TIMED BEHAVIORAL TYPING *

Arnaud Bailly and Elie Najm
École Nationale Supérieure des Télécommunications, Paris, France.
{arnaud.bailly, elie.najm}@infres.enst.fr

Abstract We discuss a formal language introduced by us to describe timed distributed computations. Our language features a type language and a calculus of timed actors. The type language allows for the specification of required and provided QoS features that need to be fulfilled by the communicating actors and by their underlying infrastructure. This language is contract-oriented. This language is contract-oriented. A contract is a written document that engages peered communicating endpoints. It contains a specification of the observed behavior of each endpoint bound by the contract. Those endpoint behaviors are written in a formalism close to Alur and Dill's timed automata. We show that deciding whether peered endpoints are well-behaved regarding each other so that they do not lose any message is statically decidable. We then propose a method to verify that computational configurations possess the same property. Our method is based on behavioral typing, a static analysis method in which types denote behaviors: well-behaved configurations have correct types. We obtain a compositional method for the verification of QoS-aware object-based distributed programs.

Keywords: QoS Contracts, Object Binding, Safety Through Behavioral Typing, Multimedia.

Introduction

Open distributed computing, where people demand to dynamically access and configure the services they need, has become a household concept. A lively set of concerns rise up today in the particular environment where real-time properties are wished to be expressed, verified, (re-)negotiated, and enforced. Multimedia systems [10], active networks [26] and QoS-enforcing middleware [24] are examples of applicative fields where the timeliness properties of specifications and programs are especially under the spotlights.

To establish *contracts* between processing entities, as described in ODP [18] for example, is now an approach widely adopted for the conception and study of systems in the fields given above [1, 11, 12]. We make a proposal for the formal specification and verification of Quality of Service (QoS) contracts. To express QoS properties we propose that contracts be based on the formalism of timed automata as defined by Alur and Dill [5]. Such contracts are said to be *behavioral*, in the sense that they declare exactly what sequences of requests a client may perform to access the service, and what sequences of replies the server may give to satisfy each request. In our case, those sequence may contain timing constraints. A service set under a behavioral contract is said *non-uniform*: the set of its possible invocations vary over time.

We propose a timed actor language to describe dynamically reconfigurable computations where actors require the use of timed behavioral contracts when invoking or performing services with QoS properties. In our language, only couples of actors may conclude contracts, no n-way multi-party contracts are not allowed. To establish a contract is then referred to as *binding* its two endpoints over some communication channel [10]. A contract

*Research supported by France TelecomT Research & Development and the RNRT MARVEL project

specification therefore encloses two automata: each automaton describes the apparent behavior of one of the engaged endpoints. This apparent behavior is called the *role* of the endpoint in the contract or its *behavioral interface*.

A commonly addressed verification problem for the peered interfaces of a contract is the *compatibility* problem [12, 11, 23]. It consists in checking that endpoints only play complementary roles in the interaction that may occur under the contract. If roles are compatible, endpoints should not retain any missed synchronization due to inconsistencies in the protocol they use to communicate. A contract enclosing compatible roles is said to be *consistent*. Our first result is that there is a way to statically check the consistency of our contracts.

To have a decision procedure answering whether configurations of communicating processes employing QoS contracts are well-behaved or not now clearly becomes a central issue. When processes are described using algebraically-founded object-oriented languages as is our timed actor language, this verification problem appears to be solvable by behavioral typing [17, 16, 21, 19, 20]. This method advocates for the use of superiorly structured *behavioral types*. A behavioral type *exactly* corresponds to the specification of a timed behavioral interface. The type of a process is therefore a set of behavioral interface types. The typing procedure then consists in verifying that the process behavior and its behavioral type are equivalent modulo some known relation (eg. bisimilarity [15]). If contracts are written by the programmer, the approach is named *explicit* typing. The dual problem, more difficult, is to infer a principal type for each process. In both cases, the sought property is that *no missed synchronization* occur in a well-behaved (ie. well-typed) process configuration. We propose an explicit behavioral type system based on our QoS contracts.

The remainder of this paper is organized as follows. In the next section, we present our language for timed actors and is operational semantics. In Section 3, we recall some necessary results on timed automata. The contract language and semantics are described in Section 4, as well as the contract consistency relation. The behavioral typing system is defined in Section 5, before a short example and discussions in Section 4.6, and our conclusion.

1. The Timed Actor Language *ArtOC*

We now present an actor-based setting for real-time contracts. Actors [4] are self-evolving objects that communicate by sending signals (usually called *messages*) through asynchronous channels. An actor may *send* a message (following in a *non-blocking* policy), wait for a message, adopt the behavior (*ie. become*) another actor, or *spawn* another actor. As our actors have to deal with time, we upgrade them with timers. A timer is a variable evolving with time, as do timed automata clocks for example. A timer allows to specify timeouts corresponding to infinite or finite, deterministic or non-deterministic, delays. This yields the power to express also *non-blocking* and *finitely-blocking* message receptions. Our actor communicate through interfaces, that are names which can be passed around among actors as in the π-calculus [15]. The communication topology may therefore be modified.

1.1. Syntax for Timed Actors

Figure 1 detail the *ArtOC* syntax. In that figure, C stands for a configuration, Dec for a set of declarations, B for a behavior, A for a named actor, p for a formal parameter, ρ for a role, x for a timer, G for a guard on one timer x, u for an interface name, T for a type name, and R for the reception on a given interface. In that context, $\tilde{u}$ represents the list of names $u_1 \dots u_k$. Finally, we will also use $\mu ::= ! \mid ?$ to name the action prefix of sending or receiving a signal.

A set of declarations Dec forms a name context for any configuration C. C is a set of behaviors put in parallel. An actor declaration is written $A(\tilde{p}) = B$, which associates

C	::=	$B \mid C \mid C$	G	::=	$\delta \leq x \leq \nu$
Dec	::=	$A(\tilde{p}) = B$	B	::=	∞
	$\mid$	$Dec\ Dec$		$\mid$	**new** $u : T$ **in** B
p	::=	$u : \rho T$		$\mid$	**become** $A(\tilde{u})$
ρ	::=	$! \mid ? \mid !! \mid ??$		$\mid$	**spawn** $A(\tilde{u}) > B$
ν	::=	$\infty \mid \delta$		$\mid$	$!v.s(\tilde{u}) > B$
δ	::=	$Integer \mid \delta + \delta \mid \delta - \delta$		$\mid$	**timer** $x = \delta$ **in** B
R	::=	$?u\ [s_1(\tilde{p}_1) > B_1 \ldots$		$\mid$	**if** G **then** B_t **else** B_f
		$\ldots s_n(\tilde{p}_n) > B_n]$		$\mid$	**timeout**$(G) \sum_{i=1}^{n} R_i > B$
				$\mid$	**timeout**$(G) > B$

Figure 1. The Syntax of *ArtOC*

name A to behavior B with formal parameters $\tilde{p}$. The syntax $u : \rho T$ associates role ρ and type T to interface name u. Each interface is said to play a particular role regarding a given service. Roles can be *client* (noted !!) or *server* (noted ??), *source* (noted !) or *target* (noted ?). The source and target roles are the ones of endpoints bound by a QoS contracts. We will describe their features in Section 4; it is just to be known that they form exclusive interaction couples, and that they can both emit and receive signals over reliable FIFO channels with fixed minimum and maximum transmission delays. Interfaces with client or server roles correspond to *uniform*, *untimed* services. They are "traditional" IDL-like interfaces, defined as a set of signals with arity and type for each of their formal parameters. A server can then concurrently receive messages from *many* clients. In the following we will often identify roles and interfaces, writing "role ρ" instead of "interface of role ρ". The distinction should usually be clear from context. In any case, the type of an interface will be described as a timed automaton.

