EMERALD: An Automated Modeling and Verification Tool for Component-Based Real-Time Systems

Yizhou Zhang*, Hao Lin*, Guoqiang Li†

* BASICS, School of Software, Shanghai Jiao Tong University
† State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
{zhangyz, linhao, li.g}@sjtu.edu.cn

Abstract—Controller automata are a formal theory to model and analyze real-time systems with mutex components. It is mainly designed for a small scale real-time system in the low-level sequential behaviors, where control passing is explicitly handled. A controller automaton properly extends a timed automaton. Given a strict partial order over states, an ordered controller automaton can be faithfully translated into a timed automaton. We present a tool named EMERALD to translate ordered controller automata into timed automata and perform the transformation between their specifications in order to use the existing model checking engine or timed automata, say, UPPAAL. Case studies show that our tool can be effectively used in modeling and verification of component-based real-time systems.

Keywords—Real-time systems, modeling, verification, controller automata, timed automata

I. INTRODUCTION

Timed automata [2], [4] are a theory for modeling and verifying real-time systems. A timed automaton is a finite-state machine extended with a finite set of clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All clocks progress synchronously. UPPAAL [21] is a widely used tool that can describe, simulate and verify a system described by timed automata. It has been applied successfully in case studies, such as schedulability analysis of task systems [15], [14].

Although timed automata provide a general method for system construction, in a low level behavior, components often behave sequentially by passing control via communication. Li et al. proposed a behavioral model, named controller automata [25], [24], to combine timed automata by focusing on the control passing between components. In a controller automaton, to each state a timed automaton is assigned. A timed automaton at a state may be preempted by the control passing to another state by a global labeled transition. A controller automaton properly extends the expressive power because of the unbounded stack. Given a strict partial order over states, an ordered controller automaton can be faithfully translated into a timed automaton [23]. It is shown that most of applications can be encoded by ordered controller automata.

This paper develops a new tool, named EMERALD, to implement the translating algorithm above, and to define an ordered controller automata graphically. An ordered controller automaton can be easily edited and saved into an XML file. EMERALD is capable of checking for reachability properties of an ordered controller automaton. We defined a specification language to describe the reachability property. Since EMERALD adopt Uppaal’s model checking engine for timed automata, after translating an ordered controller automaton into its respective timed automaton. Furthermore, we improve the algorithm proposed in [23] by removing isolated locations, so that the translation is executed more efficiently. Case studies show that our tool can be effectively used in modeling and verification of component-based real-time systems.

The remainder of the paper is organized as follows. Section 2 briefly reviews timed automata and controller automata. Section 3 will describe our EMERALD tool in detail. We will propose our translation algorithm, describe the syntax of EMERALD specification, and present graphical editor. Section 4 will give an example to show how to build a controller automaton and verify its reachability properties. Finally, Section 5 gives the related work and Section 6 concludes our work with future research directions.

II. PRELIMINARIES

This section briefly reviews timed automata [2], [18] and controller automata [25], [24].

A. Timed Automata

Timed automata are first proposed in [2]. The original timed automata use Büchi accepting conditions to enforce progress properties [2]. Later timed safety automata, introduced in [18], specify progress properties using local invariants. In this paper, we shall focus on timed safety automata, referring to them as timed automata when it is understood from the context.

Let $\mathbb{R}^0$ and $\mathbb{Q}^0$ denote the sets of non-negative real numbers and non-negative rational numbers. Time constraints over clocks are defined as follows.

**Definition 1** (Time Constraints). Let $X = \{x_1, \ldots, x_n\}$ be a finite set of clocks. The set of clock constraints, $\Phi(X)$, over $X$ is defined by the grammar:

$$\phi ::= \top | x \gg c | x - y \gg c | \neg \phi | \phi \land \phi$$
where \( c \in \mathbb{Q}^0, x, y \in X, \) and \( \preceq \in \{<, >, \leq, \geq\} \).

