Zhou Y, Baresi L, Rossi M. Towards a formal semantics for UML/MARTE state machines based on hierarchical timed automata. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(1): 188–202 Jan. 2013. DOI 10.1007/s11390-013-1322-8

Towards a Formal Semantics for UML/MARTE State Machines Based on Hierarchical Timed Automata

Yu Zhou[1,2] (周　宇), Luciano Baresi[3], and Matteo Rossi[3]

[1] *College of Computer Science, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China*

[2] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China*

[3] *Department of Electronics and Information, Politecnico di Milano, Milan 20133, Italy*

E-mail: zhouyu@nuaa.edu.cn; {baresi, rossi}@elet.polimi.it

Received November 17, 2011; revised September 24, 2012.

Abstract UML is a widely-used, general purpose modeling language. But its lack of a rigorous semantics forbids the thorough analysis of designed solution, and thus precludes the discovery of significant problems at design time. To bridge the gap, the paper investigates the underlying semantics of UML state machine diagrams, along with the time-related modeling elements of MARTE, the profile for modeling and analysis of real-time embedded systems, and proposes a formal operational semantics based on extended hierarchical timed automata. The approach is exemplified on a simple example taken from the automotive domain. Verification is accomplished by translating designed models into the input language of the UPPAAL model checker.

Keywords timed automata, state machine diagram, formal semantics

1 Introduction

UML state machine diagrams have been widely used to model the behavior of real-time reactive systems. Besides the "standard" notation, the OMG (Object Management Group) also proposes MARTE, the profile for modeling and analysis of real-time embedded systems[1], but all these notations do not go beyond a standard syntax for rendering concepts. This is not enough when we think of time-critical systems that demand for rigorous analysis and verification even on early-stage models. The underlying semantics of these notations must be defined formally to both avoid ambiguous interpretations and pave the ground to the aforementioned rigorous verification.

Motivated by this common goal, much previous work has attempted to augment UML, or rather subsets of UML (e.g., behavioral diagrams), with different formal semantics[2-5]. Among these approaches, quite a few addressed the time dimension (e.g., [4]), but they tended to treat time aspects in a simplistic way. Time can exist in multiple representations and clocks are not necessarily only chronometric ones[6]. For example, Fig.1 illustrates the state machine diagram of an engine[7]. The occurrence of events depends on the rotation angle of the engine's camshaft. In this case, seconds (or milliseconds) are not the time unit anymore, and the time flow is measured in degrees of angles. Moreover, at design level, clocks are often logical ones[6] bound to the occurrence of specific events (e.g., the execution cycles of a processor) instead of being related to the physical flow of time.

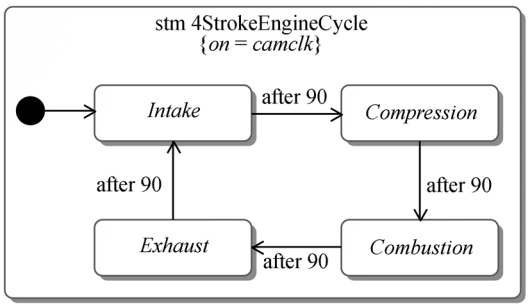

Fig.1. Example state machine with multiform time[7].

To support the aforementioned concept of time in a broader sense, MARTE introduces a general framework for representing time and time-related concepts. These concepts are not bound to physical time anymore: the

Regular Paper

This work was supported by the European Community 7th Framework Program (FP7/2007-2013) under Grant agreement No. 248864 (MADES) and the National Natural Science Foundation of China under Grant No. 61202002.

containers of instants are called time bases, and a system can contain multiple time bases, even if they are not fully independent. The partial ordering of instants characterizes the time structure of the application[6,8]. MARTE accommodates both a simple form of time structured as a totally ordered set of instants, which share the same time base, and multiple time base models.

Given the insufficient support offered by current approaches, this paper exploits hierarchical timed automata[9] to propose a formal operational semantics for UML state machine diagrams along with MARTE time-related aspects. The main reason why we use hierarchical timed automata is that they can faithfully render the structure and temporal behavior of complex real-time systems. Moreover, their well-defined, rigorous semantics paves the ground to further analysis and verification. Since reachability is decidable for timed automata, we prefer them over other tools, such as Time Petri Nets[10].

As for state machines, we select a subset of the notation guided by the relative importance and inherent ambiguity of elements[11]. We only consider basic elements, do not take into account choice, junction, exit, and terminate pseudo-states, do not model sub-machines, since they are semantically equivalent to composite states[12], assume that transitions take no time, and omit deferred events, state/transition redefinitions, and entry/do/exit clauses. Moreover, proposed notation also supports both inter-level transitions and history states (shallow and deep history). As for time, differently from previous work, the approach supports both logical clocks and multiform representations of time. The resulting modeling notation, along with its associated formal semantics, is then used for verification. To this end, the paper describes how to transform designed models into the input language of the UPPAAL model checker and provides some results on the verification of the example system.

All these elements help us highlight the novel aspects of this paper, with respect to the many attempts to augment UML with formal semantics:

- To the best of our knowledge, this proposal is the first to formalize the concept of multiform time with hierarchical timed automata in the context of UML/MARTE profile. The resulting model can facilitate automated analysis and validation.
- The proposed approach supports modelling both history pseudo-states and inter-level transitions.
- A translation algorithm is also included to complement our presentation on verification based on hierarchical timed automata.

The paper is organized as follows. The basic notions of clocks and timed automata are given in Section 2. The proposed extended hierarchical timed model with its operational semantics is presented in Section 3. Section 4 describes the process of translating designed models into the input language of the UPPAAL model checker. Section 5 re-examines our proposal, discussing its advantages as well as limitations. Section 6 presents some related approaches and Section 7 concludes the paper.

2 Clocks and Timed Automata

Timed automata are an extension of finite state automata and were first introduced by Alur and Dill to model real-time systems[9]. Given the introduction of *clock*s, a run of the automaton along a sequence of consecutive transitions is of the form: $(l_0, \nu_0) \rightarrow (l_1, \nu_1) \rightarrow \cdots \rightarrow (l_p, \nu_p)$, $(l_i)_{0 \leqslant i \leqslant p}$ denotes the state (or location) and $(\nu_i)_{0 \leqslant i \leqslant p}$ denotes the clock value. Specifically, l_i is associated with some combinations of clock-related Boolean expressions to denote time invariants, denoted as $Inv_{l_i}(\nu)$.

The general notion of a clock "CLK" is a pair (x, ν), where x is a clock variable and ν is the corresponding value. Usually ν ranges over real numbers to address continuous time and it can only be reset by users. A comparison with a given real value c results in an atomic clock constraint $g := x \sim c$, and $\sim = \{<, \leqslant, =, \geqslant, >\}$; the actual constraint is defined by combining some Boolean expressions $g := g | g \wedge g | \neg g$. If we consider the domain of real numbers to model physical dense time, the number of configurations would be infinite, but several abstractions — e.g., *region* and *zone* — have been proposed to keep the number finite and make analysis feasible.

Formally, a timed automaton $\mathcal{A}$ is a tuple $\langle \Sigma, S, S_0, C, T \rangle$, where Σ is a finite alphabet of actions, S is a finite set of states, S_0 is the finite set of initial states, C is a finite set of clocks, and $T \subseteq S \times (\Sigma \times \mathcal{CC} \times 2^C) \times S$ is a finite set of transition steps in which $\mathcal{CC}$ is a finite set of clock constraints. For example, if an action a happens and the valuation of clocks satisfies the constraint g, it can cause a state s at time t to change to s'. This transition *trans* can be formalized as $\langle s, a, g, \kappa, s' \rangle$, where the clock $c \in \kappa$ is reset and restarts from 0 as soon as the transition happens. For clarity, sometimes the transition step can also be written as $s \xrightarrow{a,g,\kappa} s'$. The source and target states can be described as $SRC(trans) = s$ and $TGT(trans) = s'$, respectively. The sequence of *trans*'s describes the trace of state transitions from the initial state as time advances. An interesting property of timed automata is that reachability is decidable.

In many cases, some state-based notations allow states to be organized hierarchically (e.g., UML). This means that if a contained state s_1 is active, there must be a containing state s_0 that is active as well. Hierarchical automata[5,13], along with their operational semantics, given by means of Kripke structures, reflect this concept. More precisely, hierarchical timed automata (HTA) are a timed extension of hierarchical automata that is defined as a tuple[4]: $\langle S, S_0, \delta, \sigma, V, C, Inv, Ch, T\rangle$, where S is a finite set of states, S_0 is the set of initial states. $\delta : S \mapsto 2^S$ is a function, mapping an element in S onto its sub-states. $\delta^*(s)$ is the set of all nested states of a super-state s. $\sigma : S \mapsto \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ associates a type to a state. V is a set of variables, C of clocks, and Ch of channels, which is crucial for parallel composition. Inv relates a state to its invariant. $T \subseteq S \times (Ch \cup \varnothing) \times \mathcal{CC} \times 2^C \times \{\text{true}, \text{false}\}) \times S$ is the set of transitions. $\{\text{true}, \text{false}\}$ defines whether a transition is urgent or not.