We now show the possible actions for the body of an actor, the so-called *behaviors*. **new** $u : T$ **in** B creates two new interfaces named u of type T and then behaves like B. According to the specification associated to type name T, either a client and server roles or a source and a target roles are created. An actor can behave as another actor: **become** $A(\tilde{u})$ behaves like A with interfaces $\tilde{u}$. **spawn** $A(\tilde{u}) > B$ creates a new actor that behaves as A, passing the effective parameters $\tilde{u}$ as arguments; A is then executed in parallel with B. $!v.s(\tilde{u}) > B$ emits a signal s through interface v with arguments $\tilde{u}$ and then behaves like B. A timer is simply introduced with the instruction **timer** $x = \delta$ **in** B. **if** G **then** P_t **else** P_f assesses guard G, then executes P_t if it is true, or P_f if it is false. A guard simply checks if the value of a timer is included in some interval.

A reception may feature a choice between many interfaces: **timeout**$(G) \sum_{i=1}^{n} R_i > B$ waits until G becomes true before executing B, but it can be interrupted before that whenever a signal delivery in one R_i occurs. Each R_i indeed comprises the name of an interface u to listen to, and the set of expected signals along with triggered behavior when the reception occurs. Finally, **timeout**$(G) > B$ waits until G becomes true, and then behaves like B.

1.2. Binding Rules

The binding rules describe how references are forged and how they can be sent away to set a service up. For contracts, the binding is said *explicit*: both roles are created by the same actor, and this actor establishes the binding by sending the target role. Once it has done so, neither role can be given to another actor. This scenario is usually referred to as *first-party* binding. The binding rules for uniform services are slightly different: both client and server roles are created by the same actor, but it is then free to send both roles away.

That is, an actor does not lose the capacity to use a client role when this latter is sent away; client roles are always duplicated, and any number of them can concurrently send signals to one server role. However, only one server role may exist for a uniform service, and when an actor sends away a server role, it loses the ability to receive signals through it. This will be reflected in the upcoming typing rules.

2. Operational Semantics

We provide a classical SOS as a congruence relation on terms and a set of rewriting rules. In addition of well-formed $\mathcal{A}rtOC$ terms, evolving configurations must enclose messages sent through contract roles. They are tagged by their full name and by the time they already spent in the channel, along with the time bounds of that very channel, as $\langle !u.s(\tilde{v}), 0\rangle^{\tau}_{\tau_0}$.

we make several simplifications, for the sake of space and simplicity. The following asserts are therefore respected by the reduction rules, but not represented in them:

- when a the two roles of a contract are created, they can let time pass infinitely without their clock changing values. When the target role is sent away, the source is activated. The target becomes active upon reception (see Section 4).
- Target roles of contracts have to be sent through uniform or behavioral services that rely on the same network characteristics τ_0 and τ.
- Signals are put in channels, of which the FIFO structure is not represented (see Section 4.3 for a formal definition).

2.1. Structural Congruence

We use lexical scoping with interface names and time variables as sorts. The interface names contained in $\tilde{p}, \tilde{p}_1, \ldots \tilde{p}_n$ are respectively bound in $B, B_1, \ldots B_n$ for terms "**actor** $A(\tilde{p}) = B$" and "$u?$**when** $s_1(\tilde{p}_1) \rightarrow B_1 \ldots$ **when** $s_n(\tilde{p}_n) \rightarrow B_n$". The interface name u is bound in B for **new** $u : T$ **in** B. The timer name x is bound in B for **timer** $x = \delta > B$. All the other names are free. We will note $C[v/u]$ the substitution by v of all free occurrences of u in C. Rules for structural congruence can be found in Figure 1. The first two rules state that the choice operator between receptions is com-

$$Recep_1 + Recep_2 \equiv Recep_2 + Recep_1$$
$$(Recep_1 + Recep_2) + Recep_3 \equiv Recep_1 + (Recep_2 + Recep_3)$$
$$C_1 \mid C_2 \equiv C_2 \mid C_1 \,, (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3) \,, C \mid \infty \equiv C$$
$$\textbf{new } u_1 \textbf{ in } (\textbf{new } u_2 \textbf{ in } B) \equiv \textbf{new } u_2 \textbf{ in } (\textbf{new } u_1 \textbf{ in } B)$$
$$\textbf{timer } x = \delta \textbf{ in } (\textbf{new } u_2 \textbf{ in } B) \equiv \textbf{new } u_2 \textbf{ in } (\textbf{timer } x = \delta \textbf{ in } B)$$
$$\textbf{timer } x = \delta \textbf{ in } (\textbf{timer } x = \delta' \textbf{ in } B) \equiv \textbf{timer } x = \delta' \textbf{ in } B$$
$$\textbf{timer } x = \delta \textbf{ in } (\textbf{timer } y = \delta' \textbf{ in } B) \equiv \textbf{timer } y = \delta' \textbf{ in } (\textbf{timer } x = \delta \textbf{ in } B) \text{ if } x \neq y$$
$$\textbf{new } u \textbf{ in } B_1 \equiv \textbf{new } v \textbf{ in } B_2 \textbf{ if } B_1[w/u] = B_2[w/v] \textbf{ for some fresh name } w$$

Table 1. Structural Congruence Rules

mutative and associative. The parallel operator is also commutative and associative. The infinitely waiting behavior (∞) is the neutral element for the parallel operator. The order of the introduction of the roles in meaningless, while the most recent values of timers always prevail. Configurations identical modulo renaming of bound variables are equivalent.

2.2. Reduction Rules

Interactions are always asynchronous. If u is either a source, target, or client role we have:

$$!u.s(\tilde{v}) > B \xrightarrow{!u.s(\tilde{v})} B \mid \langle !u.s(\tilde{v}), 0\rangle^{\tau}_{\tau_0}$$

A signal that has passed δ time units in a channel may be delivered iff $\tau_0 \leq \delta \leq \tau_0 + \tau$ and there is no other message $\langle !u.s'(\tilde{v}', \delta')\rangle$ with $\delta' > \delta$. It then rewrites:

$$?u[s(\tilde{w}) > B' + \Sigma s_i(\tilde{u}_i) > B_i] \mid \langle !u.s(\tilde{v}), \delta\rangle^{\tau}_{\tau_0} \xrightarrow{?u.s(\tilde{v})} B'[\tilde{v}/\tilde{w}]$$

For a guard G, $\langle x = \delta\rangle \not\models G$ if G does not constrain clock x. And If G constrains x:

$$\frac{\langle x = \delta\rangle \models G}{\textbf{timer } x = \delta \textbf{ in if } G \textbf{ then } B_t \textbf{ else } B_f \to B_t} \qquad \frac{\langle x = \delta\rangle \models \neg\, G}{\textbf{timer } x = \delta \textbf{ in if } G \textbf{ then } B_t \textbf{ else } B_f \to B_f}$$

The instantiation replaces an actor name by its behavior, and spawn creates a new process:

$$\frac{A(\tilde{u} : \tilde{\rho}\tilde{T}) \stackrel{dcl}{=} B}{\textbf{become } A(\tilde{v}) \to B[\tilde{v}/\tilde{u}]} \qquad \frac{A(\tilde{u} : \tilde{\rho}\tilde{T}) \stackrel{dcl}{=} B}{\textbf{spawn } A(\tilde{v}) > B' \to B' \mid B[\tilde{v}/\tilde{u}]}$$

Parallel configurations have independent discrete evolutions and common time evolution:

$$\frac{C_1 \xrightarrow{\mu u.s(\tilde{v})} C_1'}{C_1|C_2 \xrightarrow{\mu u.s(\tilde{v})} C_1'|C_2} \qquad \frac{C_1 \xrightarrow{\delta} C_1' \quad C_2 \xrightarrow{\delta} C_2'}{C_1|C_2 \xrightarrow{\delta} C_1'|C_2'}$$

A role through which a message is sent must produce the same transition as the sending behavior, whereas any other role remain unchanged (that is we have $u \neq w$ as a side condition in the right rule). Timers are not influence by signal emission/reception:

$$\frac{B \xrightarrow{\mu u.s(\tilde{v})} B' \quad T \xrightarrow{\mu s(\tilde{p})} T'}{\textbf{new } u : T \textbf{ in } B \xrightarrow{\mu u.s(\tilde{v})} \textbf{new } u : T' \textbf{ in } B'} \qquad \frac{B \xrightarrow{\mu u.s(\tilde{v})} B'}{\textbf{new } w : T \textbf{ in } B \xrightarrow{\mu u.s(\tilde{v})} \textbf{new } w : T \textbf{ in } B'}$$

$$\frac{B \xrightarrow{\mu u.s(\tilde{v})} B}{\textbf{timer } x = \delta \textbf{ in } B \xrightarrow{\mu u.s(\tilde{v})} \textbf{timer } x = \delta \textbf{ in } B'}$$

