Extending UPPAAL for the Modeling and Verification of Dynamic Real-time Systems

Abdeldjalil Boudjadar [1], Frits Vaandrager [2], Jean-Paul Bodeveix [3], Mamoun Filali [3]

[1] CISS, Aalborg University. Denmark
[2] ICIS, Radboud University Nijmegen. The Netherlands
[3] IRIT, Université de Toulouse. France

Abstract. Dynamic real-time systems, where the number of processes is not constant and new processes can be created on the fly like in object-based systems and ad-hoc networks, are still lacking a formal framework enabling their verification. Different toolboxes like Uppaal [21], Tina [10], Red [28] and Kronos [29] have been designed to deal with the modeling and analysis of real-time systems. Nevertheless, a shortcoming of these tools is that they can only describe static topologies. Other tools like Spin [18] allow the dynamic creation of processes, but do not consider time aspects. This paper presents a formal framework for modeling and verifying dynamic real-time systems. We introduce *callable timed automata* as a simple but powerful extension of standard timed automata in which processes may call each other. We show that the semantics of each call event can be interpreted either as an activation of the existing instance of the corresponding automaton (static instantiation), or a creation of a new concurrent instance (dynamic instantiation). We explore both semantical interpretations, static and dynamic, and give for each one the motivation and benefits with illustrating examples. Finally, we report on experiments with a prototype tool, which translates (a subset of) callable timed automata to UPPAAL systems.

Keywords: Dynamic real-time systems, timed automata, callable timed automata

1 Introduction

Timed automata (TA) [1] have been proposed as a powerful model for both timed and concurrent systems modelling. However, a dynamic framework for timed automata instantiation and applicability, to model dynamic system topologies like object-based systems and ad-hoc networks in which processes are created and triggered on the fly, is still lacking. Moreover, the modelling of timed automata as functional values, whereby a timed automaton can be called and applied to given parameters to generate outputs, instead of an independent component making computations and updating the system control is not explored. UPPAAL [6] is an integrated tool environment for editing, simulating and model checking real-time systems modeled as networks of timed automata. The tool has been used successfully and routinely for many industrial case studies. Nevertheless, a shortcoming of UPPAAL is that it can only describe static network topologies, and does not incorporate a notion of dynamic process creation.

Unlike UPPAAL's *C-function* actions performing local sequential computations, this study consists of encoding the call mechanism into interacting processes, whereby communication on shared variables and synchronization with the external environment are enabled. The modelling of a timed automaton as a callable function which performs communications and interactions with the external environment enables it to be callable and triggerable by any other automaton. We introduce *callable timed automata* (CTA) as a formal framework for the modelling and analysis of dynamic timed systems, where the number of components (processes) may vary. The concept of callable timed automata enables, for a set of processes, to model a common behavior as an automaton callable by any other process originally performing such a behavior.

Syntactically, a callable timed automaton is a finite timed automaton [4] parameterized by a set of data, and triggered through the execution of a calling transition from another automaton. Moreover, a callable automaton may return results to its calling component. Semantically, we interpret this syntactical extension in different ways by considering different criteria like (1) *concurrency*: the activation of a callable process may be blocking for the corresponding calling process, wherein

the former cannot progress while the callee one is running. Will both calling and callee components progress concurrently? (2) *instantiation*: the UPPAAL template's instantiation is static. Will the instantiation of callable TA be static (a constant number of instances initially created) or dynamic (for each call, a new instance is created on the fly)?

The ultimate goal of this paper is to provide a new formal framework for the modelling and verification of dynamic timed systems, where the number of processes is not constant, in terms of timed automata. To this end, we introduce an extension for structuring UPPAAL systems by integrating callable timed automata.

The rest of the paper is organized as follows. In Section 2, we cite existing related work. Section 3 motivates our proposal through a set of examples. In Section 4, we define callable timed automata and give their translation to UPPAAL TA. In Section 5, we review timed transition systems as a semantic basis. In Section 6, we define the semantics of both static and dynamic instantiations of CTA. Section 7 shows the implementation of CTA in UPPAAL. Section 8 presents the conclusion.

2 Related work

In the literature, several frameworks [15, 12, 23, 25, 22, 5, 26] have been proposed to generalize the operational model of functions to a model of concurrent processes. Most of these proposals work on the encoding of the functional computation model λ-Calculus into the concurrent computation model π-Calculus. In [22], Milner showed that λ-Calculus could be precisely encoded into π-Calculus. The SPIN tool [18] enables the verification of dynamic systems where concurrent processes can be created on the fly. Both creating and created processes progress together. The creation of a new process does not block the creating component execution i.e., a return is not needed to unlock the creating component. Similarly, the ADA language [13] enables the creation of tasks on the fly. After the creation of each task, the calling process waits until the new process is elaborated. Each process may perform a return immediately to unlock its calling component via action *accept*, or executes some actions then performs a return via statement *accept do* (RPC-like protocol [1]). Recently, there has been an amount of work focusing on recursive extensions of timed automata. Without considering synchronization, the authors of [27] define a restricted notion of recursive timed automata where their decidability results impose strong limitations on the number of clocks (at most 2 clocks). Moreover, either all clocks are passed by reference or none is passed by reference.

In our proposal, we introduce callable timed automata whereby we extend UPPAAL timed automata transition actions to concurrent process creation. Callable timed automata are referenced like functions and may interact with their environment. The semantics of each call event can be interpreted either as the activation of an existing instance of the corresponding template, or by the creation of a new concurrent instance of the callee automaton.

3 Callable Timed Automata

In this section, we introduce an extension of timed automata named *callable automata* where automata call each other. Unlike functions which are local computations getting their inputs as parameters before being triggered, a callable timed automaton is an open process which can interact with its external environment at anytime by accepting inputs, producing outputs and updating the system state. Syntactically, callable timed automata (CTA) are an extension of finite automata where transitions can be equipped by either a particular event `call`, to trigger the execution of another automaton, or again a `return` event to yield results. The call of a callable timed automaton can be parameterized by a set of expressions. Both call and return actions are used as a synchronization event instead of an update action. The execution of `call T` corresponds to the activation of an instance of template `T`. Obviously, the activation of an instance is preceded by its creation which can can be performed either when the system starts or on the fly, i.e. when an automaton calls another one, it induces both instantiation and activation of the corresponding template.

In the semantical interpretations of call events, we may distinguish static and dynamic instantiations of callable timed automata. In fact, the interpretation of each call event depends on the

[1] RPC is an acronym for *Remote Procedure Call*. It states the activation of a process (server) by another (client) such that the client process cannot progress while the server process does not perform a return.

nature of the callee template. To distinguish between static and dynamic interpretations, we associate to each CTA signature either a finite number n or an infinite one ∞. Namely, if the template signature states a finite number n of instances, then each call event for that template is considered to be static. Otherwise, in the case of ∞, the call event will be considered to be dynamic.

3.1 Static Instantiation

In this subsection, we consider the situation that each callable timed automaton is instantiable through a constant number of instances, that may be initially created when the system starts. The execution of each call event corresponds to the activation of an instance of the callee template, which may delay and interleave with the execution of other components. That is the same case as for UPPAAL-Port [17] where components trigger each others. Each of the instances will be reinitialized for each activation (call) with the corresponding parameters. In fact, the callable automaton instances are considered as any other instance associated to a normal UPPAAL template. Moreover, with such an interpretation, a callable automaton T can be called concurrently in the limits of its number of instances $\mathcal{I}(T)$.

