Under consideration for publication in Formal Aspects of Computing

Formal Semantics of an Extended Hierarchical State Transition Matrix using CSP

Yoriyuki Yamagata[1], Weiqiang Kong[2], Akira Fukuda[2],
Nguyen Van Tang[1], Hitoshi Ohsaki[1], and Kenji Taguchi[1]
[1]National Institute of Advanced Science and Technology, Amagasaki, Japan
[2]Kyushu University, Fukuoka, Japan

Abstract.
The Extended Hierarchical State Transition Matrix (EHSTM) is a table-based modeling language frequently used in industry for specifying behaviours of a system. However, assuring correctness, i.e., having a design satisfy certain desired properties, is a non-trivial task. To address this problem, a model checker dedicated to EHSTMs called Garakabu2 has been developed. However, there is no formal justification for Garakabu2, since its semantics has never been fully formalised. In this paper, we give a formal semantics to EHSTM by translating it into CSP, Communicating Sequential Processes. Our semantics covers most of the features supported by Garakabu2. We manually translate the small examples of EHSTM to CSP, and verify them by PAT, a CSP-based model checker. We also verify the examples directly using Garakabu2 and show the results are same. The experiments also show that verification using our translation and PAT is much faster than that of Garakabu2.

Keywords: embedded system, software modelling, formal semantics, model checking, CSP

1. Introduction

The Extended Hierarchical State Transition Matrix (EHSTM) [Wat98] is a table-based modelling language for specifying behaviours of systems. In an EHSTM, the horizontal axis declares the possible states of a system under consideration; the vertical axis declares the possible events that may occur to the system, and a row-column intersection cell declares behaviours of the system when the designated event is dispatched (occurs) in the designated state. Further, EHSTMs can form a hierarchy. At a point in its execution, the parent (upper) EHSTM T_1 can "delegate" all invocations of events to a child (lower) EHSTM T_2 and wait until T_2 "returns" the control. T_2 in turn, waits until T_1 calls T_2, then processes all events which occur until it reaches the "return" cell. In addition, events and states can also form hierarchies. A whole system can be described by defining each of its sub-systems as a hierarchy of EHSTMs (called a task) and by the tasks that communicate via shared variables or message passing.

EHSTMs has been used frequently in industry to develop software designs. According to a survey [Ass12] which JASA (Japan Embedded System Technology Association) conducted at ET 2011 (Embedded Technology), one of the largest conventions for embedded technology in Japan, 45% of developers are using state transition tables as a design method. In addition, the modelling tool ZIPC, which employs EHSTMs to describe software designs, has top market share among non-UML based modelling tools. Despite this popularity, assuring correctness, i.e., having the EHSTM design satisfy certain desired properties remains a non-trivial task. On the one hand, designers may inadvertently introduce subtle logical errors into an EHSTM design, especially when the design involves multiple communicating tasks, which makes it difficult for designers to maintain an overall image of the whole design. On the other hand, there is a lack of mechanised verification support for EHSTM designs, e.g., ZIPC provides facilities for syntactic check only.

To provide a mechanised verification support, Garakabu2 [KSK$^+$11], [KKQ$^+$11] has been developed. Garakabu2 translates the EHSTM design into formulae whose validity can be checked by an SMT solver. Thus, Garakabu2 performs a bounded model checking [BCCZ99] of EHSTM designs by using a Satisfiability Modulo Theories (SMT) solving technique [BSST]. More specifically, in this approach, all execution sequences within a given bound of an EHSTM design and the negation of a LTL-property to be checked, are encoded into a quantifier-free formula whose satisfiability is to be checked with respect to some decidable background theories, such as the theory of integers and the theories of various data structures such as arrays. Satisfiability of the resulting 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 to some bad behaviours of the design.

One of the remaining problems is, then, to justify the encoding from EHSTM designs to SMT formulae. There are three reasons why justification of the encoding is difficult. The first reason is that the semantics of EHSTMs is given by a simulator and a generated C code. Thus, there are ambiguities in the semantics of EHSTMs. The second reason is that EHSTMs introduce a non-trivial extension such as hierarchies of tables and events to a simple transition table. Therefore, we cannot just look the translation and believe its correctness. The third reason is that the rules for encoding EHSTMs to SMT formulae are buried in C++ code and have never been fully formalised (partial formulation, however, can be found in [KSK$^+$11], [KKQ$^+$11]).

To address this problem, we introduce a translation of a subset of EHSTMs into CSP [Hoa04], [RHB98]. Among other formal methodologies, we choose CSP since CSP is a language that models concurrent processes, and thus is suitable for interpreting EHSTM designs which are inherently parallel. Further, CSP has a long history of research, and is equipped with a well-defined operational and denotational semantics. Thus, the translation gives EHSTMs a formal semantics, albeit in an indirect way. CSP is a concise and relatively high-level language; thus, CSP can facilitate discussions among non-experts. Finally, CSP is supported by good verification tools such as FDR2 (http://www.fsel.com) and PAT [SLD09, LSD11].

We translate only a subset of EHSTMs; this subset covers the basic functionality supported by Garakabu2. By translating EHSTMs into CSP, we determine the semantics of composed statements, event virtual frames, event hierarchies, state virtual frames and parallel states with/without synchronisation. Also, we consider the problem of the atomicity of compound actions. Further, we compare the results and time required for verification of EHSTMs by Garakabu2, and our translation of these EHSTMs by PAT. We find that by using our approach, verification is much faster than Garakabu2, at least on the small examples we use in the experiments.

As stated earlier, Garakabu2 is an automated model checker for EHSTMs. Garakabu2 translates EHSTMs designs into formulae which are interpreted by SMT solvers, and performs bounded model checking. The translation from EHSTMs into a formal language by Garakabu2 can be seen as a sort of formal semantics of EHSTMs. However, translation rules are buried in C++ code and have never been fully formalised. Although Garakabu2 handles such as event virtual frames, event hierarchy and state virtual frame, the method to handle these features has not been published to date. By using CSP we have an advantage in that we can use all CSP-related research for EHSTMs. In particular, we apply PAT to our translation, and find that the time required for verification is greatly reduced from Garakabu2.

In related work, statecharts appear closely related to EHSTMs. Statecharts are diagrams which capture state transitions. Statecharts have similarities to EHSTMs such as submachine and hierarchical states. Their standard semantics is given by Harel and Naamad [HN96]. Uselton and Smolka introduce compositional semantics for statecharts [US94]. Although comparison with our work is difficult, we note that they all assume the notion of *steps* in the semantics, that is, all actions executable are executed in one step, and then the next inputs only take effect in the next step. On the contrary, our semantics assumes that the inputs and

the actions take effect one by one in an unspecified order. This choice is justified because ZIPC translates concurrent actions into multi-task C programs.

There are many works on formal semantics and formal verification on statecharts, in particular in the context of UML. Zhang and Liu [ZL10] give a translation of UML state machines into CSP. Their translation is quite similar to ours, but our translation is complicated by the presence of a "global transition" and the semantics of "return" in EHSTMs. For additional references, the reader is encouraged to refer to Zhang and Liu [ZL10].

This paper is organised as follows. In Section 2, we introduce CSP. In Section 3, we introduce EHSTM and its translation into CSP using examples which gradually become more complex. In Section 4, we show a formal definition of the translation from EHSTM to CSP. In Section 5, we compare the results and time required to verify the several properties of EHSTMs and their translation into CSP using Garakabu2 and PAT respectively. Finally, we present our conclusion and propose future research in Section 6.

2. CSP

We use our own definition of *CSP*, which is an extension of *CSP#* [SLD09]. Comparing traditional CSP [Hoa04], [RHB98] CSP# has the advantages of having built-in global variables and channels with finite queues. Further, we introduce several notions which do not appear in CSP#, such as concatenation of alphabets and records. Since they can be translated to the traditional CSP, we use such constructs freely.

CSP is a language for concurrent computation. CSP models concurrent computation through a set of processes that communicate with each other. Each process is composed of a sequence of *events*. To construct events, we assume *alphabets* $a, b, a.b, a.b.c, \ldots$ which act as abstract events, or values of expressions and names of global variables. Events are either abstract event e, or assignments $x := exp$, or an action $c!exp$ that sends the value exp to a channel c where exp is a C-like expression without side-effects, or an action $c?x$ which receives a value from the channel c and assign it to x. In addition to integers and real numbers, we allow alphabets and records as values in our version of CSP. If v stores an alphabet, expression in the form $(v).a$ is the expression which concatenates the value of the variable v with a. If v stores a record, its field can be accessed by v.name. v.name can be used as a value in expressions. v.name $:= a$ substitutes a into the name field of a record stored in v. To denote a value explicitly, we use the notation $\{\text{field1} = a, \text{field2} = b, \ldots\}$.

