

Environmental Simulation of Real-Time Systems with Nested Interrupts

Guoqiang Li Shoji Yuen
NCES, Graduate School of Information Science, Nagoya University
li.g@nces.is.nagoya-u.ac.jp
yuen@is.nagoya-u.ac.jp

Masakazu Adachi
Toyota Central R&D Labs. INC.
adachi@mosk.tytlabs.co.jp

Abstract

Interrupts are important aspects of real-time embedded systems to handle events in time. When there exist nested interrupts in a real-time system, and an urgent interrupt is allowed to preempt the current interrupt handling, the design and analysis of the system become difficult due to the lack of appropriate behavioral models. This paper proposes a compositional model for nested interrupts and an analysis named environmental simulation. We present a new kind of timed transition system, named controller automata, to treat interrupts. Together with an interrupt environment modeled as a timed automaton, and a scheduler as a timed automaton with semaphores, the system behaviors with nested interrupts are realized by a sequence of transitions with time. Although various verification problems for this model are undecidable in general, it is shown that the reachability to error states is practically solvable with our implementation of the environmental simulation by Maude.

1. Introduction

Real-time systems, due to their requirements to complete their works and deliver their services on a timely basis, are easily caused pitfalls when they are not properly designed. In order to guarantee correctness of properties that real-time systems hold, lots of formal models [13], such as *timed automata* [1, 8] and a variation, *timed automata with semaphores* [10] have been proposed and widely used for verification of real-time systems.

Interrupts are one of the important aspects of real-time embedded system designs. An interrupt is an event triggered by a signal from hardware or software, indicating the need for attention, which makes the processor suspend the current running task, and begin an execution of an *interrupt*

handler. When there are more than one interrupt, and urgent interrupts are allowed to preempt the current executing interrupt, an *interrupt controller* is used to allocate different interrupts and to provide priorities for them.

This paper proposes a technique, named *environmental simulation*, to analyze real-time embedded systems with nested interrupts. Behaviors of interrupts are characterized by the communication between a target system and an environment. The target system consists of interrupt handlers and a controller to invoke the handlers according to the urgency of the interrupts. In order to model the behavior of the real-time system with nested interrupts, we propose *controller automata* [12], extended from timed automata, which integrates interrupt behaviors with the non-interrupt behaviors of the system. A controller automaton overrides the (non-interrupt) system behavior by interrupts, and resumes the behavior after the interrupt handling is over. Environmental simulation provides an interrupt generator to check whether the whole system works as intended under the certain interrupt environment.

We implement the environmental simulation by Maude [3]. Maude represents model generation rules by rewriting, instead of describing a model directly. A property is thus analyzed at the same time when a model is generated. Hence even many properties, e.g., reachability problem, on controller automata are undecidable in general, it can uncover subtle counterexamples of a specification in an early step.

Related Work In [12], a *strict partial order* was imposed over the state set of a controller automaton. With this restriction, an *ordered controller automaton* could be translated to a timed automaton. Hence schedulability analysis was performed by the reachability problem of timed automata, which was solved by translating a timed automaton to a *region automaton* [1]. It was already implemented by several tools, e.g., UPPAAL [11]. Hence, we drew a conclu-

sion that the reachability problem of ordered controller automata is decidable, while the reachability problem of controller automata is in general undecidable.

As an approach to handle time, *Real-Time Maude* [14] was proposed, which is a tool for the high-level formal specification, simulation and analysis of real-time and hybrid systems. Based on Real-Time Maude, the CASH scheduling algorithm were formally described and analyzed [15]. We independently implement the time issues with near 2000 lines by Maude, with the aim of features for controller automata, timed automata, timed automata with semaphores, and parallel compositions among them. These features cannot be directly implemented by Real-Time Maude. Our implementation can be used to describe, simulate and analyze real-time systems with nested interrupts, as well as other real-time systems modeled by these automata.

Task automata [5, 4, 6] were a specific model for schedulability analysis. It assumed that time was dense, and tasks can come at any time, periodically or sporadically. When considering the fixed priority scheduling, two extra clocks were enough to represent a scheduler timed automaton for checking schedulability [7]. A tool, TIMES [2] was developed for schedulability analysis of tasks. In this approach, although dependence of tasks was considered, all tasks were represented atomically. It could not describe complex interactive models, such as 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 controller automata.

Paper Organizations The rest of the paper is organized as follows. Section 2 proposes the formal definition of controller automata. Section 3 introduces the environmental simulation. Section 4 shows the experimental results by Maude. Section 5 concludes the paper.

2. Controller Automata

This section presents *controller automata* to model nested interrupts. We assign a timed automaton to the control location where an interrupt handler is invoked.

2.1. Timed Automata

This subsection briefly reviews *timed automata* [1, 8].

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

$$\phi ::= \top \mid x \bowtie c \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

where $c \in \mathbb{R}^+$, $x \in X$, and $\bowtie \in \{<, >, \leq, \geq\}$.

For the set of clocks X , a *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}^+$, which assigns a value to each clock $x \in X$.

For a clock valuation ν and a clock constraint ϕ , we write $\nu \models \phi$ to denote that ν satisfies the constraint ϕ . Given a set of clocks $\lambda \subseteq X$ and a clock valuation ν , 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 ν and a time $t \in \mathbb{R}^+$, we define $(\nu + t)(x) = \nu(x) + t$, for $x \in X$.

Timed automata have been first proposed in [1]. The original timed automata use Büchi accepting conditions to enforce progress properties [1]. Later *timed safety automata* are introduced in [8], to specify progress properties using local invariant conditions. In this paper, we shall focus on timed safety automata, referring them as timed automata, when it is understood from the context. In order to define the parallel compositions, we distinguish actions of timed automata by internal actions and external actions [9].

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

- E is a finite set of external actions, and H is a finite set of internal actions.
- Q is a finite set of control locations.
- $q_0 \in Q$ is the initial location.
- X is a finite set of clocks.
- $I : Q \rightarrow \Phi(X)$ 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 $q_1 \xrightarrow{a, \phi, \lambda} q_2$. If we let $\Sigma = E \cup H$, then the definition above is exactly as same as the definition in [8].

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 sets of external actions, internal actions and control locations, initial location, and set of clocks, respectively. We also use the same notations for other kinds of automata.

In [11], *semaphores*, shared variables with the type of *clock*, a bounded domain and an initial value are inserted to timed automata. Predicates over semaphores are also used as guards on transitions. On each transition, semaphores can be assigned to any value within their respective domains. Before defining timed automata with semaphores, we generalize time constraints in Definition 1 by inserting a new element $x \bowtie s$, where s ranges over a set of semaphores.

Definition 3 (Timed automata with semaphores). A timed automaton with semaphores (SA) is a tuple $\mathcal{S} = (E, H, Q, q_0, X, S, I, M, \delta)$, where

- S is a finite set of semaphores.

- $M : Q \rightarrow \Phi(S)$ is a function assigning each location with a semaphore predicate.
- $\delta \subseteq Q \times (E \cup H) \times \Phi(X) \times 2^X \times \Phi(S) \times 2^{\nu(S)} \times Q$.

Other elements are the same as those in Definition 2.

Timed automata with semaphores do not enrich the expressiveness of timed automata. A timed automaton with semaphores just compacts bounded number of control locations and transitions in a timed automaton [11].