Formally, the call event is blocking where the calling component cannot run any other transition while its callee automaton has not performed a return. Likewise, a CTA may block call events from components other than the current callers if free instances are not available. The calling component gets the control back when the execution of the callee instance emits a return event. The return event of a callee instance does not state its termination. The execution of a callee instance can be atomic, which agrees with the UPPAAL action semantics.

The static instantiation applicability of callable timed automata covers a large spectrum of the RPC-based systems. An example of such an instantiation can be found in UPPAAL-Port, where a system is structured as a set of hierarchical components executed in a sequence. When the execution of a component has completed, it triggers the (non-atomic) execution of another component by activating its trigger-ports. Without considering hierarchy, one can distinguish that an UPPAAL-port system can be translated to a set of callable automata in a systematic way. Such a translation consists of replacing the activation of trigger-ports of each component by a call made by the last transition of its triggering component.

3.2 Example 1 (Static instantiation)

We reuse the UPPAAL expression of the well known *Train-Gate* example [6], depicted in Figure 1. In fact, such an example models the train crossing concurrency, where a set of trains request concurrently access to a unique crossing point, *the critical section*, in order to continue on their respective routes. The crossing point is governed by a gate which each train must signal to gain crossing authorization.

In order to distinguish between train instances, each one has a unique identifier *Id*. As the access request is the same for all trains, we model this common behavior (access request) by a new parameterized callable timed automaton named *Register*, and by that trains get rid of requesting their own access authorization. The automaton *Register* can be called by any train intending to cross the gate.

When a train *Id* approaches the crossing point, it calls the automaton *Register* with its own identifier *Id*. The automaton *Register* notifies the *Gate*, which the train *Id* is approaching, through a synchronization on channel *appr*, and inserts *Id*

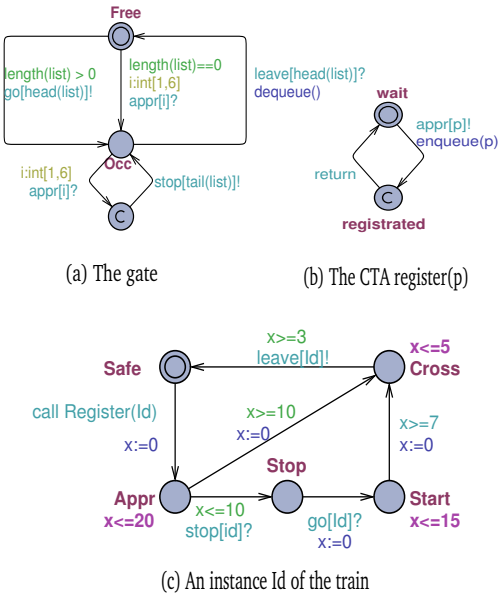

Fig. 1. The *Train-Gate* Example.

into the waiting list *list*. Whenever the execution of automaton *Register* is over for a given call by reaching the return action, the corresponding calling train can resume. Depending on the availability of the *Gate*, such a train (*Id*) crosses immediately or stops for a delay specified by a constraint on clock x, waiting to be on the front of *list* then crosses the gate. Accordingly, the automaton *Register* becomes available for accepting other calls by any train intending to cross the gate.

3.3 Dynamic Instantiation

In this interpretation, a varying number of instances can be dynamically associated to each callable automaton: each call event corresponds to the creation of a new instance of the callee automaton. Template instances are created on the fly through the execution of the corresponding calls. Each newly created instance will be simultaneously triggered. Hence, the call event is not blocking for other calling components. Moreover, both calling and callee instances may progress concurrently, after performing a return. In fact, in the dynamic instantiation the return event of an instance enables to yield its results but does not state its termination. i.e. an instance may run other transitions after performing a return. The termination of an instance execution is stated by reaching a final location. The dynamic instantiation of callable timed automata leads to building the structure of the system on the fly: the system has different numbers of instances on different executions and at different dates.

The dynamic feature of such an instantiation is suitable to model object-based systems, ad-hoc networks, fault tolerant and DataBase Management systems (DBMS) where components (objects, hosts, processes) are created on the fly. For example, in the case of DataBase Management systems, when the execution of a process requires to read data from a database, it calls the Reader module of DBMS by creating an instance of the former to fetch data.

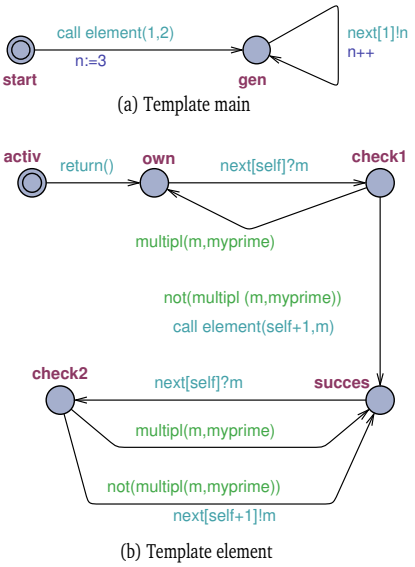

Fig. 2. The sieve of Eratosthenes.

3.4 Example 2 (Dynamic instantiation)

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a given integer M. Given a list of numbers, the algorithm iteratively marks as a non-prime the multiples of each prime, starting with the multiples of 2. It runs across the table until the only numbers left are prime.

As depicted in Figure 2, we have implemented this algorithm by the parallel composition of 2 automata: `main` and `element`. In fact, we model the table elements by the automaton so-called `element`. Each instance of template `element` is parameterized by a natural number (1 of template `main`) which states its identifier, and another integer number (2 of template `main`) to retrieve its prime number. Moreover, each instance has 2 local variables: *self* to store its identifier (parameter), and *myprime* to store the value of the corresponding prime number (parameter). To allow the communication of instances, we declare a vector *next* of M channels.

The system is managed by another automaton so-called `main`, which creates the first instance of automaton `element`. Such an instance gets as effective parameters the identifier of the first instance (1), and the corresponding prime number (2). After that, automaton `main` increments iteratively the number n to be checked and sends it to that instance (of template `element`) through channel $next[1]$ [2].

Once the system is triggered, the automaton `main` moves from location *start* to location *gen* (generate) by executing the call action $call\ element(1, 2)$, and updating n to 3. Such a call creates the first instance of template `element`, which is identifiable by $self = 1$ and $myprime = 2$. This

[2] In fact, channels *next* are parameterized by the number to be checked. We may consider shared variables to implement the data communication over channels.

instance performs a return to unlock its caller and moves to its location *own*. The automaton `main` sends the first value n to be checked to the newly created instance, of template `element`, on channel *next[1]*. Through the reception of the first message $next[self]?m$, the current `element` instance checks whether or not the received value of m is a multiple of its own prime number *myprime*.

If m is a multiple of *myprime* then the received value of m will be ignored, and the current instance of `element` moves back from location *check1* to location *own*. Otherwise, the current instance of template `element` requests the creation of another instance, through the statement $call\ element(self + 1, m)$, and moves to location *succ* (successor). At this level, the first instance of `element` is waiting for the reception of another number to be checked, sent by `main`. On a reception $next[self]?m$ of a new value which is not a multiple of *myprime*, the instance of `element` sends that value to its successor instance via channel $next[self + 1]$, which corresponds in this case to $next[2]$. Similarly, each new instance of `element` behaves in the same way as the first one. Herein, one can distinguish that each new number, sent by automaton `main`, crosses a sequence of `element` instances until it is dropped, the case of a multiple of a discovered *myprime*, or registered as a new prime number with the creation of a new instance of template `element`.