For the set of clocks \( X \), a clock valuation is a function \( \nu : X \to \mathbb{R}^0 \). It assigns a value to each clock \( x \in X \). For a clock valuation \( \nu \) and a clock constraint \( \phi \), we write \( \nu \models \phi \) to denote that \( \nu \) satisfies the constraint \( \phi \). Given a set of clocks \( \lambda \subseteq X \) and a clock valuation \( \nu \), let a clock reset function \( \nu[\lambda] \) be a clock valuation, defined as follows:

\[
(\nu[\lambda])(x) = \begin{cases} 
0 & \text{if } x \in \lambda \\
\nu(x) & \text{otherwise}
\end{cases}
\]

Given a clock valuation \( \nu \) and a time \( t \in \mathbb{R}^0 \), we define \( (\nu + t)(x) = \nu(x) + t \), for \( x \in X \).

**Definition 2** (Timed Automata). A timed (safety) automaton is a tuple \( \mathcal{A} = (E, H, Q, q_0, X, I, \delta) \), where

- \( E \) is a finite set of external symbols, composed of two disjoint sets \( E = E_a \cup E_i \), where \( E_a \) is the set of triggering symbols, and \( E_i \) is the set of triggered symbols.
- \( H \) is a finite set of internal symbols.
- \( q_0 \in Q \) is the initial location.
- \( X \) is a finite set of clocks.
- \( I : Q \to \Phi(X) \) is a function assigning each location with a clock constraint, called an invariant.
- \( \delta \subseteq Q \times (E \cup H) \times \Phi(X) \times 2^X \times Q \).

When \( \langle q_1, a, \phi, \lambda, q_2 \rangle \in \delta \), we write \( q_1 \xrightarrow{a,\phi,\lambda} q_2 \).

Given a timed automaton \( \mathcal{A} \in \mathcal{A} \), we use \( E(\mathcal{A}), H(\mathcal{A}), Q(\mathcal{A}), q_0(\mathcal{A}) \) and \( X(\mathcal{A}) \) to represent its set of actions, control locations, initial location, and set of clocks, respectively. We will use similar notation for controller automata.

The semantics of timed automata includes progress transitions, for time elapsing within one control location, and discrete transitions, for transference between two control locations [4].

**Definition 3** (Semantics of Timed Automata). A configuration of TA is a pair \( (q, \nu) \) of a control location \( q \in Q \), and a clock valuation \( \nu \) on \( X \). The labeled transition system (LTS) of timed automata is represented as follows,

- **Progress transition**: \( \langle q, \nu \rangle \xrightarrow{a} \langle q, \nu + a \rangle \), where \( t \in \mathbb{R}^0 \) and \( (\nu + t) \models I(q) \).
- **Discrete transition**: \( \langle q_1, \nu \rangle \xrightarrow{a,\phi,\lambda} \langle q_2, \nu[\lambda] \rangle \), if \( q_1 \xrightarrow{a,\phi,\lambda} q_2 \), and \( \nu \models \phi \), and \( \nu[\lambda] \models I(q_2) \).

**Fact 1.** The reachability problem for timed automata [2], [4] is decidable.

**B. Controller Automata**

Intuitively, a controller automaton [24] provides a sequential policy of execution for a set of timed automata (e.g. An example is shown in Fig. 1). We equip these timed automata with three relations, internal, push, and pop, to handle competition, preemption and resumption, respectively.

**Definition 4** (Controller Automata). A controller automaton (CA) is a tuple \( \mathcal{C} = (E, H, S, s_0, M, x_{run}, T, \delta) \), where

- \( E \) is a finite set of external symbols, and \( H \) is a finite set of internal symbols.
- \( S \) is a finite set of states, and \( s_0 \in S \) is the initial state.
- \( M : S \to \mathbb{R}^+ \) is a function assigning each state with a timed automaton, where the assigned timed automata share external symbols with \( E \), and their internal symbols (including \( H \)) are pairwise disjoint.
- \( T : S \to \mathbb{R}^+ \) is a function assigning each state with a time, named expected running time.
- \( x_{run} \) is a global clock to accumulate the running time of all states.
- \( \delta \subseteq S \times (E \cup H) \times S \), which is partitioned into three disjoint subsets, \( \delta_{push}, \delta_{pop}, \delta_{int} \), for push, pop and internal relations, respectively.