Timed automata are composed by a *parallel composition* (PA) [10] over a common set of external actions, which also does not enrich the expressiveness of timed automata [10].

Fact 1. *A parallel composition of two timed automata is a timed automaton.*

2.2. Time Lag on Timed Automata

When a timed automaton is preempted by another one, the system will stop running current timed automaton, store the current status, and begin to run the latter timed automaton. A *time lag* transforms a timed automaton to wait a certain time when preempted by another timed automata.

A time lag occurs at a given control location q in a timed automaton \mathcal{A} . We need an extra idle location q^{id} for q . The definition of $\text{TimeLag} : \mathcal{A} \times Q(\mathcal{A}) \times \mathbb{R}^+ \rightarrow \mathcal{A}$ accepts a timed automaton, a control location on this timed automaton, a time interval, and returns a timed automaton, with the following definition, $\text{TimeLag}(\mathcal{A}, q, t) = (E, H \cup \{r, l\}, Q \cup \{q^{id}\}, q_0, X \cup \{x_p\}, I', \delta')$, where

- $I'(q) = I(q) \wedge \{x_p \leq 0 \vee x_p \geq t\}$, $I'(q^{id}) = \{x_p \leq t\}$, and $I'(q') = I(q')$ for all $q' \neq q$.
- Define $\delta'' = \{q' \xrightarrow{a, \phi, \lambda \cup \{x_p\}} q \mid q' \in Q, q' \xrightarrow{a, \phi, \lambda} q \in \delta\} \cup \{q' \xrightarrow{a, \phi, \lambda} q'' \mid q', q'' \in Q, q' \xrightarrow{a, \phi, \lambda} q'' \in \delta \wedge q'' \neq q\}$, and $\delta' = \delta'' \cup \{q \xrightarrow{r, T, \emptyset} q^{id}, q^{id} \xrightarrow{l, x_p \geq t, \emptyset} q\}$.

When the location q transits to the idle location q^{id} , it may not return back to q after t time units, due to the violation of $I'(q)$. It also cannot stay in q^{id} , since $I'(q^{id})$ is also violated after t time. Thus an empty timed automaton is produced, which explains that when some unexpected break happens, a system aborts the execution after restored.

2.3. Formal Definition

To avoid conflicts with terminology of timed automata, we name control locations of controller automata *states*.

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

- E is a finite set of external actions, and H is a finite set of internal actions.

- S is a finite set of states, and $s_0 \in S$ is the initial state.
- X is a finite set of clocks.
- $I : S \rightarrow \Phi(X)$ is a function assigning each state with a clock constraint.
- $M : S \rightarrow \mathcal{A}$ is a function assigning each state with a timed automaton.
- $\delta \subseteq S \times (E \cup H) \times \Phi(X) \times 2^X \times S$.
- $T : S \rightarrow \mathbb{R}^+$ is a function assigning each state with a time, named *expected running time*.
- $x_{run} \in X$ is the special clock to accumulate the running time of each state.

Note that a controller automaton shares external actions with timed automata assigned to its states. For each $M(s)$, $X(M(s)) \cap X = \emptyset$. δ is partitioned into three disjoint subsets, δ_{push} , δ_{pop} , δ_{int} , for push, pop and internal actions, respectively, which are disjoint.

We prepare a stack for configurations of a controller automaton, recording the state and the running control location when a push event performed, and restoring the running context when a pop event performed.

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

- s is the running state;
- q is the current running location in $M(s)$;
- ν is the clock valuation for all clocks of the controller automaton and timed automata in the states;
- κ is the set of clocks keeping frozen when the time elapses, named *frozen clocks*;
- \mathbf{S} is the stack.
- Progress transitions:
 - $(s, q, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s, q, (\nu + t)[\kappa], \kappa, \mathbf{S})$, if $(\nu + t)[\kappa] \models I(s)$, and $(q, \mu) \rightarrow_{\mathcal{A}} (q, \mu + t)$, where $\mu (\subseteq \nu)$ is a clock valuation on $X(M(s))$.
- Discrete transitions:
 - Intra-action: $(s, q, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s, q', \nu[\lambda], \kappa, \mathbf{S})$, if $(q, \mu) \rightarrow_{\mathcal{A}} (q', \mu[\lambda])$ in $M(s)$, where $\mu (\subseteq \nu)$ is a clock valuation on $X(M(s))$.
 - Push: $(s, q, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s', q_0(M(s')), \nu[\lambda \cup \{x_{run}\}], \kappa \setminus X(M(s')), (s, q) :: \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, \nu(x_{run}))$ for all (s, q) in \mathbf{S} , if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{push}$, $\nu \models \phi$, and $\nu[\lambda] \models I(s')$.
 - Pop: $(s, q, \nu, \kappa, (s', q') :: \mathbf{S}) \rightarrow_{\mathcal{C}} (s', q', \nu[\lambda \cup \{x_{run}\}], \kappa \cup X(M(s)), \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, T(s))$ for all (s, q) in $(s', q') :: \mathbf{S}$, if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{pop}$, $x_{run} = T(s)$, $\nu \models \phi$, and $\nu[\lambda \cup \{x_{run}\}] \models I(s')$.

- Inter-action: $(s, q, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s', q_0(M(s')), \nu[\lambda \cup \{x_{run}\}], \kappa \setminus X(M(s')) \cup X(M(s)), \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, T(s))$ for all (s, q) in the stack, if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{int}$, $x_{run} = T(s)$, $\nu \models \phi$, and $\nu[\lambda \cup \{x_{run}\}] \models I(s')$.

Each transition has the intuitive meanings as follows,

- Progress transitions mean that if the control location of the timed automaton in the state can run t time, and after t time elapsed, the invariant of the state is not violated, then the state can run t time.
- Intra-action transitions mean that if one control location transits to another one in the state, the the controller automaton can also perform such a transition.
- Push transitions mean that if time conditions are satisfied (constraints on the transition and invariant of the state), the system pushes the running state and control location into the stack, and transits to the initial location of the time automaton in the latter state. Simultaneously, all timed automata in the states pushed in the stack have a time lag, with the time recorded by x_{run} .
- Pop transitions mean that if the latter state is on the top of the stack, and time conditions are satisfied, then after the execution of the expected running time, the system pops the previous state and control location from the stack, and begins to run the timed automaton from the popped control location. Simultaneously, all timed automata in the states within the stack have a time lag, with the time recorded by $T(s)$.
- Inter-action transitions mean that if time conditions are satisfied, then after the execution of the expected running time, the system transits to the initial location of the time automaton in the latter state. Simultaneously, all timed automata in the states within the stack have a time lag, with the time recorded by $T(s)$.

Generally, if an interrupt has lower priority than another one, then the transition from the former to the latter belongs to δ_{push} , and from the latter to the former belongs to δ_{pop} . When two interrupt handlers have the same priority, the transition between them belongs to δ_{int} .

Consider a timed automaton \mathcal{A} and a controller automaton \mathcal{C} , where for each $s_i \in S(\mathcal{C})$, $M(s_i) = (E_i, H_i, Q_i, q_i^0, X_i, I_i, \delta_i)$. Assume that \mathcal{A} shares common sets of external actions, and clocks with \mathcal{C} and all $M(s_i)$. A parallel composition of \mathcal{A} and \mathcal{C} is denoted as $\mathcal{A} \parallel \mathcal{C}$.