For any $\delta > 0$, role behavior temper time-passing transitions, but timers do not:

$$\frac{B \xrightarrow{\delta} B' \quad T \xrightarrow{\delta} T'}{\textbf{new } u : T \textbf{ in } B \xrightarrow{\delta} \textbf{new } u : T' \textbf{ in } B'} \qquad \frac{B \xrightarrow{\delta} B}{\textbf{timer } x = \delta' \textbf{ in } B \xrightarrow{\delta} \textbf{timer } x = \delta' + \delta \textbf{ in } B'}$$

Time passing transitions carry a delay $\delta > 0$. Infinitely waiting can be done through the ∞ behavior: $\infty \xrightarrow{\delta} \infty$ unconditionally. Oppositely timeouts are conditioned:

$$\frac{\begin{array}{c}((\langle x = \delta'\rangle \not\models G) \wedge (\langle x = \delta + \delta'\rangle \not\models G) \wedge (\forall \delta'' \leq \delta,\ \langle x = \delta' + \delta''\rangle \not\models G)) \vee \\ ((\langle x = \delta'\rangle \not\models G) \wedge (\langle x = \delta + \delta'\rangle \models G)) \vee \\ ((\langle x = \delta'\rangle \models G) \wedge (\langle x = \delta + \delta'\rangle \models G))\end{array}}{\textbf{timer } x = \delta' \textbf{ in timeout}(G) \xrightarrow{\delta} \textbf{timer } x = \delta + \delta' \textbf{ in timeout}(G)}$$

When its guard is true, a timeout can call its continuation:

$$\frac{\langle x = \delta \rangle \vDash G}{\textbf{timer } x = \delta \textbf{ in timeout}(G) \sum_{i=1}^{n} R_i > B \rightarrow \textbf{timer } x = \delta \textbf{ in } B} \qquad \frac{\langle x = \delta \rangle \vDash G}{\textbf{timer } x = \delta \textbf{ in timeout}(G) > B \rightarrow \textbf{timer } x = \delta \textbf{ in } B}$$

Silent and unlabelled transitions can be performed without restriction:

$$\frac{T \xrightarrow{\epsilon} T'}{\textbf{new } u : T \textbf{ in } B \rightarrow \textbf{new } u : T' \textbf{ in } B} \qquad \frac{B \rightarrow B'}{\textbf{new } u : T \textbf{ in } B \rightarrow \textbf{new } u : T \textbf{ in } B'}$$

$$\frac{B \rightarrow B'}{\textbf{timer } x = \delta \textbf{ in } B \rightarrow \textbf{timer } x = \delta \textbf{ in } B'}$$

Finally, congruent configurations behave identically:

$$\frac{C_1 \equiv C_1' \xrightarrow{\mu u.s(\tilde{v})} C_2 \equiv C_2'}{C_1 \xrightarrow{\mu u.s(\tilde{v})} C_2}$$

3. Essential Recalls on Timed Automata

3.1. Timed Automata Construction

We start by giving elementary definitions, similar to those in [5]. Let $\mathbb{X}$ be an at most countable set of time-bound variables called *clocks*. Clocks and timers are behave identically. The assumed time domain is the positive reals $\mathbb{R}_+$. A clock valuation $v : \mathbb{X} \rightarrow \mathbb{R}_+^{|\mathbb{X}|}$ therefore maps each clock of $\mathbb{X}$ to a value in $\mathbb{R}_+$. Given a clock valuation v and a value $\tau \in \mathbb{R}_+$, we note $v + \tau$ the valuation v' such that $v'(x) = v(x) + \tau$ for any clock $x \in \mathbb{X}$.

Let x and y be members of a set $X \subseteq \mathbb{X}$, let $c \in \mathbb{Q}_+$ and $\sim \in \{<, \leq, =, \neq, \geq, >\}$. The domain of aperiodic clock constraints over X is noted $\mathcal{C}(X)$. An aperiodic clock constraint $\phi \in \mathcal{C}(\mathbb{X})$ is a predicate over valuations of clocks in X, that has the form: $\Gamma_a ::= x \sim c \mid x - y \sim c \mid \Gamma_a \wedge \Gamma_a \mid \neg \Gamma_a \mid true$. We write $v \vDash \phi$ when the clock valuation v satisfies the constraint ϕ. A *simple update* is a function $su(v, x)$ taking a valuation $v : X \rightarrow \mathbb{R}_+^{|\mathbb{X}|}$ and a clock $x \in X$ as parameters and returning a valuation v' such that $v'(z) = v(z)$ if $z \neq x$ and:

$$\begin{cases} v'(x) = c & \text{if } su ::= x := c \\ v'(x) = v(y) - c & \text{if } su ::= x := y - c\,. \end{cases}$$

A *local update* lu that belongs to the class $\mathcal{U}(X)$ is a function from valuations to valuations, defined for each clock y_i of a subset $Y = \{y_1, \ldots, y_k\} \subseteq X$ as a finite set of simple updates $\{su_y^0, \ldots, su_y^k\}$. It returns a valuation v'' such that each su_y^i is applied to v:

$$\begin{cases} v''(z) = v(z) & \text{if } z \notin Y \\ v''(y_i) = su_y^i(v, y_i) & \text{if } y_i \in Y. \end{cases}$$

We define reset(X) as the local update resetting all the clocks of X to 0.

Definition 3.1 *A timed automaton $\mathcal{A}$ is a tuple $\langle L, X, \Sigma, \Lambda, I, E \rangle$, where L is a finite set of* locations*, X is a finite set of* clocks*, Σ is a finite set of* action labels*, $\Lambda \subseteq L \times \mathcal{C}(X)$ is a function labeling each location l with a* timing invariant *$\Lambda(l)$, I is the set of* initial *locations, and $E \subseteq L \times [\mathcal{C}(X) \times \Sigma \times \mathcal{U}(X)] \times L$ is a finite set of* edges*. Each edge connects one* source *and one* target *location.*

Those automata slightly differ from the "classical" automata by Alur and Dill [5]: they allow periodic updates of the form $x := y - c$ where $c \in \mathbb{Q}_+$. They are named *periodic*

updatable timed automata in [6]. It has then been shown that they allows a much easier specification of *jitter-periodic* phenomena (ie. periodic phenomena with jitter) than classical timed automata. Their expressive power is however identical to classical timed automata, as long as silent transitions (noted ϵ) are allowed. This prolongs the essential result established in [5]: *reachability analysis* can be performed on such timed automata.

3.2. Timed Automata Semantics

Since we are interested in computational aspects of timed automata, we define their semantics according to a *branching time* model. An equivalent *linear time* semantics can be found in language-theoretic studies such as [5, 6]. A directed graph $G_{\mathcal{A}}$ is used to represent the possible executions of a timed automaton $\mathcal{A}$. $G_{\mathcal{A}}$ is a pair $(\mathcal{P}, \rightarrow)$ where $\mathcal{P}$ is a set of *nodes*, and $\rightarrow \subseteq V \times V$ is a set of *transitions*. Each node $P \in \mathcal{P}$ (also called a *process*) bears a label of the form $\langle l, v \rangle$ where $l \in L$ and v is a clock valuation. A time-passing transition $\langle l, v \rangle \xrightarrow{\delta} \langle l, v+\delta \rangle$ can occur whenever $v+\delta \vDash \Lambda(l)$ for a delay $\delta \in \mathbb{R}_+ \setminus \{0\}$. A discrete transition from $\langle l, v \rangle$ through an edge $e = \langle l, \phi, \sigma, u, l' \rangle \in E$ may occur whenever $v \vDash \phi$, leading to a state $\langle l', u(v) \rangle$; it is noted $\xrightarrow{\sigma}$. It easily appears that $G_{\mathcal{A}}$ is infinitely branching, and hence infinite. A finite sequence of transitions is noted $(\rightarrow)^*$. The time domain respects *time determinism* (*ie.* $P \xrightarrow{\delta} P' \wedge P \xrightarrow{\delta} P'' \Rightarrow P' = P''$) and *time continuity* (*ie.* $P \xrightarrow{\delta+\delta'} P'' \iff \exists P',\ P \xrightarrow{\delta} P' \xrightarrow{\delta'} P''$). The time domain is then *stuttering closed.*

3.3. How to Analyse Timed Automata

The *region graph* technique has been proposed in [5] so as to provide a finite-state representation of the infinite branching-time state space reached by a finite timed automaton. The reduced state-space is computed by, for each node, grouping all the time-passing transitions that do not alter the set of available discrete transitions leaving this node. Such a group is then called a *region*, and the set of all reachable regions provide a finite partitioning of the space of clock valuations. Each region r is represented by a node in region graph; the transitions that leave r then represent what regions can be reached from r either by taking a discrete transition or by letting time pass. Hence, the region graph is finitely-branching. Since region-equivalence is decidable, the graph is also well-founded and thus finite.