We prepare a stack for configurations of a controller automaton, in order to record the state and the running control location when a push action is performed, and to resume the running context when a pop action is performed. By introducing the time lag, the semantics of a controller automaton can be formally defined as follows:

**Definition 5** (Semantics of Controller Automata). A configuration for a controller automaton \( \mathcal{C} \) is a tuple \( (s, q, \nu, \kappa, S) \), where

- \( s \) is the running state of \( \mathcal{C} \);
- \( q \) is the current running control location in \( M(s) \);
- \( \nu \) is the clock valuation for all clocks \( \bigcup_{s \in S} X(M(s)) \cup \{x_{run}\} \) in \( \mathcal{C} \);
- \( \kappa \) is the set of clocks keeping frozen when the time elapses, named frozen clocks;
With this restriction, there is an unbounded but finite number of execution paths, which also leads to difficulty in analysis and verification. A restrictive subclass of controller automata is introduced, in which a strict partial order is imposed on the state set of a controller automaton. With this restriction, there is an unbounded but finite number of fresh control locations and clocks inserted to timed automata within a controller automaton, the length of the stack is also finite. Hence, an ordered controller automaton can be translated to a timed automaton.

Definition 6 (Ordered Controller Automata [24]). For a controller automaton $C \in \mathcal{C}$, if there is a strict partial order (irreflexive, asymmetric and transitive) on the set of states, $(S(C), \prec)$, that satisfies the followings:

- $s_1 \prec s_2$, if $s_1 \xrightarrow{a} s_2 \in \delta_{push}$.
- $s_1 \succ s_2$, if $s \xrightarrow{a} s_2 \in \delta_{pop}$.
- $s_1$ and $s_2$ are incomparable, if $s_1 \xrightarrow{a} s_2 \in \delta_{int}$.

then $C$ is an ordered controller automaton.

Fact 2. Given an ordered controller automaton, there exists an upper bound on the length of its stack.

Actually, when applying controller automata to a real application, e.g., an interrupt system. A controller automaton is usually adopted to model a structured system in which all components run sequentially with a given strategy. The usage of the controller automaton is to give a global view of the execution of these components. The given strategy is usually able to be modeled as a partial order, e.g. priority of various interrupts, to avoid possible deadlocks and livelocks of the system.

Remark 1. Ordered controller automata prohibit self-preempting loop transitions which is uncommon in real applications. Almost all real-time systems with mutex components can be modeled as ordered controller automata. That is, ordered controller automata are enough for our purpose of representations.

III. TRANSLATION FROM ORDERED CONTROLLER AUTOMATA TO TIMED AUTOMATA

A. The Descriptive Algorithm

We present a translation from an ordered controller automaton to a timed automaton, satisfying soundness and completeness with respect to set of symbols and the time to release them.

Given an ordered controller automaton $C = (E, H, \Sigma, s_0, M, x_{run}, T, \delta)$, where there is a strict partial order on the set of states, $(S, \prec)$, and for each $s_i \in S$, $M(s_i) = (E, H_i, Q_i, q_i^0, X_i, I_i, \delta_i)$, a timed automaton $\mathcal{A}^C = (E^*, H^A \times \delta^A, Q^A, q_0^A, X^A, I^A, \delta^A)$ is constructed by:

- $E^A = E$.
- $H^A = \bigcup_{i=1}^{n} H_i \cup H$.
- $Q^A \subset (\bigcup_{i=1}^{n} Q_i) \times (\bigcup_{i=1}^{n} Q_i)^*$, where for $(q, \tilde{q}) \in Q^A$, $\tilde{q}$ is an ascending chain. That is, for every two elements $q_1, q_2$ in $\tilde{q}$ such that $q_1 \prec q_2$, then $q_1, q_2 \in Q$.
- $q_0^A = (q_0(M(s_0)), \varepsilon)$, where $\varepsilon$ is the empty sequence.
- $X^A = \bigcup_{i=1}^{n} X_i \cup \{x_{\text{run}}\}$.
- For each $(q_i, \tilde{q}_i) \in Q^A$, where $q_i \in Q(M(s_i))$, $I^A(q_i, \tilde{q}_i) = I(q_i) \cup \{x_{\text{run}} \leq T(s_i)\}$.
- $\delta^A$ is defined as follows,
Definition 7 (Subbisimilarity). Let $R$ be a relation from the set of configurations of controller automata to that of timed automata. It is a subbisimilarity if the following properties hold:

1) If $dR^{-1}c \xrightarrow{\sigma} c'$ for some $\sigma \in E \cup H \cup \mathbb{R}^+$, then there exists some $d'$ such that $d \xrightarrow{\sigma} d'R^{-1}c'$;

2) If $cRd \xrightarrow{\sigma} d'$ for some $\sigma \in E \cup H \cup \mathbb{R}^+$, then there exists some $c'$ such that $c \xrightarrow{\sigma} c'Rd'$.