Definition 2.1 (CSP processes).

$$\begin{aligned} P ::= {} & STOP \mid SKIP \mid \text{assert(false)} \mid e \to P \mid P_1 \lhd b \rhd P_2 \\ & \mid P_1 \,||_A\, P_2 \mid P_1; P_2 \mid P_1 \triangle_A P_2 \mid P \backslash A \\ & \mid P_1 | \cdots | P_n \end{aligned} \tag{1}$$

where e is an event, b is a Boolean expression, and A is a set of events. As for the special case, we write $||_\emptyset$ by $|||$.

We briefly review the meaning of each construct of the process expressions.

The process *STOP* does nothing and immediately terminates abnormally. The process *SKIP* does nothing and immediately terminates successfully. The process assert(false) causes termination of the entire system.

The process $e \to P$ first executes e and then behaves as P. The process $P_1 \lhd b \rhd P_2$ first evaluates b and if b is true, then the process behaves as P_1. Otherwise, the process behaves as P_2.

The process $P_1||_A P_2$ is obtained by running P_1 and P_2 interleavingly using events in A as synchronisation points. Thus, $P|||Q$ means that P and Q interleave without synchronisation between the 2 processes.

The process $P; Q$ first behaves like P, and after P terminates successfully, then $P; Q$ executes Q. If P terminates abnormally, $P; Q$ terminates abnormally too. The process $P \triangle_A Q$ first behaves like P, but if an event in A occurs in the execution of P, the process behaves like Q. Usually, an event which causes an interrupt comes from the outside of P and Q, but we depart from this usual definition. Hiding $P \backslash A$ represents the process obtained by "hiding" all events which belong in A from P. The process $P_1 | \cdots | P_n$ first waits for one of $P_1, \ldots, P_n$ to become active, and executes the activated process. This construct is called *choice*.

3. EHSTM and its interpretation by CSP

In this section, through examples, we present a subset of EHSTMs and its translation to CSP. We progressively define the translation, by first translating a simple EHSTM, an EHSTM with a event virtual frame, state virtual frame, parallel states with/without synchronisation, a event hierarchy, and compound actions. Finally we consider EHSTMs in multiple tasks.

Each EHSTM is contained in a task. Here, a task represents the set of tables that are sequentially executed. EHSTMs that belong to a task form a hierarchy, which we later explain in detail. Each task is either a *message type* or a *flag type*. If the task is a message type, the EHSTMs of this task share a message queue. If the task is a flag type, EHSTMs of this task share a flag. In this and the next section, we only consider the message type task, since the flag type task can be treated similarly.

We translate states of an EHSTM as values in global variables; events of an EHSTM as events of CSP and an EHSTM as a process. We translate parallel states as tuples of states. We translate hierarchical EHSTMs using parallel composition and interleaving. We translate multiple tasks using interleaving.

3.1. Simple EHSTM

An EHSTM in its simplest form is just a state transition table in automata theory.

T	S_0	S_1
e_0	S_1 P_1	...
e_1	...	...

(2)

If we have the EHSTM above, and the event e_0 occurs when the EHSTM is in the state S_0, then the EHSTM performs action P_1 and changes the state to S_1.

The simplest translation of this table to CSP is to translate states as processes. However, to encode a *global transition* (see below), we need to change the state of the table from the other tables. To achieve this, we need to encode (current) states as global variables.

The table is interpreted by choice and recursion. The states and the events of an EHSTM are interpreted as alphabets. For each EHSTM T, we create the global variable T.state which stores the current state of the EHSTM.

$$\begin{aligned} T = \ & (T.\text{state}?S_0 \to (a?e_0 \to P_1; (T.\text{state} := S_1); T) \\ & \qquad\qquad \mid (a?e_1 \to ...)) \\ & \mid (T.\text{state}?S_1 \to ...) \end{aligned} \tag{3}$$

P_1 can contain atomic actions, assignments to global variables, and can send events e to queue q which is encoded as action $q!e$. The entire system $(T.\text{state} := S_0); T$ is obtained from T by initialising the T.state to the default state S_0.

If the transition is omitted, the EHSTM does not change its state after action P_1 is executed.

T	S_0	S_1
e_0	– P_1	...
e_1	...	...

(4)

In this case, there is no assignment to the T.state in the corresponding clause.

$$\begin{aligned} T = \ & T.\text{state}?S_0 \to (a?e_0 \to P_1; T \mid a?e_1 ...) \\ & \mid T.\text{state}?S_1 \to ... \end{aligned} \tag{5}$$

If the entire cell is a crossed $\times$, this means that the combination of the event e_0 and the state S_0 will never happen, or equivalently, if such a combination occurs, the system will terminate abnormally. Such cells are represented by assert(false).

T	S_0	S_1
e_0	$\times$	...
e_1	...	...

(6)

$$\begin{aligned} T = \;& T.\text{state}?S_0 \rightarrow (a?e_0 \rightarrow \text{assert(false)} \mid a?e_1 ...) \\ & \mid T.\text{state}?S_1 \rightarrow ... \end{aligned} \tag{7}$$

3.2. Event virtual frame

Events can form a tree, as shown next. In this example, if the EHSTM receives event e_0, then the EHSTM waits for events e_2 or e_3. If the EHSTM receives e_2 then it executes P_1 and changes its state to S_1. If the EHSTM receives e_3 then it executes and changes the state as specified in the corresponding cell. If the EHSTM does not receive e_2 or e_3, then the behaviour is undefined. We call e_0 an event virtual frame.

T		S_0	S_1
e_0	e_2	S_1 / P_1	...
	e_3	...	...
e_1		...	...

(8)

This behaviour is interpreted by successive choices.

$$\begin{aligned} T = \;& T.\text{state}?S_0 \rightarrow (a?e_0 \rightarrow \; (a?e_2 \rightarrow ... | a?e_3 \rightarrow ...) | a?e_1 ...) \\ & \mid T.\text{state}?S_1 \rightarrow ... \end{aligned} \tag{9}$$

Another way to interpret the event virtual frame would be to interpret the table using *hierarchical alphabets* $e_0.e_1, e_0.e_2, \ldots$.

$$\begin{aligned} T = \;& T.\text{state}?S_0 \rightarrow \; (a?e_0.e_2 \rightarrow ... | a?e_0.e_3 \rightarrow ... | a?e_1 ...) \\ & \mid T.\text{state}?S_1 \rightarrow ... \end{aligned} \tag{10}$$

This makes the branching on events an atomic operation. But according to the information obtained from CATS engineers, ZIPC interprets hierarchical events by nested if-then-else clauses, thus the branching is not an atomic operation.

3.3. State virtual frame

States can also form a tree. In the next example, S_1 has sub-states S_2 and S_3 and S_3 has sub-states S_4 and S_5. We call S_1, S_3 state virtual frames. $^\blacktriangledown$ means that S_2 and S_5 are *default states*.

T	S_0	S_1		
		S_2	$^\blacktriangledown S_3$	
			S_4	$^\blacktriangledown S_5$
e_0	$S_1(F)$ / P_1	...		
e_1	$S_1(M)$ / P_2	...		
e_2	$S_1(D)$ / P_3	...		
e_3	S_1 / P_3	...		

(11)

Transitions to a state virtual frames are divided into *fixed transitions*, *memorised transitions* and *deep memorised transitions*. The type of transition is specified by putting symbols (F), (M), or (D) after the transition. (F) represents a fixed transition, (M) represents a memorised transition and (D) represents a deep memorised transitions.

A fixed transition $S_1(F)$ first changes the system's state to the virtual state S_1. Then, the system changes its state to the default sub-state (the state with $^\blacktriangledown$). If the sub-state also has sub-states, the system further changes its state to the default sub-states recursively. In the example above, $S_1(F)$ causes the transition to S_2. But if S_3, and not S_2 is the default state, $S_1(F)$ causes the transition to S_5.