The region-graph method partitions the state space according to a *strong time-abstracting bisimulation* [27]. The original article from Alur and Dill [5] however do not permit the computation of the *coarsest* such partition, implementations using the region graph are very inefficient. A better solution is given by so-called *symbolic* techniques [13], and the optimal solution is obtained through *partition refinement* [27]. However, an effective way to represent and operate on *sets* of regions is then necessary. A set of regions is called a *zone*, and it constitutes a union of convex polyhedra constraining the clocks of the examined automaton. Zones have an interesting property: all possible discrete and time successors (resp. predecessors) of a zone can be computed, and each of them is a zone. Zones can be represented using *Difference Bound Matrices* (DBMs) [27]: for a n-clock automaton, a DBM is a $(n+1) \times (n+1)$ square matrix where each couple of clocks x and y of the automaton receive a couple of constraints $M_{xy} = (c, \prec)$ and $M_{yx} = (d, \prec)$ respectively representing $x - y \prec c$ and $y - x \prec d$ with $\prec \in \{<, \leq\}$ and $c, d \in \mathbb{N} \cup \{\infty\}$. The additional clock x_0 always evaluates to zero. It allows to represent constraints of the type $x \prec c$ and $x \succ d$, for any clock x. For any DBM, a *canonical* DBM can be computed; it is the tightest representation of the same zone. As an example, consider a DBM D on clocks x and y with $x \leq 5 \wedge z \leq 7 \wedge z - x \geq 4$. The canonical representation of D is obviously: $x \leq 3 \wedge z \leq 7 \wedge z - x \geq 4$. Zones (and DBMs) are closed under union, intersection, and complement. However, those latter two operations may yield non-convex polyhedra which are not directly representable by one DBM but rather as a union of them.

So-called *on-the-fly* techniques [7] make use of symbolic representations but avoid the computation of unions of DBMs by doing a *forward symbolic execution* of the examined automaton. A graph is built which nodes are tagged by a location of the automaton and a zone represented by a DBM. This technique is used by the most successful tools of today (eg. Kronos [27] or Uppaal [14]). It is the one we adapt in our type-checking algorithm. For a more detailed survey on verification principles and implementations for timed systems, see *eg.* [27].

4. Timed Behavioral Contracts

4.1. The Use of Bounded-Time Channels with Jitter

The general specification framework for contracts is a timed extension of Communicating Finite-State Machines (CFSMs) [8, 9]: processes are described as timed input/output automata that communicate asynchronously through finite or infinite reliable FIFO channels. When channels are infinite, any Turing machine can be simulated by some CFSM, and most problems of interests are therefore undecidable for CFSMs [8]. Many contributions therefore restrain to *unreliable* infinite FIFO channels [3]. We choose to restrain to reliable but *finite* channels. We thus impose that some timed FSM may emit only a finite number of signals in a finite time (it is then said *non-zeno*), and that those signals have finite delivery times. Each channel delivers signals in FIFO order and has two rational parameters τ_0 and τ: the transmission time of a signal is superior to τ_0 and inferior to $\tau_0 + \tau$. If the destination FSM is not ready to receive when an incoming signal is ready to be delivered, the signal is lost. This model is similar to the one in [9].

4.2. The Structure of Contracts for Reconfigurable QoS

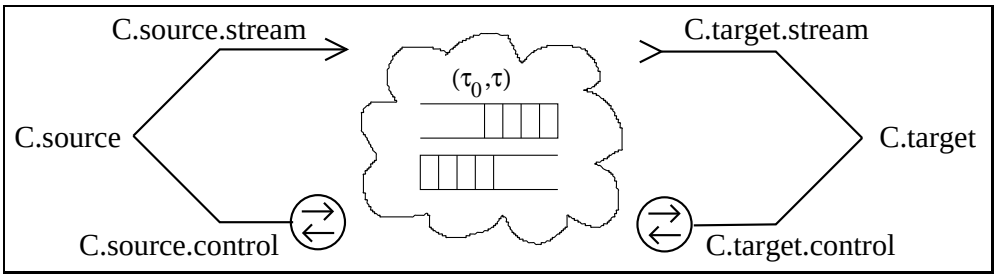

Figure 2. A Representation of Contract-Bound Endpoints

A contract is intended to bind endpoints relying on opposite-way channels which transmission delay and jitter characteristics are known at verification time. This situation is represented on Figure 2. A contract C has a *source* and a *target* role representing the bounded endpoints. As we target *interactive* and *open* systems, QoS *reconfigurability* is central to our effort. In order to describe contracts with reconfigurable QoS, each role is actually composed of a *stream* and a *control* part. Between the stream parts, signals always go from the source to the target. Distinctively, control signals may go in either direction between the control parts of peered roles. The control part of a contract can therefore be used to specify (re-)configuration scenarios that can be performed by this contract. In that purpose, the control part of a role may order the stream part of the same role to change its behavior (ie. to adopt new QoS characteristics). The coupling of stream and control parts can be compared to that of the RTP/RTCP [22] and RTSP [25] protocols: the stream part describes which levels of QoS are available through the contract (as when

using RTP/RTCP), and the control part describes under which circumstances those levels can be reached (as a remote control definable with RTSP).

Definition 4.1 *A contract C is a tuple $\langle S, T, \tau_0, \tau \rangle$ where S is the source role, T is the target role and τ_0 and τ are the channel characteristics. S and T have an identical structure $\langle \mathcal{A}_s, \mathcal{A}_c, \mathcal{R} \rangle$ where $\mathcal{A}_s = \langle L_s, X_s, \Sigma_s, \Lambda_s, I_s, E_s \rangle$ is the stream automaton, $\mathcal{A}_c = \langle L_c, X_c, \Sigma_c, \Lambda_c, I_c, E_c \rangle$ is the control automaton, and $\mathcal{R} \subseteq E_c \times L_s \times \mathbb{Q}_+^{|X_s|}$ is a function associating to some edges of $\mathcal{A}_c$ the power to preempt the state of $\mathcal{A}_s$. In that case, both the location and a constant rational values for* all *the clocks of $\mathcal{A}_s$ must be given.*

4.3. Contract Semantics

The contract semantics is obtained through a straightforward specialization of timed automata semantics. That is, contract roles may bear distinctive labels describing types and arities for signal parameters, as found in the syntax of $\mathcal{A}rtOC$. We restrict to the case where edge labels are either written like $\mu s(\tilde{p})$ or viewed like they are the silent label ϵ. In Table 2, we give semantic rules for two roles R_1 and $R_2 \in \{S, T\}$ of a contract $C = \langle S, T, \tau_0, \tau \rangle$. These rules take only into account the sending capacities of R_1 and the receiving capacities of R_2: the state of only one channel γ delivering messages from R_1 to R_2 is considered. As both roles may send and receive messages, those rules must be combined with their converses in order to obtain the exact behavior of C.