A controller automaton is subbisimilar to a timed automaton iff there is a subbisimilarity containing the pair of their initial configurations.

Theorem 1 (Operational Correspondence). Any ordered controller automaton $C$ is subbisimilar to its corresponding timed automaton $A^C$.

B. Implementing the Algorithm

The intuitive meaning of the algorithm above is to enumerate all legitimate locations $(q, \hat{q}) \in \bigcup_{i=1}^{n} Q_i \times \bigcup_{i=1}^{n} \{q_0(M(s))\}$ and then examine each pair of locations against the four criterions to determine whether there exists a transition between them and what type of transition it is.

This translation strategy may result in isolated locations that are not reachable from the initial location and should be removed. In our implementation, we follow the basic idea of this translation. However, instead of enumerating all locations, we construct a minimal set of candidate locations through pre-pruning some isolated locations and eliminate remaining redundant locations through post-pruning.

Isolated Locations In a timed automaton resulting from the translation of an ordered controller automaton, an isolated location is one that is not reachable from the initial location $q_0(M(s_0))$, following the directed edges, considering the timed automaton as a directed graph without labels. An isolated location is redundant and should be removed from the timed automaton.

The pre-pruning exploits the following observation: For any location $q \in Q(M(s))$ and ascending chain $\hat{q}q'$ where $q' \in Q(M(s))$, if $s_i \nless s_j$, location $(q, \hat{q}q')$ is an isolated location. In our implementation of translation, we construct a set of locations that cannot be determined as isolated according to the observation above. We then add transitions between these locations according to the intra-, push-, pop- and inter-relations in the controller automaton.

Finally we prune the timed automaton to remove all remaining isolated locations. Considering the timed automaton without labels as a directed graph, we perform a depth-first traversal on this graph and mark all visited nodes (locations). Those locations that are not visited in the graph traversal are isolated from the initial location and are deleted.

C. Pseudo Code

Here we present the pseudo code of our translation algorithm (shown in Algorithm 1). Five procedures are also presented, to generate control locations, intra transitions, push transitions, pop transitions, internal transitions, respectively. We encapsulate the tricky part like calculate locations and transitions into procedures.

Function and operation definition:

- $L(s_i)$: Location set \{(q, \hat{q})|q \in Q(i)\}.
- $F(s_i)$: Location chain set \{\hat{q}|(q, \hat{q}) \in L(s_i)\}.
- $T(s_i)$: Transition set \{(q, \hat{q}) \to (q', \hat{q}')|q \in Q(i)\}.
- $find((q, \hat{q}), L(s_i))$: Find location $(q, \hat{q})$ in $L(s_i)$ and return it.

Algorithm 1: Translation from OCA to Timed Automata

input : $C = (E, H, S, s_0, M, x_{run}, T, \delta)$, where for each $s_i \in S, M(s_i) = (E, H(i), Q(i), q_0(i), X(i), I(i), \delta(i))$
output: $A^C = (E_A, H_A, Q_A, q_0^0, X_A, I_A, \delta_A)$

$E_A \leftarrow E$;
$H_A \leftarrow H$;
$q_0^0 \leftarrow (q_0(M(s_0)), \varepsilon)$;
$X_A \leftarrow \{x_{run}\}$;
for $s_i \in S$ do
  genLocations($s_i$);
  $Q_A \leftarrow Q_A \cup L(s_i)$;
  $H_A \leftarrow H_A \cup H(i)$;
  $X_A \leftarrow X_A \cup X(i)$;
end
for $s_i \in S$ do
  genIntraTransitions($s_i$);
  genPushTransitions($s_i$);
  genPopTransitions($s_i$);
  genInerTransitions($s_i$);
  $\delta_A \leftarrow \delta_A \cup T(s_i)$;
end

D. EMERALD Specifications

This very first version of EMERALD is capable of checking for reachability properties, in particular whether certain combinations of states or locations and clock constraints are reachable from an initial configuration.
The abstract syntax of EMERALD specifications is defined as in Figure 2. For $A[] \beta$ to be satisfied, all reachable states must satisfy the state formula $\beta$. And for $E<> \beta$ to be satisfied, there should exist some state that satisfies $\beta$.