The operational semantics associated with HTA is defined over the transitions between configurations. A configuration represents an abstraction or a snapshot of system settings. In [4], it is of the form: (ρ, μ, ν, θ). $\rho : S \mapsto 2^S$ maps a super-state onto its active sub-states (self-included), $\mu : V \mapsto (\mathbb{Z})^*$ gives the values of the local integer variables, $\nu : C \mapsto (\mathbb{R}^+)^*$ gives the values of local clocks, and θ reflects history information. Since history information is related to both states and variables, θ consists of two functions: θ_{state} and θ_{var}, which denote the history information of state and variable respectively.

These automata allow for three types of transitions $t := (l, v) \rightarrow (l', v')$. We can have delays, that is the time passes without changing states, synchronized transitions, labeled with the pair of actions *send* and *receive*, and non-synchronized transitions, labeled with silent or event triggers. A predicate function *TransitionEnabled* says whether t is enabled, while *UrgentEnabled* whether it is urgent. For space limitations, details are not included here, interested readers can refer to [4] for a complete presentation.

3 Proposed HTA Model

The HTA presented in the previous section can give a concise and formal representation of UML state diagrams[4]. However, the implicit binding of clocks to physical ones and the lack of support for inter-level transitions limit its application in settings with richer clocks and complex transitions, and motivate the introduction of our extended HTA model.

Instead of defining HTA through a single tuple, we use a compositional notation, along with a refinement function to associate a composite state with a set of automata. We also render *Clock* through a more complex structure rather than a simple real-time value, and use a separate history indicator function to elaborate on the different history types since *shallow* and *deep* history need different semantic interpretations.

All these concepts are illustrated through the example of Fig.2, which describes the simplified behavior of a vehicle. A car can move between states *Stop* and *Running*. *Running* is further decomposed into three orthogonal regions. The first region describes the state transitions of the engine. The second region contains the state transitions of the spark plug. The last region contains another composite state, called *Elec_device*, which contains two further regions: one for the state transitions of a light and one for the air conditioner. *Elec_device* also contains a shallow history pseudo-state as the target of transition *turn_on*. This means that the first time one enters *Elec_device*, the target of the transition returns the default entry sub-states; otherwise it always returns the sub-states that were active when the state was left. The behavior of *Running* is parametric with respect to two clocks: *camclk* and *chronclk*. The first clock measures the engine's cycle by means of the angles of camshaft, while the other one is the physical clock for the spark plug. In this diagram, edges are annotated with some guards (after), synchronization (ignite event of spark plug and engine cycle) and resets (angle:=0 for the engine, t:=0 for the spark plug).

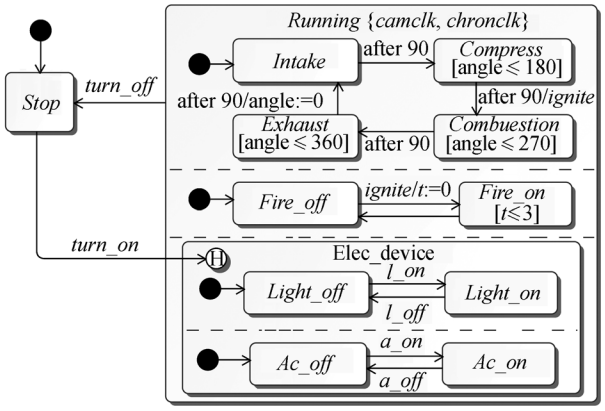

Fig.2. Example state machine.

3.1 Extended Clocks

A clock $\mathcal{C}$ can be represented as a tuple $\langle \mathcal{V}, \nu_0, \prec, \mathcal{D}, \lambda, u\rangle$, where $\mathcal{V}$ represents the set of possible clock values, $\nu_0 \in \mathcal{V}$ is the lower bound and also the default reset value of the clock, $\prec$ is the pre-order relation over the set of values, $(\mathcal{V}, \prec)$ forms a total order relation. This total ordered set of instants can model the simple

form of time owned by a time base (corresponding to the first type of concepts covered by MARTE). $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{V} \mapsto \mathcal{D}$ is a partial labeling function that associates a meaningful name to a time instant. These labels can model the entities bound to time (corresponding to the fourth type of concepts covered by MARTE). u is a time unit. $\mathcal{V}$ can range over both continuous and discrete domains, which can be used to model physical and logical clocks. In physical clocks, u can be second, millisecond and alike; while in logical clocks, u can be any tick. Basic operations over $\nu \in \mathcal{V}$ also include advancement $+$. If we do not use labels, a clock C can be written as $\langle \mathcal{V}, \prec, u \rangle$. This extended notion of clock can model the access to the time (corresponding to the third type of concepts covered by MARTE.)

In presence of multiform time, different kinds of clocks co-exist in one system. Our proposal borrows from the MARTE notion of *time structure*, and we define a time structure $W = \langle CS, \preccurlyeq, \zeta \rangle$, where CS denotes a set of clocks, $\preccurlyeq$ establishes a partial order relation between two clocks, and $\zeta_{u_2}^{u_1} : \nu_1 \mapsto \nu_2$ defines a value mapping between two different time units and returns undefined $\perp$ if the two units are not comparable. Therefore, with time structure, the multiple time bases can be modeled (corresponding to the second type of concepts covered by MARTE).

Usually, during transitions, clock values need to be reset or updated. An evaluation function for clocks can be defined as $v : \mathcal{C} \mapsto \mathcal{V}$, in which $\mathcal{V}$ is the time domain. For $\forall x$ of type $\mathcal{C}$, clock advances and resets can be defined: $(v+t)(x) = v(x)+t$ and $v(x) = \nu_0$, respectively. Specifically, to update a subset of Y of type $\mathcal{C}$ with value t, we can write $([Y \leftarrow t]v(x) = t)$; the clock values that are not members of Y remain unchanged. If $t = \nu_0$, the clocks in the set Y get reset.

Clock constraints are mainly used for time related invariants in states and guards for transitions. Normally, a clock constraint g over variables x of $\mathcal{C}$ is defined by $g := x \preccurlyeq c \mid x \succcurlyeq c \mid x = c \mid \neg g \mid g \wedge g \mid \text{true}$, where c is a valid time value constant. g is a function from Boolean expressions to Boolean values. $\mathcal{CC}$ is a finite set of g. As a concrete example, Fig.2 shows two kinds of clocks: *camclk*, and *chronclk*, measuring *camshaft* angles and physical time, separately. $camclk = \langle \mathbb{R}^+, 0, <, \mathcal{D}_1, \lambda_1, \text{degree} \rangle$, and $chronclk = \langle \mathbb{R}^+, 0, <, \mathcal{D}_2, \lambda_2, \text{ms} \rangle$. We can select some representative angles for *camclk* and associate them with meaningful labels. For example, the labels *intake_bottom*, *compress_top*, *combustion_bottom*, *exhaust_top* can denote $90, 180, 270, 360$ degrees respectively. Similarly, for *chronclk*, labels *ignite_start*, *ignite_stop* can be defined and associated with particular time instants to denote the start and stop time for the spark plug. If we assume that the revolution per minute (RPM) of the engine is n, since one revolution of the camshaft implies two revolutions of the crankshaft mechanically, then by physical laws, we can get the function $\zeta_{\text{ms}}^{\text{degree}}(v) = (2 \times 60 \times 1000 \times v)/(n \times 360)$ transforming the rotation of v degrees into the physical time duration. We can define a $\preccurlyeq$ relation between two clocks, e.g., the instances of the following labels, $(intake\_bottom_{camclk},\ ignite\_start_{chronclk})_{\preccurlyeq}$, specifying that the clock instance of *ignite_start* should be no earlier than that of the *intake_bottom*.

Advancement Function for Multiform Clocks. When multiform clocks co-exist in a system, that is, $|W.CS| > 1$, the synchronization among them becomes crucial. Generally, these clocks can advance at their own paces measured by separate clocks. However, to ease the process of analysis, normalization is necessary, i.e., to establish some relationship among these clocks. The clock set in the time structure W is partitioned into several blocks based on clocks' property. We can have both continuous and discrete clocks. For continuous ones, a specific clock unit can be chosen as the base unit, denoted by u_{base}. For logical clocks, the advancement is represented by the occurrence of the events in the sequence ordered by ticks. The precedence relation $R_{\preccurlyeq}$ establishes the connection between them. The advancement functions are given in the following (1). d is the advancement value measured in the base clock, and $adv_T(\boldsymbol{\nu} + d)$ can be adv_{T_c} or adv_{T_l} depending on the properties of clocks in context. adv_{T_c} is the advancement function for continuous clock, and adv_{T_l} for logic clock. ν_c is the clock value. Notice that in the case of clocks incomparable with the base clock, ν_c is unchanged, as d is the value measured in the base clock, which is somehow orthogonal with ν_c. However, this does not mean that ν_c cannot advance. In this case, ν_c would advance independently of the base clock.