A state of γ is a list of couples $[\langle s_1(\tilde{p}_1), \nu_1 \rangle, \ldots, \langle s_k(\tilde{p}_k), \nu_k \rangle]$ showing the order of messages currently in γ along with their respective sojourn times $\nu_1 \ldots \nu_k$. The state of some role is written $R = \langle \langle v^s, l^s \rangle, \langle v^c, l^c \rangle \rangle$ where $\langle v^s, l^s \rangle$ and $\langle l^c, v^c \rangle$ are the current clock valuations and locations of $\mathcal{A}_s$ and $\mathcal{A}_c$, respectively. Role semantics is the straightforward extension of automata semantics, where R let time pass when both $\mathcal{A}_s$ and $\mathcal{A}_c$ agree to do so, and a discrete transition may be triggered when either $\mathcal{A}_s$ or $\mathcal{A}_c$ may do so. Rule (1) is

$$\frac{\begin{array}{c}\langle\langle v_1^s, l_1^s\rangle, \langle v_1^c, l_1^c\rangle\rangle \xrightarrow{\delta} \langle\langle v_1^s + \delta, l_1^s\rangle, \langle v_1^c + \delta, l_1^c\rangle\rangle \wedge \\ \langle\langle v_2^s, l_2^s\rangle, \langle v_2^c, l_2^c\rangle\rangle \xrightarrow{\delta} \langle\langle v_2^s + \delta, l_2^s\rangle, \langle v_2^c + \delta, l_2^c\rangle\rangle \wedge (\nu_k + \delta \leq \tau_0 + \tau)\end{array}}{\left[\begin{array}{c}\langle\langle v_1^s, l_1^s\rangle, \langle v_1^c, l_1^c\rangle\rangle \\ \mid [\langle s_1, \nu_1\rangle, \ldots, \langle s_k, \nu_k\rangle] \mid \\ \langle\langle v_2^s, l_2^s\rangle, \langle v_2^c, l_2^c\rangle\rangle\end{array}\right] \xrightarrow{\delta} \left[\begin{array}{c}\langle\langle v_1^s + \delta, l_1^s\rangle, \langle v_1^c + \delta, l_1^c\rangle\rangle \\ \mid [\langle s_1, \nu_1 + \delta\rangle, \ldots, \langle s_k, \nu_k + \delta\rangle] \mid \\ \langle\langle v_2^s + \delta, l_2^s\rangle, \langle v_2^c + \delta, l_2^c\rangle\rangle\end{array}\right]} \quad (1)$$

$$\frac{R_1 \xrightarrow{!s(\tilde{p})} R_1'}{R_1 \mid [M] \mid R_2 \xrightarrow{!s(\tilde{p})} R_1' \mid [\langle s(\tilde{p}), 0\rangle, M] \mid R_2} \quad (2)$$

$$\frac{R_2 \xrightarrow{?s(\tilde{p})} R_2' \ \wedge \ \tau_0 \leq \nu_s \leq \tau_0 + \tau}{R_1 \mid [M, \langle s(\tilde{p}), \nu_s\rangle] \mid R_2 \xrightarrow{?s(\tilde{p})} R_1 \mid [M] \mid R_2'} \quad (3)$$

$$\frac{(\nexists R_2', R_2 \xrightarrow{?s(\tilde{p})} R_2') \ \wedge \ \tau_0 \leq \nu_s \leq \tau + \tau_0}{R_1 \mid [M, \langle s(\tilde{p}), \nu_s\rangle] \mid R_2 \xrightarrow{Err} Error} \quad (4)$$

Table 2. Operational Semantics for Contracts

a time-passing transition that may occur only if the maximum queuing time of the oldest message is not reached yet. Rule (2) is a signal emission, adding $s(\tilde{p})$ to γ with a zero queuing time. Rule (3) is a message reception retrieving $s(\tilde{p})$ from γ: R_2 must be ready to receive, and $s(\tilde{p})$ must have a sufficient queuing time. Rule (4) applies whenever a signal

that is ready to be delivered cannot be received by R_2. Such a message can be either unexpected, early, or late. We finally introduce a slight difference between roles and automata semantics: silent transitions have a lower priority than others, *ie.* there may exists R' so that $R \xrightarrow{\epsilon} R'$ only if there is no R'' such that $R \xrightarrow{\delta} R''$ or $R \xrightarrow{\rho s(\tilde{p})} R''$.

4.4. Conjoining Roles Within Contracts

In order to be able to assemble processes bound by contracts in a consistent way, we need each contract to apply some sort of *assume/guarantee* rule [2] between its source and target roles. In our case, the rule states that the target endpoint has to guarantee the respect of the obligations given by its role if and only if the other endpoint exhibits the behavior assumed by the source role. We therefore need our language for assumptions and guarantees to be *enforcement-oriented*: any action that is time-constrained has to occur on time unless both peers explicitly agree to do otherwise. Roles have therefore the *urgency* to act when they can. This yields the following principle:

> *For any role R of a contract, never the availability window of a signal $s \in R$ should close without s being triggered through R.*

This implies that the language for roles is actually a syntactically recognizable subclass of timed automata. Precisely, roles are timed automata for which any location with outgoing edges $e_1, \cdots, e_n$ bearing constraints $\phi_1, \cdots, \phi_n$ has an invariant $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$.

4.5. Contract Consistency

Definition 4.2 *Role Compatibility Relation.*
Considering two distinct channels γ_{12} and γ_{21} of characteristics τ and τ_0, we will write that roles R_1 and R_2 belong to the relation $\mathcal{C}_{\tau_0,\tau}(\gamma_{12}, \gamma_{21})$ if and only if:

$$\begin{cases} R_1 \mid (\gamma_{12}, \gamma_{21}) \mid R_2 \xrightarrow{Err} \not\longrightarrow \quad \wedge \\ R_1 \mid (\gamma_{12}, \gamma_{21}) \mid R_2 \rightarrow R_1' \mid (\gamma_{12}', \gamma_{21}') \mid R_2' \Rightarrow (R_1', R_2') \in \mathcal{C}_{\tau_0,\tau}(\gamma_{12}', \gamma_{21}') \quad \wedge \\ \forall Seq \in \{R_1 \mid (\gamma_{12}, \gamma_{21}) \mid R_2 \rightarrow_*\}, \ \neg\, zeno(Seq) \end{cases}$$

The first two clauses forbid the initial configuration to produce an execution error. The last one forces the model to possess the previously mentioned *non-zeno* property.

Definition 4.3 (Contract Consistency) *Let $\mathcal{A} = \langle L, X, \Sigma, \Lambda, \{l_1, \ldots, l_n\}, E\rangle$ be a timed automaton. Let $\mathcal{W}(\mathcal{A}) = \langle L \cup \{l_1', \ldots, l_n'\}, X, \sigma \cup \{bind\}, \Lambda, \{l_1', \ldots, l_n'\}, E \cup \{e_1, \ldots, e_n\}\rangle$ a timed automaton where $l_1, \ldots, l_n$ is a set of new initial locations, and $e_1, \ldots e_n$ is a set of new edges such that $e_i = \langle l_i, true, ?bind(), reset(X), l_i'\rangle$ for $1 \leq i \leq n$. For a role $R = \langle \mathcal{A}_s, \mathcal{A}_c, \mathcal{R}\rangle$, we also define $\mathcal{W}(R) = \langle \mathcal{W}(\mathcal{A}_s), \mathcal{W}(\mathcal{A}_c), \mathcal{R}\rangle$. We then write that a contract $C = \langle S, T, \tau_0, \tau\rangle$ is (τ_0, τ)-consistent iff:*

$$(S, \mathcal{W}(T)) \in \mathcal{C}_{\tau_0,\tau}([\langle bind(), 0\rangle], \emptyset).$$

The transformation $\mathcal{W}$ applied to T characterizes the *binding scenario*. We chose to apply *first-party binding*, where the allegedly source endpoint creates both roles of the contract then sends the target role to the intended peer as an argument of the *bind* signal. The clocks of the source role start to progress when *bind* is sent. The clocks of the target role start to progress when *bind* is delivered.