Naturally, we have to adapt EMERALD specifications in order to be verified by Uppaal’s model checking engine. EMERALD uses a hierarchical namespace, denoted by the names of ordered controller automata, states, locations and clocks. For instance, a location named $q_{s}$ of the state $s$ in the controller automaton $\text{OCA}$ is expressed as $\text{OCA} . s . q$, while the running clock $x_{\text{run}}$ of $\text{OCA}$ is expressed as $\text{OCA} . x_{\text{run}}$. We map the clocks and locations of the ordered controller automaton to those of the produced timed automaton. A location $q \in Q(M(s))$ is mapped to the disjunction of all of its counterparts $(q, \hat{q})$ in the produced timed automaton. A clock $x \in X(M(s))$ is mapped to its counterpart $x_{s}$ in the timed automaton.

Procedure genLocations($s_{i}$)
/* This procedure will calculate location set $\{(q, \hat{q})|q \in Q^{(i)}\}$. */
if $s_{i}$ is visited then return;
for $q_{j}^{(i)} \in Q^{(i)}$ do
    $I((q_{j}^{(i)}, \varepsilon)) \leftarrow I(q_{j}^{(i)}) \cup (x_{\text{run}} \leq T(s_{i}))$;
end
$L(s_{i}) \leftarrow L(s_{i}) \cup \{(q_{j}^{(i)}, \varepsilon)\}$;
end
$L(s_{i}) \leftarrow F(s_{i}) \cup \{\varepsilon\}$;
for $s_{j}$ that exists a Relation $s_{j} \xrightarrow{a} s_{i}$ do
    genLocations($s_{j}$);
end
for $l \in L(s_{j})$ do
    for $q_{j}^{(i)} \in Q^{(i)}$ do
        $I((q_{j}^{(i)}, l)) \leftarrow I(q_{j}^{(i)}) \cup (x_{\text{run}} \leq T(s_{i}))$;
        $L(s_{i}) \leftarrow L(s_{i}) \cup \{(q_{j}^{(i)}, l)\}$;
    end
end
$L(s_{i}) \leftarrow L(s_{i}) \cup \{l\}$;
end
set $s_{i}$ as visited;

Procedure genInterTransitions($s_{i}$)
/* This procedure will calculate inter transitions $(q, \hat{q}) \rightarrow (q', \hat{q}')$ where $(q, \hat{q}) \in L(s_{i})$. */
for $q \xrightarrow{a, \phi, \lambda} q'$ do
    for $l_{f} \in F(s_{i})$ do
        $l \leftarrow \text{find}((q, l_{f}), L(s_{i}))$;
        $l' \leftarrow \text{find}((q', l_{f}), L(s_{i}))$;
        $T(s_{i}) \leftarrow T(s_{i}) \cup \{l \xrightarrow{a, \phi, \lambda} l'\}$;
    end
end

Procedure genPushTransitions($s_{i}$)
/* This procedure will calculate intra transitions $(q, \hat{q}) \rightarrow (q', \hat{q}')$ where $(q, \hat{q}) \in L(s_{i})$. */
for $s_{j}$ that exists a Relation $s_{i} \xrightarrow{a} s_{j}$ do
    for $l \in L(s_{j})$ do
        $l' \leftarrow \text{find}((q_{j}^{(i)}, l), L(s_{j}))$;
        $T(s_{i}) \leftarrow T(s_{i}) \cup \{l \xrightarrow{a, \phi, \lambda} l'\}$;
    end
end
E. The Graphical Editor

EMERALD is motivated by the concept of ordered controller automata. It provides a graphical user interface for editing ordered controller automata and specifying properties of the model to be verified. EMERALD translates the ordered controller automata into timed automata, which UPPAAL model checking engine will accept and verify.