4 Timed Automata Extension

The modeling and verification of real-time systems, via timed automata, are mature topics to which a large amount of work has been devoted during the last two decades. However, the modeling and verification of dynamic real-time systems, where the topology (global architecture and number of components) may change during the execution of systems, constitute a perspective and an active field of research.

In this section, we give the formal basis of callable timed automata (CTA) where transition actions can be internal, external, a `call` of another callable timed automaton, or again a `return`. Then, we show how callable timed automata can be translated to UPPAAL ones, and establish an important result stating that the semantics of CTA and that of their translation to UPPAAL timed automata are bisimilar (Figure 3). In fact, the translation enables us to reuse the UPPAAL toolbox for the verification of dynamic timed systems modeled with CTA. Let us introduce the following notation.

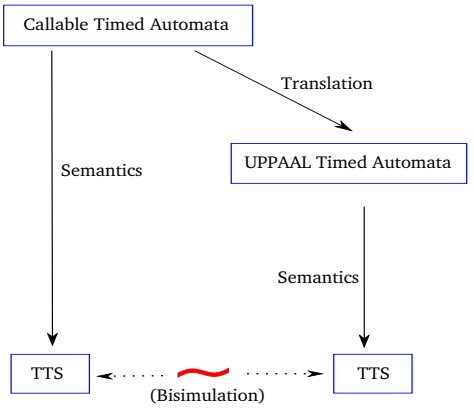

Fig. 3. Semantics and translation of CTA.

Notation. We assume a universe $\mathcal{V}$ of variables. To each variable $v \in \mathcal{V}$ we associate a nonempty set of values, referred to as the type of v and denoted $type(v)$. Moreover, we associate to each variable $v \in \mathcal{V}$ a default initial value $d_v^0 \in type(v)$. A variable v whose type equals the set $\mathbb{R}_{\geq 0}$ of non-negative real-numbers is called a *clock*. We assume that the default initial value of all clocks equals 0. Let $V \subseteq \mathcal{V}$ be a set of variables.

- A valuation of V is a function that maps each variable to an element of its type. We use $Val(V)$ to denote the set of valuations of V.
- $\mathfrak{E}(V)$ defines the set of expressions built over V. To each expression $e \in \mathfrak{E}(V)$ we assign a type $type(e)$. Each expression induces a state transformer, that is, $[\![e]\!] : Val(V) \rightarrow Val(V)$. We call an expression side effect free if $[\![e]\!]$ is the identity function. Each expression also denotes a value for any valuation: $\langle\langle e \rangle\rangle : Val(V) \rightarrow type(e)$.
- $\mathfrak{P}(V)$ defines the set of predicates built over V. If ϕ is a predicate over V then $[\![\phi]\!] : Val(V) \rightarrow Bool$ gives the truth value of ϕ for any given valuation of V.

– For a function f defined on a domain $dom(f)$, we write $f\lceil X$ the restriction [8] of f to X, that is the function g with $dom(g) = dom(f) \cap X$ such that $g(z) = f(z)$ for each $z \in dom(g)$.
– Two functions f and g are compatible [8], denoted $f \heartsuit g$, if they agree on the intersection of their domains, that is, $f(z) = g(z)$ for all $z \in dom(f) \cap dom(g)$.
– We denote by $f \triangleright g$ the left overriding function defined on $dom(f \triangleright g) = dom(f) \cup dom(g)$ where f overrides g for all elements in the intersection of their domains. For all $z \in dom(f \triangleright g)$,

$$(f \triangleright g)(z) \triangleq \begin{cases} f(z) \text{ if } z \in dom(f) \\ g(z) \text{ if } z \in dom(g) - dom(f) \end{cases}$$

  Similarly, we define the dual right overriding operator by $f \triangleleft g \triangleq g \triangleright f$.
– We define $f \parallel g \triangleq f \triangleright g$ when f and g are compatible.

4.1 UPPAAL Timed Automata

UPPAAL is an integrated tool environment for editing, simulating and model checking real-time systems modeled as networks of timed automata. The tool has been used successfully and routinely for many industrial case studies. Nevertheless, a shortcoming of UPPAAL is that it can only describe static network topologies, and does not incorporate a notion of dynamic process creation. Moreover, UPPAAL does not incorporate a notion of one automaton calling another, like a function, even though this last concept can be encoded within UPPAAL using a pair of handshakes.

In fact, UPPAAL timed automata [6] are extensions of the classical ones [1] where one level hierarchy of local/global variables, committed locations, communication and priorities have been introduced. Besides, in the UPPAAL language timed automata are defined within a global common context.

Definition 1 (Global context). *A global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$ consists of a finite set of automata names $\Sigma \subseteq \mathcal{T}$, a finite set of global variables $V^g \subseteq \mathcal{V}$, the initial valuation $Init^g$ of global variables V^g and a finite set of channels C.*

Throughout this paper we do not distinguish between clock and normal variables. Each variable of $\mathcal{V}$ is either a clock or a normal variable. By now, we give the structure of a timed automaton defined on a global context.

Definition 2 (Timed automaton). *Given a global context $\mathcal{C}$, a timed automaton (TA) is a tuple $\langle Q, q^0, K, V^l, Init^l, Inv, \rightarrow \rangle$ where Q is the set of locations, $q^0 \in Q$ is the initial location, V^l is the set of local variables, $Init^l$ is the initial valuation of local variables, $Inv : Q \rightarrow \mathfrak{P}(V)$ associates an invariant to each location, $K \subseteq Q$ is a set of committed locations, and $\rightarrow \subseteq Q \times \mathfrak{P}(V) \times \Lambda \times \mathfrak{E}(V) \times Q$ is the transition relation, where $V = V^l \cup V^g$ and $\Lambda = C? \cup C! \cup \{\tau\}$.*

For the sake of simplicity, we write $q \xrightarrow{G/\lambda/a} q'$ for $(q, G, \lambda, a, q') \in \rightarrow$. The composition of timed automata, so-called networks of timed automata (NTA), enables to model a system as a flat set of interconnected components. Each component (TA) interacts with its external environment through communication on shared variables and synchronization of actions.

In a variant of UPPAAL called UPPAAL-Port [17], hierarchical compositions are enabled whereby the system can be modeled as a set of components. Each component may encapsulate other components. Several proposals [6, 8, 11, 14] studying the composition of UPPAAL timed automata have analyzed their properties. The authors of [6] define a non compositional semantics of UPPAAL NTA. In [11, 8], the authors define a compositional semantics of NTA and establish some properties like the preservation of system invariants. In [14, 20], the semantics of TA composition is not compositional because the product of TA semantics is not associative. Counter-examples are given in [9, 7].

4.2 Callable Timed Automata

Callable timed automata provide a formal framework for the modelling and analysis of dynamic timed systems. In fact, the concept of callable timed automata enables, for a set of processes, to

model a common behavior as an automaton callable by any other process originally performing such a behavior.

Unlike UPPAAL callable C-functions, a callable timed automaton can interact with the other components and call other callable automata. However, in the case of static instantiation, in order to avoid deadlock due to mutually dependent executions, a callable timed automaton cannot call its own hierarchical calling components. In fact, for the static interpretation, the calling component cannot progress while its current callee component is running. Once the callee TA execution is over, the corresponding calling component may resume the control and continue its execution. However, for the dynamic instantiation, after performing a return to unlock its calling component, a callee component may progress together with the execution of its calling component. Thus, in the static instantiation, the *return* action represents the end of the call execution of callable TA whereas, in the dynamic instantiation, it is considered as an ordinary action. Obviously, a system of CTA must contain at least one triggering TA (root) to activate CTA.