The contract consistency relation is essentially decidable. One may indeed consider a region-graph-based interpretation where each channel able to contain n signals has n clocks. The non-zeno property ensuring that channels only need a finite capacity, the model is finite and the complete set of reachable states of any contract may be computed. A similar way has been explored in [9], where only one-clock automata were considered.

As a complementary element of proof, a toy example contract ready for verification is available[5] in the Uppaal [14] syntax.

4.6. A Small Example

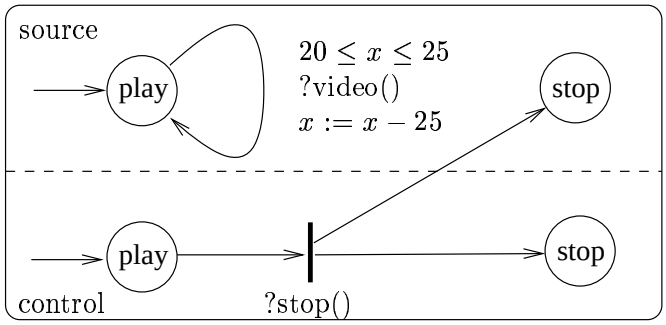

Figure 3. A Video Target Interface

The target role of Figure 3 can be used to receive video signals with a period of 25 milliseconds (we take milliseconds for time unit) and a jitter of 5 milliseconds. Initially, both automata are in the location *play*. The control part may receive a signal *stop* at any moment, which makes it preempt the stream part to reach the *stop* location.

A compatible behavior would be VideoClient on 4, that receives a behavioral target interface that must wait 5 seconds for the delivery of the VideoTarget interface and then do nothing. The VideoRcv behavior waits infinitely for video until the signal *stop*() is received.

VideoRcv(chan:?VideoTarget) =
  timer $y = 0$ **in**
  timeout($y \leq 25$) ? chan
    [video() > **timeout**($y = 25$) > **become** VideoRcv(chan:?VideoTarget)
      stop() $> \infty$
    ] $> \infty$
VideoClient(reply: ?Connect) =
  timer $x = 0$ **in**
  timeout($x \geq 5000$) ?reply
    [getChan(chan:?VideoTarget) >
      spawn VideoRcv(chan) $> \infty$
    ] $> \infty$

Figure 4. A Small *ArtOC* Behavior

5. Static Semantics

The typing system we define must check for two kinds of properties in order to guarantee the safe reduction of well-formed *ArtOC* terms. The first is that any actor respects the timeliness constraints required by the contract roles it owns. The second is that all the

[5] at `http://www.perso.enst.fr/~abailly/ArtOC`.

name invocations it performs are legal, that is, they respect the scoping and binding rules defined above.

5.1. Using Zones as Representatives

The correction regarding binding rules can be checked easily by enclosing a set of currently defined names in the typing context. This name-set can be maintained using classical set-theoretic operations. The verification of timing constraints can be done using the *strong time-abstracting bisimulation* to determine the set of states that will be reached after each step of computation. We will represent the behavior of an interface u as a tuple $T = \langle \mathcal{A}_T, l_T, Z_T \rangle$, that encloses an automaton $\mathcal{A}_T$, a location $l \in \mathcal{A}_T$ and a zone Z_T represented as a DBM. We proceed in a similar manner for actors, maintaining a zone Z_A for all the timers X_A of any behavior A. If A possesses u, we will have to compare the behavior of A with the behavior of T. It yields that Z_T must actually be a set of constraints relating the clocks of T with the timers of A: Z_T is a DBM on $X_T \cup X_A$. Upon creation of interface u, its type T has for location any initial location of $\mathcal{A}_T$, and Z_0 for zone. For any dimension, we define Z_0 to be a zone in which all clocks evaluate exactly to 0.

We will need to perform intersections, projections and expansions on polyhedra (*ie.* zones represented as DBMs). The *dimension-restricting* projection of a zone Z_X of dimension $|X|$ on a set of clocks $Y \subseteq X$ is the zone containing all valuations v' such that $\exists v \in Z_X, \forall y \in Y, v(y) = v'(y)$. This projection is noted $Z_X \rfloor_Y$. It basically corresponds to selecting the constraints in Z_X that only relate the clocks of Y among themselves. The converse operation is the expansion: if Z_Y is a zone of dimension $|Y|$ and $Y \subseteq X$, $Z_Y \rceil^X$ is the union all the zones Z_X of dimension $|X|$ such that $Z_X \rfloor_Y \subseteq Z_Y$. Expansion enlarges the constraints in Z_Y by allowing any values for the clocks in X and not in Y.

To be able to use zones to represent behavior states, we need to show that the successor states of a zone can be computed, and that the result is a zone. We rely on the truthfulness of the same result for timed automata [27].

Lemma 5.1 (Computability of Zone-Successors in $\mathcal{A}rtOC$) *Consider a zone Z_X on a timer set X. The zone obtained by letting time progress unboundedly from Z_X is obviously the same for timers and for timed automata. The same transformation can thus be applied to Z_X. Its is called $succ_\varepsilon(Z_X)$. When executing timeout behavior, one must be able to compute whether the timeout zone Z_G corresponding to guard G will be reached. This can be done as in the timed automata case by computing $Z_t = Z_G \cap succ_\varepsilon(Z)$. The only behavior that induces a discrete modification on some timer value is the timer setting operation itself. It resets a timer x to a value δ, as an automaton does for a clock. The resulting zone is therefore computable by applying the operation $Z_X[x := \delta]$ to Z_X defined for timed automata, standing for the set of all clock valuations in Z_X where the constraints on x are replaced by the constraint $x = 0$. We generalize the notation $Z[\mathfrak{u}]$ to any local update $\mathfrak{u}$ performed by a transition. A similar reasoning can be done easily for predecessors $pred_\varepsilon(Z_X)$ and $pred_e(Z_X)$.*

Example 1 (taken from [27]) *Consider the zone Z do be $1 < y < 2 \wedge 2 < x \wedge x - y < 2$. Then the update of x to 0 yields:*

$$\begin{aligned} succ_e(Z) &= \exists x', y'.\ \phi[X/x', y/y'] \wedge y = 0 \wedge x = x' \\ &= \exists x', y'.\ 1 < y' < 2 \wedge 2 < x' \wedge x' - y' < 2 \wedge y = 0 \wedge x = x' \\ &= \exists y'.\ 1 < y' < 2 \wedge 2 < x \wedge x - y' < 2 \wedge y = 0 \\ &= \exists y'.\ 1 < y' < 2 \wedge 2 < x \wedge x - y' < 2 \wedge y = 0 \\ &= 2 < x \wedge x < 4 \wedge y = 0 \end{aligned}$$

Since we intend to represent the type of an interface as sole automaton, we need to compute the product of its stream and control automata before verification. This is possible

using a straightforward modification of the synchronous product for timed automata defined in [5]. This extension is needed to deal with preemption from the control automaton over the stream automaton, that is of course not present in Alur and Dill's automata. We recall that when preemption occurs both the location and clock values of the stream automaton are given. Hence, when encountering preemption, the computation of the product automaton can continue by abandoning the former state of the stream part and taking the new one. The behavior of a type can consequently be described by only one automaton.

5.2. Typing Contexts

A typing context Γ is a set of tuples of the forms $u : \rho T$ and $A : \langle \tilde{u} : \tilde{\rho}\tilde{T}, Z_A \rangle$. Γ also contains the current zone of the process in typing, noted Z_{self}. Γ is well-formed if all names A and all couples $\langle A, \rho \rangle$ are unique, and if all role names $\tilde{u}$ contained in the actor tuples are present in Γ. We will use the comma operator for the extension of the typing contexts. The typing judgments are summarized in Table 5.

judgment	meaning
$\Gamma \vdash u : \rho T$	in the context Γ, the interface u has the role ρ of type T
$\Gamma \vdash G$	in the context Γ, $Z_{\mathsf{self}} \vDash G$
$\Gamma \vdash B$	in the context Γ, the behavior B is well typed
$\Gamma \vdash C$	in the context Γ, the configuration C is well typed

Figure 5. Judgments of the typing rules

Roles of contracts for which the binding has not been done yet will carry a tag, like $u^{\perp}$. The suppression of the tag, *ie.* the activation, can be done with $\top$: $\top(u) = u$ and $\top(u^{\perp}) = u$. It can also be applied to lists: $\top(\tilde{u}) = \top(u_1)\ \ldots\ \top(u_n)$. It is finally applied to typing contexts, where $\top(\Gamma, \tilde{u})$ is Γ where all the inactive role names contained in $\tilde{u}$ have been activated.