In our example, we can use physical time as the base time and $\zeta_{\text{ms}}^{\text{degree}}$ to establish the synchronization between these two different kinds of clocks. The advancement d degrees of *camshaft* angles indicates the $d \times \zeta_{\text{ms}}^{\text{degree}}$ pass of physical time.

$$adv_{T_c}(\boldsymbol{\nu}, d) = \begin{cases} \nu_c + d, & \text{if } u_c = \text{base}, \\ \nu_c + d \times \zeta_{u_{\text{base}}}^{u_c}, & \text{else if } \zeta_{u_{\text{base}}}^{u_c} \neq \perp, \\ \nu_c, & \text{otherwise.} \end{cases} \tag{1}$$

Logical clock advancement is measured on the occurrences of events. If these occurrences refer to — explicitly or implicitly — some specific values of the base clock, which can be specified by the pre-order relation in the time structure, then we can derive the tick

advancement function during d time measured in the base clock unit and the result is the latest event occurrence. next(ν) denotes the event exactly happens after ν.

$$adv_{T_l}(\boldsymbol{\nu}, d) = \begin{cases} \nu'_c, & \text{if } \exists \nu'_c, \nu'_c \preccurlyeq (\nu_{\text{base}} + d) \wedge \\ & (\nu_{\text{base}} + d) \prec \text{next}(\nu'_c), \\ \nu_c \text{ (unchanged)}, & \text{otherwise.} \end{cases} \quad (2)$$

In our example, if we abstract away the actual degrees of *camshaft*, we can select the four critical angles as described by the previously-introduced labels: *intake_bottom*, *compress_top*, *combustion_bottom*, *exhaust_top*. Since these labels correspond to four timed events in a cycle, and they display an exact pre-order relation, they constitute a logical clock. This clock is denoted as *clk*1 in Fig.3. In mechanical engineering, to keep the engine work, there are additional components to participate, for example, spark plug, inlet (intake) valves, outlet (exhaust) valves. Generally, in one working cycle of the engine, the inlet valves are opened (*inlet_open*) at the beginning of the intake stroke and closed (*inlet_close*) when the piston travels to the bottom of the cylinder. The spark plug would ignite (*ignite_start*) when the piston reaches the top of the cylinder and the spark exists shortly (*ignite_stop*) within the combustion stroke. The outlet(exhaust) valve is opened (*outlet_open*) and the piston travels back up expelling the exhaust gases through. When the piston reaches the top of the cylinder, the outlet valve is closed (*outlet_close*). Fig.3 describes the simultaneity of these events by dashed lines, for example, *compress_top* and *ignite_start* (i.e., *compress_top* $\preccurlyeq$ *ignite_start* $\wedge$ *ignite_start* $\preccurlyeq$ *compress_top*).

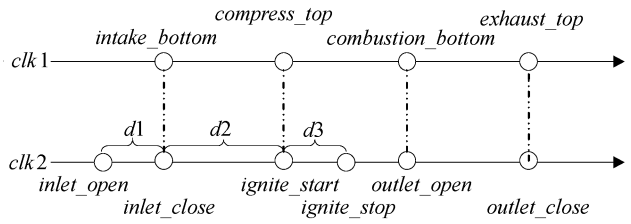

Fig.3. Example clock synchronization.

In Fig.3, $d1$ is the period between labels *inlet_open* and *inlet_close*, $d2$ is that between *inlet_close* and *ignite_start*, and $d3$ between *ignite_start* and *ignite_stop*. The measurement is bounded to physical time, and therefore, *clk2* denotes chronometric clock. In the example, we use it as the base clock. Assume currently value for *clk*1 is *intake_bottom* and *clk*2 is *inlet_close*, after $d2$ time, the derived value for *clk*1 is *compress_top* according to (2). If all the clocks in the time structure are chronometric ones, this reduces to the case of multiple clocks in ordinary timed automata.

3.2 Extended HTA

Since the hierarchy is mainly introduced by the containment relation inside composite states, we use the notion of sequential timed automata to accommodate states in the same level and the sub-states of a composite state are delegated to another sequential timed automaton along the hierarchy. A refinement function is employed to establish the hierarchical connection. A sequential timed automaton is a tuple $\langle S, s_0, \sigma, C, Inv, \mu, \Sigma, T \rangle$. S is a set of states, s_0 is the initial state, σ is a typing function, mapping a state $s \in S$ to a specific type and $\sigma(s) \in \{BASIC, COMPOSITE, HISTORY, ENTRY\}$. C, Inv denote sets of clocks and invariants respectively. μ is a history indicator function, and $\mu : S \mapsto \{N, DEEP, SHALLOW\}$. It indicates whether a given state contains history pseudo-state (i.e., *DEEP*, *SHALLOW*) or not (N). Σ is the action set and contains three kinds of actions, i.e., synchronized, non-synchronized and internal actions, represented by *Ch*, $\{action\}$, and $\{\tau\}$ respectively. Similar to [4], *Ch* consists of channel action elements (synchronization), and has two types, send and receive, represented by ! and ? respectively. $\mathcal{CC}$ is the set of clock constraints. $T \subseteq S \times \Sigma \times \mathcal{CC} \times 2^C \times S$, is the set of transitions, and can be written as $t = s \xrightarrow{a,g,r} s'$.

We define an extended hierarchical timed automaton as a tuple: $\langle F, E, W, \rho \rangle$ where:

• F is a finite set of sequential timed automata with mutually disjoint sets of states, i.e., $\forall A_i, A_j \in F, i \neq j, S_{A_i} \cap S_{A_j} = \varnothing$.

• E is a finite set of triggering events associated with related guards and reset clock sets, $E \subseteq (\bigcup_{A \in F} S_A \times \bigcup_{A \in F} \mathcal{CC}_A \times \bigcup_{A \in F} 2^{C_A} \times \bigcup_{A \in F} \Sigma_A \times \bigcup_{A \in F} S_A)$, whose elements denote triggers satisfying the clock constraints. E consists of local transitions as well as inter-automata transitions.

• W is the time structure which is composed of the underlying clocks and their relationship as described previously.

• ρ is a refinement function, mapping a state to a set of automata, i.e., $\rho : \bigcup_{A \in F} S_A \mapsto 2^F$. ρ constructs a tree (hierarchical) structure to the related automata. There is a unique root automaton which cannot be derived by ρ, i.e., $\exists_1 A \in F$ and $A \notin \bigcup_{s \in S_A} \rho(s)$, and this root automaton is denoted by A_{root}; for simple states, the refinement function returns an empty set, i.e., $\forall s, \sigma(s) = BASIC \Rightarrow \rho(s) = \varnothing$. The hierarchy should not introduce loops, i.e., $\forall S \subseteq \bigcup_{A \in F} S_A, \exists s \in S$ and $S \cap \bigcup_{A \in \rho(s)} S_A = \varnothing$. Besides, to be well-formed, for every non-root automaton, there exists a unique ances-

tor state, i.e., $\forall A_i \in F \setminus \{A_{\text{root}}\}, \exists_1 s \in \bigcup_{A \in F \setminus \{A_i\}} S_A$. Based on the definition of ρ, we recursively define a function $\rho^*(s) = \rho(s) \cup (\bigcup_{s_i \in S_{\rho^*}(s)} \rho(s_i))$ which returns all the descendant automata generated by s. Among these descendant automata, the leaf automata sub-set is of particular interest and is defined specifically as $\rho_{\text{leaf}}(s) = \{A' | A' \in \rho^*(s) \wedge (\forall s' \in A', \rho(s') = \varnothing)\}$.

Fig.4 illustrates the corresponding hierarchical structure of Fig.2. It has five sub-automata, $\{A_{\text{root}}, A_1, A_2, A_3, A_4, A_5\}$. $E = \{$*turn_off*, *turn_on*, *l_on*, *l_off*, *ignite*, *a_on*, *a_off*$\} \cup \{\tau\}$①, $\rho($*Running*$) = \{A_1, A_2, A_3\}$, and $\rho($*Elec_device*$) = \{A_4, A_5\}$. $\rho^*($*Running*$) = \{A_1, A_2, A_3, A_4, A_5\}$, and $\rho_{\text{leaf}}($*Running*$) = \{A_1, A_3, A_4, A_5\}$. For all other states s, $\rho(s) = \varnothing$. $\sigma($*Running*$) = \sigma($*Elec_device*$) =$ *COMPOSITE*, and $\mu($*Elec_device*$) =$ *SHALLOW*. Fig.4 also illustrates the invariants associated with the states, *Intake, Compress, Combustion, Exhaust, Fire_on* and some transition guards. Dashed arrows represent the refinement relation.

State Precedence. It is a pre-order relation describing the presence of one state in an automaton refined by another state, i.e., for $s_1, s_2 \in \bigcup_{A \in F} S_A$, $s_1 \prec^s s_2 \iff s_2 \in S_{\rho(s_1)}$. Automaton precedence can be derived similarly, i.e., $A_1 \prec^A A_2 \iff \exists s_1, s_2 . s_1 \in A_1 \wedge s_2 \in A_2 \wedge s_1 \prec^s s_2$. The reflexive closure of $\prec^s$ is denoted by $\preccurlyeq^s$ (and $\prec^A$ by $\preccurlyeq^A$). In Fig.4, *Running* $\preccurlyeq^s$ *Intake*, and *A_root* $\preccurlyeq^A A_1$.