We assume a universe $\mathcal{T}$ of automata names, and associate to each automaton name $T \in \mathcal{T}$ a return type $\mathcal{R}(T)$, the number of instances to be created $\mathcal{I} : \mathcal{T} \to \mathbb{N}_{>0} \cup \{\infty\}$ and a formal parameter $p_T \in \mathcal{V}$. In fact, the maximal number of instances to be created for each template is either a strictly positive number (> 0) if the template is statically instantiable, or an infinity (∞) in the case of dynamic instantiation.

In this paper, we only consider automata with a single formal parameter. Automata with multiple parameters may be encoded using variables of type vector, record or union in the same way as simple types and without affecting our framework.

We introduce expressions of type automaton and write $\mathfrak{E}(\mathcal{T}, \mathcal{V})$ for the set of expressions $\{T(e) \mid T(p_T) \in \Sigma \wedge e \in \mathfrak{E}(\mathcal{V}) \wedge type(e) = type(p_T) \wedge e$ is side effect free$\}$. Formally, a callable timed automaton is given by:

Definition 3 (Callable timed automaton). *Let $\mathcal{C} = \langle \Sigma, V^g, Init^g, C\rangle$ be a global context. A callable timed automaton (CTA) for $\mathcal{C}$ is a tuple $\langle T, Q, q^0, F, V^l, Init, Inv, \to\rangle$ where Q, q^0, $Init^l$ and Inv are the same as for TAs:*

- *$T \in \Sigma$ is the automaton name,*
- *$F \subseteq Q$ a set of final locations,*
- *$V^l \subseteq \mathcal{V}$ is a set of local variables; we require $V^g \cap V^l = \emptyset$, $p_T \in V^l$, and write $V = V^g \cup V^l$,*
- *$\to \subseteq Q \times \mathfrak{P}(V) \times \Lambda \times \mathfrak{E}(V) \times Q$ is the transition relation which, for each transition, consists of a source location, a guard, a label, an action and a target location. Here $\Lambda = C? \cup C! \cup \{\tau\} \cup (V \times \Sigma \times \mathfrak{E}(V)) \cup \mathfrak{E}(V)$ is the set of transition labels. Each transition label can be a synchronization on a channel, an internal event, a call of another automaton, or a return action.*

We write $q \xrightarrow{G/\lambda/a} q'$ for $(q, G, \lambda, a, q') \in \to$. Moreover, if $\lambda = (x, T', e) \in V \times \Sigma \times \mathfrak{E}(V)$ then we refer to the transition as a call transition and write $q \xrightarrow{G/x:=call\ T'(e)/a} q'$. In this case, we require that $type(e) = type(p_{T'})$ and $type(x) = \mathcal{R}(T')$. Similarly, if $\lambda = e \in \mathfrak{E}(V)$ then we refer to the transition as a return transition and use the notation $q \xrightarrow{G/return(e)/a} q'$. In this case we require that $type(e) = \mathcal{R}(T)$. Intuitively, via a *call* $T'(e)$-transition automaton T calls automaton T' with a parameter value that can be obtained by evaluating expression e. A $return(e)$-transition is used to return the value of expression e. If the return type of an automaton T is void, we use $return()$ and just keep *call* $T(E)$ to call the automaton T, omitting the assignment "$x :=$" . Furthermore, callable timed automata should satisfy the following wellformedness conditions: final locations do not have outgoing transitions, and return actions are side effect free. We call *subprogram* a CTA of which each return transition leads to a final location. Moreover, we associate to each automaton name a CTA template (record): $\mathcal{D} : \mathcal{T} \to CTA$.

4.3 Translation of Callable TA to UPPAAL TA

In order to reuse the UPPAAL toolbox, we translate callable timed automata to UPPAAL TA. Hence, as stated in the previous section, to make the translation and implementation of CTA easier the user provides the nature of each template instantiation. In fact, through $\mathcal{I}(T)$ the user states whether the template T is instantiable statically or dynamically. Moreover, the user specifies

the number of instances to be created, for each template, in the case of static instantiation. Since calling and callee components may not access each others local variables, we consider the UPPAAL communication through shared variables.

As shown in Figure 4, for translating the calling transition $q \xrightarrow{x<=0/y:=call\ T(e)/} q'$, the expression e is assigned to a new shared variable $param$ [3]. Thereafter, the value of such a shared variable will be copied into the local variable $p_T \in V^l$ of the callee automaton ($T(p_T) \in \Sigma$), as depicted in the bottom of Figure 5.

Mainly, the translation consists of splitting each calling transition of CTA into two synchronizing transitions, as shown in Figure 4. The first transition is an output on a particular channel cal, to activate the corresponding callee CTA, which engages with the assignment of expression e to shared variable $param$, whereas the second transition is an input on a particular channel ret, with the assignment of value $result$ to variable y requesting the call. The execution of the former transition states the termination of the call execution. Both transitions, resulting from the translation of a call, are linked through a new intermediate location q_{int} relative to each pair (y, t), where t is the original calling transition and y is the variable requesting the call. In fact, we use the notation $t : q \xrightarrow{G/\lambda/a} q'$ to state that t is the current transition name, which will be used to reference this transition.

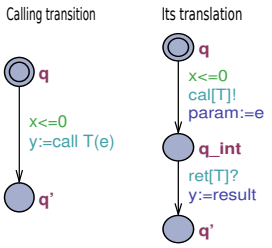

Fig. 4. The translation of calls.

In Figure 5, we show how the structure of a callable automaton (top) can be translated to that of an UPPAAL one (bottom). The translation consists of adding a new initial location q_{init} as the triggering point (activation) of the corresponding UPPAAL TA. This location will be linked to the original initial location q^0 of CTA through an input synchronizing transition on channel $cal[T]$, engaging with the assignment of shared variable $param$ to the CTA local variable p, dedicated to receive the parameter value.

When it meets a return event, the callee CTA yields its result to its calling through a synchronizing transition on channel $ret[T]$, which assigns the result r to shared variable $result$ and unlocks the calling component. Moreover, all CTA final locations are linked to newly inserted location q_{init} via an empty committed transition in order to make CTA available for other calls.

Remark. We have associated to each callable timed automaton T a pair of channels $(cal[T], ret[T])$. Such channels can be used by any other automaton T' intending to call automaton T. Moreover, the set of parameters $\{param\}$, respectively $\{result\}$, depends on the number and types of call, respectively return, parameters. Such variables are re-used for the whole model because the synchronizations on cal and ret channels are atomic transitions.