A location pair is a pair $\tilde{q} = (q_a, q_i)$, where $q_a \in Q(\mathcal{A})$ and $q_i \in \bigcup Q(M(s_i))$. The invariant function over location pairs is composed by $I((q_a, q_i)) = I(q_a) \wedge I_i(q_i) \wedge I(s_i)$ where $q_i \in Q(M(s_i))$. Let $\tilde{q}[q'_i/q_i]$ denote the location pair where the i -th element q_i is replaced by q'_i . We distinguish external actions E as two disjoint sets $E = E_o \cup E_i$,

where E_o is the set of *triggering actions*, ranged over by $a!$, $b!$, and E is *triggered actions*, ranged over by $a?$, $b?$.

Definition 6. A configuration of parallel composition between a timed automaton and a controller automaton is a tuple $(s, \tilde{q}, \nu, \kappa, \mathbf{S})$, where s is a state of the controller \tilde{q} is a location pair, $\tilde{q} \in Q(\mathcal{A}) \times \bigcup Q_i(M(s_i))$, ν is a clock valuation, κ is a set of frozen clocks, and \mathbf{S} is a stack.

- Progress transition:
 - $(s, \tilde{q}, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s, \tilde{q}, (\nu + t)[\kappa], \kappa, \mathbf{S})$, where $t \in \mathbb{R}^+$ and $(\nu + t)[\kappa] \models I(\tilde{q})$.
- Discrete transition:
 - $(s, \tilde{q}, \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s, \tilde{q}[q'_i/q_i], \nu[\lambda], \kappa, \mathbf{S})$, if $q_i \xrightarrow{a, \phi, \lambda} q'_i$, $\nu \models \phi$ where $a \in H_i$ and $\nu[\lambda] \models I(\tilde{q}[q'_i/q_i])$.
 - $(s, \tilde{q}, \nu, \mathbf{S}) \rightarrow_{\mathcal{C}} (s, \tilde{q}[q'_i/q_i, q'_j/q_j], \nu[\lambda_1 \cup \lambda_2], \mathbf{S})$, if $q_i \xrightarrow{a?, \phi_1, \lambda_1} q'_i$, $q_j \xrightarrow{a!, \phi_2, \lambda_2} q'_j$ where $a! \in E_o$, $a? \in E_i$, $\nu \models \phi_1 \wedge \phi_2$, and $\nu[\lambda_1 \cup \lambda_2] \models I(\tilde{q}[q'_i/q_i, q'_j/q_j])$.
 - $(s, (q_a, q'), \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s', (q_a, q_0(M(s'))), \nu[\lambda \cup \{x_{run}\}], \kappa \setminus X(M(s')), (s, q) :: \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, \nu(x_{run}))$ for all (s, q) in \mathbf{S} , if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{push}$ and $a \in H(\mathcal{C})$, $\nu \models \phi$, and $\nu[\lambda] \models I(s')$.
 - $(s, (q_a, q'), \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s', (q'_a, q_0(M(s'))), \nu[\lambda \cup \lambda' \cup \{x_{run}\}], \kappa \setminus X(M(s')), (s, q) :: \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, \nu(x_{run}))$ for all (s, q) in \mathbf{S} , if $s \xrightarrow{a?, \phi, \lambda} s' \in \delta_{push}$, $q_a \xrightarrow{a!, \phi', \lambda'} q'_a$ where $a! \in E_o$, $a? \in E_i$, $\nu \models \phi \wedge \phi'$, $\nu[\lambda \cup \lambda' \cup \{x_{run}\}] \models I(q'_a, q_0(M(s')))$ and $\nu[\lambda \cup \lambda' \cup \{x_{run}\}] \models I(s')$.
 - $(s, (q_a, q), \nu, \kappa, (s', q') :: \mathbf{S}) \rightarrow_{\mathcal{C}} (s', (q_a, q'), \nu[\lambda \cup \{x_{run}\}], \kappa \cup X(M(s)), \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, T(s))$ for all (s, q) in $(s', q') :: \mathbf{S}$, if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{pop}$ and $a \in H(\mathcal{C})$, $x_{run} = T(s)$, $\nu \models \phi$, and $\nu[\lambda \cup \{x_{run}\}] \models I(s')$.
 - $(s, (q_a, q), \nu, \kappa, \mathbf{S}) \rightarrow_{\mathcal{C}} (s', (q_a, q_0(M(s'))), \nu[\lambda \cup \{x_{run}\}], \kappa \setminus X(M(s')) \cup X(M(s)), \mathbf{S})$, and $M(s) := \text{TimeLag}(M(s), q, T(s))$ for all (s, q) in the stack, if $s \xrightarrow{a, \phi, \lambda} s' \in \delta_{int}$ and where $a \in H(\mathcal{C})$, $x_{run} = T(s)$, $\nu \models \phi$, and $\nu[\lambda \cup \{x_{run}\}] \models I(s')$.

The symmetric forms for parallel push, and parallel rules for pop and internal actions in discrete transitions are elided.

A parallel composition between a timed automaton \mathcal{A} and a controller automaton \mathcal{C} can be translated to parallel compositions of the \mathcal{A} and the timed automaton assigned to each state of \mathcal{C} . Hence it does not enrich the expressiveness of controller automata.

Fact 2. A parallel composition of a timed automaton and a controller automaton is a controller automaton.

A parallel composition between a timed automaton with semaphores and a controller automaton can also be defined.

3. Environmental Simulation

In existing formal models [5, 9], a real-time system is usually described as a closed model, without interacting with the environment, e.g., a system composed of a task system and a scheduler [5]. However, behaviors of interrupts may vary according to the time and frequency of occurrences of interrupt signals that can trigger these interrupt handlers. Hence a real-time system with interrupts should be analyzed under a context of its environment.

In our approach, nested interrupts are described as a composition of two parts: an *interrupt environment*, represented as a timed automaton, describing the time and the frequency of occurrences of interrupt signals, and an interrupt controller, represented as a controller automaton. A system usually needs a scheduler to arrange task instances, described as a timed automaton with semaphores [4, 6, 5].

Usually, priorities assigned to interrupts are integers. When two interrupts with the same priority are triggered simultaneously, an interrupt controller allocates them non-deterministically. It is appropriate to model the priorities as a *well-quasi-order*. Controller automata, in which priorities are not considered, are a superclass for the description of nested interrupts, whose reachability problem is in general undecidable. Ordered controller automata [12], in which priorities are modeled as a strict partial order, are a subclass, whose reachability problem is proven decidable [12].

In what follows, we model the behavior of a robot puppy as a running example. Suppose 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.

3.1. Interrupt Environments

Interrupt signals of the puppy come from the sensor of its skin, triggered by human's touch. A group of interrupt signals can be represented by a timed automaton \mathcal{A}^{sig} , with external actions as events that trigger the interrupt handler.

The timed automaton in Figure 1 describes the signal *pat* occurs once each 10 time units; after 30 time units, the signal *turn* for low battery power warning is triggered.

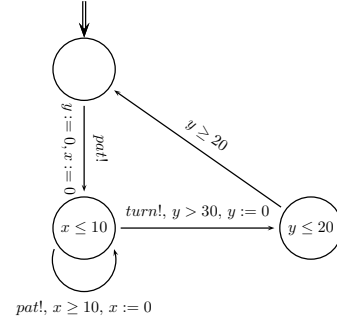


Figure 1. An Interrupt Signal Timed Automaton

An interrupt environment is an external assumption for the system to work correctly with respect to interrupts. This may be derived by the physical constraints, or by the experiences and restrictions of operations.