State Descendants. First we define the transitive closure of state precedence recursively. $s \prec^{s*} s' = s \prec^s s' \vee (\exists s'' . s \prec^{s*} s'' \wedge s'' \prec^{s*} s')$. $\preccurlyeq^{s*}$ is the reflexive closure of $\prec^{s*}$. The set of state descendants of s is defined as $S_{\preccurlyeq^{s*}}(s) = \{s' | s \preccurlyeq^{s*} s'\}$. Similarly, operator $\preccurlyeq^{A*}$ (and $\prec^{A*}$) can be applied to automata. Given s, s', and $s \in A, s' \in A'$, if $s \preccurlyeq^{s*} s'$, then $A \preccurlyeq^{A*} A'$, and the set of state descendants of automaton A is $S_{\preccurlyeq^{A*}}(A) = \{s' | A \preccurlyeq^{A*} A' \wedge s' \in A'\}$. In our example, $S_{\preccurlyeq^{s*}}($*Running*$)$ consists all the states except *stop*.

State Closure. Based on the notions of state descendants, we define state closure function. The function yields a closure set of a state s in the designated set S. It represents all states in the hierarchy between s and a state of S. $closure_S(s) = \{s' | \exists s'' \in S. s \preccurlyeq^{s*} s' \preccurlyeq^{s*} s''\}$. We regulate that if S is an empty set, $closure_S(s) = \{s\}$. In Fig.4, $closure_{S_{A_4}}($*Running*$) =$ {*Running*, *Elec_device*, *Light_off*, *Light_on*}.

State Ancestors. We define the set of state ancestors of a given state s' in an HTA as $S_{\prec^{-s*}}(s') = \{s'' | s'' \prec^{s*} s'\}$. Similarly, $S_{\prec^{-A*}}(A') = \{s'' | A'' \prec^{A*} A' \wedge s'' \in A''\}$ returns a set of ancestors of states in A'. In our example of Fig.4, $S_{\prec^{-A*}}(A_4) = \{$*Elec_device*, *Running*$\}$.

Orthogonal States. Given two states in an HTA, $s, s' \in \bigcup_{A_i \in F} S_{A_i}$, s and s' are orthogonal, written as $s \parallel s'$, if and only if $\exists s'' \in \bigcup_{A \in F} S_A$. $A, A' \in \rho(s'') \wedge A \neq A' \wedge s \in S_{\preccurlyeq^{A*}}(A) \wedge s' \in S_{\preccurlyeq^{A*}}(A')$. Intuitively speaking, if two states are descendant states of sibling sub-automata which have the same parent automaton, they are orthogonal. For instance, in Fig.4, *Intake* $\parallel$ *Fire_off*. We can define *orthogonal automata* similarly, $A \parallel A'$, given $\exists s, s' . (s \in A \wedge s' \in A' \wedge s \parallel s')$.

Invariant Compatibility. Given two states in an HTA, i.e., $s, s' \in \bigcup_{A \in F} S_A$, s' is compatible with s, written as $s' \vdash s$ if and only if $inv(s')$ entails $inv(s)$. Invariant compatibility is a partial order relation. Suppose, for example, we have two invariants, $x < 10$ and $x < 6 \wedge y < 3$, in state s and s' respectively. $inv(s)$ states that the value of clock x should be smaller than 10, and $inv(s')$ asserts the value of x should be smaller

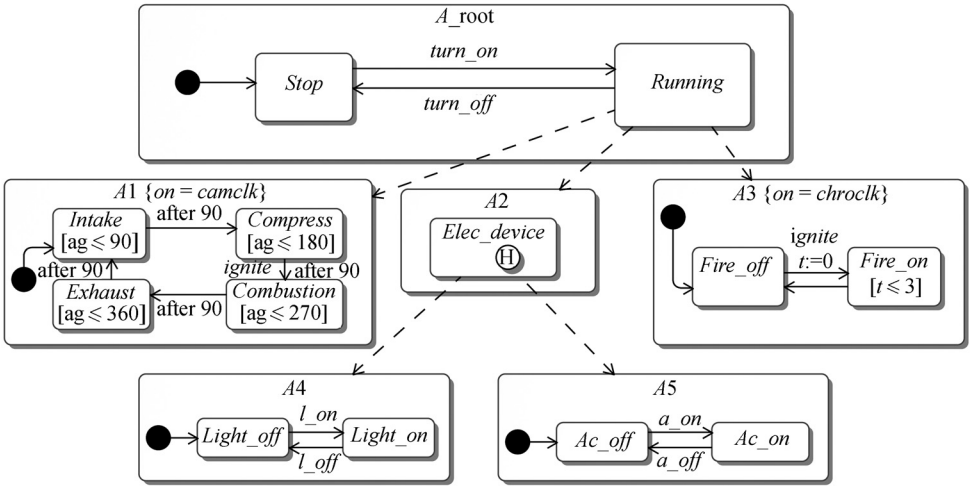

Fig.4. Example hierarchy illustration.

① For simplicity, we use event names in the diagram to represent elements of E instead of tuples, and τ denotes a silent transition.

than 6 and the value of clock y should be smaller than 3. In this case, $s' \vdash s$. Obviously, to be well-formed, for two states, s and s', if $s \preccurlyeq s'$, it entails that $s' \vdash s$.

Initial and History State Functions. We define two functions $init(s)$ and $hist(s)$ to retrieve the set of initial sub-states and history sub-states of a composite state s. $init(s) = \{s' | s' \in \bigcup_{A_i \in \rho^*(s')} S_{A_i} \wedge \sigma(s') = ENTRY\}$. To define $hist(s)$, we use an indicator function $lactive(A)$ to denote the most recently active states in an automaton A. If it is the first time to access A, *lactive* simply returns a set of entry states. If $\mu(s) = DEEP, hist(s) = \bigcup_{A_i \in \rho^*(s)} lactive(A_i)$. If $\mu(s) = SHALLOW$, $hist(s) = \bigcup_{A_i \in \rho(s)} lactive(A_i) \cup \{s' | s' \in \bigcup_{A_i \in \rho^*(s)/\rho(s)} S_{A_i} \wedge \sigma(s') = ENTRY\}$. In the example, if it is the first time to access the composite state *Elec_device*, $hist(Elec\_device) = \{Elec\_device, Light\_off, Ac\_off\}$, but other possible combinations exist depending on the last active situation.

Inter-level transitions are those whose actual source and target states are at different levels of the hierarchy. It is desirable in the cases when we want a transition directly targeting at or originating from a particular state inside a composite one. If such a transition is enabled, not only is the source state exited but also the parent of the source state is exited if it has. Similarly, entering a target state implies the fact that its parent state is also entered. In hierarchical timed automata models, transitions normally stay in the same level. Since inter-level transitions cross the boarders of hierarchy, they cannot directly be supported. To simulate inter-level transitions in hierarchical timed automata, we can lift the transition to the upmost states which are exited or entered[13]. Meanwhile the actual source and target state information are recorded and determined by specific functions, i.e., source and target restriction functions as described in the following.

Source and Target Restrictions. If a transition t is involved with composite states, the source or target is composed of several orthogonal basic states. This corresponds to *join* and *fork* transitions. In *A_root* of Fig.4, the transition target of *turn_on* is *Running*. Since $\rho(Running) = \{A1, A2, A3\}$, the actual transition target is the set of states inside the above refined automata separately. In this case, the default target set is $\{Intake, Fire\_off\} \cup hist(Elec\_device)$. Similar to [5], source and target restriction functions are defined as $sr : t \to S$ where $S \subseteq \bigcup_{A_i \in \rho^*(SRC(t))} S_{A_i}$ and $tr : t \to T$ where $T \subseteq \bigcup_{A_i \in \rho^*(TGT(t))} S_{A_i}$. For both sets, elements are pairwise orthogonal states if any. As their names hint, these functions are mainly to restrict the transitions involved with those composite states, and they are yielding the actual sources or targets of a transition. With these two functions, the inter-level transitions can be supported[13]. But different from their notions, we also augment the function the ability to yield the history sub-states. Thus there are three cases to derive the target restriction of a transition which decides the entry set of a composite state. If the transition terminates on the outside edge of the composite state, the target restriction function will generate a set of initial sub-states of the composite state: $TGT(t)$, i.e., $tr(t) = init(TGT(t))$. If the transition goes to a sub-state s' of the composite state directly, then $tr(t) = \{s'\} \cup init(s') \cup \{s'' | (s'' \parallel s') \wedge TGT(t) \prec^{s*} s'' \wedge \sigma(s'') = ENTRY\}$, and otherwise, if the transition goes to a history state s of an automata, then $tr(t) = hist(s) \cup \{s' | (s' \parallel s) \wedge TGT(t) \prec^{s*} s' \wedge \sigma(s') = ENTRY\}$. As an example, in Fig.2, the transition *turn_on* corresponds to our third case and it points to the history pseudo-state contained in *Elec_device*, i.e., $hist(Elec\_device)$. While in Fig.4, $TGT(turn\_on) = Running$, and the set of $\{s' | (s' \parallel Elec\_device) \wedge Running \prec^{s*} s' \wedge \sigma(s') = ENTRY\}$ is $\{Intake, Fire\_off\}$, thus the real target information returned by the restriction function $tr(turn\_on)$ is $hist(Elec\_device) \cup \{Intake, Fire\_off\}$.