Definition 4 (TA corresponding to a CTA). *Given a CTA $\langle T, Q, q^0, F, V^l, Init^l, Inv, \rightarrow_T \rangle$ for a global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$ with $T(p_T) \in \Sigma$, its translation to a TA is defined by $\langle Q \cup Q^{int} \cup \{q_{init}\}, q_{init}, F, V^l, Init^l, Inv', \rightarrow \rangle$ over the global context $\langle \Sigma, V^g \cup \{param, result\}, Init^g, C \cup \{cal, ret\} \rangle$ where $Inv'(q) = Inv(q)$ if $q \in Q$ else true and $\rightarrow$ is the smallest relation such that:*

$$\frac{q \xrightarrow{G/\lambda/a}_T q' \quad \lambda \in C! \cup C? \cup \{\tau\}}{q \xrightarrow{G/\lambda/a} q'}\ Action \qquad \frac{q \in F}{q \xrightarrow{\top/\tau/skip} q_{init}}\ Restart$$

$$\frac{}{q_{init} \xrightarrow{\top/cal[T]?/p_T:=param} q^0}\ Activate \qquad \frac{q \xrightarrow{G/return(e)/a}_T q'}{q \xrightarrow{G/ret[T]!/result:=e,a} q'}\ Return$$

$$\frac{t : q \xrightarrow{G/x:=call\ T'(e)/a}_T q'}{q \xrightarrow{G/cal[T']!/param:=e} q_t \xrightarrow{\top/ret[T']?/x:=result,a} q'}\ Call$$

[3] In fact, the type $type(param)$, resp $type(result)$, is the union of all of the parameter, resp return, types used in the model.

where skip is an empty action (identity), $Q^{int} = \{q_t \mid t : q \xrightarrow{G/x:=call\ T'(e)/a} q'\}$ *is a set of intermediate locations introduced when splitting the calling transitions as shown in Figure 4, and* q_{init} *is the new initial location of the resulting TA, again illustrated in Figure 5.*

In fact, this definition translates a CTA and its global context to a timed automaton, where the final locations are marked committed in order to get instances immediately available after the end of each call. Each instance of the template T is processed in the same way. Therefore, the translation of a CTA is a network of timed automata defined on the translation of the global context $\mathcal{C}$ where to each instance of the CTA T corresponds a TA.

Transition rule *Action* states that non calling transitions of the CTA are held without any change in the corresponding translation. Rule *Restart* enables the resulting TA to join its new initial location q_{init} from each final location. Via rule *Activate*, the execution of a callable TA translation is activated through an enabled (guard $= \top$) synchronizing transition. The former leads to reach the old initial location q^0 of the CTA, and updates the value of parameter p_T according to value of variable *param*. Rule *Return* states that whenever a callable automaton emits a `return` event, its translation yields the results to its calling (parent) TA through a synchronization on channel *ret*, with the assignment of result e to shared variable *result*. Finally, rule *Call* is explained via Figure 4.

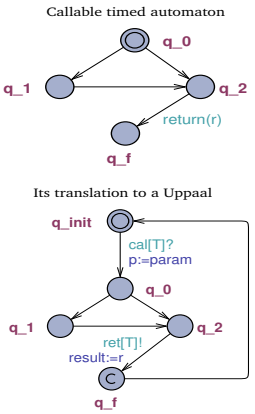

Fig. 5. TA of a CTA.

In the same way, the translation of a network of CTA, defined by a root CTA, a set of template definitions $\mathcal{D}$ and the maximal number of instances associated to each template $\mathcal{I}$ which is supposed to be **bounded**, is a NTA containing the translation of each CTA replicated according to their number of instances.

In the case of dynamic-instantiable CTA (infinite number of instances), for each CTA T we choose a finite number n_T of instances for each infinite number $\mathcal{I}(T)$, then we translate the new CTA model to UPPAAL. Thus, if the number of simultaneously active instances of each T is lower than the corresponding chosen number, the properties of the checked TA model are those of the original CTA model. In order to check that the chosen numbers are sufficient, we use the UPPAAL model-checker to prove that for each T there always exists an instance in its initial state.

Otherwise, we retry with higher values n_T, for each T whose the number of instances has been reached, and redo the checking process. However, such a process may not terminate. A perspective of this section is to provide a tool for inferring automatically the sufficient number of instances for each dynamic-instantiable CTA. Such a tool could be based on the decision procedure for the boundedness of Petri nets.

5 Semantical Model: TTSs

In order to ensure the translation correctness, we define the semantics of both UPPAAL timed automata and callable TA in terms of timed transition systems (TTS). We study then the bisimilarity between the CTA direct semantics and the translation-based one. In fact, we study the bisimilarity between the semantics of CTA composition and that of their translation, defined in a compositional way. To this end, we extend timed transition systems with local and global variables, and review their timed bisimulation relation and associative product, according to [8]. Moreover, we consider the static priority *Committedness*, which is useful to specify that certain behaviors need to be executed atomically, without interleaving of lower priority behaviors from other components. In general, the states of a TTS constitute a proper subset of the set of all valuations of the state variables. This feature is used to model the concept of location invariants in timed automata.

Definition 5 (TTS). *A Timed Transition System over a set of channels* C *is a tuple* $\langle \mathcal{G}, \mathcal{L}, S, s^0, \rightarrow \rangle$ *where* $\mathcal{G}$ *and* $\mathcal{L}$ *are respectively the sets of global and local variables,* $S \subseteq Val(V)$ *is the set of states with* $V = \mathcal{G} \cup \mathcal{L}$*,* $s^0 \in S$ *the initial state and* $\rightarrow \in S \times (C! \cup C? \cup \{\tau\} \cup \Delta) \times \mathbb{B} \times S$ *is the transition relation.* Δ *is the time domain and* $\mathbb{B}$ *states whether or not a transition is committed.*

A state s of a TTS is called committed, denoted $Comm(s)$, if it enables an outgoing committed transition $(s, l, \top, s')$.

Furthermore, a TTS must satisfy a wellformedness condition : in a committed state neither time-passage steps nor uncommitted τ may occur. Thus, time transitions (with labels in Δ) are non committed.

In fact, the state space S will be used to encode the location invariants of timed automata. Here and elsewhere, we write $s \xrightarrow{\lambda,b} s'$ for a transition $\langle s, \lambda, b, s' \rangle \in \rightarrow$ linking the state s to another state s' through an event λ and having the committedness priority b. This former is considered to be false ($\perp$) if absent. Formally, the predicate $Comm$ is defined by:

$$Comm(s) = \begin{cases} \top & \text{If } \exists\ \lambda\ s' \mid s \xrightarrow{\lambda,\top} s' \\ \perp & \text{Otherwise} \end{cases}$$

Through location committedness, certain (lower-priority) behavior are ruled out which may lead to serious reductions in the state space of a model. By now, we define the simulation relation of TTSs [8]. In fact, such a relation is used to show whether a TTS implements another. The simulation relation can be established through the inclusion of traces where, from a common state, we check that each transition of the simulated system can be triggered in the simulating one.

Definition 6 (Timed step simulation). *Given two TTSs T_1 and T_2 having the same set of global variables, we say that a relation $R \subseteq S_1 \times S_2$ is a timed step simulation from T_1 to T_2, provided that $s_1^0 R s_2^0$ and if $s\ R\ r$ then*

- $s\lceil \mathcal{G}_1 = r\lceil \mathcal{G}_2$,
- $\forall u \in Val(\mathcal{G}_1) : s[u] R\ r[u]$,
- *if $Comm(r)$ then $Comm(s)$,*
- *If $s \xrightarrow{\lambda,b} s'$ then either there exists an r' such that $r \xrightarrow{\lambda,b} r'$ and $s' R r'$, or $\lambda = \tau$ and $s' R r$.*

where $s[u]$ states the update of state s according to valuation u. We write $T_1 \preceq T_2$ when there exists a timed step simulation from T_1 to T_2. In fact, such a definition maps each transition of T_1 to a transition of T_2 given that global variables have the same valuations. Accordingly, T_1 and T_2 are bisimilar if there exists a timed step simulation R from T_1 to T_2 such that R^{-1} is a timed step simulation from T_2 to T_1. In order to study the semantics of timed automata composition, we define the product of TTSs, according to [8], which is a partial operation that is only defined when TTSs initial states are compatible, i.e. $s_1^0 \heartsuit s_2^0$.