The graphical user interface is developed as an Eclipse application using the Standard Widget Toolkit (SWT). A system is modeled as a parallel composition of ordered controller automata as outlined in the tree-style view. (In EMERALD, a timed automaton parallel with a controller automaton is treated as a controller automaton with exactly one state wherein the timed automaton resides.) The main part of the editor is a drawing canvas, where the model of the system can be defined graphically. In a controller automaton, each state is labeled with a name and expected running time. A timed automaton can be directly edited within each state of the controller automaton. Relations between states are labeled with synchronization symbols. Different relation types (viz., push, pop and internal) are represented by different line styles. In a timed automaton, locations are labeled with names and invariants, and transitions between locations are labeled with guards, clock resets and synchronization symbols. All of these labels can be edited in the Properties View. When saved, the ordered controller automata defined in the editor are transformed to persistent storage as an XML file, which allows restoring the model to its graphical form.

Figure 3 gives a flavor of the EMERALD file, which allows restoring the model to its graphical form.

IV. CASE STUDIES OF EMERALD: THE ROBOT PUPPY

We give an example to illustrate how to use EMERALD to model and verify real-time systems. The following example models the behavior of a robot puppy featuring nested interrupts.

A. Description

The robot puppy has two functions, turning around and moving forward. If one pats the puppy's body, it will turn around; after 25 time units, if no one touch the puppy, it will stop. When the puppy is turning around, and patted again, the puppy will move forward; after 30 time units, it will stop. At any time, when it is doubly patted, the puppy will stop. An interrupt handler is used to handle the interrupt signal from its skin sensor; another interrupt handler is assumed to handle the interrupt with a higher priority, triggered by low battery power.

B. Modeling in EMERALD

The model of the robot puppy is a parallel composition of four ordered controller automata, including three one-state controller automata that represent three timed automata respectively.

1) Signal Automaton: The timed automaton (one-state controller automaton) in Figure 4 describes the time and the frequency of occurrences of interrupt signals. The signal "pat" occurs once every 10 time units; after 30 time units, the signal "turn" for low battery power warning is triggered.

2) Interrupt Handlers Automaton: The ordered controller automaton in Figure 5 is used to represent nested interrupts, allocating different interrupt handlers. An interrupt handler is represented as a timed automaton in a state of the controller automaton. The controller automaton has three states. The right-top state Beginning is the initial state, representing the situation when no interrupts are invoked. NormalHandler, the timed automaton in the left, presents the interrupt handler to handle interrupt signals from the sensor. The right-bottom one, LowPowerHandler, is for the interrupt handler triggered by low battery power.

3) TaskScheduler Automaton: The task scheduler, represented as the timed automaton in Figure 6, shows the way to schedule tasks when they are released. The robot puppy has two tasks, P(3,7) and Q(2,5), for the functionality of turning around and moving forward respectively. The two parameters of a task type are its execution time and relative deadline respectively. FPS (fixed priority scheduling) is enforced as the robot puppy's task scheduling strategy.

The timed automaton has five control locations. Idle means that no task instance is being executed. Check indicates that an instance of the task P(3,7), which is released and possibly being executed, is being analyzed for schedulability. This location is entered when a task instance of P(3,7) is non-deterministically chosen for analyzing. RunP and RunQ means that a task instance of P(3,7) or Q(2,5) is being executed. The task being executed is not the analyzed one. Error tells that the analyzed task queue is not schedulable, which happens when the analyzed task instance meets its deadline before completion of execution, or the whole released tasks cannot be scheduled.

4) System Automaton: Some tasks in a general system may be executed independently from interrupts. The timed automaton in Figure 7 represents a system with tasks.

C. Verification

We check a few reachability and safety properties, including the absence of deadlock.
The following reachability properties check whether a given combination of locations is reachable.

- $E<> \text{System.System.P and TaskScheduler.TaskScheduler.RunQ}$:
  It is possible that a task instance of P is being executed in System, while the TaskScheduler is running another task instance in RunQ.

The following safety properties must hold for all reachable states.