To enforce our name management policy, we will use a name incorporation operator for contexts. This operator performs differently when adding uniform or behavioral role names. Precisely, $\Gamma \oplus u : \rho$ is equal to $(\Gamma, u : \rho T)$ for some T when defined, and is undefined in the following cases: if $\Gamma \vdash u : \rho T$ and $\rho = ??$, or if $\Gamma \vdash u : \rho T$ and $\rho = ?$, or if $\rho = !$. This relation is easily extended to lists of role names $\tilde{u} : \tilde{\rho}$, by defining addition as the fixpoint of the equation $(\Gamma \oplus \tilde{u} : \tilde{\rho}) = (\Gamma \oplus u_1 : \rho_1) \oplus (u_2 : \rho_2, \ldots, u_n : \rho_n)$.

5.3. Operations on Zones and Typing Contexts

In the following, we consider $T = \langle \mathcal{A}_T, l_T, Z_T \rangle$, and $\Gamma \vdash \tilde{u} : \tilde{\rho}\tilde{T}$. We first generalize the notion of time successor to types, by stating that $succ_\varepsilon(T) = \langle \mathcal{A}_T, l_T, succ_\varepsilon(Z_T) \rangle$. We then generalize it to typing contexts: we note $succ_\varepsilon(\Gamma) \vdash u_1 : \rho_1\, succ_\varepsilon(T_1),\ \ldots\ , u_n : \rho_n\, succ_\varepsilon(T_n)$. The function $reset(T)$ places T in an initial location and changes its zone to Z_0. This function is also extended to lists: $reset(\tilde{T}) = reset(T_1), \ldots, reset(T_n)$. The discrete successor of T by an edge $e = \langle l_T, \phi, \sigma, \mathfrak{u}, l'_T \rangle$ will be noted $succ_e(T) = \langle \mathcal{A}_T, l'_T, Z_T[\mathfrak{u}] \rangle$. We will use the predicates $possible(T)$ for the set of possible discrete transitions in Z_T. We then define the predicate $possible(\sigma, T)$ if there is an edge e in $possible(T)$ that has label σ, and the type $succ(\sigma, T) = succ_e(T)$ successor of T by e. We will also use the predicate $timelock(T)$ indicating that $possible(T) = \emptyset$ and that the current location l_T of T has an invariant $x \prec c$ for some clock $x \in \mathcal{A}_T$. Time progression is then impossible after a finite delay. Roles with such property ought to be ruled out. We define $guard(T) = \langle \mathcal{A}_T, l_T, I_T \rceil^{X_T} \cap Z_T \rangle$ where I_T is the invariant of location l_T in $\mathcal{A}_T$. We extend it also to typing contexts: $guard(\Gamma) \vdash u_1 : \rho_1\, guard(T_1),\ \ldots\ , u_n : \rho_n\, guard(T_n)$.

We also extend zone (*ie.* set) intersection and projection to zones that appear in typing contexts. For a type $T = \langle \mathcal{A}_T, l_T, Z_T \rangle$ and a clock set $X \subseteq X_T$ of the clocks X_T of $\mathcal{A}$,

we note $T\rfloor_X$ for $\langle \mathcal{A}_T, l_T, Z_T\rfloor_X\rangle$. Similarly, when $\Gamma \vdash \tilde{u} : \tilde{\rho}\tilde{T}$, then we note $\Gamma\rfloor_Y \vdash u_1 : \rho_1 T_1\rfloor_Y, \ \ldots\ , u_n : \rho_n T_n\rfloor_Y$. The type and typing context intersections are defined in an asymmetric fashion. For Z_Y of domain $Y \subseteq X_T$, we have $T \cap Z_Y = \langle \mathcal{A}_T, l_T, Z_T \cap (Z_Y\rceil^{X_T})\rangle$ and $\Gamma \cap Z_Y \vdash u_1 : \rho_1 T_1 \cap Z_Y, \ \ldots\ , u_n : \rho_n T_n \cap Z_Y$. We also have $Z_Y \cap T = Z_T\rfloor_Y \cap Z_Y$ and $Z_Y \cap \Gamma = (Z_Y \cap T_1) \ \cap \ \ldots \ \cap \ (Z_Y \cap T_n)$.

5.4. Typing Rules

First, a behavior can be infinitely idle only if it has nothing to do, that is to say it only owns uniform interfaces or behavioral interfaces that do not impose some timing constraints on the sending and reception of signals:

$$\frac{\Gamma \ \vdash \ u : T \Rightarrow \left\{\begin{array}{l}(\rho \in \{!!, ??\} \vee \\ (possible(succ_\varepsilon(T)) = \emptyset \wedge \neg\, timelock(guard(succ_\varepsilon(T))))) \vee \\ (\forall e = \langle l, \phi, \sigma, u, l'\rangle \in \ possible(succ_\varepsilon(T)), \\ \qquad\qquad (\sigma = \epsilon \ \wedge \Gamma, u : succ_e(succ_\varepsilon(T)) \ \vdash \ \infty))\end{array}\right.}{\Gamma, u : T \ \vdash \ \infty}$$

Message sending is conditioned by timeliness and by adequacy of name references. The type of u must allow the sending action, the parameters must be of the required types, and the non-duplicable roles should never be used again in B. The behavioral roles one of which is sent are activated.

$$\frac{\begin{array}{l}possible(T_u, !s(\tilde{\rho}\tilde{T})) \\ T'_u = succ(!s(\tilde{\rho}\tilde{T})) \\ \top(\Gamma, \tilde{v}), u : \rho_u T'_u \ \vdash \ B \\ \Gamma \oplus \tilde{v} : \tilde{\rho} \text{ defined}\end{array}}{\Gamma \oplus \tilde{v} : \tilde{\rho}, u : \rho_u T_u \ \vdash !u.s(\tilde{\rho}\tilde{v}) > B}$$

In a reception state, an actor must be able to catch any signal that may arrive during the time for which the process waits. To identify those signals, we compute the zone in which time can progress: first by computing the zone in which time may progress infinitely, then by applying to it the guards found in the current location of each active interface owned by the process. The resulting component is applied to the current local zone Z_{self}. The obtained result constrains in return the evolution of each interface. Then the precondition on $possible(T)$ imposes that the set of expected signals is a superset of the possible ones. Finally, any incoming role starts evolving from one of its possible initial states.

$$\frac{\begin{array}{l}Z'_{\mathsf{self}} = (succ_\varepsilon(Z_{\mathsf{self}}) \cap pred_\varepsilon(G)) \cap guard(succ_\varepsilon(\Gamma, u : \rho T)) \\ \Gamma' = guard(succ_\varepsilon(\Gamma)) \cap Z'_{\mathsf{self}} \\ T' = guard(succ_\varepsilon(T)) \cap Z'_{\mathsf{self}} \\ possible(T') = \{s_1, \ldots, s_n\} \subseteq \{s_1, \ldots, s_n\} \\ T'_1 = succ(?s_1(\tilde{\rho}_1 \tilde{T}_1), T') \ \ldots \ T'_n = succ(?s_n(\tilde{\rho}_n \tilde{T}_n), T') \\ \tilde{T_1}'' = reset(\tilde{T}_1) \ \ldots \ \tilde{T_n}'' = reset(\tilde{T}_n) \\ \Gamma, u : \rho T'_1, \tilde{v}_1 : \tilde{\rho}_1 \tilde{T}''_1 \ \vdash \ B_1 \ \ldots \ \Gamma, u : \rho T'_n, \tilde{v}_n : \tilde{\rho}_n \tilde{T}''_n \ \vdash \ B_n\end{array}}{\Gamma, u : \rho T \ \vdash \ \textbf{timeout} \ (G)?u[s_1(\tilde{v}_1 : \tilde{\rho}_1 \tilde{T}_1) > B_1, \ \ldots\ , s_n(\tilde{v}_n : \tilde{\rho}_n \tilde{T}_n) > B_n]}$$