3.2. Nested Interrupts

A controller automaton 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, with external actions as awaited requirements of signals, or as events to trigger tasks.

A controller automaton to represent the system of robot puppy is shown in Figure 2, which has three states. The right-top state is the initial state, representing the situation when no interrupts are invoked. The timed automaton in the left one represents the interrupt handler to handle interrupt signals from the skin sensor. The right-bottom state is for the interrupt handler triggered by low battery power.

3.3. Schedulers

Assume there is a bounded set of task types \mathcal{P} , ranged over by $P(C, D)$, which has two parameters C and D , with $C \leq D$, where C is its *execution time*, and D is its *relative deadline*. Given a task type $P(C, D)$, we use $C(P)$ and $D(P)$ to represent its execution time and relative deadline, respectively. A task may have several instances released in a system. A task queue is a list of task instances awaited to be executed by the processor. A *scheduling strategy*, e.g. FPS (fixed priority scheduling), is a sorting function on the task queue, which inserts a new task instance into the queue, according to the task parameters.

The length of a schedulable task queue is bounded, since the number of instances of each task type $P_i(C_i, D_i) \in \mathcal{P}$ in a schedulable queue is bounded by $\lceil D_i/C_i \rceil$, and thus the length of a schedulable queue is bounded

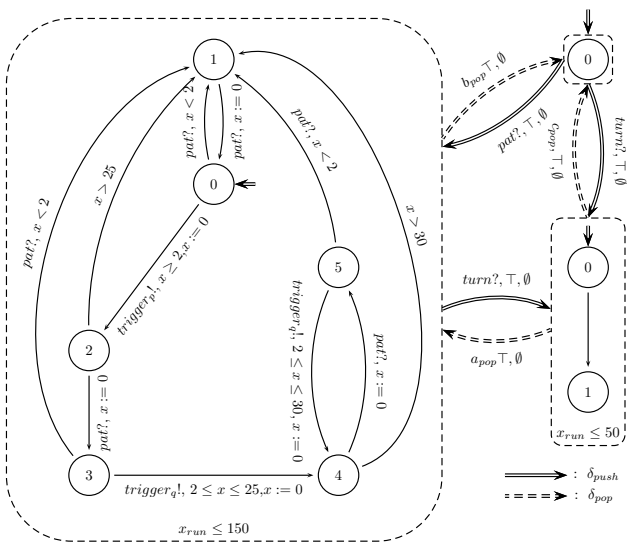


Figure 2. The Controller Automaton of The Robot Puppy

by $\sum_{P_i(C_i, D_i) \in \mathcal{P}} [D_i/C_i]$. With this result, a scheduler can be represented by a timed automaton with semaphores [4, 6, 5]. Here we only show a scheduler of our running example. Assume that our puppy robot has two tasks, $P(3, 7)$ and $Q(2, 5)$, for the functionalities of turning around and moving forward respectively, with the priority that $Q(2, 5) \succ P(3, 7)$. A timed automaton \mathcal{A}_p^{sch} for the task $P(3, 7)$ is constructed in Figure 3, following the approach in [6].

Clocks and semaphores of \mathcal{A}_p^{sch} :

- c is a clock to measure time when a task instance of P is executed (no other task instances with higher priorities is released).
- d is a clock to measure time when a task instance of P is released.
- r is a semaphore to calculate the time needed to complete all released tasks.

Constants of \mathcal{A}_p^{sch} :

- τ is the minimal execution time of a task instance. It can be any value greater than 0. Usually, it is assigned to the maximal value of all execution time of tasks in the system $\tau = \text{Max}_{P_i(C_i, D_i) \in \mathcal{P}} (C(P_i))$.
- R_p is the value of P that leads the task queue unschedulable. The maximum value of the executable task queue, \mathbf{r}^{max} is calculated by $\sum_{P_i(C_i, D_i) \in \mathcal{P}} ([D_i/C_i] \times C_i)$. Note that $r - c$ is the time remaining until all released tasks are executed. Thus $r - c < \mathbf{r}^{max} + D(i)$ where $D(i)$ is one

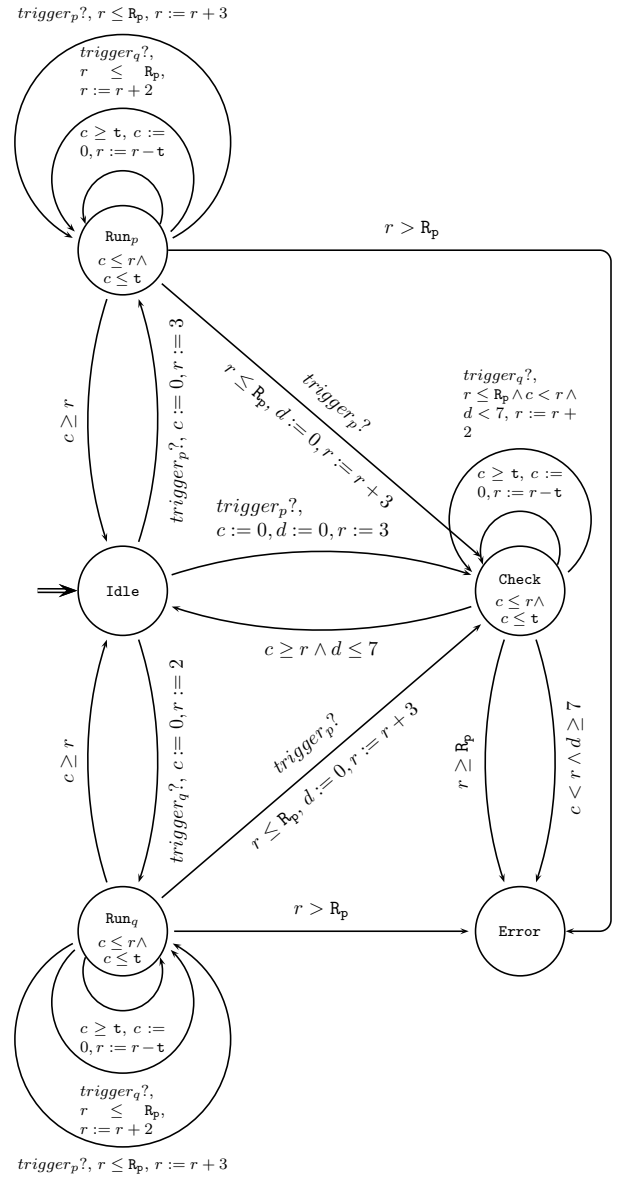


Figure 3. The Scheduler of the Task $P(3, 7)$

of the relative deadline. We also have $c \leq \tau$, Hence $R_p = \mathbf{r}^{max} + D(P) + \tau$.

General explanations for control locations of the \mathcal{A}_p^{sch} :

- **Idle** means that no task instances are being executed. It is reentered when the released tasks has been executed (by the guard $c \geq r$).
- **Check**: An instance of the task $P(3, 7)$, which is released and possibly executed, is being analyzed for schedulability. It is entered when a task instance of $P(3, 7)$ is non-deterministically chosen for analyzing. Two self-loops are in the Check: one represents the

execution of the current task instance; the other represents the arrival of the new task instance of $Q(2, 5)$.