3.3 Operational Semantics

The operational semantics is closely related to the notion of transition between configurations. Generally, a configuration denotes a snapshot of computation which contains active states, values of related clocks and history information. In this subsection, we present the associated operational semantics based on the concepts introduced above.

Configuration. A configuration (*conf*) of EHTA contains a set of active states, clock values, and history, represented as a tuple: (S, ν, θ). UML specifies that the current active "state" is represented by a set of trees of states along the hierarchy down to the innermost active sub-state[12]. Therefore, for the active state set S in a configuration, according to our aforementioned concepts, the following properties hold. There is one and only one state from the root automaton which is in the active state set, i.e., $\exists_1 s \in S_{A_{\text{root}}} \wedge s \in S$; for any composite state in the active state set, there is one and only one state from each of its refined automata which is in the active state set, i.e., $\forall s, A.(s \in S \wedge A \in \rho(s)) \Rightarrow \exists_1 s' \in A \wedge s' \in S$. The definitions of ν, θ are similar to those of [4].

Conflict Transitions. According to UML 2.0, if the intersection of the exit state sets of two transitions is non-empty, we say the two transitions are in conflict, written as $t_1 \# t_2$[5], i.e., $t_1, t_2 \in E$, $t_1 \neq t_2$, $(SRC(t_1) \cup sr(t_1)) \cap (SRC(t_2) \cup sr(t_2)) \neq \varnothing$.

Priority Schema. Priority schedule is a partial order

relation over transitions. A priority schema is defined by a tuple $(E, \preccurlyeq^p, p)$. p is a function, mapping a transition $t \in E$ to a specific priority. The default schema states that the transition originating from sub-states has higher priority over that originating from enclosing states[12]. But it does not prohibit the application of priority schemas from domain-specific profiles, e.g., MARTE[1].

Transition Selection. Due to the run-to-completion (RTC) semantics required by UML, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. Thus at one time, only one transition in the enabled set can be selected for execution. The enabled transition set contains maximal number of enabled and non-conflicting transitions satisfying the following conditions: it is prohibitive to have conflicting transitions within the set and transitions outside the set with higher priority than a transition inside. Therefore, given a configuration *conf:* (S, ν, θ), the enabled transitions under action set Σ_A is ET_{Σ_A} which is the subset of: $\{t | t \in T \wedge (\{SRC(t)\} \cup sr(t)) \subseteq conf.S \wedge \nu \vdash g_t\}$②. Accordingly, the elements in the set satisfy the following conditions: $\forall t_1,\ t_2 \in ET_{\Sigma_A}.\ \neg t_1 \# t_2$; $\forall t \in ET_{\Sigma_A},\ \nexists t'.\ t' \notin ET_{\Sigma_A} \wedge t' \# t \wedge (\{SRC(t)\} \cup sr(t)) \subseteq conf.S \wedge \nu \vdash g_{t'} \wedge p(t) \prec^p p(t')$. The enabled transition set under action set local to Σ_A is similarly defined as LET_{Σ_A}, but the transitions belonging to the descendants of A are not included.

We use a Kripke structure $k = (conf, conf_0, \xrightarrow{\text{STEP}})$ to define the operational semantics of the EHTA. *conf* is a tuple (S, ν, θ) defined as above. $\xrightarrow{\text{STEP}}$ has further several cases, i.e., *delay, progress, composition* and *synchronization.*

Similar to [4], a predicate $Inv(S, \nu)$ is defined to assert whether all the invariants of S hold at the clock values ν, i.e., $Inv_S(\nu) = \bigwedge_{s \in S} Inv_s(\nu)$.

Delays. Given a configuration (S, ν, θ), if the clocks advance, and the invariants of states in the set S hold, then it may not cause actual state transition. In this case, states and history information do not change except clock values.

$$\frac{Inv_S(\nu + d)}{(S, \nu, \theta) \xrightarrow{d} (S, adv_{T_S}(\nu + d), \theta)}.$$

Usually there is an upper limit value associated with the clock invariant of a state. If a local clock advances to this value and no enabled transitions exist, it will cause a deadlock.

Progresses. Progress type denotes the transitions which occur inside a sequential timed automaton[5,13]. In this case, the event dispatcher module selects a transition in the enabled transition set based on their priority if defined. As this step usually involves the exit of composite state, it would modify the history information accordingly. In our example, if the transition *turn_off* is triggered, the history information of *Elec_device* will be recorded. If the target composite state has a history tag attached (i.e., shallow history or deep history pseudo-state), the most recently active sub-states should be restored. For example, the transition *turn_on* will cause the state change of configurations from {*Stop*} to {*Running*, *Intake*, *Fire_off*, *Elec_device*, *Light_off*, *Ac_off*}. To state formally, the new history information θ' is recorded as follows: $\bigcup_{A_i \in H_t} lactive(A_i)$, where $H_t = \{A | \exists s.\ (SRC(t) \preccurlyeq^* s \wedge (\mu(s) = DEEP \vee \mu(s) = SHALLOW) \wedge A \in \rho(s))\}$. The update of history information is denoted by $\theta \mapsto \theta'$ in the following rule.

$$\frac{\begin{array}{c} t \in LET_{\Sigma_A} \\ \nexists t' \in ET_{\Sigma_A}.p(t) \prec p(t') \\ Inv_{tr(t)}([reset(t) \leftarrow v_0]\nu) \end{array}}{(S, \nu, \theta) \xrightarrow{t} (closure_{tr(t)}\, TGT(t), [reset(t) \leftarrow v_0]\nu, \theta')}.$$

Composition Transitions. This step describes the automaton A delegates the transitions to its descendant automata, meaning that A collects the transitions of its sub automata. This step happens on two conditions, i.e., either there are no enabled transitions in the $LEA_{\Sigma A}$, or the priority of the transitions in the descendants are higher than any elements in $LEA_{\Sigma A}$. The new configuration consists of the combined union of the new active states caused by the underlying transitions in the sub automata. Since the invariant compatibility exists between the states along the hierarchy, the invariant still holds for the ancestor state s in the new configuration.

$$\frac{\begin{array}{c} \{s\} = S_{conf} \cap S_A, \rho(s) = \{A_1, \ldots, A_m\} \neq \varnothing \\ A_1 :: (S_1, \nu_1, \theta_1) \xrightarrow{t_1} (S_1', \nu_1', \theta_1') \\ \vdots \\ A_m :: (S_m, \nu_m, \theta_m) \xrightarrow{t_m} (S_m', \nu_m', \theta_m') \\ (LET_{\Sigma_A} = \varnothing \vee\ \forall t \in LET_{\Sigma_A}, \\ \exists t_i \in ET_{\Sigma_A}, p(t) \prec p(t_i)) \end{array}}{(S, \nu, \theta) \xrightarrow{t_1, \ldots, t_m} (S', \nu', \theta')}.$$

In the above semantic formula, $(S', \nu', \theta') = (\{s\} \cup S_1' \cup \cdots \cup S_m',\ \nu_1' \cup \cdots \cup \nu_m',\ \theta_1' \cup \cdots \cup \theta_m')$.

Synchronized Transitions. This is a special kind of composite transition. There exists synchronized transitions, as the send and receive actions establish a

②$\nu \vdash g_t$ denotes the clock value of current configuration respects the transition guard of t.

connected channel of actions.

$$\frac{\begin{array}{c}\{s\} = S_{conf} \cap S_A, \rho(s) = \{A_1, \ldots, A_m\} \neq \varnothing \\ A_i :: (S_i, \nu_i, \theta_i) \xrightarrow{t?} (S_i', \nu_i', \theta_i') \wedge A_i \in \rho(s) \\ A_j :: (S_j, \nu_j, \theta_j) \xrightarrow{t!} (S_j', \nu_j', \theta_j') \wedge A_j \in \rho(s) \\ (LET_{\Sigma_A} = \varnothing \vee \forall t' \in LEA_{\Sigma_A}, \\ \exists t \in ET_{\Sigma_A}, p(t') \prec p(t))\end{array}}{(S, \nu, \theta) \xrightarrow{t!,t?} (S', \nu', \theta')}.$$

In the above semantic formula, $(S', \nu', \theta') = (\{s\} \cup S_i' \cup S_j' \cup \bigcup_{k \neq i,j} S_k, \nu_i' \cup \nu_j' \cup \bigcup_{k \neq i,j} \nu_k, \theta_i' \cup \theta_j' \cup \bigcup_{k \neq i,j} \theta_k)$.

4 Translation

4.1 Algorithms

To translate UML/MARTE state machine diagrams into the input language of existing model checkers, we develop a prototype tool. It consists of three main components: parser, model constructor, and target translator. The input state machine diagram is provided in XMI (XML Metadata Interchange) format, and the parser re-constructs the UML state machine encoded in the XML file. The constructor produces the hierarchical timed automata based on the operational semantics presented previously; finally, the translator produces the input code for the selected model checker.