Definition 7 (Parallel composition of TTSs). *Given two TTSs T_1 and T_2 with $s_1^0 \heartsuit s_2^0$, their parallel composition $T_1 \parallel T_2$ is defined by the tuple $\langle \mathcal{G}, \mathcal{L}, S, s_i^0, \rightarrow \rangle$ where $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2$, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$, $S = \{r \parallel s \mid r \in S_1 \wedge s \in S_2 \wedge r \heartsuit s\}$, $s^0 = s_1^0 \parallel s_2^0$ and $\rightarrow$ is the smallest relation such that:*

$$\frac{r \xrightarrow{\lambda,b}_i r'}{r \parallel s \xrightarrow{\lambda,b} r' \triangleright s}\ Ext \qquad \frac{r \xrightarrow{\tau,b}_i r' \quad Comm(s) \Rightarrow b}{r \parallel s \xrightarrow{\tau,b} r' \triangleright s}\ Tau$$

$$\frac{r \xrightarrow{c!,b}_i r' \quad s[r'] \xrightarrow{c?,b'}_j s' \quad i \neq j \quad Comm(r) \vee Comm(s) \Rightarrow b \vee b'}{r \parallel s \xrightarrow{\tau, b \vee b'} r' \triangleleft s'}\ Sync \qquad \frac{r \xrightarrow{\delta}_i r' \quad s \xrightarrow{\delta}_j s' \quad i \neq j}{r \parallel s \xrightarrow{\delta} r' \parallel s'}\ Time$$

$i,\ j$ range over $\{1, 2\}$ and $b,\ b'$ range over $\mathbb{B}$. The set of variables of the product is simply obtained by the union of both component variables. Moreover, the states, respectively initial states, of the product are obtained by merging the states, respectively initial states, of individual TTSs. The notation $Comm(q) \Rightarrow b$ with $t : s \xrightarrow{\lambda,b}_i s'$ states that t must be committed if there exists another outgoing committed transition from s. Otherwise stated: a transition cannot be hidden by a lower-priority transition.

Rule Ext represents potential synchronizations that the TTS T_i may be willing to engage in with its environment. The committedness of such transitions is not checked because it may be that a compatible committed transition will synchronize with the current transition of T_i making then the resulting transition committed. Rule Tau induces an internal transition of the composition from an internal transition of a component T_i. Rule $Sync$ describes the synchronization of components T_i and T_j on channels $c \in C$ if their labels are compatible, and the input transition is still triggerable according to the valuation associated to the output transition target state r'. The resulting transition, labelled by the internal event τ, is committed if at least one of the involved transitions (output, input) is committed. Hence, a non-committed synchronization may only occur if both components are in uncommitted states. Finally, rule $TIME$ states that a delay δ of the composition may occur when both components perform a delay δ.

Theorem 1 (Associativity). *Let T_1, T_2 and T_3 be TTSs with their initial states pairwise compatible, then $(T_1 \parallel T_2) \parallel T_3 = T_1 \parallel (T_2 \parallel T_3)$.*

In the following, we define the semantics of UPPAAL timed automata through TTS where committed transitions of TTS are those outgoing from TA committed locations.

Definition 8 (TTS semantics of a TA). *Given a global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C \rangle$, the TTS associated to a timed automaton $\langle Q, q^0, K, V^l, Init^l, Inv, \rightarrow_{ta} \rangle$ is defined by $\langle V^g, V^l \cup \{loc\}, S, s^0, \rightarrow \rangle$ where `loc` is a fresh variable with type Q, $W = V^g \cup V^l \cup \{loc\}$, $S = \{v \in Val(W) \mid v \models Inv(v(loc))\}$, $s^0 = Init^g \cup Init^l \cup \{loc \mapsto q^0\}$ and the transition relation is defined by:*

$$\frac{q \xrightarrow{G/\lambda/a}_{ta} q' \quad s(loc) = q \quad s \models G \quad b \Leftrightarrow (q \in K)}{s \xrightarrow{\lambda,b} a(s \lhd \{loc \mapsto q'\})} \; Act \qquad \frac{s(loc) \notin K}{s \xrightarrow{\delta,\perp} s \oplus \delta} \; Time$$

We have introduced a new local variable `loc` to state the TA current location. Each state of the TTS corresponds to a valuation of TA variables where the invariant of the corresponding location holds. Moreover, the TTS transitions are inferred from the transitions and locations of TA. In fact, rule *Act* states that to each TA transition, we associate a TTS transition if the current location `loc` corresponds to the source location q of TA transition, and the TTS current state s satisfies the guard G of the TA transition. Through rule $Time$, we associate to each non-committed location of TA a TTS non-committed transition. The former adds an amount δ to all clock variables. One may distinguish that $Time$ transitions do not update local states and non-clock variables.

6 Semantical Interpretations

By now, we define the semantics of callable timed automata instantiation in terms of TTS. In fact, such a semantics considers a callable automaton (template) together with its instances. Mainly, we distinguish two different instantiations: static and dynamic. In fact, the static instantiation corresponds to implement each callable template through a finite (constant) number of instances, may be initially created, whereas the dynamic instantiation of a callable automaton consists of creating a (possibly infinite) set of instances on the fly when executing the system. Each instantiation mechanism is suitable for a given kind of applications, whereby the modelling of systems becomes much more natural. Let us introduce the following elements:

- We extend the set of locations by introducing, for each calling transition t a new location $\bar{t}$. Such a location will be used to wait for a return of the call made over transition t.
- In order to distinguish between different instances of the same template, a fresh identifier Id is assigned to each instance.
- We introduce a new local variable **templ** such that, an instance Id is an instantiation of the template T if $Id.\mathbf{templ} = T$.
- We have also introduced a new local variable **ParId** in order to identify for whom (parent identifier) an instance (Id) performs a return. In fact, the variable $Id.\mathbf{ParId}$ stores the identifier of the current caller of Id.

– The local variables of instance Id are renamed by prefixing each one by the identifier Id.
– The notation $[\![e]\!]_s^{Id}$ states the valuation of expression e according to state s, where the template local variables occurring in e are replaced by the corresponding local variables of instance Id.

Definition 9 (CTA instantiation semantics). *Given a global context $\mathcal{C} = \langle \Sigma, V^g, Init^g, C\rangle$, the instantiation semantics of the callable timed automaton $\langle T(p_T), Q, q^0, F, V^l, Init^l, Inv, \rightarrow_T\rangle$ is defined by the TTS $\langle V^g, Id.V^l \cup Id.\{\mathbf{loc}, \mathbf{templ}, \mathbf{ParId}\}, S, s^0, \rightarrow\rangle$ [4] over the set of channels C where $Id = fresh(\emptyset)$ is the identifier of the initial instance, $S = \{s \in Val(W) \mid s \models Inv(s(\mathbf{loc}_T))\}$, $s^0 = Init^g \cup Init^l \cup \{Id.\mathbf{loc} \mapsto q^0\}$, $W = V^g \cup \bigcup_{i\in\mathcal{I}(T)}\{Id_i.V^l \cup \{Id_i.\mathbf{loc}, Id_i.\mathbf{templ}, Id_i.\mathbf{ParId}\}$ and $\rightarrow$ is the smallest relation such that:*

$$\frac{q \xrightarrow{G/\lambda/a}_T \quad q' \quad s(Id.\mathbf{templ}) = T \quad s(Id.\mathbf{loc}) = q \quad s \models G}{s \xrightarrow{\lambda,\perp} a_{Id}(s \lhd \{Id.\mathbf{loc} \mapsto q'\})}\ Act \qquad \frac{s(Id.\mathbf{loc}) \overset{return}{\nrightarrow}}{s \xrightarrow{\delta,\perp} s \oplus \delta}\ Time$$