A memorised transition $S_1(M)$ first changes the system's state to the virtual state S_1. Then, the system changes its state to the sub-state which is activated the last time. If the state also has sub-states, the system further changes its state to the default sub-states recursively. In the example above, $S_1(M)$ causes the transition to S_3 if either S_4 or S_5 are activated the last time. After this, the system further changes its state to the default sub-states S_5.

A deep memorised transition $S_1(D)$ first changes the system's state to the virtual state S_1. Then, the system changes its state to the sub-state which is activated the last time, similarly to a memorised transition. If the sub-state also has sub-states, however the system further changes its state to the sub-states which is visited the last time recursively. In the example above, $S_1(M)$ causes the transition to S_3 if S_4 is activated the last time. After this, the system further changes its state to the last visited state S_4.

If the transition S_1 does not have notations $(F), (M)$, or (D), deep memorised transition is assumed.

This behaviour is interpreted by memories $S_1.\mathrm{mem}, S_3.\mathrm{mem}$ and the default children $S_1.\mathrm{default}$ and $S_3.\mathrm{default}$

$$
\begin{aligned}
T = \;& T.\mathrm{state}?S_0 \to a?e_0 \to P_1; S_1.\mathrm{mem} := *; T.\mathrm{state} := S_1; T \\
& \quad \mid a?e_1 \to P_2; (S_1.\mathrm{mem}).\mathrm{mem} := *; T.\mathrm{state} := S_1; T \\
& \quad \mid a?e_2 \to P_2; T.\mathrm{state} := S_1; T \\
& \quad \mid a?e_3 \to P_2; T.\mathrm{state} := S_1; T \\
& \mid T.\mathrm{state}?S_1 \to T.\mathrm{state} := S_1.\mathrm{mem} \lhd S_1.\mathrm{mem} = * \rhd (S_1.\mathrm{default}).\mathrm{mem} := *; T.\mathrm{state} := S_1.\mathrm{default}; T \\
& \mid T.\mathrm{state}?S_2 \to S_1.\mathrm{mem} := S_2; ...; T \\
& \mid T.\mathrm{state}?S_3 \to S_1.\mathrm{mem} := S_3; (T.\mathrm{state} := S_3.\mathrm{mem} \lhd S_3.\mathrm{mem} = * \rhd \\
& (S_3.\mathrm{default}).\mathrm{mem} := *; T.\mathrm{state} := S_3.\mathrm{default}); T \\
& \mid T.\mathrm{state}?S_4 \to S_3.\mathrm{mem} := S_4; ...; T \\
& \mid T.\mathrm{state}?S_5 \to S_3.\mathrm{mem} := S_5; ...; T
\end{aligned} \tag{12}
$$

Each super-state S has a memory for the history $S.\mathrm{mem}$. The contents of $S.\mathrm{mem}$ are used for memorised and deep memorised transitions. Deep memorised transitions are the simplest to interpret, since they just follow the contents of memories. A fixed transition first resets the memory ($S.\mathrm{mem} := *$), and then goes to target state S. Then, S goes to the default state. In addition, S resets the contents of the memory of its default sub-state. Thus the sub-state further makes a transition to its default sub-state. Interpretation of a memorised transition is a bit tricky. Before going to the target state, a memorised transition resets the memory of $S.\mathrm{mem}$, that is, the sub-state of S which is visited for the last time. Therefore, the transition follows the memory of S, but then chooses default sub-states to follow.

3.4. Parallel states

EHSTMs can be in multiple states at once, if the states are *parallel* states. In the next example, $^\circ$ signifies that states S_3, S_4 are parallel states, that is, the matrix can be in the states S_3, S_4 simultaneously. Our notation for parallel states is different from that of ZIPC. In ZIPC, parallel states are signified by surrounding state names in dashed lines.

In the next example, if the matrix moves its state to S_1, then the matrix enters multiple states S_2 and S_3. Therefore, the matrix can be simultaneously in S_4 or S_5 and S_6 or S_7. Finally, if the transition to S_0 occurs in one of the state, then the matrix makes a transition to S_0, and another state is discarded.

T	S_0	S_1				
		S_2°		S_3°		
		▼S_4	S_5	▼S_6	S_7	
e_0	S_1	S_5	S_0	S_7		-
	...	...	...	...		
e_1	...	...	...	...		S_0
						...

(13)

This behaviour is translated by allowing $T.\mathrm{state}$ to have a tuple of the states.

$$
\begin{aligned}
T = \ & T.\text{state}?S_0 \to a?e_0 \to ...; T.\text{state} := S_1; T \\
& \qquad\qquad\quad | \ a?e_1 \to ... \\
& | \ T.\text{state}?S_1 \to T.\text{state} := \{S_2 = S_2, S_3 = S_3\}; T \\
& | \ T.\text{state}.S_2?S_2 \to T.\text{state}.S_2 := S_2.\text{mem}; T \\
& | \ T.\text{state}.S_3?S_3 \to T.\text{state}.S_3 := S_3.\text{mem}; T \\
& | \ T.\text{state}.S_2?S_4 \to a?e_0 \to S_2.\text{mem} := S_4; ...; T.\text{state}.S_2 := S_5; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_2?S_5 \to a?e_0 \to S_2.\text{mem} := S_5; ...; T.\text{state} := S_0; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_3?S_6 \to a?e_0 \to S_3.\text{mem} := S_6; ...; T.\text{state}.S_3 := S_6; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_3?S_7 \to a?e_0 \to S_3.\text{mem} := S_7; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 \to ...; T.\text{state}; = S_0; T
\end{aligned} \tag{14}
$$

In this translation, when the matrix is in the parallel states, the state of the matrix becomes a tuple of states which run parallelly. The state to run is chosen by nondeterministic choices. Thus, each parallel states run interleavingly without a priority.

In the previous example, if one state transits to the outside of the state virtual frame, other parallel states terminate immediately. However, in the next example, the transition to the outside of the virtual frame is synchronised. $^\bullet$ signifies that S_1 requires its children to exit to the same state simultaneously. We call such states synchronised states. Again, our notation is different from that of ZIPC. In ZIPC, synchronised is signified by surrounding state names in doubled-lines.

T	S_0	$S_1^\bullet$			
		S_2°		S_3°	
		$\blacktriangledown S_4$	S_5	$\blacktriangledown S_6$	S_7
e_0	S_1	S_5	S_0	S_7	-
	...	...	...	...	
e_1	...	...	...	...	S_0
					...

(15)

This behaviour is translated by changing the translation of the transition to S_0.

$$
\begin{aligned}
T = \ & T.\text{state}?S_0 \to a?e_0 \to ...; T.\text{state} := S_1; T \\
& \qquad\qquad\quad | \ a?e_1 \to ... \\
& | \ T.\text{state}?S_1 \to T.\text{state} := \{S_2 = S_2, S_3 = S_3\}; T \\
& | \ T.\text{state}.S_2?S_2 \to T.\text{state}.S_2 := S_2.\text{mem}; T \\
& | \ T.\text{state}.S_3?S_3 \to T.\text{state}.S_3 := S_3.\text{mem}; T \\
& | \ T.\text{state}.S_2?S_4 \to a?e_0 \to S_2.\text{mem} := S_4; ...; T.\text{state}.S_2 := S_5; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_2?S_5 \to a?e_0 \to S_2.\text{mem} := S_5; ...; T.\text{state}.S_2 := S_0; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_3?S_6 \to a?e_0 \to S_3.\text{mem} := S_6; ...; T.\text{state}.S_3 := S_6; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 ... \\
& | \ T.\text{state}.S_3?S_7 \to a?e_0 \to S_3.\text{mem} := S_7; T \\
& \qquad\qquad\qquad\quad | \ a?e_1 \to ...; T.\text{state}.S_3 := S_0; T \\
& | \ T.\text{state}?\{S_2 = S_0, S_3 = S_0\} \to T.\text{state} := S_0; T
\end{aligned} \tag{16}
$$

Instead of $T.\text{state} := S_0$, the synchronised states changes its own state to S_0 ($T.\text{state} := \langle p_1(T.\text{state}), S_0\rangle$). We add the new clause $T.\text{state}?\langle S_0, S_0\rangle \to T.\text{state} := S_0; T$ so if all states of the synchronised states are S_0, the entire state of the matrix is changed to S_0.