- $A[] not \text{deadlock}$: The execution of robot puppy is deadlock-free.
- $A[] not \text{TaskScheduler.TaskScheduler.Error}$: The scheduler will never reach the Error location. This means that the tasks are always schedulable by the FPS scheduling strategy.
- $A[] \text{InterruptHandlers.LowPowerHandler.L0 or InterruptHandlers.LowPowerHandler.L1 imply InterruptHandlers.x_run} \leq 50$:
  Execution will leave the interrupt handler for low battery power before it reaches the interrupt handler’s expected running time.
- $A[] \text{TaskScheduler.TaskScheduler.Check imply TaskScheduler.TaskScheduler.d} \leq 7$:
  Whenever the scheduler is in the Check location,
Controller automata were first proposed in [25], as a formal model to represent and analyze real-time systems with nested interrupts. An analysis technique named environmental simulation adopted controller automata to treat interrupts. Together with an interrupt environment modeled as a timed automaton, and a scheduler as a timed automaton with semaphores [20], the system behaviors with nested interrupts were realized by a sequence of transitions with time. It was shown that the reachability to error states is practically solvable with the implementation of the environmental simulation by Maude [10]. The environmental simulation in [25] is not a verification technique, since it cannot guarantee termination generally.

Task automata [14], [13], [15], extended from timed automata, were a specific model for schedulability analysis. It assumed that time was dense, and tasks could come at any time, periodically or sporadically. There were many papers based on task automata. For example, when considering the fixed priority scheduling, two extra clocks were enough to represent a scheduler timed automaton for checking schedulability [16]. A tool, TIMES [3] was developed for schedulability analysis of tasks. In this approach, although dependence of tasks was considered, all tasks were represented atomically. It cannot describe complex interactive models, such that an object triggers another one during the running time, or an object may have multiple functions due to different environments, which can be represented by our controller automata.

BIP (Behavior, Interaction, Priority) system [17], [27] provided a language and a theory for incremental composition of heterogeneous components, ensuring correctness-by-construction for essential system properties, such as mutex, deadlock-freedom and progress. The construction of BIP system was composed of three layers. The lower layer described behavior. The intermediate layer included a set of connectors describing the interactions between transitions of the behavior. The upper layer was a set of priority rules describing scheduling policies for interactions. It also enabled formal verifications, providing formal tools such as D-Finder [5]. Controller automata have a similar viewpoint to model real-time systems in a component-based way. Behaviors are described by built-in timed automata, interactions are represented in synchronization and priorities are shown as mutexes in controller automata.

Abdeddaïm developed a methodology [1] for solving the

V. RELATED WORK

UPPAAL [21] was a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. It had been applied successfully in case studies ranging from communication protocols [11] to multimedia applications [19]. The tool was designed to verify systems that can be modeled as a parallel composition of timed automata, extended with integer variables, structured data types, user defined functions, and channel synchronization. EMERALD adopted the engine of the UPPAAL. Basically, it provided a graphical editor facility for analyzing component-based real-time systems.
scheduling problem based on timed automata. In her work, schedulers corresponded to paths in a timed automaton. The aim was to find an optimal scheduler, which corresponded to the shortest path. Furthermore, for the scheduling problem, there are many well-studied methods [9], [12], such as rate monotonic scheduling [26]. These researches arranged a scheduling policy under discrete time, and all tasks run at the same time periodically.

VI. CONCLUSION

Controller automata are a formal theory to model and analyze real systems with mutex components. Our work focuses on developing a tool that adopts a subclass of controller automata, named ordered controller automata, to describe behaviors and to verify real-time systems with preemption, resumption and competition relations among components, aiming to decrease model complexity and facilitate modeling and verification. This includes implementing and optimizing the translation from ordered controller automata to timed automata (the two have the same expressiveness), developing the graphics user interface for modeling and verification, and transforming the reachability specification of ordered controller automata into that of timed automata. Future work should consider investigating the semantics of ordered controller automata and developing graphical simulation capability for running a controller automaton. Furthermore, we will develop our own model checking engine for controller automata, based on bounded model checking, adopting the SAT solver.

ACKNOWLEDGEMENT

This research is supported by the National Nature Science Foundation of China (61100052, 61011140074, 61033002).

REFERENCES