In this paper, we use UPPAAL[14] as the test-bed since its formalism is based on timed automata, and this similarity eases both the translation process and the later phase of mimicking the proposed semantics. However, as UPPAAL does not support multiform clocks, a normalization procedure has to be performed. The transformation is not straightforward but the space limits a detailed explantation. Hence in the following, we give a brief description of the normalization process. A base clock needs to be designated first. For continuous clocks, according to the relation specification of the time structure, the clock associated constraints values (i.e., guards and invariants) are normalized separately. If there exist clocks which are incomparable to the base clock, this is contrary to the assumption of UPPAAL that clocks advance with the same speed. However, we can use a separate counter[14] to simulate these clocks, the frequency of which can be set arbitrarily depending on the application context. For logical clocks, since they actually describe the "happen-before" relation among events[15], state machines naturally describe the ordered execution traces of these events (one state machine considered as a process). However, to model the pre-order relation of sending/receiving messages, since UPPAAL's channel abstracts away the delay, we add an intermediate location denoting the state of transmitting the message. In this way, the pre-order relation can be established. If there exists relationship between logic clock and the base clock specified by the time structure, we use specific channels to synchronize the related states of different automata to guarantee the order.

In our example, we use the $\zeta_{\text{ms}}^{\text{degree}}$ function in the previous section to model the relationship and normalize the multiform clocks. Specifically, we choose millisecond (ms) as the base time unit. Moreover, since UPPAAL only operates on the parallel composition of "plain" timed automata, we have to flatten the hierarchy while carrying out the actual transformation.

Space limitations oblige us to require that readers be familiar with UPPAAL and its features. The flattening process — embedded in the target translator — comprises three steps. The pseudo-code of the algorithmic steps is attached in appendix A1∼A3. Generally, each sequential timed automaton of the EHTA is translated into an UPPAAL template③. Then the hierarchy is rendered as follows. A state *inactive* is added to each generated template, but the root one, to denote its current status. Then, for each transition entering a composite state S_{comp} with no history, we add a transition from its state *inactive* to all its default entry states in their corresponding templates. If S_{comp} is an initial state, then the default entry states of the templates generated from its sub automata are marked as initial. Otherwise, the newly added inactive states are as initial. To synchronize the transitions between different templates, we use the facilities provided by UPPAAL. The corresponding transition actions are annotated with *broadcast* channels. In our example, the action associated with the transition entering *Running* is *turn_on*. We turned this transition into a broadcast channel, and each template that corresponds to a sub-state of *Running* has a transition synchronized with it (i.e., with *turn_on*?).

If there is a history pseudo-state S_{hist}, we use a Boolean variable associated with each sub-state to record history information. The variable that corresponds to the default entry state is initialized to true; the others to false. Values change as transitions fire and a true value always corresponds to the last visited state. A selection state is added to each of the templates that correspond to the sub automata $\rho(S_{\text{hist}})$, and these templates are marked as $TP_{\rho(S_{\text{hist}})}$. Again, we use UPPAAL's facility and mark the selection state as *committed*[14], because we restrict it with no delays. For any incoming transition t whose target restriction set④

③By UPPAAL, a template denotes an automaton that can be instantiated and composed to define a system.

④The set returned by the target restriction function of t.

contains S_{hist}, in each template of $TP_{\rho(S_{\text{hist}})}$, we add a transition from *inactive* to the selection state associated with corresponding channels and guards. Then we add transitions from the selection state to the historical sub-states of S_{hist} guarded by the selection variables. If S_{hist} is a shallow history state, the process stops at the first sub-level. If it is a deep history state, the process continues recursively until the last level. Fig.5 shows the UPPAAL model that corresponds to our example. *System* is the parallel composition of the six templates. In every template except *A_root*, an *inactive* state has been added. *Elec_device* has a history pseudo-state, thus templates *A*4 and *A*5 of Fig.5 are augmented with a committed selection state and a set of Boolean variables. Since the target restriction set of *turn_on* contains *Elec_device*, in *A*4 and *A*5, a transition from the *inactive* state to the *selection* state annotated with a broadcast channel is added respectively.

Eventually, we must add global join information. For this purpose, a special state is added to each template but the root one. Since this newly added state requires no delay, it is marked as *committed* in UPPAAL. For any transition t whose source restriction set[5] contains a composite state or a history pseudo-state S_{hist}, then in each of the templates $TP_{\rho(S_{\text{hist}})}$, there will be transitions from each of its states to this global join state, and further there is a transition from this join state to the inactive state marked with corresponding channels, guards and temporal invariables. If S_{hist} has a history pseudo-state, the history selection variables associated with the states in the templates $TP_{\rho(S_{\text{hist}})}$(or $TP_{\rho^*(s_{\text{hist}})}$, depending on the types of the history pseudo-state) will be set to *true*. Meanwhile, the ones associated with other states are set to false to assure that only the value of the variable associated with the state that last exits is true.

In our example, transition *turn_off* in *A_root* triggers a global join action in the templates which are generated from $\rho(Running)$. There must be a transition from every state to a committed join state synchronized with *turn_off*?, and these committed states will have transitions to the corresponding inactive states in their templates. Since $A2(elec\_device)$ has a history pseudo-state, the transitions to the global join states in *A*4 and *A*5 also set the corresponding Boolean history variables.

4.2 Verification

This section presents an initial assessment of the verification based on the semantics proposed above. The experiments are based on the automotive example used throughout the paper: to play with the size of the example, we use the number of lights in the vehicle as parameter. We measure the number of explored states, the peak consumption of memory, and the time needed to perform the complete verification. The experiments are conducted on a PC with an AMD Athlon XP 2000+ processor and 1.0 GB RAM running Ubuntu 8.04 OS. The version of UPPAAL is 4.1.3 with an academic licence.

In our example, the spark plug ignites when the camshaft advances over 90 degrees, and triggers the transition of the engine's state from *compression* to *combustion*. Meanwhile, the spark plug itself changes from *fire_off* to *fire_on*. The channel action *ignite* establishes the synchronization of the two transitions. The *fire_on* state exists for a very short chronometric

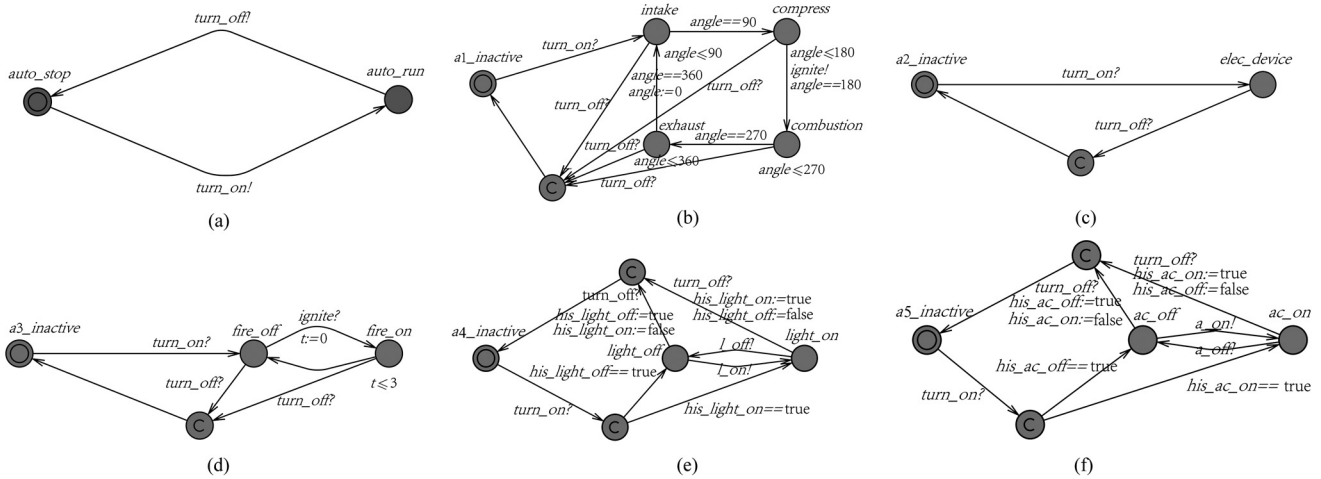

Fig.5. Translated UPPAAL model (parallel composition of templates). (a) *auto*(A_root). (b) *engine*(*A*1). (c) *elec_device*(*A*2). *spark_plug*(*A*3). (e) *light*(*A*4). (f) *ac*(*A*5).

[5]The set returned by the source restriction function of t.

time interval and goes back to state *fire_off* again. The example limits this interval to 3 milliseconds with an invariant: $t \leqslant 3$. For the engine, initially, we set the RPM value to 3 000. In this context, we want to check whether the system can enter a problematic configuration, for example, the co-existence of *intake* and *fire_on*. This safety property (SP_1) can be rendered in CTL as: $(A[]\neg(engine.exhaust \wedge spark\_plug.fire\_on))$. As the two timed events are associated with different forms of clocks, the traditional assumption that clocks are of the same type does not hold in this case. We need the time structure presented in Section 3 to model the relationships between the two clocks.