3.5. Event Hierarchy

EHSTMs can form a hierarchy. In the next example, T_0 is a *root EHSTM* and $T_{0.1}$ is a *child EHSTM*. The root EHSTM, that is, the EHSTM without parents, is denoted by T_0. There is only one root EHSTM in each task. EHSTMs can *call* their child EHSTMs as in the next example. EHSTMs can call only their direct children. They cannot call their indirect descendants.

The execution starts from the default state of T_0. Transition occurs inside of T_0 as a simple EHSTM, until T_0 executes the action $\square T_{0.1}$. Then "control" is transferred to the child EHSTM $T_{0.1}$. The child EHSTM $T_{0.1}$ receives all events and makes transitions until it executes a "return" action.

T_0	S_0	S_1	S_2
e_0	S_1 $\square T_{0.1}$	...	...
e_1	...	...	...

(17)

$T_{0.1}$	S_3	S_4
e_2	S_4	S_3
	P_1	return
e_3	...	...

(18)

T_0 is interpreted as in the case of a simple EHSTM, but the action is replaced by the *call* of the child EHSTM.

$$\begin{aligned} T_0 = \; & T_0.state?S_0 \to (a?e_0 \to \text{call}(T_{0.1}); T_0.state := S_1; T_0 \\ & \qquad\qquad \mid a?e_1 ...) \\ & \mid T_0.state?S_1 \to ... \end{aligned} \tag{19}$$

where $\text{call}(T_{0.1})$ is defined by

$$\text{call}(T_{0.1}) = T_{0.1}.start \to T_{0.1}.return \to SKIP. \tag{20}$$

The child EHSTM is interpreted as a simple EHSTM but the "return" construct of an action field needs special attention. Construct "return" is interpreted as a process $return = T_{0.1}.return \to T_{0.1}.start \to SKIP$, which means that it returns the control to the caller, and waits for the next call.

$$\begin{aligned} T_{0.1} = \; & T_{0.1}.state?S_3 \to (a?e_2 ...) \\ & \mid T_{0.1}.state?S_4 \to (a?e_2 \to return; T_{0.1} \mid ...) \end{aligned} \tag{21}$$

The whole task is translated into the initialisation and the parallel composition

$$(T_0.state := S_0); (T_{0.1}.state := S_3); \cdots ; (T_0 ||_A (T_{0.1}.start \to T_{0.1})). \tag{22}$$

We must guard $T_{0.1}$ by $T_{0.1}.start$: otherwise, $T_{0.1}$ begins receiving events before $T_{0.1}$ is called. Even-though T_0 and $T_{0.1}$ appear to run in parallel, since $T_{0.1}$ is prefixed by $T_{0.1}.start$, $T_{0.1}$ never runs except when it is called by the root EHSTM.

If T_0 has multiple children, then we translate each EHSTM as before, and compose all children by interleaving.

$$\begin{aligned} & (T_0.state := S_0); (T_{0.1}.state := S_3); (T_{0.2}.state := S_5); \cdots ; \\ & \qquad\qquad (T_0 ||_A ((T_{0.1}.start \to T_{0.1}) ||| (T_{0.2}.start \to T_{0.2}) ||| ...)). \end{aligned} \tag{23}$$

If the EHSTM contains a notation such as $T_0 > S_2/S_4$, then the EHSTM has a *global transition*. In the example below, if e_2 occurs when $T_{0.1}$ is in state S_3, then $T_{0.1}$ changes its state to S_4. In addition,

$T_{0.1}$ changes the state of T_0 to S_2. This change occurs before $T_{0.1}$ returns the control to T_0. Therefore if T_0 specifies a certain transition after $\Box T_{0.1}$, the change T_0 to S_2 is overridden after the control returns to T_0.

$T_{0.1}$	S_3	S_4
e_2	$T_0 > S_2/S_4$ P_1	...
e_3	...	...

(24)

To interpret global transitions, we only need to append the assignment $(T_0.state := S_2)$ to the action P_1. Therefore, the translation of $T_{0.1}$ is defined by

$$\begin{aligned} T_{0.1} =& T_{0.1}.start \rightarrow \\ & T_{0.1}.state?S_3 \rightarrow (a?e_2 \rightarrow P_1'; T_{0.1}.return \rightarrow T_{0.1} \mid ...) \\ & \mid T_{0.1}.state?S_4 \rightarrow ... \end{aligned} \tag{25}$$

where $P_1' = P_1; (T_0.state := S_2); (T_{0.1}.state := S_4)$. The translation of the root EHSTM is not changed.

Before going to the next section, we discuss the difference of our semantics and that of ZIPC briefly. In fact, Garakabu2 and ZIPC employ different semantics for hierarchical events. In the interpretation by Garakabu2, the control which is transferred to a child EHSTM returns the parent EHSTM only when "return" command is executed in the child EHSTM. On the other hand, in the interpretation of ZIPC, the control is returned whenever the actions in a cell is executed. According to CATS engineers, the both interpretations are acceptable. In this paper, we follow the interpretation by Garakabu2.

3.6. Compound Action

So far, all EHSTMs that appeared contain only "simple" actions. Now let us consider the composite actions. First, we consider the case where the actions do not contain "return". In this case, the translation is straightforward. Each action is translated as follows.

$v := e$ is translated into $v := e$ itself.

The child EHSTM call is translated into a $call(T)$

event(e, t) which sends event e to task t, and is translated into $t.q!e$

The atomic action is translated into itself, assuming that such action is among abstract events.

"$s; P$" is interpreted as the sequential composition of s and P, and "if b then P_1 else P_2" is interpreted as the Boolean choice $[\![P_1]\!] \lhd b \rhd [\![P_2]\!]$ where $[\![P_1]\!]$ and $[\![P_2]\!]$ are translations of P_1 and P_2 respectively.

For example, "if $z == 0$ then $x := y + 1$ else $x := y - 1$" is translated into "$x := y + 1 \lhd z == 0 \rhd x := y - 1$".

Note that in this translation, actions in a single cell are not executed atomically. This is different from the behaviour of Garakabu2, in which the actions in a single cell are executed atomically. To realise this Garakabu2 semantics, we use an "atomic" construct in CSP#. In the next section, we use this translation to compare PAT and Garakabu2. Arguably, translation without an "atomic" construct makes the semantics closer to the informal semantics of ZIPC, which is determined by translation into C.

Next, we consider the actions which include "return". "return" is translated into $T.return \rightarrow SKIP$. The other actions are translated as before. Let P be a CSP process obtained by this translation. We translate the entire action by

$$P \triangle_{\{T.return\}} T.straightforwardart \rightarrow SKIP \tag{26}$$

using interrupt. Interrupt is used to discard the actions after "return". For example

```
if z == 0 then return else SKIP;
z := 0;
```

is translated into

$$((T.return \rightarrow SKIP) \lhd \ z == 0 \rhd SKIP; z := 0) \triangle_{\{T.return\}} T.start \rightarrow SKIP \tag{27}$$

Our translation is easy to extend to a system with multiple tasks. Multiple tasks are translated into the interleaving of the translations of tasks. The action sending an event e to a message queue $t_i.q$ of task t_i is translated into $t_i.q!e$.

4. Formal Semantics

In this section, we present a formal definition of *system design* D by EHSTMs [KKQ+11] and its translation. We denote the set of compound actions by $\mathcal{L}_{act}$ and Boolean expressions by $\mathcal{L}_B$. Also we denote the set of all expressions by $\mathcal{L}_{exp}$.

D is a finite set of *tasks* $t_1, \ldots, t_n$ which are executed in an interleaving manner.

Each task t consists of a hierarchy of EHSTMs H_0, $H_{0.0}$, $H_{0.1}$, ... and a FIFO message queue $t.q$ which is equipped with operations $\mathrm{pop}(t.q), \mathrm{push}(t.q)$, predicate $\mathrm{empty}(t.q)$, and $\mathrm{full}(t.q)$. There is only one root EHSTM in each task.

Each EHSTM H is a tuple $\langle S, E, C\rangle$. S is a tree whose nodes except the root are labelled by *states* $S_0, S_1, \ldots, S_n$. Each state S_i is used as a label only once in the entire system design. Also, each node can be labelled by $^{\blacktriangledown}$,$^{\circ}$,$^{\bullet}$. If a node is labelled by $^{\blacktriangledown}$, we call the node a default state. If a node is labelled by $^{\circ}$, all its siblings must be labelled by $^{\circ}$. We call the nodes labelled by $^{\circ}$ parallel states. If a node is not labelled by $^{\circ}$, we call the node an exclusive state. If the parent of parallel states is marked by $^{\bullet}$, its children are synchronised.