- Run_p and Run_q : A task instance of $P(3, 7)$ or $Q(2, 5)$ is current executed, respectively. The task instance executed in the Run_p is not the analyzed one. When a task instance of $P(3, 7)$ is released, it can non-deterministically enter Check or Run_p . There are three self-loops in Run_p and Run_q , respectively, to represent the execution of the task instance, the arrival of the new task instances of two task types.
- **ERROR**: The analyzed task queue is not schedulable. It is entered when the analyzed task instance met its deadline ($d \geq 7$) before completion of execution ($c \leq r$), or the whole released tasks cannot be scheduled ($r > R_p$).

3.4. System Representations

The controller automaton \mathcal{C} shows conditions that task instances are released by interrupt handlers. A scheduler \mathcal{A}^{sch} shows the way to schedule tasks when they are released. An timed automaton \mathcal{A}^{sig} represents an environment where the system executes. The puppy is thus represented by the parallel composition of the three automata,

$$\mathcal{A}^{sig} \parallel \mathcal{C} \parallel \mathcal{A}^{sch}$$

Note that in the example, all tasks are triggered by interrupts. However, some tasks in a general system may be executed independently from interrupts. A system with tasks can also be represented by a timed automaton, \mathcal{A}^{sys} [4]. A general representation of a real-time system with nested interrupts is as follows,

$$\mathcal{A}^{sig} \parallel \mathcal{C} \parallel \mathcal{A}^{sys} \parallel \mathcal{A}^{sch}$$

As a comparison, a real-time system without interrupts [4] is represented by

$$\mathcal{A}^{sys} \parallel \mathcal{A}^{sch}$$

The former representation, due to the occurrence of a controller automaton, is more expressive than the latter. The price to pay for the expressiveness is that reachability problem in general is undecidable. In [12], we show that with a strict partial order over the controller automaton, the controller automata can be translated to a timed automaton. Thus the reachability problem becomes decidable.

4. Experimental Results

We take *schedulability analysis*, one of the most important issues in developing real-time systems, as a case study to show how the environmental simulation works. The simulation is implemented by Maude [3], a language and system supporting both equational and rewriting logic computation. The basic units of Maude specifications are *modules*.

In Core Maude, there are two kinds of modules: *functional modules* and *system modules*. Elementary types, definitions and functions are defined in functional modules, and their respective semantics and parallel composition among three kinds of automata are implemented by system modules.

Maude provides many strategies to expand models, *rewrite* for simulating one possible path of the execution, *search* for exploring all possible execution paths from the initial term, and *frewrite* for fair rewriting, etc. Maude also provides LTL model checker on the transition system defined in system modules. Hence, a system can be generated following these strategies. Users can also define strategies flexibly in Maude, if needed.

The implementation details and their explanations are in the Appendix A. By our implementation, an error state of the running puppy example is found within 0.2 sec., after rewriting 457 states, which is shown in Figure 4. If we use `show search graph` command in maude, we can obtain the trace from the initial state to the error state.

In the Figure 4, EN1, SN1 and LN1 show the location number of the interrupt environment, the state number of the controller, and the location number of the current running handler, respectively. VAL11 and SVAL11 are the clock valuation and the semaphore valuation, respectively. FRE1 indicates the list of frozen clocks. STACK is the stack. CTRL1, ETRL1, STRL1, and TRL1 are the list possible next transitions of the controller, the interrupt environment, the scheduler and the current handler, respectively. CA shows the current controller automaton.

5. Conclusion

We presented an environmental simulation to analyze properties for real-time systems with nested interrupts by introducing controller automata. The behavior of interrupts was characterized by the communication between a target system and an environment. A virtue of our modeling was that since an interrupt behavior was combined as a separate component of a real-time embedded system, interrupts could be flexibly modeled as an embedded mechanism and an environment. The research was to investigate the analysis technique of controller automata. The experimental result showed that although the reachability analysis of a controller automaton was in general undecidable, it was reasonably feasible with the environmental simulation by Maude.

The future work will be: firstly, to perform the analysis on some complex and practical examples. Secondly, to find more efficient time rewriting rules that can handle dense time. Thirdly, to combine the simulation methods to our previous model checking approach [12], developing a more powerful and useful tool for the analysis of real-time systems with nested interrupts.

```

Solution 1 (state 457)
states: 458 rewrites: 105853 in 292082197lms cpu (257ms real) (0
rewrites/second)
EN1 --> EN(1)
SN1 --> SN(1)
LN1 --> LN(12)
VALI1 --> Vclock(c(20), 3) Vclock(c(21), 3) Vclock(c(30), 0) Vclock(c(31), 3)
Vclock(c(10), 0) Vclock(c(88), 3)
SVALI1 --> Vclock(s(0), 3)
FREL --> (nil).List{Clock}
STACK --> SN(0),LN(0) # empty
CTRLI --> (nil).List{ConTransition}
ETRLI --> Tr(EN(1), a(10) !, c(20) ge 3, c(20), EN(1))
STRLI --> ExTr(CN(2), Null, c(30) ge 1, nil, top, nil, CN(0)) ExTr(CN(2), Null,
c(30) ge 3, c(30), top, minus(s(0), 3), CN(2)) ExTr(CN(2), a(11) ?, top,
nil, s(0) le 14, plus(s(0), 3), CN(2)) ExTr(CN(2), a(12) ?, top, nil, s(0)
le 14, plus(s(0), 2), CN(2)) ExTr(CN(2), a(12) ?, top, c(31), s(0) le 14,
plus(s(0), 2), CN(3)) ExTr(CN(2), Null, top, nil, s(0) gt 14, nil, ERR)
TRLI --> Tr(LN(12), a(10) ?, top, c(10), LN(13)) Tr(LN(12), Null, c(10) gt 25,
nil, LN(10))
CA -->
CA([SN(0),TA([LN(0),top], LN(0), nil, nil),top,3] [SN(1),TA([LN(10),top] [LN(
11),top] [LN(12),top] [LN(13),top] [LN(14),top] [LN(15),top], LN(10), c(
10), Tr(LN(10), a(10) ?, top, c(10), LN(11)) Tr(LN(11), a(10) ?, c(10) le
2, nil, LN(10)) Tr(LN(11), a(11) !, c(10) gt 2, c(10), LN(12)) Tr(LN(12),
a(10) ?, top, c(10), LN(13)) Tr(LN(12), Null, c(10) gt 25, nil, LN(10)) Tr(
LN(13), a(10) ?, c(10) le 2, nil, LN(10)) Tr(LN(13), a(12) !, c(10) gt 2
and c(10) le 25, c(10), LN(14)) Tr(LN(14), a(10) ?, top, c(10), LN(15)) Tr(
LN(14), Null, c(10) gt 30, nil, LN(10)) Tr(LN(15), a(12) !, c(10) gt 2 and
c(1) le 30, c(10), LN(14)) Tr(LN(15), a(10) ?, c(10) le 2, nil, LN(10)),c(
88) le 150,150] [SN(2),TA([LN(20),top] [LN(21),top], LN(20), nil, Tr(LN(
20), a(13) ?, top, nil, LN(21))),c(88) le 50,50],
SN(0),
c(88),
push(SN(0), Null, top, nil, SN(1)) push(SN(0), Null, top, nil, SN(2))
pop(SN(2), Null, top, nil, SN(0)),c(88))