$$\frac{\begin{array}{c} t : q \xrightarrow{G/v:=call\ T'(e)/a}_T \quad q' \quad s(Id.\mathbf{templ}) = T \quad s(Id.\mathbf{loc}) = q \\ s \models G \quad Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T') \quad Id' := fresh(s) \end{array}}{s \xrightarrow{\tau,\perp} s \lhd \{Id.\mathbf{loc} \mapsto \bar{t}\ \} \parallel Init\_Id' \parallel fInit^l(Id', T')}\ Call$$

$$\frac{s(Id.\mathbf{loc}) \in \mathcal{D}(s(Id.\mathbf{templ})).F}{s \xrightarrow{\tau/\perp} s/Id}\ Destroy$$

$$\frac{\begin{array}{c} s(Id.\mathbf{loc}) = \bar{t} \quad s(Id'.\mathbf{loc}) = q \quad s(Id'.\mathbf{templ}) = T \\ q \xrightarrow{G/return\ e/a}_T \quad q' \quad s \models G \quad s(Id'.\mathbf{ParId}) = Id \end{array}}{s \xrightarrow{\tau,\top} t.a_{Id}(a_{Id'}(s \lhd \{Id.\mathbf{loc} \mapsto t.q', Id'.\mathbf{loc} \mapsto q', t.v \mapsto [\![e]\!]_s^{Id'}\}))}\ Return$$

where $Init\_Id' = \{Id'.p_{T'} \mapsto [\![e]\!]_s^{Id}, Id'.\mathbf{ParId} \mapsto Id, Id'.\mathbf{templ} \mapsto T', Id'.\mathbf{loc} \mapsto \mathcal{D}(T').q^0\}$ is the initialization of parameters and newly created variables of instance Id' (rule Call), and the function $fInit^l(Id', T') = \|_{v\in\mathcal{D}(T').V^l} \{Id'.v \mapsto \mathcal{D}(T').Init^l(v)\}$ is the initialization of the instance original local variables according to the initial valuation $Init^l$ of its template $\mathcal{D}(T')$ identified by T'.

The semantics of the CTA instantiation is given through the former definition together with the TTS product. It consists of compiling dynamically CTA instances to TTSs and computing simultaneously the parallel product of these TTSs. In fact, the semantics of a CTA T creates the first instance of T. Such an instance is recognizable by a fresh identifier $Id = fresh(\emptyset)$. The set of local variables of the underlying TTS corresponds to the union of the local variables of all instances of T that can be created according to the maximal number of instances $\mathcal{I}(T)$ i.e., $(\bigcup_{i\in\mathcal{I}(T)}\{Id_i.V^l)$, together with the newly introduced variables $(Id_i.\mathbf{loc}, Id_i.\mathbf{templ}, Id_i.\mathbf{ParId})$. Moreover, TTS states are partial functions where only variables of created instances are valued.

About transitions, rule *Act* states a non-calling transition of an instance Id of template T ($Id.\mathbf{templ} = T$) if the current location of Id corresponds to q. Such a transition is enabled if the current source state s satisfies the guard G, and consists of updating local and global variables according to action a_{Id}, with a jump to location q'. The update action a_{Id} is a rewriting of action a where the local variables of template T, occurring in a are replaced by that of instance Id.

Rule *Time* corresponds to a delay of an instance Id from state s. The notation $q \overset{return}{\nrightarrow}$ states the absence of outgoing transitions labelled with a `return` event, from location q. Implicitly, return events have priority over others. Thus, we do not allow delays from locations having outgoing transitions labelled by a return. Such a restriction is useful to enable instances unlocking their callers once they reach a state having an outgoing return.

After checking that the current location **loc** of an instance Id of template T corresponds to location q, the current state s satisfies the guard G, and the cardinality of the current set of the

[4] $Id.E = \{Id.e \mid e \in E\}$ consists of prefixing each variables $e \in E$ by the identifier Id of a CTA instance. Such a renaming is used to distinguish between variables of different instances, in particular between instances of the same template where variables have the same original names.

callee template (T') instances does not cross up the maximal number allowed for this template i.e., $Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T')$, rule *Call* creates a new instance Id' of the callee template T'. Such a newly created instance is concurrently run with its calling instance Id of template T, and has the parent (calling) instance identifier $\mathbf{ParId} = Id$. Without executing the update action a, the calling instance Id moves to an intermediate location $\bar{t}$ waiting for a return. The update action a is stored in location $\bar{t}$, and will be applied after assigning the result returned by Id' to variable v. On its creation, the instance Id' initializes its parameter and its new local variables ($\mathbf{loc}, \mathbf{templ}, \mathbf{ParId}$) according to $Init\_Id'$, and also initializes its original local variables V^l according to $fInit^l$.

Rule *Destroy* states that an instance Id will be destroyed when it reaches a final location. Such a destruction consists of removing the variables and locations of Id from the system state.

Rule *Return* specifies how an instance Id' of template T performs a return, for its calling instance Id waiting on an intermediate location $\bar{t}$. In fact, after ensuring for whom ($Id'.\mathbf{ParId} = Id$) the return action should be made, the instance Id' yields the result expression e, evaluated to $[\![e]\!]_s^{Id'}$ according to the valuation of state s, to its calling (parent) instance Id. The former joins the target location q', stored in $t.q'$, of its calling transition t after the reception of the returned value $t.v = [\![e]\!]_s^{Id'}$. Through such a transition, from location $\bar{t}$ to $t.q'$, the update action $t.a$ [5] of the transition t, originally performing the call of Id', is applied after the execution of the local action $a_{Id'}$ of the returning transition and the assignment of $[\![e]\!]_s^{Id'}$ to local variable v of Id.

Remark. One may remark that we have unified both static and dynamic instantiations in one semantics. The difference between both instantiation semantics can be clearly distinguished over the following condition $Card\{Id \mid Id.\mathbf{loc} \in dom(s)\} < \mathcal{I}(T')$ of rule *Call*. In fact, in the dynamic instantiation we can create an infinite set of instances because the above condition is always satisfied, i.e., the maximal number ($\mathcal{I}(T') = \infty$) of instances to be created cannot be reached. Whereas in the static instantiation semantics, we are allowed to create a new instance if the number of the current active instances does not cross up the maximal (finite) bound $\mathcal{I}(T')$.

Theorem 2 (Subprogram call safety). *An instance of a subprogram CTA T is either in its own initial location q^0 or there exists a unique component which is in a waiting location $\bar{t}$ associated to a call to T. Formally, the property P such that $P(s) \equiv \forall Id\ s(Id.\mathbf{loc}) \neq \mathcal{D}(s(Id.\mathbf{templ})).q^0 \Rightarrow \exists! Id'\ \exists! t\ s(Id'.\mathbf{loc}) = \bar{t} \wedge s(Id.\mathbf{ParId}) = Id'$ is an invariant of the system.*

Theorem 3 (Instantiation semantics and translation). *The semantics of a system of CTA and TA, defined by the product of TTS associated to its individual components and that based on the translation of CTA to TA are bisimilar.*

Theorem 4 (Liveness). *For an instance Id, a location q with a call as unique outgoing transition which is locally enabled and such that time elapse is bounded [6], then the call is eventually accepted. Formally, for each calling location q, we have: $(s(Id.\mathbf{loc}) = q) \wedge G \rightsquigarrow \exists Id'\ \exists s, (Id'.\ \mathbf{ParId} = Id) \wedge (s(Id'.\mathbf{loc}) = \mathcal{D}(s(Id'.\mathbf{templ})).q^0)$, where $\rightsquigarrow$ is the UPPAAL **Leads to** operator.*

Theorem 5. *If the NTA translation of a CTA system has always a free instance for each template i.e., $\forall s\ \forall T\ \bigvee_i (s(Id_i.\mathbf{loc}) = \mathcal{D}(T).q^0 \mid Id_i.\mathbf{templ} = T)$, then the TTS associated to the NTA translation and the TTS associated to the dynamic instantiation semantics of the CTA system are bisimilar.*

7 Implementation and Experiments

In order to make our extension profitable, we have designed a PYTHON SCRIPT program converting callable timed automata systems to UPPAAL NTA. In fact, our converter uploads an XML file designed using UPPAAL graphical editor, as an input where the interface of each CTA states a

[5] The notation $t.v$ refers to variable v occurring in the left side of the calling transition t label. Similarly, $t.a$ is the update action of transition t.