Let $L(T)$ be the set of leafs of T. E is a tree whose nodes except the root are labelled. Labels are either *events* or Boolean expressions in $\mathcal{L}_B$ called *guards*. C is a map from an element of $L(S) \times L(E)$ to a *cell*. The set of cells is divided into normal cells C_{nor}, ignore cells C_{ign} and invalid cells C_{inv}. A normal cell $c_N \in C_{nor}$ is a tuple $\langle a, t\rangle \in (\mathcal{L}_{act} \cup \{/\}) \times R$ where R is a sequence of transitions or $-$. A transition is either the form $S_i(F), S_i(M), S_i(D)$ where $S_i \in S$, or a form $T > S_i(F), T > S_i(M), T > S_i(D)$ where T is an EHSTM. An ignore cell is represented by the symbol / and an invalid cell by the symbol $\times$.

We translate a system design D to CSP as follows. We write the translation function by $[\![\cdot]\!]$. We construct the translation in a top-down fashion.

A system design $D = t_1, \ldots, t_m$ is translated using interleaving.

$$[\![t_1]\!] \;|||\; [\![t_2]\!] \;|||\; \cdots \;|||\; [\![t_m]\!]\,. \tag{28}$$

The translation of a task is defined by recursion on the tree structure of the matrices in the task. Let G be the translation function. Assume that $T_{i_1.....i_k}$ has children $T_{i_1.....i_k.0}, \ldots, T_{i_1.....i_k.n_{i_1.....i_k}}$. Then, we define

$$\begin{aligned} &G(T_{i_1...i_k}) = \\ &\big[T_{i_1.....i_k} ||_{A_{i_1.....i_k}} \{(T_{i_1.....i_k.0}.start \to G(T_{i_1.....i_k.0}))||| \ldots ||| \\ &\qquad\qquad (T_{i_1.....i_k.n_{i_1.....i_k}}.start \to G(T_{i_1.....i_k.n_{i_1.....i_k}}))\}\big]. \end{aligned} \tag{29}$$

Let t be a task whose root EHSTM is T_0. Using G, we define the translation $[\![t]\!]$ of the task by $G(T_0)$ plus an initialisation code. An initialisation code first initialises the states of all EHSTMs in t, and next initialises all S.default and S.state for all states S which appear in t, finally resets (substitutes *) all memory S.mem. The translation of each EHSTM $T = \langle S, E, C\rangle$ is defined by gathering the interpretation of S. First, we associate each node $S_i \in S \backslash *_S$ where $*_S$ is the root of S with $[\![S_i]\!]$. Let σ be the sequence of parents of parallel states which appear in the path from the root of S to S_i. Let $S\uparrow$ be the parent of S_i and $S^1, \ldots, S^k$ be the children of S_i. If S_i is a leaf, let $[\![S_i]\!]$ be $T.\mathrm{state}.\sigma?S_i \to S\uparrow.\mathrm{mem} := S_i; [\![*_E]\!](\sigma, S_i)$, and T where $[\![*_E]\!](\sigma, S_i)$ is defined later. For the node which is neither the root nor a leaf, we distinguish two cases.

$S^1, \ldots, S^k$ are exclusive states. Let $[\![S_i]\!]\,\sigma$ be

$$\begin{aligned} &T.\mathrm{state}.\sigma?S_i \to S\uparrow.\mathrm{mem} := S_i; (T.\mathrm{state}.\sigma := S_i.\mathrm{mem} \lhd S_i.\mathrm{mem} = * \rhd \\ &\qquad\qquad (S_i.\mathrm{default}).\mathrm{mem} := *; T.\mathrm{state}.\sigma := S_i.\mathrm{default}); T. \end{aligned} \tag{30}$$

$S^1, \ldots, S^k$ are parallel states. Let $[\![S_i]\!]\,\sigma$ be

$$T.\mathrm{state}.\sigma?S_i \to S\uparrow.\mathrm{mem} := S_i; T.\mathrm{state}.\sigma := \{S^1 = S^1, \ldots, S^k = S^k\}; T. \tag{31}$$

We gather these $[\![S_i]\!], i = 1, \ldots, n$ by choice operators and define $[\![T]\!]$. To interpret synchronised parallel states, we add further clauses. For each synchronised parallel state $S^1, \ldots, S^k$, let $M_{S^1,\ldots,S^k,S_i}$ be $T.\text{state}.\sigma?\{S^1 = S_j, \ldots, S^k = S_j\} \to T.\text{state}.\tau := S_j$ where σ is the sequence of parents of parallel states which appears in the path from the root of S to $S^1, \ldots, S^k$ and τ be the sequence obtained from σ by dropping the last element of σ. If τ is empty, we consider $T.\text{state}.\tau \equiv T.\text{state}$. We define $[\![T]\!]$ be

$$[\![S_1]\!] \mid \cdots \mid [\![S_n]\!] \mid \cdots \mid M_{S^1,\ldots,S^k,S_j} \mid \cdots \tag{32}$$

where $S^1, \ldots, S^k$ are synchronised parallel states and $S_j \in S$.

$[\![*_E]\!](\sigma, S_i)$ is defined by the recursion on E. We associate each leaf e of E with $[\![C(S_i, e)]\!]\sigma$ which is defined later. If $e \in E$ is not a leaf, it has children $e_1, \ldots, e_l$. We define $[\![e]\!](\sigma, S_i)$ as

$$e_1 \to [\![e_1]\!](\sigma, S_i) \mid \cdots \mid e_l \to [\![e_l]\!](\sigma, S_i). \tag{33}$$

$[\![*_E]\!](\sigma, S_i)$ is obtained by letting e be $*_E$.

Let C be a cell. We define $[\![C]\!]\sigma$ by distinguishing three cases of the form of C. If C is an ignore cell, $[\![C]\!]\sigma =_{\text{def}} SKIP$. If C is an invalid cell, $[\![C]\!]\sigma =_{\text{def}} \text{assert(false)}$. Otherwise, C is a normal cell. C is a tuple $\langle a, t\rangle$ where a is a (compound) action and t is the sequence $t_1, \ldots, t_m$ of transitions or $-$. In Section 3.6, the translation $[\![a]\!]$ of a is already defined. If t is $-$, the translation $[\![t]\!]$ is $SKIP$. Otherwise, t is the sequence $t_1, \ldots, t_m$. The translation $[\![t]\!]\sigma$ is $[\![t_1]\!]\sigma; \cdots; [\![t_m]\!]\sigma$. Let j be the one of $1, \ldots, m$. We define $[\![t_j]\!]\sigma$ analysing the location of source and target of the transition in the tree S. Let $t_j \equiv T^1 > S_i(\alpha)$ where T is an EHSTM and α is one of F, M, D. If t_j is a local transition, that is, there is no part corresponding to $T >$ in the notation t_j, we assume $T^1 \equiv T$. Let τ be the sequence of parents of parallel states from the root to the $T^1.\text{state}$. Let S_0 be the lowest (most far away from the root) synchronised state in τ and τ_0 be the subsequence of τ up to S_0. If there is no synchronised state in τ, let τ_0 be empty and $T.\text{state}.\tau_0 \equiv T.\text{state}$. We define

$$\begin{aligned}
[\![T^1 > S_i(F)]\!] &\equiv S_i.mem := *; T.\text{state}.\tau_0 := S_i \\
[\![T^1 > S_i(M)]\!] &\equiv (S_i.mem).mem := *; T.\text{state}.\tau_0 := S_i \\
[\![T^1 > S_i(D)]\!] &\equiv T.\text{state}.\tau_0 := S_i.
\end{aligned} \tag{34}$$

This completes the formal definition.

5. Experiments by PAT and Garakabu2

In this section, we compare the result of the verification obtained from our translation using PAT, and the result directly obtained from Garakabu2. We consider three examples. The first example is of EHSTMs using a hierarchy, but all EHSTMs belong to the same one task. The second is an example of EHSTMs belonging to multiple tasks, but essentially without calling-to and returning-from child EHSTMs. The third is an extension of the second example in which invocation of child EHSTMs are considered. We experiment with these three examples using PAT 3.4.4 and Garakabu2 1.1.3 (with optimisation for the type of EHSTMs that communicate with shared variables, which has not been publicly released) for 32-bit Windows. We use a machine with Intel Core i7-2640M CPU 2.80GHz, 8GB memory, and Windows 7 64-bit operating system.