The same conditions are distributed to all the receptions in a choice:

$$\frac{\Gamma \ \vdash \ Recep_1 \ \ldots \ \Gamma \ \vdash \ Recep_n}{\Gamma \ \vdash \ \sum_{i=1}^{n} Recep_i}$$

A timeout simply lets timers progress until the guard G becomes true, at the condition that no message may potentially be ready to be delivered with that period:

$$\frac{\begin{array}{l} Z'_{\mathsf{self}} = (succ_\varepsilon(Z_{\mathsf{self}}) \cap pred_\varepsilon(G)) \\ \Gamma' = guard(succ_\varepsilon(\Gamma)) \cap Z'_{\mathsf{self}} \\ possible(T') = \emptyset \\ \Gamma, u : \rho T'_1, \tilde{v}_1 : \tilde{\rho}_1 \tilde{T}''_1 \vdash B_1 \ \ldots \ \Gamma, u : \rho T'_n, \tilde{v}_n : \tilde{\rho}_n \tilde{T}''_n \vdash B_n \end{array}}{\Gamma, Z_{\mathsf{self}} \vdash \textbf{timeout}\ (G) > B}$$

In a conditional action, zones are updated according to the branch that is taken:

$$\frac{\begin{array}{c} \Gamma, Z_{\mathsf{self}} \cap G \vdash B_t \\ \Gamma, Z_{\mathsf{self}} \cap \overline{G} \vdash B_f \end{array}}{\Gamma, Z_{\mathsf{self}} \vdash \textbf{if}\ G\ \textbf{then}\ B_t\ \textbf{else}\ B_f}$$

In a **new** both roles are created and reset. Contract roles are tagged as inactive.

$$\frac{\begin{array}{l} T' = reset(T) \\ \Gamma, u :!! T', u :?? T' \vdash B \end{array}}{\Gamma \vdash \textbf{new}\ u : T\ \textbf{in}\ B} \qquad \frac{\begin{array}{l} T' = reset(T) \\ \Gamma, u^\perp :! T', u^\perp :? T' \vdash B \end{array}}{\Gamma \vdash \textbf{new}\ u : T\ \textbf{in}\ B}$$

We must check that during actor impersonation and actor instantiation the resulting behaviors are consistent with their types. Naming must be correct in parallel configurations

$$\frac{\begin{array}{l} A \stackrel{dcl}{=} A(\tilde{u} : \tilde{\rho}\tilde{T}) = B \\ \Gamma \vdash \tilde{u} : \tilde{\rho}\tilde{T} \\ \Gamma, Z_\Omega \vdash B \end{array}}{\Gamma, Z_{\mathsf{self}} \vdash \textbf{become}\ A(\tilde{u})} \qquad \frac{\begin{array}{l} A \stackrel{dcl}{=} A(\tilde{u} : \tilde{\rho}\tilde{T}) = B \\ \Gamma, \oplus\tilde{\rho}\tilde{u}, Z_\Omega \vdash B \\ \Gamma, Z_{\mathsf{self}} \vdash B' \\ \Gamma \oplus \tilde{\rho}\tilde{u}, Z_{\mathsf{self}}\ \text{defined} \end{array}}{\Gamma \oplus \tilde{\rho}\tilde{u} \vdash \textbf{spawn}\ A(\tilde{u}) > B'} \qquad \frac{\begin{array}{l} \Gamma_1 \vdash C_1 \\ \Gamma_1 \vdash C_1 \\ \Gamma_1 \oplus \Gamma_2\ \text{defined} \end{array}}{\Gamma_1 \oplus \Gamma 2 \vdash C_1 | C_2}$$

6. Results and Conclusions

Our type system guarantees that a well-typed closed configuration waits for signals when it has to and emits signals only when it can. Since the number of regions for the timer and interfaces of a given behavior is finite, our typing algorithm always terminates. This verification method actually combines enumerative and deductive methods: model-checking techniques are used on contracts to check their consistency, and then on purely sequential timed actors to check type correctness. Deductive aspects appear in rules dealing with name transmission and actor spawning. Indeed, an infinite number of actors can be created, and our method provides a finite abstraction to enumeratively check such infinite-space configurations.

In conclusion have shown how, by enhancing plain timed automata with periodic transitions and preemptive control behaviors, behavioral QoS contracts can be easily written and their consistency automatically verified. The introduction of timed behavioral contracts answers to current trends in software engineering that wish to model and specify applications providing some explicit QoS offers and retaining explicit requirements. This problem is an open field of research for Real-Time ORBs, such as RT-CORBA-compliant ORBs. Finally, We consider that enhancing actors and types with counters and introducing lossy channels constitute natural extensions, and should be interesting further works.

References

[1] *Open Distributed Processing and Multimedia.* In Press. Addison-Wesley, 1997.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] Abdulla and Jonsson. Verifying programs with unreliable channels. *INFCTRL: Information and Computation (formerly Information and Control)*, 127, 1996.

[4] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *J. Functional Programming*, 7:1–72, 1997.

[5] R. Alur and D. Dill. Automata for modelling real-time systems. In *Theoretical Computer Science*, volume 126(2), pages 183–236, April 1994.

[6] Arnaud Bailly and Elie Najm. Timed automata with periodic clock updates.

[7] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model-checking for real-time systems.

[8] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the Association for Computing Machinery*, 30(2):323–342, 1983.

[9] L. Cacciari and O. Rafiq. Validation of protocols with temporal constraints, 1996.

[10] G. Coulson, G. S. Blair, J. B. Stefani, F. Horn, and L. Hazard. Supporting the real-time requirements of continuous media in open distributed processing. *Computer Networks and ISDN Systems*, 27(8):1231–1246, 1995.

[11] F. Eliassen and S. Mehus. Type checking stream flow endpoints. In *Middleware'98, The Lake District, England*, pages 305 – 322, 1998.

[12] S. Frølund and J. Koistinen. QML: A language for quality of service specication. Technical report, Technical Report HPL-98-10, February 1998.

[13] T.A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for realt-time systems. In *Logic in Computer Science*, 1992.

[14] K. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, October 1997.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts i and ii). In *Information and Computation*, volume 100, pages 1–77, 1992.

[16] Elie Najm, Abdekrim Nimour, and J-B Stefani. Infinite types for distributed objects interfaces. In *Proc. of IFIP conf. FMOODS'99*. Kluwer, Feb 1999.

[17] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*. Prentice Hall, 1995.

[18] Open distributed processing reference model, parts 1,2,3,4. ISO/IEC IS 10746-1..4 or ITU-T X901..4, 1995.

[19] Franz Puntigam. Types for active objects based on trace semantics. In Elie Najm et al., editor, *Proceedings of FMOODS'96*, Paris, France, 1996. Chapman & Hall.

[20] S. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *SAS 01: Static Analysis*, volume 2126 of *LNCS*, pages 375–394. Springer-Verlag, 2001.

[21] A. Ravavra and V.T. Vasconcelos. Behavioral types for a calculus of concurrent objects. In *Euro-Par'97*. Springer-Verlag, 1997.

[22] I. RFC. Rtp: A transport protocol for real-time applications, 1996.

[23] Jim Rumbaugh and Bran Selic. Using UML for modeling complex real-time systems. Rational Software Corp. and ObjectTime Limited.

[24] D. C. Schmidt. Real time CORBA with TAO (the ACE ORB). Technical report, http://www.cs.wustl.edu/schmidt/TAO.html, 2000.

[25] H. Schulzrinne, A. Rao, and L. Lanphier. Real time streaming protocol (RTSP). In *IETF draft 09*, 1998. http://www.cs.columbia.edu/ hgs/rtsp/.

[26] I. Wakeman, A. Jeffrey, R. Graves, and T. Owen. Designing a programming language for active networks. Technical report, 1998.

[27] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.