Since RPM can change while the engine is working, we are interested in knowing whether SP_1 keeps holding true. We increase the value gradually and discover that when it becomes equal to 12 000, UPPAAL reports an error and generates the trace of a counter example. The error corresponds to the fact that the engine's rotation speed cannot exceed a certain upper limit. Above the threshold, the spark plug and the engine would not be coordinated properly: for example, the spark plug may still be ignited while the engine already enters an *exhaust* state. To distinguish the property in these two settings, in our experiment, we mark SP_1 as SP'_1 when the value of RPM is equal to 12 000.

Besides the aforementioned safety property, we are also interested in probing the correctness of proposed translation. To this end, we propose another example property: we want to guarantee that whenever the root template is in state *Running*, all the other templates cannot be *inactive*. Rendered in CTL as SP_2, this becomes: $A[](auto.Running \rightarrow (\neg engine.a1\_inactive \wedge \neg elec\_device.a2\_inactive \wedge \neg spark\_plug.a3\_inactive \wedge \neg light[1,N].a4\_inactive \wedge \neg ac.a5\_inactive))$.

Table 1 summarizes the different metrics when incrementing the number of lights (our parameter) from 1 to 15. When the parameter is equal to or below 12, SP_1 and SP_2 hold for the multi-clock configurations of interest. The verification of these two properties runs out of memory when the parameter is equal to 15 or greater. For SP'_1, since UPPAAL can find a counter example without exploring the whole state space, the states listed in Table 1 are only those explored before finding the counter example.

5 Discussion

Rigorous semantics is a prerequisite for formal analysis and verification. This is why the formalization of UML diagrams has been an active research topic over the last years. This paper defines a step-based formal semantics for UML/MARTE state machine diagrams using extended hierarchical timed automata model. Our proposal takes into account the basic elements of the diagrams and the new features of multiform time support introduced by MARTE: the notions of logical and physical clocks are both covered by our solution. Thus the approach can be seen as an extension (evolution) over the earlier work on the formalization of UML state machine diagrams with real-time extensions.

Our proposal concentrates on a sub-set of the elements of UML state machine diagrams and MARTE, and this incomplete coverage is the main limitation of the approach. This is mainly due to the feasibility nature of the work, aimed to pave the ground to wider and more complete solutions, but the limitation can also be alleviated by means of semantically-equivalent transformations of models' elements. For example, UML defines three types of states: simple states (basic states, in our context), composite states, and submachine states[12]. Since submachine states are semantically equivalent to composite states, their transformation comes for free.

6 Related Work

Since MARTE is a new member of the UML family, and the standard itself is still under development[1,8], many efforts are devoted to improving the precision and applicability of the profile. Thus there exist two parallel threads of research. The first emphasizes the formalization of the constituent components of the profile, for example, the clock constraint language, inner packages. The second emphasizes the formal annotations on UML diagrams to enable the formal verification. This section discusses some representative approaches of the two threads.

Table 1. Summary of Our Verification Efforts

N	States	R/V Memory (MB)	Time (s)
1	(348, 12, 348)	(3.3/21.78, 3.3/21.79, 3.4/21.79)	(0.02, 0.02, 0.04)
2	(718, 18, 718)	(3.9/22.31, 3.61/22.07, 3.9/22.32)	(0.02, 0.02, 0.04)
4	(3118, 46, 3118)	(4.41/22.8, 4.24/22.7, 4.66/22.6)	(0.28, 0.02, 0.26)
8	(76318, 534, 76318)	(8.76/25.66, 5.31/23.83, 9.06/25.96)	(1.86, 0.06, 1.82)
12	(3353518, 8222, 3353518)	(208.7/216.1, 7.4/25.2, 209.1/216.6)	(110.6, 0.38, 112.7)
15	(NC, 65572, NC)	(OM, 14.55/27.25, OM)	(NC, 2.06, NC)

Note: N: number of lights, R/V: Residential/Virtual, NC: Not Concluded, OM: Out of Memory

Mallet and André[16] provided a precise semantics for the Clock Constraint Specification Language (CCSL) associated with MARTE in terms of Time Petri Nets[10] and synchronous languages with well-defined semantics. This work focuses on a language of MARTE, while our proposal focuses on the behavior of UML/MARTE models. Clearly, high level behavior analysis benefits from precise language-level semantic foundations. Also Ge *et al.*[17] presented a model transformation based approach to analyzing and verifying UML/MARTE models based on Time Petri Nets[10]. Proposed verification is mainly against time properties and a lot of time-irrelevant parts are abstracted away. Moreover, as admitted by the authors, the correctness of the translation itself cannot be proved, and reachability is undecidable for Time Petri Nets.

As for the formal semantics of UML state machine diagrams, Crane and Dingel[18] proposed a comparison framework organized around three main categories: mathematical models, rewriting systems, and translation approaches. Since most of the translation-based approaches concentrate on establishing the mapping, instead of verification, the rest of this section only addresses the first two categories.

Harel and Naamad[19] proposed a step-based operational semantics for statecharts without considering the time dimension. Mikk *et al.*[13] first introduced the notion of hierarchical automata as intermediate representation for the implementation of tools for statecharts. They also introduced the notion of source and target restriction functions to model inter-level transitions. Based on this work, Latella and Massink[5] explicitly introduced priorities into the step-based transitions, which allows flexible priority schemas to be associated with transitions. Some of our concepts are based on this work, which provides a concise semantics for state machine diagrams, but this work does not model the time dimension and history states explicitly[5,13].

David *et al.*[4,20-21] proposed an approach based on hierarchical timed automata to model UML diagrams with real-time extensions and also a solution for flattening the models into UPPAAL timed automata. However, their assumptions preclude the support to inter-level transitions and the treatment of time is implicitly bounded to physical clocks. To overcome these disadvantages, our proposal introduces the concept of source/target restriction functions and time structures. The former is to support inter-level transitions, while the latter is to model multiform time. Moreover, we gave a hierarchical definition of HTAs which to give a more intuitive structural correspondence. André *et al.*[6] provided a notion of rich clocks and multiform time representation similar to ours, but theirs is restricted to discrete sets and is not equipped with a formal operational semantics. Akshay *et al.*[22] proposed distributed timed automata with independently evolving clocks. However, the assumption that the rates of distributed clocks depend on some absolute time is not necessary in our model.

There have been also proposals based on Petri nets and graph rewriting. Baresi and Pezzè[2] proposed a mapping from UML behavioral diagrams to high level Petri nets through pairs of transformation rules. Although the approach can exploit existing Petri net theory and tools to support the automated analysis of UML specifications, it does not address the temporal aspects. Similarly Hu and Shatz[23] did not cover the temporal dimension. Hölscher *et al.*[24] presented a semantics for UML based on the translation of a model into a graph transformation system. The graph transformation system comprises both rules and an initial graph that represents the current state of the system. These states however are limited to simple states. Also Kong *et al.*[25] proposed a graph-based formal model to augment UML behavioral diagrams with formal semantics. However, as summarized in [18], graphical rewriting-based models are not too sophisticated in covering the features of UML state machines.

7 Conclusions and Future Work

This paper presented a proposal for augmenting UML state machine diagrams with MARTE clocks and for ascribing the resulting notation with a formal operational semantics based on extended hierarchical timed automata. Specifically, the approach addresses the core concept of multiform clocks in MARTE, formalizes it, and embeds it into the operational semantics associated with state diagrams. This is a progress over past proposals that formalize (subsets of) UML through hierarchical automata since, to the best of our knowledge, this is the first proposal that embeds and formalizes multiform time in hierarchical timed automata. It also deals with inter-level transitions and history states. The result can be used for further analysis and verification: the paper explained how to translate produced diagrams into the input language of UPPAAL.

The experiments provided interesting results and also paved the ground to exploiting models' hierarchy during analysis. In fact, in some cases, when properties predicate on states at a given level, there is no need to also consider their sub-states: this reduces the number of generated states and allows for more compositional reasoning techniques[26]. The formalization of these optimizations is part of our future work.

Acknowledgements The authors would like to

thank Mieke Massink for the discussions at the summer school of SEFM 2010 and Alexander Knapp for sharing HUGO/RT with us.

References

[1] OMG. UML profile for MARTE: Modeling and analysis of real-time embedded systems. Version 1.0, formal/2009-11-02, 2009, http://www.omg.org/spec/MARTE/1.0/.

[2] Baresi L, Pezze M. On formalizing UML with high-level Petri nets. In *Concurrent Object-Oriented Programming and Petri Nets*, Springer Verlag, 2001, pp.276-304.

[3] Crane M, Dingel J. Towards a formal account of a foundational subset for executable UML models. In *Proc. the 11th International Conference on Model Driven Engineering Languages and Systems*, October 2008, pp.675-689.

[4] David A, Möller M, Yi W. Formal verification of UML statecharts with real-time extensions. In *Proc. the 5th Int. Conf. Fundamental Approaches to Software Engineering*, Apr. 2002, pp.218-232.

[5] Latella D, Majzik I, Massink M. Towards a formal operational semantics of UML statechart diagrams. In *Proc. the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems*, March 1999, p.465.

[6] André C, Mallet F, Peraldi-Frati M. A multiform time approach to real-time system modeling: Application to an automotive system. In *Proc. the International Symposium on Industrial Embedded Systems*, July 2007, pp.234-241.

[7] Mallet F, de Simone F. MARTE: A profile for RT/E systems modeling, analysis–and simulation? In *Proc. the 1st Simutools*, June 2008, Article No.43.