5.1. Example 1

The first example is a hierarchy of EHSTMs which can reach an invalid cell (EHSTMs communicate with shared variables). This is an artificial example to show how the hierarchy in EHSTMs works.

<table>
<tr><td>$\square 0$</td><td>S_0</td><td>S_1</td><td>S_2</td></tr>
<tr><td rowspan="2">$e_0 == 0$</td><td>S_1
$e_0 = 1;$</td><td rowspan="2">$\times$</td><td>S_0
$\square 0.1;$
$e_0 = 1;$</td></tr>
<tr><td>S_1
$\square 0.2;$</td><td>S_2
$e_0 = 0;$</td></tr>
<tr><td>$e_1 == 1$</td><td colspan="2"></td><td>$\times$</td></tr>
</table>

(35)

```
enum {s0, s1, s2, s01, s02, s011, s012, s013};

var state0 = s0;  var state1 = s01; var state2 = s011;
var e0 = 0; var e1 = 0; var e2 = 0; var tmp = 0;

STM0 = [state0==s0 && e0==0]{e0=1; state0 = s1} -> STM0()
[]
[state0==s0 && e0==1]tau -> STM2_start -> STM2_return -> {state0=s1;} -> STM0()
[]
[state0==s1 && e0==0]tau -> assert(false)
[]
[state0==s1 && e0==1]{e0=0; state0=s2} -> STM0()
[]
[state0==s2 && e0==0]tau-> STM1_start -> STM1_return -> {e0=1; state0=s0} -> STM0()
[]
[state0==s2 && e0==1]tau -> assert(false);

STM1 =  [state1==s01 && e1==0]{e1=1; state1=s02} -> STM1()
[]
[state1==s01 && e1==1]tau -> STM1_return -> STM1_start -> STM1()
[]
[state1==s02 && e1==0]tau -> assert(false)
[]
[state1==s02 && e1==1]{e1=1; state1=s01} -> STM1();


STM2 =  [state2==s011 && e2==0]{e2=1; state2=s012} -> STM2()
[]
[state2==s011 && e2==1]tau -> assert(false)
[]
[state2==s012 && e2==0]{e2=1; state2=s013} ->  STM2()
[]
[state2==s012 && e2==1]{e2=0; tmp=0} -> STM2()
[]
[state2==s013 && e2==0]tau -> STM2_return -> STM2_start -> STM2()
[]
[state2==s013 && e2==1]{tmp=1; e2=1; state2=s011} -> STM2();

System = STM0() || (STM1_start -> STM1() ||| STM2_start -> STM2());
```

Fig. 1. Translated CSP# programs for the first example

$\Box 0.1$	S_{01}	S_{02}
$e_1 == 0$	S_{02} $e_1 = 1;$	$\times$
$e_1 == 1$	return;	S_{01} $e_1 = 1;$

(36)

$\Box 0.2$	S_{011}	S_{012}	S_{013}
$e_2 == 0$	S_{012} $e_2 = 1;$	S_{013} $e_2 = 1;$	return;
$e_2 == 1$	$\times$	$e_2 = 0$ $tmp = 0;$	S_{011} $tmp = 1;$ $e_2 = 1;$

(37)

We translate the EHSTMs of the first example into the CSP# programs (shown in Fig. 1) and check the reachability of invalid cells of these models by PAT and Garakabu2. In PAT, we verify the unreachability of invalid cells by verifying $0 = 0$ and looking at whether a runtime error occurs or not. Both find the reachability of an invalid cell quickly (in less than 1 second). An invalid cell is reached after $\Box$ 0 successfully

□ CHANGER		IDLE	WAIT_REQUEST		WAIT_MONEY_TAKEN
		0	1		2
xPrepare	0	WAIT_REQUEST balance=balance+10000;	/		/

x10KRequest	1	/	balance > 10000
			WAIT_MONEY_TAKEN
			payment=1000;
			balance=balance-payment;
			event(RETURNER, paid);
				else
				IDLE
				payment=0;
				event(RETURNER, paid);
					/

taken	2	/	×		WAIT_REQUEST payment=0;

□ RETURNER		WAIT	RETURN
		0	1
paid	0	RETURN /	×
xReceive	1	/	WAIT event(CHANGER, taken);

Fig. 2. A Simplified money exchange machine (MEM) system.

calls to and returns from □ 0.1, and then calls to □ 0.2, during the execution of which, □ 0.2 gets stuck in a situation where its state is S011 and the variable e2 equals 1. (Such a situation is called the reachability of an invalid cell in Garakabu2's terminology.)

5.2. Example 2

The second example shown in Fig. 2 is an example of a simplified money exchange machine (MEM), which is the motivating example in [KSK+11] and [KKQ+11]. To make the explanation easier to understand, we attach indexes (i.e., give numbers) for both events and states of the EHSTMs in Fig. 2. The two EHSTMs belong to two different tasks and communicate through message passing.

The informal meaning of the exchange machine system is explained as follows. The device CHANGER supplies smaller denominations when an equivalent amount of money in a larger denomination is inserted. The device RETURNER gets notified when small denominations are ready and notifies the CHANGER if the users who made the exchange have taken the small denominations. The MEM system is modelled in a greatly simplified means by abstracting away details irrelevant to the purpose of the demonstration.

We translate the EHSTMs of the second example into CSP# programs (Fig. 3). Note that we use a slightly different translation than the one we presented in the previous section. We insert "atomic {...}" statements around the actions and transitions in cells so that they are executed atomically.

We verify the properties of reachability of invalid cells, STC1, STC2, and DYN defined in [KKQ+11]. Generally, STC1, STC2, and DYN specify certain correlations between the states of the EHSTMs. Specifically, STC1 states that: it is always that the device CHANGER is in state "wait_money_taken" if the device RETURNER is in the state "ret". STC2 has a similar meaning. DYN states that: it is always that if the device CHANGER is in the state "wait_request" and is going to switch to the state "wait_money_taken" in the next step, then the device RETURNER will be in the state "wait" in the next step. These are general LTL properties that can be defined in pseudo CSP# programs as follows.

```
STC1 := [](state_return == ret -> state_changer == wait_money_taken)
STC2 := [](state_changer == wait_request -> state_return == wait)
DYN  := [](((state_changer == wait_request && X state_changer == wait_money_taken)
            -> X state_return == wait)
```

where `[]` denotes the LTL-operator "always" and `X` denotes the LTL-operator "next". Note that `[]` is used for two meanings: to denote the choice operator in the body of program, and to denote the "always" operator in LTL-formula. We set the length of the message queues (namely channel size in the CSP# terminology) as 2.

Both PAT and Garakabu2 report counterexamples for these properties. The time required (in seconds) to find a counterexample is shown in Table 1. Note that we choose the explicit breadth-first search (BFS) algorithm for PAT since the counterexamples found using its depth-first search (DFS) are not the shortest ones and are hard to follow. We use 25 as the bound for Garakabu2's bounded model checking (BMC). PAT and Garakabu2 return the same results (Invalid or runtime error, i.e., counterexamples). However, PAT requires much less time to find counterexamples.