```

Figure 4. The Snapshot of an Error State of the Running Example

References

- [1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [2] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Proceedings of the FORMATS'03*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer-Verlag, 2003.
- [3] M. Clavel, F. Durán, S. Eker, P. Lincolnand, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.4)*. <http://maude.cs.uiuc.edu/maude2-manual/>, 2008.
- [4] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of the RTCSA'99*, pages 182–189. IEEE Computer Society, 1999.
- [5] E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task Automata: Schedulability, Decidability and Undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
- [6] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability Analysis of Fixed-Priority Systems Using Timed Automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.
- [7] E. Fersman, P. Pettersson, and W. Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In *Proceedings of the TACAS'02*, volume 2280 of *Lecture Notes in Computer Science*, pages 67–82. Springer-Verlag, 2002.
- [8] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.
- [9] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan&Claypool, 2006.
- [10] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proceedings of the RTSS '95*, pages 76–87, 1995.
- [11] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
- [12] G. Li, S. Yuen, and M. Adachi. Schedulability Analysis of Real-Time Systems with Nested Interrupts (extended version). <http://www.agusa.i.is.nagoya-u.ac.jp/person/li.g/paper/InterruptFull.pdf>, 2008.
- [13] J. Mattai. *Real-Time Systems: Specification, Verification, and Analysis*. Prentice Hall, 1995.
- [14] P. C. Ölveczky. *Real-Time Maude 2.3 Manual*. <http://heim.ifi.uio.no/~peterol/RealTimeMaude/intro.pdf>, 2007.
- [15] P. C. Ölveczky and M. Caccamo. Formal Simulation and Analysis of the CASH Scheduling Algorithm in Real-Time Maude. In *Proceedings of the FASE'06*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372, 2006.

A. Implementing the Simulation by Maude

A.1. Basic Time Definitions

In Maude, time is defined as non-negative rational number. The functional module for time is defined by

```
fmod TIME is
  protecting RAT .
  sorts NNegRat Time .
  subsorts Zero PosRat < NNegRat < Rat.
  subsorts Nat < NNegRat < Rat.
  subsort NNegRat < Time .
endfm
```

The types and constructors of Clocks, valuations, and time constraints, and types of several important functions are defined in their respective functional modules.

```
sorts Clock Valuation .
  op c : Nat -> Clock [ctor] .
  op f : Nat -> Clock [ctor] .
  op Vclock : Clock Time ->
    Valuation [ctor] .

sort Constraint .
  op top : -> Constraint .
  op _le_ : Clock Time ->
    Constraint [ctor prec 37] .

  op sat : Valuation Constraint ->
    Bool .
  op satisfy : List{Valuation}
    Constraint -> Bool .
  op Accumulate : List{Valuation} Time
    -> List{Valuation} [memo] .
  op ResetClock : List{Valuation} Clock
    -> List{Valuation} .
  op Reset : List{Valuation} List{Clock}
    -> List{Valuation} [memo] .
```

Here, we only show the two constructor equations for the type `Constraint`. Other constructor equations, such as `lt`, `ge`, `gt`, `and`, `or`, and `not` are defined similarly.

The function `satisfy` judges that whether a list of valuations satisfies a constraint. It is implemented as follows. Other functions are simple and are omitted here.

```
eq sat (VL, top) = true .
eq sat (Vclock (CL1,T1), CL1 le T2) =
  sat (Vclock (CL1,T1),
    CL1 lt T2) or T1 == T2 .
eq sat (Vclock (CL1,T1), CL1 ge T2) =
  not sat (Vclock (CL1,T1), CL1 lt T2) .
eq sat (Vclock (CL1,T1), CL1 gt T2) =
  not sat (Vclock (CL1,T1), CL1 le T2) .
```

```
eq sat (Vclock (CL1,T1), CL1 lt T2) =
  if T1 < T2 then true else false fi .
eq sat (VL, C1 and C2) =
  sat (VL,C1) and sat (VL,C2) .
eq sat (VL, C1 or C2) =
  sat (VL,C1) or sat (VL,C2) .
eq sat (VL, not C1) = not sat (VL,C1) .
eq sat (VL, C1) = true [owise] .

eq satisfy (nil,C1) = true .
eq satisfy (VL VLLI, C1) =
  sat (VL,C1) and satisfy (VLLI,C1) .
```

A.2. Implementation of Components

As mentioned in Subsection 4.4, a system configuration is a parallel composition of three kinds of components: a timed automaton as an environment, a timed automaton with semaphores as a scheduler and a controller automaton as an interrupt controller.

A.2.1. Timed Automata

The main types of timed automata include:

- Location numbers that identify locations: `LoNumber`.
- A Location is a pair, including a location number, and a constraint, as its invariant: `Location`.
- An external action includes an I/O marker (with the type `IO`) and its action name (with the type `Action`). Here we use a singleton internal action `Null`, since each internal action is unrevealed by the environment.
- A transition is a tuple, containing a source location number, an action, a constraint on clocks, a list of clocks that should be reset and an object location number.
- A timed automaton is a tuple, including a list of locations, a location number as its initial location, a list of clocks and a list of transitions.

```
sorts LoNumber Location .
sorts IO Action IOAction .
sort Transition .
sort Timedautomata .
```

```
ops LN CN : Nat -> LoNumber [ctor] .
ops SN EN : Nat -> LoNumber [ctor] .
op [_,_] : LoNumber Constraint ->
  Location [ctor] .
ops ? ! : -> IO [ctor] .
op a : Nat -> Action [ctor] .
op Null : -> IOAction [ctor] .
op ___ : Action IO -> IOAction [ctor] .
op Tr : LoNumber IOAction
  Constraint List{Clock} LoNumber
```

```

-> Transition [ctor] .
op TA : List{Location} LoNumber
      List{Clock} List{Transition}
      -> Timedautomata [ctor] .

```

In Maude, a timed automaton can be defined as a constant with the type of `Timedautomata`. For example, the timed automaton showed in Figure 1 is defined as follows:

```

op Env : -> Timedautomata .

eq Env =
  TA([EN(0), top] [EN(1), c(20) le 3]
     [EN(2), c(21) le 20],
     EN(0),
     c(20) c(21),
     Tr (EN(0), a(10) !, top,
         c(20) c(21), EN(1))
     Tr (EN(1), a(10) !, c(20) ge 3,
         c(20), EN(1))
     Tr (EN(1), a(13) !, c(21) gt 30,
         c(21), EN(2))
     Tr (EN(2), Null , c(21) ge 20,
         nil, EN(0))
  ) .

```

where `a(10)` denotes the action *pat* and `a(13)` denotes the action *turn*.

A.2.2. Timed Automata with Semaphores

The main difference between timed automata and timed automata with semaphores is the way to handle semaphores and clocks. Semaphores can be regarded as a especial kind of clocks, which do not accumulate when time elapses. On transitions of timed automata, clocks can only be reset to zero, while on transitions of timed automata with semaphores, semaphores can be assigned to any time value, be increased or decreased any time interval. Thus we use `ParaVal` to define valuations of semaphores, as follows,

```

sort ParaVal .

op value : Clock Time ->
          ParaVal [ctor] .
op plus  : Clock Time ->
          ParaVal [ctor] .
op minus : Clock Time ->
          ParaVal [ctor] .