[6] In UPPAAL, such a property can be enforced by assigning $clock \leq B$ as an invariant to this location.

finite number of instances. After performing a deeper analysis of callable automata syntax, in particular template interfaces, *call* and *return* transitions, the converter generates the corresponding UPPAAL NTA format, written in a new XML file that will be then reloaded in the UPPAAL tool, as an ordinary system to be analyzed and checked.

The interface of each callable TA is given by the number of instances, the type of *return*, the name of template and the set of parameters. That is an example of a template signature with 3 instances, a void return type, the template name *Use_Case* and a set of parameters.

```
3;void Use_Case(int ind, int arrival_time, int memory_usb)
```

After replicating template instances in the system declaration, according to the template signatures, the source XML file will be explored template by template and transition by transition. For each callable template occurring in a calling transition, both that transition and the callee template will be translated as stated in Section 4.3.

The converter translates each callable TA, occurring in a calling transition, to a UPPAAL TA by adding an extra synchronizing transition (from q_{init} to q^0) to activate the automaton, another transition (from a final location to q_{init}) to get the instances available for other calls after reaching final locations, a shared variable to hold the name of the current calling template, and splitting each calling transition to a sequence of call and return transitions as shown in Figure 5.

In the case of dynamic interpretation where templates have an infinite number of instances, we infer a finite (sufficient) number simulating the infinite bound of each CTA instantiation as stated in Section 4.3. Then, for each call, we reuse an existing instance instead of creating a new one.

As an application, we have remodeled the Océ printer system using callable timed automata, where each job (use-case) is modeled by a callable automaton. We consider 6 templates where only 3 are callable (3 CTA). We have also introduced another template USER to manage the system. The USER triggers dynamically different jobs at different respective dates. We have successfully translated the new model of the Océ system to an UPPAAL NTA, and also proceeded on the verification of the property stating that all jobs reach their final locations $DONE$. Such a property is satisfied by both original model [19] and the translation. The Oce protocol with CTA is available on `http://www.irit.fr/~Abdeldjalil.Boudjadar/EXEMPLES/Oce/oce-model.xml`. The corresponding translation is also available on `http://www.irit.fr/~Abdeldjalil.Boudjadar/EXEMPLES/Oce/oce-translation.xml`.

8 Conclusion and Perspectives

Throughout this paper, we have introduced and formalized the concept of callable timed automata for the modelling and structuring of real-time and interactive systems. Such a syntactical extension can be interpreted in different semantical ways: static and dynamic. In the dynamic case, we propose to reuse UPPAAL by giving bounds to the numbers of simultaneously active instances of templates. Such a technique can be interesting for the study of population protocols [3] when the population happens to be bounded.

Thanks to the UPPAAL translation, we have validated our proposal through an UPPAAL "plugin".

As a challenging continuation of our work, we envision to consider existing work related to Petri nets as well as to logics that take into account CALL and RETURN like CaRet [2] and Spade [24]. Moreover, we have in mind model checking support for architecture description languages, where subprograms with their own resources are considered [16]. Another point worth studying is related to compositionality. It would be interesting to study how the results of [8] and [11] could be extended to the context of CTA.

References

1. R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP '90, pages 322–335, 1990.
2. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *TACAS'04*, LNCS, volume 2988, pages 467–481. Springer, 2004.

3. J. Aspnes and E. Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98 – 117, 2007.
4. C. Baier, N. Bertrand, P. Bouyer, and T. Brihaye. When are timed automata determinizable? In *ICALP'09*, pages 43–54, 2009.
5. E. Beffara. Functions as proofs as processes. *CoRR*, abs/1107.4160, 2011.
6. G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal 4.0. Department of computer science, Aalborg university, 2006.
7. J. Berendsen and F. Vaandrager. Parallel composition in a paper of Jensen, Larsen and Skou is not associative. technical note available at http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html. 2007.
8. J. Berendsen and F. Vaandrager. Compositional Abstraction in Real-Time Model Checking. In *6th International Conference FORMATS'08*, pages 233–249, 2008.
9. J. Berendsen and F. Vaandrager. Parallel composition in a paper by de Alfaro e.a. is not associative. technical note available at http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html. 2008.
10. B. Berthomieu, P. O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. In *Intl Journal of Production Research*, page Vol 42, 2004.
11. J. P. Bodeveix, A. Boudjadar, and M. Filali. An alternative definition for timed automata composition. In *ATVA'11*, pages 105–119. LNCS 6996, 2011.
12. G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *TAPSOFT, Vol.1'89*, pages 149–161, 1989.
13. A. Burns and A. Wellings. *Concurrency in Ada (2nd edition)*. Cambridge university press edition, 1998.
14. L. De Alfaro, L. Dias da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *Proceedings of 5th international workshop FroCoS'05, September 2005*, pages 81–105, 2005.
15. U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical report, Computer Science Department, University of Aarhus, 1986.
16. P. H. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis and design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop*, pages 1–10, 2003.
17. J. Håkansson and P. Pettersson. Partial order reduction for verification of real-time components. In *Proc of FORMATS'07*, volume 4763 of *LNCS*. Springer–Verlag, 2007.
18. G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
19. G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, and L. Somers. Formal modeling and scheduling of datapaths of digital document printers. In *FORMATS'08*, pages 169 – 186, 2008.
20. H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up Uppaal:Automatic Verification of Real-Time Systems using Compositionality and Abstraction. In *FTRTFT'00*, pages 19–30. LNCS, 2000.
21. K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. In *Journal on Software Tools for Technology Transfert*, 1997.
22. R. Milner. Functions as processes. In *ICALP'90*, pages 167–180, 1990.
23. F. Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE '89*, pages 357–373. Springer-Verlag, 1989.
24. G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *Proccedings of CAV'07*, pages 254–257, 2007.
25. B. Thomsen. A calculus of higher order communicating systems. In *Proceedings of the 16th ACM conference POPL '89*, pages 143–154. ACM, 1989.
26. B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. *CoRR*, Paper available on http://ctp.di.fct.unl.pt/∼lcaires/papers/, 2011.
27. A. Trivedi and D. Wojtczak. Recursive timed automata. In *Proceedings of ATVA'10*, pages 306–324, Berlin, Heidelberg, 2010. Springer-Verlag.
28. F. Warn. Red: Model-checker for timed automata with clock-restriction diagram. In *Workshop on Real-time Tools*, 2001.
29. S. Yovine. Kronos: A verification tool for real-time systems. In *Journal of Software Tools for Technology Transfer*, pages 123–133, 1997.