As noted in [KKQ+11], the cause of the unsoundness of the model resides in the action definition of the cell (event=1, state=1) of CHANGER: even if CHANGER does not have enough balance (< 10000), the event `paid` is still sent to RETURNER. The trace of the counterexample for property STC1 is illustrated below to increase understanding. Message passing (push-into/pop-from queues) is omitted from the trace, and we use `C` and `R` to denote EHSTMs CHANGER and RETURNER, respectively, where `X.(event=A, state=B)` denotes the execution of a cell with event index `A` and state index `B` of EHSTM `X`.

```
enum {wait, ret, idle, wait_request, wait_money_taken, paid,
      xReceive, xPrepare, x10kRequest, taken};
channel event_return 2;     channel event_changer 2;
var balance = 0;     var payment = 0;     var state_return = wait;     var state_changer = idle;

Return =  [state_return==wait]event_return?paid -> {state_return=ret} -> Return()
[]
[state_return==wait]event_return?xReceive -> Return()
[]
[state_return==ret]event_return?paid -> assert(false)
[]
[state_return==ret]event_return?xReceive ->
atomic{ event_changer!taken ->
{state_return = wait} -> Skip};
Return();

Changer =  [state_changer==idle]event_changer?xPrepare ->
{balance=balance+10000; state_changer=wait_request} ->
Changer()
[]
[state_changer==idle]event_changer?x10kRequest -> Changer()
[]
[state_changer==idle]event_changer?taken -> Changer()
[]
[state_changer==wait_request]event_changer?xPrepare ->
Changer()
[]
[state_changer==wait_request]event_changer?x10kRequest ->
atomic{
if (balance >= 10000) {
{payment=10000;balance=balance-payment;} ->
event_return!paid ->
{state_changer = wait_money_taken;} -> Skip
}
else {
{payment=0; state_changer=idle} ->
event_return!paid  ->
Skip
}};
Changer()
[]
[state_changer==wait_request]event_changer?taken -> assert(false)
[]
[state_changer==wait_money_taken]event_changer?xPrepare ->
Changer()
[]
[state_changer==wait_money_taken]event_changer?x10kRequest -> Changer()
[]
[state_changer==wait_money_taken]event_changer?taken ->
{payment=0;
state_changer=wait_request} ->
Changer();

Env  = event_return!xReceive -> Env()
       [] event_changer!xPrepare -> Env()
       [] event_changer!x10kRequest -> Env();

system = Env() ||| Return() ||| Changer();
```

Fig. 3. Translated CSP# programs for the second example

Table 1. Verification results of the MEM system

Property	Verdict	PAT	Garakabu2
invalid cell	Invalid	< 1 (runtime error)	384
STC1	Invalid	0.085	26
STC2	Invalid	0.13	47
DYN	Invalid	0.0094	136

Table 2. Verification results of the revised MEM system

Property	Verdict	PAT	Garakabu2
Invalid cell	Valid	0.0094	2716
STC1	Valid	0.098	895
STC2	Valid	0.098	1573
DYN	Valid	0.12	3591

```
C.(event=0, state=0) --> lhs of C.(event=1, state=1) --> R.(event=0, state=0) -->
R.(event=1, state=1) --> C.(event=2, state=2) --> rhs of C.(event=1, state=1) -->
R.(event=0, state=0)
```

Therefore we comment out the message-sending action of this cell and verify the EHSTMs again. Nonetheless we use the explicit DFS algorithm for PAT, and use 25 as the bound for Garakabu2. The results are shown in Table 2.We can see that the performance is greatly improved by using the CSP translation of EHSTMs and PAT. The results of PAT and Garakabu2 are the same (Valid).

5.3. Example 3

In the second example described above, no calling-to and returning-from statements in the EHSTMs exist. In this subsection, we consider an extended money exchange machine (E-MEM) system that contains such statements. The extended example has been used in [KLY+12] and its figure is shown in Fig. 4, in which EHSTMs MainInterface and Exchanger are of one task, and ReturnController and Returner are another task. The hierarchical structure is shown at the bottom-right of the figure. Note that EHSTMs in the E-MEM system use shared variables to communicate.

Following the rules defined in Section 3, the translated CSP# programs of the example are shown in Fig. 5. We verify the properties of reachability of an invalid cell and SSC1, SSC2, FCF1, and FCF2 as in [KLY+12], which can be defined as LTL properties in pseudo CSP# programs as follows, where `<>` denotes LTL-operator "finally". The meaning of the properties should be straightforward as the variable names indicate.

```
SSC1 := [](state_ReturnControl == R_ACTIVE -> state_MainControl == M_WAIT_BILL_TAKEN)
SSC2 := [](state_MainControl == M_ACTIVE -> state_ReturnControl == R_IDEL)
FCF1 := []<>(BillOutputAmount > 0)
FCF2 := []<>(xUserBillTake -> ((BillOutputAmount > 0) -> <>UserBillObtained))
```

The verification results of PAT and Garakabu2 are shown in Table 3, where time is counted in seconds. Note again that we choose to use the explicit breadth-first search (BFS) algorithm for PAT, and we use 40 as the bound for Garakabu2. In addition, among the optimisation made in Garakabu2 for the shared-variable (type) EHSTMs, we choose to use the non-iteration hybrid (i.e., a combination of explicit and BMC)

Table 3. Verification results of the E-MEM system

Property	Verdict	PAT	Garakabu2
Invalid cell	Valid	0.59	1
SSC1	Valid	0.58	1
SSC2	Valid	0.55	1
FCF1	Invalid	0.023	91
FCF2	Valid	0.094	83

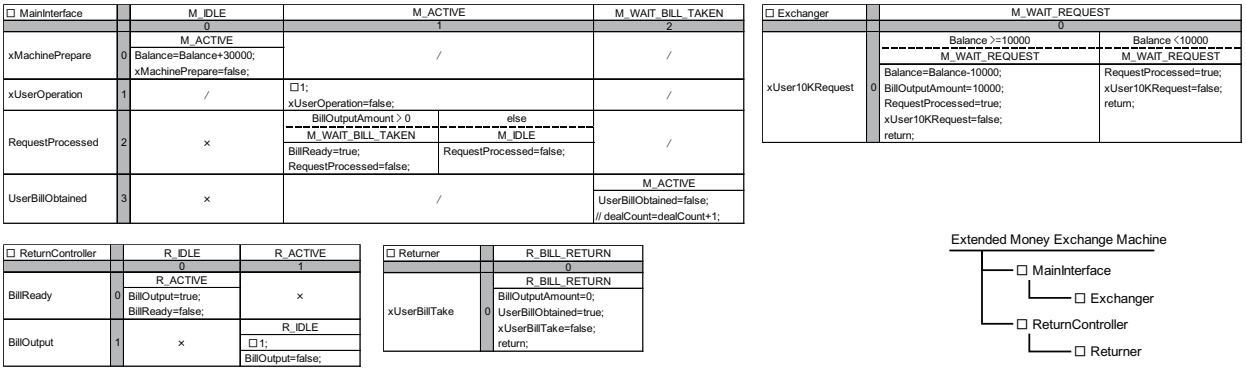

Fig. 4. The extended money exchange machine (E-MEM) system.

algorithm [KLY+12]. PAT and Garakabu2 return the same results. Again, PAT requires less time to check the model.

5.4. Analysis of the Experimental Results

From the experimental results of the three examples, we know that PAT and Garakabu2 get the same verification results, but PAT is generally much faster than Garakabu2, especially for message-passing (type) EHSTMs. A fundamental reason behind this is that PAT basically applies explicit state exploration algorithms and Garakabu2 applies bounded model checking (BMC) [BCCZ99] with satisfiability modulo theory (SMT) solving [BSST]. BMC was originally proposed for coping with the state explosion problem [CGP99] faced by explicit model checking, rather than for improving checking speed; therefore, our comparison of the speed may be unfair. However, one of the purposes of such a comparison is to suggest that using PAT, through our translation described in the paper, might be a better choice for end users who may not care about the model checking algorithms used. Note that Garakabu2 is extremely slow for checking message-passing EHSTMs because SMT solvers (such as the CVC3 solver [BT07] used in Garakabu2) are generally not quite efficient for solving arrays that are used in Garakabu2 for representing message queues.

On the other hand, however, we would also like to point out that Garakabu2 is more effective than our method (through PAT) for checking systems with infinite state space. Let us revisit the E-MEM system in Example 3. If the commented statement is made available in the cell (event=3, state=2) of the EHSTM MainInterface and we try to check the same properties again with PAT, we would find that the state explosion problem occurs. The reason for this is simply that the statement keeps increasing the value of the variable `dealCount` to infinite value and PAT does not handle this situation, unless a specific range of the variable is set. To summarise, PAT is designed to check systems with finite state space, while Garakabu2 can directly read and check systems with infinite state space although only a bounded part of the state space is checked.

The next question, then, is about the scalability of the method described in the paper. Although there must be room for optimising our translation, we believe that the scalability largely depends on that of PAT. PAT implements a set of optimisation techniques, e.g., partial order reduction, process counter abstraction, etc. As mentioned in [LSD11], PAT3 has been used to model and verify a variety of systems, ranging from recently proposed distributed algorithms, security protocols to real-world systems like the pacemaker system; in addition, their experiment results show that PAT3 is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers like SPIN [Hol97, Hol08] in many cases.