```

Constraints on clocks of timed automata with semaphores are also included comparisons between a clock and a semaphore (with type `Clock`). Hence we use `ParaCon` to denote the type of *parametric constraints* on clocks. Note that type `Constraint` is a sub-type of `ParaCon`.

The transitions for timed automata with semaphores (with the type `ExtTransition`) are also different,

each of which include a source location number, an action, a parametric constraint with the type `ParaCon` on clocks, a list of clocks that should be reset, a constraint with the type `Constraint` on semaphores, a list of assignments to semaphores, and an object location number.

Each control location for timed automata with semaphores (with the type `ExtLocation`) includes a location number and a parametric constraints. And a timed automaton with semaphores that usually represents a scheduler (with the type `Scheduler`) is a tuple, including a list of locations, a location number as its initial location, a list of clocks, a list of semaphores and a list of transitions.

```

sorts ExtTransition ExtLocation .
sort ParaCon .
subsort Constraint < ParaCon .
sort Scheduler .

op _le_ : Clock Clock ->
        ParaCon [ctor prec 37] .

op ExTr : LoNumber IOAction ParaCon
         List{Clock} Constraint
         List{ParaVal} LoNumber
         -> ExtTransition [ctor] .
op [_;_] : LoNumber ParaCon
         -> ExtLocation [ctor] .

op SCH : List{ExtLocation} LoNumber
        List{Clock} List{Clock}
        List{ExtTransition}
        -> Scheduler [ctor] .

```

We also omit several constructor equations for `ParaCon`, such as `lt`, `ge`, `gt`, `and`, `or`, and `not`. They are defined straightforwardly.

An important function `InsConstraint` in timed automata with semaphores is to instantiate a parametric constraint to a constraint under the context of a list of valuation. It instantiates all semaphores to their respective valuations so that one can decided whether the constraint is satisfied in transitions.

```

op InsCon : ParaCon Valuation ->
           ParaCon .
op InsConstraint : ParaCon
                 List{Valuation} -> Constraint .

eq InsCon (C1 le C2, Vclock(C2,T))
          = C1 le T .
eq InsCon (C1 lt C2, Vclock(C2,T))
          = C1 lt T .
eq InsCon (C1 ge C2, Vclock(C2,T))
          = C1 ge T .

```

```

= C1 ge T .
eq InsCon (C1 gt C2, Vclock(C2,T))
= C1 gt T .
eq InsCon (PCON1 and PCON2, VAL) =
InsCon (PCON1, VAL) and
InsCon (PCON2, VAL) .
eq InsCon (PCON1 or PCON2, VAL) =
InsCon (PCON1, VAL) or
InsCon (PCON2, VAL) .
eq InsCon (not PCON1, VAL) =
not InsCon (PCON1, VAL) .
eq InsCon (PCON1, VAL) =
PCON1 [owise] .

eq InsConstraint (PCON1, nil) = PCON1 .
eq InsConstraint (PCON1, VAL VALI) =
InsConstraint (InsCon (PCON1, VAL),
VALI) .

```

For example, the timed automaton with semaphores showed in Figure 3 is also defined as a constant.

```

op Sch : -> Scheduler .

eq Sch =
SCH ([CN(0) ; top]
[CN(1) ; c(30) le s(0) and c(30) le 3]
[CN(2) ; c(30) le s(0) and c(30) le 3]
[CN(3) ; c(30) le s(0) and c(30) le 3]
[ERR ; top]),
CN(0),
c(30) c(31),
s(0),
ExTr(CN(0), a(11) ?, top, c(30), top,
value(s(0),3), CN(1))
ExTr(CN(0), a(12) ?, top, c(30), top,
value(s(0),2), CN(2))
ExTr(CN(0), a(11) ?, top, c(30) c(31),
top, value(s(0),3), CN(3))
ExTr(CN(1), Null, c(30) ge s(0), nil,
top, nil, CN(0))
ExTr(CN(2), Null, c(30) ge s(0), nil,
top, nil, CN(0))
ExTr(CN(3), Null, (c(30) ge s(0)) and
(c(31) lt 7), nil, top, nil, CN(0))
ExTr(CN(1), Null, c(30) ge 3, c(30),
top, minus(s(0),3), CN(1))
ExTr(CN(1), a(12) ?, top, nil,
s(0) le 25, plus(s(0),2), CN(1))
ExTr(CN(1), a(11) ?, top, nil,
s(0) le 25, plus(s(0),3), CN(1))
ExTr(CN(1), a(11) ?, top, c(31),
s(0) le 25, plus(s(0),3), CN(3))
ExTr(CN(1), Null, top, nil,

```

```

s(0) gt 25, nil, ERR)
ExTr(CN(2), Null, c(30) ge 3, c(30),
top, minus(s(0),3), CN(2))
ExTr(CN(2), a(12) ?, top, nil,
s(0) le 25, plus(s(0),2), CN(2))
ExTr(CN(2), a(11) ?, top, nil,
s(0) le 25, plus(s(0),3), CN(2))
ExTr(CN(2), a(11) ?, top, c(31),
s(0) le 25, plus(s(0),3), CN(3))
ExTr(CN(2), Null, top, nil,
s(0) gt 25, nil, ERR)
ExTr(CN(3), Null, c(30) ge 3, c(30),
top, minus(s(0),3), CN(3))
ExTr(CN(3), a(12) ?, (c(30) lt s(0))
and (c(31) lt 7), nil,
s(0) le 25, plus(s(0),2), CN(3))
ExTr(CN(3), Null, top, nil,
s(0) gt 25, nil, ERR)
ExTr(CN(3), Null, (c(30) lt s(0))
and (c(31) ge 7), nil,
s(0) gt 25, nil, ERR)
) .

```

where ERR is a special location number defined by

```
op ERR : -> LoNumber .
```

to denote the error control location in the scheduler (see Subsection 4.3). a(10) and a(13) denote the same actions as the above subsection. Besides that, a(11) denotes *trigger_p* and a(12) denotes *trigger_q*.

A.2.3. Controller Automata

Time Lag The most important part in controller automata is the time lag, which includes the following functions:

```

op TimeLag : Timedautomata LoNumber
NNegRat -> Timedautomata .
op UpdateCon : List{Location} LoNumber
Clock NNegRat -> List{Location} .
op UpdateTri : List{Transition}
LoNumber Clock ->
List{Transition} .
op getLastFre : List{Location} Nat
-> Nat [memo] .
op getLastClo : List{Clock} Nat
-> Nat [memo] .
op fresh : Nat -> Nat .

```

- As the definition, TimeLag accepts a timed automaton, a location number, and a time, and returns a new timed automaton.
- UpdateCon and UpdateTri update the constraint in the given control locations, and the constraint in the specific transitions, respectively.

- `getLastFre` and `getLastClo` get the indexes of the last inserted control location and the last inserted clock, respectively, which are used to generate fresh inserted control location and clock. `fresh` generates a fresh nature number for the index.