[8] OMG. UML profile for MARTE: Modeling and analysis of real-time embedded systems. Version 1.1, formal/2011-06-02, 2011, http://www.omg.org/spec/MARTE/1.1.

[9] Alur R, Dill D. A theory of timed automata. *Theoretical Computer Science*, 1994, 126(2): 183-235.

[10] Berthomieu B, Ribet P, Vernadat F. The tool TINA–Construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 2004, 42(14): 2741-2756.

[11] Fecher H, Schönborn J, Kyas M, de Roever W. 29 new unclarities in the semantics of UML 2.0 state machines. In *Proc. the 7th International Conference on Formal Methods and Software Engineering*, November 2005, pp.52-65.

[12] OMG. OMG unified modeling language^{TM} (OMG UML), superstructure. Version 2.2, 2009, http://www.omg.org/spec/UML/2.2/Superstructure.

[13] Mikk E, Lakhnechi Y, Siegel M. Hierarchical automata as model for statecharts. In *Proc. the 3rd Asian Computing Science Conf. Advance in Computing Science*, December 1997, pp.181-196.

[14] Behrmann G, David A, Larsen K. A tutorial on UPPAAL. In *Proc. the International Conference on Formal Methods for the Design of Real-time Systems*, July 2004, pp.33-35.

[15] Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978, 21(7): 558-565.

[16] Mallet F, André C. On the semantics of UML/MARTE clock constraints. In *Proc. the Int. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing*, Mar. 2009, pp.305-312.

[17] Ge N, Pantel M. Time properties dedicated semantics for UML-MARTE safety critical real-time system verification. In *Proc. the 8th European Conference on Modelling Foundations and Applications*, July 2012, pp.25-39.

[18] Crane M, Dingel J. On the semantics of UML state machines: Categorization and comparision. Technical Report 2005-501, Queen's University, 2005.

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

[20] David A, Möller M. From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. Technical Report, University of Aarhus, 2001.

[21] Giese H, Burmester S. Real-time statechart semantics. Technical Report TR-RI-03-239, University of Paderborn, 2003.

[22] Akshay S, Bollig B, Gastin P, Mukund M, Kumar K N. Distributed timed automata with independently evolving clocks. In *Proc. the 19th International Conference on Concurrency Theory*, August 2008, pp.82-97.

[23] Hu Z, Shatz S. Explicit modeling of semantics associated with composite states in UML statecharts. *Automated Software Engineering*, 2006, 13(4): 423-467.

[24] Hölscher K, Ziemann P, Gogolla M. On translating UML models into graph transformation systems. *Journal of Visual Languages & Computing*, 2006, 17(1): 78-105.

[25] Kong J, Zhang K, Dong J, Xu D. Specifying behavioral semantics of UML diagrams through graph transformations. *Journal of Systems and Software*, 2009, 82(2): 292-306.

[26] Bindelli S, Di Nitto E, Furia C *et al.* Using compositionality to formally model and analyze systems built of a high number of components. In *Proc. the 15th Int. Conf. Eng. of Complex Computer Systems*, Mar. 2010, pp.85-94.

Yu Zhou is an assistant professor at Nanjing University of Aeronautics and Astronautics, China. He received his Ph.D. degree in software engineering from Nanjing University in 2009. In 2010, he was a post-doc at DEEPSE (DEpendable, Evolvable, Pervasive Software Engineering) Research Group of Politecnico di Milano, Italy. His research interests are mainly in software architecture, software evolution and verification with particular emphasis on leveraging domain-specific knowledge to verify large-scale software systems.

Luciano Baresi is an associate professor at Politecnico di Milano, Italy, where he spent most of his professional life and got both his Master degree in electronic engineering and Ph.D. degree in computer science. Luciano is a regular member of the program committee of important conferences. He was/will be the program chair of ICECCS 2002, FASE 2006, ICWE 2007, ICSOC 2009, SEAMS 2012 and ESEC/FSE 2013. Luciano is also currently a member of the editorial board of the Transactions on Autonomous and Adaptive Systems (ACM) and of Service Oriented Computing and Applications (Springer). Luciano co-authored some 120 papers and his research interests touch different aspects of software engineering: formal modeling approaches, specification languages, UML and distributed and ubiquitous software systems.

Matteo Rossi received his Laurea Degree in computer engineering from Politecnico di Milano in 1999. At the same time, he received his Diplôme d'Ingénieur from the École nationale supérieure de Techniques Avancées (ENSTA), Paris, as part of the Top Industrial Managers Europe program. He received his Ph.D. degree in computer engineering and automation in 2003, also from Politecnico di Milano. He was a post-doc at Politecnico di Milano from April 2003 until January 2005. Since then he is assistant professor at Politecnico di Milano. His research interests are mainly in formal methods for safety-critical and real-time systems, with particular reference to architectures for real-time distributed systems, and formal techniques for modeling security vulnerabilities of computer networks. He is also interested in the application of formal methods in fields other than computer science, and to production systems in particular.

Appendix Flattening Algorithms

A.1 Algorithm 1

Algorithm 1. Step 1: Generating Basic Elements in Templates

Data: M: model of EHTA

Result: T: set of templates

begin

- $T \longleftarrow \varnothing$;
- HashMap *map*;
- **forall** $A_i \in M.F$ **do**
  - Create a template t;
  - Add locations and transitions based on $A_i.S$ and $A_i.\Sigma$;
  - Normalize the multiform clock based on $M.W$;
  - **if** $A_i \neq A_{\text{root}}$ **then**
    - Add A_i_*inactive* location in t;
    - Mark A_i_*inactive* as committed;
  - $T \longleftarrow T \cup \{t\}$;
  - *map*.add(A_i, t);
- **forall** $A_i \in M.F$ **do**
  - **forall** $s \in A_i.S \wedge \sigma(s) = COMPOSITE$ **do**
    - **forall** $A_i \in \rho(s)$ **do**
      - *temp* $\longleftarrow$ *map*.get(A_i);
      - **forall** *in_tr* $\in$ {incoming transitions to s} **do**
        - Add transition *trans* from A_i_*inactive* to the entry location in *temp*;
        - Augment the action of *trans* with *broadcast* channel;
      - **if** s is initial **then**
        - Default entry location in A_i is marked as initial in *temp*;
      - **else**
        - A_i_*inactive* is marked as initial in *temp*
    - *map*.update;
    - *T*.update;

A.2 Algorithm 2

Algorithm 2. Step 2: Augmenting History Information in Templates

Data: *map*: Hashmap by step 1; M: model of EHTA

Result: T: set of templates

begin

- AA: set of Automata $\longleftarrow \varnothing$;
- $T \longleftarrow \varnothing$;
- Template *temp*;
- **forall** $s \in \bigcup_{A_i \in M.F} S_{A_i} \wedge \sigma(s) = HISTORY$ **do**
  - **if** $\mu(s) = SHALLOW$ **then**
    - $AA \longleftarrow \rho(s)$;
  - **else**
    - $AA \longleftarrow \rho^*(s)$;
  - **forall** $A_i \in AA$ **do**
    - *temp* $\longleftarrow$ *map*.get(A_i);
    - Add selection location A_i*_selection* in *temp*;
    - Mark A_i*_selection* as committed;
    - Add Boolean variable guards based on $A_i.S$;
    - Add transitions from A_i*_selection* to locations generated by $A_i.S$;
    - Attach the transitions with guards;
    - **forall** transition $tt \in M.\Sigma \wedge s \in tr(tt)$ **do**
      - Add transitions from A_i*_inactive* to A_i*_selection* in *temp*;
      - Attach related channels, guards and clock resets;
    - *map*.update;
- $T \longleftarrow \bigcup_{A_i \in M.A}$ *map*.get(A_i);

A3 Algorithm 3

Algorithm 3. Step 3: Add Global Joins in Templates

Data: *map*: Hashmap by step 2; M: model of EHTA

Result: T: set of templates

begin

- template *temp*;
- **forall** $A_i \in M.F \wedge A_i \neq A_{\text{root}}$ **do**
  - *temp* $\longleftarrow$ *map*.get(A_i);
  - Add join location A_i*_join* in *temp*;
  - Mark A_i*_join* as committed;
- **forall** $t \in \bigcup_{A_i \in M.F} \Sigma_{A_i}$ **do**
  - **forall** $s \in sr(t) \wedge (\sigma(s) = COMPOSITE \vee \sigma(s) = HISTORY)$ **do**
    - **forall** $A_j \in \rho^*(s)$ **do**
      - *temp* = *map*.get(A_j);
      - Add transitions tt from each location generated by $A_j.S$ to A_j*_join* in *temp*;
      - Add transitions tt' from A_j*_join* to A_j*_inactive*;
      - Associate tt with channels, guards and clock resets based on t;
      - **if** $\mu(s) = SHALLOW \wedge A_j \in \rho(s)$ **then**
        - Associate tt with actions setting history variables;
      - **if** $\mu(s) = DEEP$ **then**
        - Associate tt with actions setting history variables;
      - *map*.update;
- $T \longleftarrow \bigcup_{A_i \in M.A}$ *map*.get(A_i);