6. Concluding Remarks and Future Work

In this paper, we gave a formal semantics to EHSTMs. EHSTMs are widely used in industry as a design method; therefore, their formal semantics is important. To obtain formal semantics, we gave a translation of EHSTMs to CSP. Since CSP has operational, denotational, and algebraic semantics, through translation we gave EHSTMs these semantics. We introduced the function of EHSTMs and their translation step by

```
enum {M_IDLE, M_ACTIVE, M_WAIT_BILL_TAKEN, R_IDLE, R_ACTIVE, R_BILL_RETURN, M_WAIT_REQUEST};
var xMachinePrepare = false;          var Balance = 0;          var xUserOperation = false;
... ... // some variable declarations are omitted.
var state_Returner = R_BILL_RETURN;   var state_Exchanger = M_WAIT_REQUEST;

MainController =
  [xMachinePrepare && state_MainController == M_IDLE]
     {Balance = Balance + 30000; xMachinePrepare = false; state_MainController = M_ACTIVE}
          -> MainController()
  []
  [xMachinePrepare && state_MainController == M_ACTIVE]tau -> MainController()
  []
  [xMachinePrepare && state_MainController == M_WAIT_BILL_TAKEN]tau -> MainController()
  []
  [xUserOperation && state_MainController == M_IDLE]tau -> MainController()
  []
  [xUserOperation && state_MainController == M_ACTIVE]tau ->
     Exchanger_start -> Exchanger_return -> tau{xUserOperation = false} -> MainController()
  []
  [xUserOperation && state_MainController == M_WAIT_BILL_TAKEN]tau -> MainController()
  []
  [RequestProcessed && state_MainController == M_IDLE]tau -> assert(false)
  []
  [RequestProcessed && state_MainController == M_ACTIVE]
     atomic{if (BillOutputAmount>0) {tau{BillReady = true; RequestProcessed = false;
                                    state_MainController = M_WAIT_BILL_TAKEN} -> Skip}
            else {tau{// BillReady=true;
                      RequestProcessed=false; state_MainController=M_IDLE} -> Skip}
           }; MainController()
  []
  [RequestProcessed && state_MainController == M_IDLE]tau -> MainController()
  []
  [UserBillObtained && state_MainController == M_IDLE]tau -> assert(false)
  []
  [UserBillObtained && state_MainController == M_ACTIVE]tau -> MainController()
  []
  [UserBillObtained && state_MainController == M_WAIT_BILL_TAKEN]
    {UserBillObtained = false; state_MainController = M_ACTIVE} -> MainController();

ReturnControl =
  [BillReady && state_ReturnControl == R_IDLE]
    {BillOutput = true; BillReady = false; state_ReturnControl = R_ACTIVE} -> ReturnControl()
  []
  [BillReady && state_ReturnControl == R_ACTIVE] assert(false)
  [] [BillOutput && state_ReturnControl == R_IDLE] assert(false)
  []
  [BillOutput && state_ReturnControl == R_ACTIVE]
     Returner_start -> Returner_return -> {BillOutput = false; state_ReturnControl = R_IDLE}
            -> ReturnControl();

Returner =
  [xUserBillTake && state_Returner == R_BILL_RETURN]
     {BillOutputAmount = 0; UserBillObtained = true; xUserBillTake = false} ->
     Returner_return -> Returner_start -> {state_Returner = R_BILL_RETURN} -> Returner();

Exchanger =
  [xUser10KRequest && state_Exchanger == M_WAIT_REQUEST]
     if (Balance >= 10000)
      {tau{Balance = Balance-10000; BillOutputAmount = 10000;
           RequestProcessed = true; xUser10KRequest = false} -> Exchanger_return
       -> Exchanger_start -> tau{state_Exchanger = M_WAIT_REQUEST} -> Skip}
     else {tau{RequestProcessed = true; xUser10KRequest = false} ->
           Exchanger_return -> Exchanger_start
           -> tau{state_Exchanger = M_WAIT_REQUEST} -> Skip}; Exchanger();

Env =
  {xMachinePrepare = true} -> Env() [] {xUserOperation = true} -> Env() []
  {xUserBillTake = true} -> Env() [] {xUser10KRequest = true} -> Env();

system = Env() ||| (MainController() || Exchanger_start -> Exchanger())
         ||| (ReturnControl() || Returner_start -> Returner());
```

Fig. 5. Translated CSP# programs for the third example

step. The translation covers major functionalities supported by Garakabu2 model checkers, that is, event virtual frames, state virtual frame, parallel state with/without synchronisation, hierarchical EHSTMs, global transitions, composite actions, and multiple tasks.

As to our knowledge to date, no formal semantics for EHSTMs has been established. Therefore, it is impossible to prove the soundness of our translation. Still, once we clearly understand the informal semantics of EHSTMs, we can convince ourselves the correctness of translation relatively easily since the translation is clean and easy to understand.

We also compared the results and time required for verification of EHSTMs with Garakabu2, as well as our translation of these EHSTMs with PAT. We found that the results of Garakabu2 and our translation coincide, and furthermore, PAT verifies our translation of EHSTMs much faster than Garakabu2. We note that the number of states required to verify (in PAT) is about < 10000. Therefore, PAT finishes the verification very quickly. It would be interesting to compare PAT and Garakabu2 for the case where more states or/and complex arithmetic is required for verification.

Ideally it would be better to prove the correctness of Garakabu2 by our translation, but it is very difficult. As stated before, Garakabu2 translates EHSTMs to SMT formulae for model checking. However, the translation rules are buried in C++ code, and have never been fully formalised.

For future work, we plan to extend the translation to more functions of EHSTMs such as hierarchical states and interrupts, because these functions are practically important. These features are left out since Garakabu2 does not support them, and their semantics is still unclear for us.

Furthermore, we would like to implement this translation as a PAT plug-in, and compare its performance to Garakabu2 on a more realistic example.

Acknowledgements We are grateful to Yoshinao Isobe and Liu Yang for their comments and suggestions. Song Songzheng and Zhang Shaojie helped debug the CSP# code. Cyrille Artho carefully checked the manuscript and proposed numerous improvements. This research was partially supported by a program for the "Regional Innovation Cluster (Global Type, the 2nd Stage)" by The Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan. CATS CO.,LTD. provided us with a copy of ZIPC for the preparation of this paper. We would also like to thank all relevant organisations for their support.

References

[Ass12] Japan Embedded System Technology Association. A tentative report on questionnaires of spread of design methods 2011 (japanese). `et2011_questionnaire.pdf` file on JASA web site, 2012.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th TACAS*, pages 193–207. Springer, 1999.

[BSST] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Handbook of Satisfiability*, chapter 26, pages 825–885.

[BT07] C. Barrett and C. Tinelli. CVC3. In *19th CAV*, pages 298–302. Springer, 2007.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes. By C.A.R. Hoare. Prentice-Hall International, London, 1985, viii+256 pages.*, volume 9. August 2004.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hol08] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2008.

[KKQ+11] W. Kong, N. Katahira, W. Qian, M. Watanabe, T. Katayama, and A. Fukuda. An SMT-based approach to bounded model checking of designs in communicating state transition matrix. In *the 11th International Conference on Computational Science and Its Application (ICCSA 2011)*, pages 159–167. IEEE CS, 2011.

[KLY+12] W. Kong, L. Liu, Y. Yamagata, K. Taguchi, H. Ohsaki, and A. Fukuda. On accelerating smt-based bounded model checking of hstm designs. In *APSEC (accepted)*, 2012.

[KSK+11] W. Kong, T. Shiraishi, N. Katahira, M. Watanabe, T. Katayama, and A. Fukuda. An SMT-based approach to bounded model checking of design in state transition matrix. *IEICE Transactions on Information and Systems*, E94-D(5):946–957, 2011.

[LSD11] Y. Liu, J. Sun, and J. Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199. IEEE, 2011.

[RHB98] A.W. Roscoe, C.A.R. Hoare, and R. Bird. *The theory and practice of concurrency*, volume 216. Prentice Hall, 1998.

[SLD09] J. Sun, Y. Liu, and J.S. Dong. Model checking CSP revisited: Introducing a process analysis toolkit. *Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322, 2009.

[US94] A. Uselton and Scott A. Smolka. A compositional semantics for Statecharts using labeled transition systems. *CONCUR'94: Concurrency Theory*, 1994.

[Wat98] M. Watanabe. Extended Hierarchy State Transition Matrix Design Method-Version 2.0. Technical report, CATS Technical Report, 1998.

[ZL10] S. J. Zhang and Y. Liu. An Automatic Approach to Model Checking UML State Machines. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 1–6. IEEE, 2010.