We just show the first three functions, as follows,

```

eq UpdateCon (nil, LN, C, R) = nil .
eq UpdateCon ([LN,CON] LOLI, LN, C, R) =
  append([LN, CON and (C le 0 or
            C ge R)], LOLI) .
eq UpdateCon ([LN1,CON] LOLI, LN2, C, R) =
  append([LN1, CON],
    UpdateCon(LOLI, LN2, C, R)) [owise] .

eq UpdateTri (nil, LN, C) = nil .
eq UpdateTri (Tr(LN1, IOA, CON, CLLI, LN2)
              TRLI, LN2, C) =
  append(Tr(LN1, IOA, CON, append(C, CLLI),
            LN2), UpdateTri(TRLI, LN2, C)) .
eq UpdateTri (Tr(LN1, IOA, CON, CLLI, LN2)
              TRLI, LN, C) =
  append(Tr(LN1, IOA, CON, CLLI, LN2),
    UpdateTri(TRLI, LN, C)) .

eq TimeLag(TA(LOLI, LN0, CLLI, TRLI), LN, R)
  = TA (append([IDLE(fresh(getLastFre
(LOLI, 0))), f(fresh(getLastClo(CLLI, 0))
le R ], UpdateCon(LOLI, LN, f(fresh
(getLastClo(CLLI, 0))), R)), LN0,
append (f(fresh(getLastClo(CLLI, 0))),
          CLLI),
append ( Tr(LN, Null, top, nil, IDLE
(fresh(getLastFre(LOLI, 0))))
  Tr(IDLE(fresh(getLastFre(LOLI, 0))),
  Null, f(fresh(getLastClo(CLLI, 0))
ge R, nil, LN), UpdateTri(TRLI, LN,
  f(fresh(getLastClo(CLLI, 0)))))) .

```

Controllers A controller automaton is defined like a timed automaton, except that each state is a tuple of a location number, a timed automaton, a constraint on clocks and a time as expect running time of the state. Furthermore, the transitions of a controller automaton are partitioned into three disjoint sets, for push, pop and internal transitions, respectively.

```

sort ConState .
sorts Push Pop Internal ConTransition .
subsort Push < ConTransition .
subsort Pop < ConTransition .
subsort Internal < ConTransition .

op [_,_,_,_] : LoNumber Timedautomata
              Constraint Time

```

```

-> ConState [ctor] .
op push : LoNumber IOAction Constraint
List{Clock} LoNumber -> Push [ctor] .
op pop : LoNumber IOAction Constraint
List{Clock} LoNumber -> Pop [ctor] .
op int : LoNumber IOAction Constraint
List{Clock} LoNumber
-> Internal [ctor] .

```

The controller automaton showed in Figure 2 is defined as a constant. as follow,

```

op Con : -> Controller .

eq Con = CA(
[SN(0),
TA([LN(0), top], LN(0), nil, nil), top, 3]
[SN(1),
TA([LN(10), top] [LN(11), top]
[LN(12), top] [LN(13), top]
[LN(14), top] [LN(15), top],
LN(10),
c(10),
Tr(LN(11), a(10) ?, top, c(10), LN(10))
Tr(LN(10), a(10) ?, c(10) le 2,
nil, LN(11))
Tr(LN(10), a(11) !, c(10) gt 2,
c(10), LN(12))
Tr(LN(12), a(10) ?, top, c(10), LN(13))
Tr(LN(12), Null, c(10) gt 25,
nil, LN(10))
Tr(LN(13), a(10) ?, c(10) le 2,
nil, LN(10))
Tr(LN(13), a(12) !, c(10) gt 2
and c(10) le 25, c(10), LN(14))
Tr(LN(14), a(10) ?, top, c(10), LN(15))
Tr(LN(14), Null, c(10) gt 30,
nil, LN(11))
Tr(LN(15), a(12) !, c(10) gt 2
and c(1) le 30, c(10), LN(14))
Tr(LN(15), a(10) ?, c(10) le 2,
nil, LN(11))
),
c(Run) le 150, 150]
[SN(2),
TA([LN(20), top] [LN(21), top],
LN(20),
nil,
Tr(LN(20), Null, top, nil, LN(21))
),
c(Run) le 50, 50],
SN(0),
c(Run),
push(SN(0), a(10) ?, top, nil, SN(1))

```



```

[ CA ]
=>
[[ EN2 || CN1 || SN1 | LN2 ||
  Reset (VALI1,append (CLLI1, CLLI2)) ;
    SVALI1 ; FRE1 ; STACK]]
|>>>| CTRLI
|<E>| getTransition(getAllTransition
                    (Env), EN2)
|<S>| STRLI
|<C>| getTransition(getAllTransition
                    (getConTimedautomata (CA, SN1)), LN2)
[ CA ]
if satisfy (VALI1, CON1) and
  satisfy (Reset (VALI1,append (CLLI1,
  CLLI2)), getInvariant (Env, EN2)) and
  satisfy (VALI1, CON2) and
  satisfy (Reset (VALI1,append
  (CLLI1, CLLI2)), getInvariant
  (getConTimedautomata (CA, SN1), LN2)) and
  comm (IOA1, IOA2) .

```

Furthermore, Maude does not allow nondeterministic rewriting. Thus we need to remove the first rule in the list of transitions, so that other transitions can be applied to the system. There are four such kind of rules, for the environment, the scheduler, the controller and the current handler, respectively, we only show the rule for the environment.

```

rl [E-Decrease] :
  [[EN1 || CN1 || SN1 | LN1 || VALI1 ;
    SVALI1 ; FRE1 ; STACK]]
  |>>>| CTRLI
  |<E>| TR1 TR2 ETRLI
  |<S>| STRLI
  |<C>| TRLI
  [ CA ]
  =>
  [[ EN1 || CN1 || SN1 | LN1 || VALI1 ;
    SVALI1 ; FRE1 ; STACK]]
  |>>>| CTRLI
  |<E>| TR2 ETRLI
  |<S>| STRLI
  |<C>| TRLI
  [ CA ] .

```

Init is used to denote the initial state with the type INTState, defined by

```
op Init : -> INTState .
```

```

eq Init =
[[getInit (Env) || getSchInit (Sch) ||
  getConInit (Con) | getInit
  (getConTimedautomata
  (Con, getConInit (Con))) ||

```

```

Initial (append (append (getClock (Env),
  getSchClock (Sch)), getAllClock (Con)));
  Initial (getSchSemaphore (Sch));
  getInitFre (Con);
  empty]]
|>>>| getConTransition
  (getConAllTransition (Con),
  getConInit (Con))
|<E>| getTransition (getAllTransition
                    (Env), getInit (Env))
|<S>| getSchTransition
  (getSchAllTransition (Sch),
  getSchInit (Sch))
|<C>| getTransition (getAllTransition (
  getConTimedautomata (Con,
  getConInit (Con))),
  getInit (getConTimedautomata
  (Con, getConInit (Con))))
[ Con ] .

```

We omit all definitions of get-like functions here, which are just projection functions on their respective tuples. In the definition of Init, constants Env, Sch, and Con are an interrupt environment, a scheduler, and a controller, defined in subsection 5.2.1, 5.2.2 and 5.2.3, respectively. They can be modified by the user according to different contexts and different target systems.

With the strategies and engines rewrite, search, frewrite etc. provided by Maude, the target system can be flexibly simulated, and its properties are checked at the specification level. For example, schedulability analysis of the system can be checked by the following command.

```

search [1] Init =>* [[EN1 || ERR ||
  SN1 | LN1 || VALI1 ; SVALI1 ;
  FRE1 ; STACK]] |>>>| CTRLI
|<E>| ETRLI <S>| STRLI
|<C>| TRLI [ CA ] .

```