A contract-oriented middleware

Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia,
Alessandro Sebastian Podda, and Livio Pompianu

Università degli Studi di Cagliari, Italy

Abstract. Developing distributed applications typically requires to integrate new code with legacy third-party services, e.g., e-commerce facilities, maps, etc. These services cannot always be assumed to smoothly collaborate with each other; rather, they live in a "wild" environment where they must compete for resources, and possibly diverge from the expected behaviour if they find it convenient to do so. To overcome these issues, some recent works have proposed to discipline the interaction of mutually distrusting services through *behavioural contracts*. The idea is a dynamic composition, where only those services with *compliant* contracts can establish sessions through which they interact. Compliance between contracts guarantees that, if services behave honestly, they will enjoy safe interactions. We exploit a theory of timed behavioural contracts to formalise, design and implement a message-oriented middleware where distributed services can be dynamically composed, and their interaction monitored to detect contract violations. We show that the middleware allows to reduce the complexity of developing distributed applications, by relieving programmers from the need to explicitly deal with the misbehaviour of external services.

1 Introduction

Modern distributed applications are often composed by loosely-coupled services, which can appear and disappear from the network, and can dynamically discover and invoke other services in order to adapt to changing needs and conditions. These services may be under the governance of different providers (possibly competing among each other), and interact through open networks, where competitors and adversaries can try to exploit their vulnerabilities.

In the setting outlined above, developing trustworthy services and applications can be a quite challenging task: the problem fits within the area of computer security, since we have *adversaries* (in our setting, third-party services), whose exact number and nature is unknown (because of openness and dynamicity). Further, standard analysis techniques from programming languages theory (like e.g., type systems) cannot be applied, since they usually need to inspect the code of the whole application, while under the given assumptions one can only reason about the services under their control.

A possible countermeasure to these issues is to discipline the interaction between services through *contracts*. These are formal descriptions of service behaviour, in terms of, e.g., pre/post-conditions and invariants [21], behavioural types [17], etc. Contracts can be used at static or dynamic time to discover and bind Web services, and to guarantee they interact in a protected manner: when a service does not behave as prescribed by its contract, it can be blamed (and punished) for breaching the contract [32]. Although several models and architectures for contract-oriented services have been proposed in the last few years [13,36,38], further evidence is needed in order to put this paradigm at work in everyday practice. We also believe that contract-oriented services should be equipped with a formal semantics, in order to make their analysis possible.

Contributions. We formalise, design, implement, and validate a middleware which uses contracts to allow disciplined interactions between mutually distrusting services. The middleware is designed to support different notions of contract, which only need to share some high-level features:

– a *compliance* relation between contracts, which specifies when services conforming to their contracts interact correctly. The middleware guarantees that only services with compliant contracts can interact.
– an *execution monitor*, which checks if the actions done by the services conform to their contracts, and — otherwise — detects which services are *culpable* of a contract violation.

Building upon these basic ingredients, our middleware extends standard message-oriented middleware [4] (MOMs) by allowing services to advertise contracts, establish sessions between services with compliant contracts, and interact through these sessions. The execution monitor guarantees that, whenever a contract is violated, the culprit is sanctioned. Sanctions negatively affect the reputation of a service, and consequently its chances to establish new sessions.

We explore several ways to validate our middleware. First, we perform some scalability tests, to measure the execution time of the core primitives as a function of the number of advertised contracts. Second, we develop a distributed application (to solve an RSA factoring challenge [31]), involving a master and a population of workers, some of which do not always respect their contracts. We show that our service selection mechanism allows to automatically marginalize the dishonest services, without requiring the master to explicitly handle their misbehaviour. Third, we use the middleware as a (contract-oriented) communication layer for a real distributed application, i.e. a reservation marketplace where service providers can advertise resources, and clients can reserve them. Resources can be of heterogeneous nature, and their usage protocols are specified by contracts, which are handled by the middleware to guarantee safe interactions.

A public instance of the middleware is accessible from [7], together with all examples and experiments we carried out, and a suite of development tools.

Structure of the paper. In Section 2 we overview the middleware features. In Section 3 we introduce a process calculus to specify services. In Section 4 we illustrate the main design choices of the middleware, and in Section 5 we discuss its architecture; validation is then accomplished in Section 6. In Section 7 we discuss some related approaches, and in Section 8 we conclude.

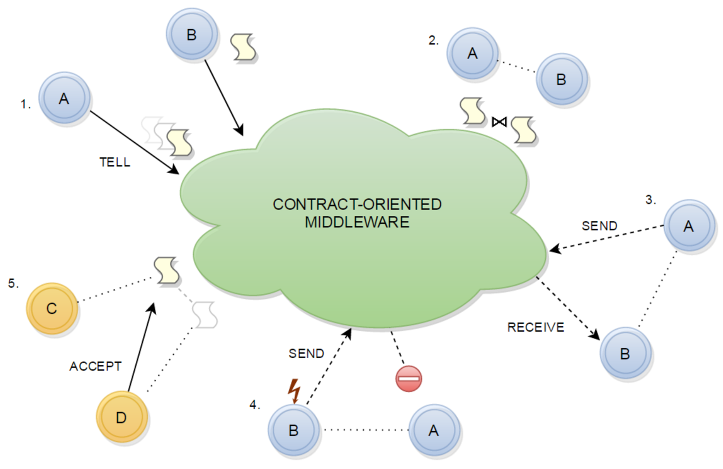


Fig. 1: A schema of the primitive behaviours.

2 The middleware at a glance

Figure 1 illustrates the main features of our middleware. In (1), the participant A advertises its contract to the middleware, making it available to other participants. In (2), the middleware determines that the contracts of A and B are compliant, and then it establishes a session through which the two participants can interact. This interaction consists in sending and receiving messages, similarly to a standard MOM [4]: for instance, in (3) participant A delivers to the middleware a message for B, which can then collect it from the middleware.

Unlike standard MOMs, the interaction happening in each session is monitored by the middleware, which checks whether contracts are respected or not. In particular, the execution monitor verifies that actions can only occur when prescribed by their contracts, and it detects when some expected action is missing. For instance, in (4) the execution monitor has detected an attempt of participant B to do some illegal action. Upon detection of a contract violation, the middleware punishes the culprit, by suitably decreasing its *reputation*. This is a measure of the trustworthiness of a participant in its past interactions: the lower is the reputation, the lower is the probability of being able to establish new sessions with it. The reputation system exploits some of the techniques in [35] to mitigate self-promoting attacks [23].

Item (5) shows another mechanism for establishing sessions: here, the participant C advertises a contract, and D just *accepts* it. Technically, this requires the middleware to construct the *dual* of the contract of C, to associate it with D, and to establish a session between C and D. The interaction happening in this session then proceeds as described previously.

Some simple examples of contract-oriented programs are shown in Appendix A.

3 Specifying contract-oriented services

In this section we introduce TCO$_2$ (for *timed CO$_2$*), a specification language for contract-oriented services. This is a timed extension of the process calculus in [9], through which we can specify services interacting through primitives analogous to those sketched in Section 2. Rather than giving a *tour de force* formalization of the whole middleware behaviour, we focus here on the core functionalities. Extending the calculus with more advanced features (like e.g. value passing, exceptions, reputation, etc.) can be done using standard techniques. A more detailed account of TCO$_2$ is contained in Appendix B.

The formalisation of TCO$_2$ is independent from the chosen contract language, as we only pivot on a few abstract operators and relations on contracts. In particular, we assume: (1) a *compliance relation* $\bowtie$, which relates two contracts whenever their interaction is "correct" [8]; (2) a predicate which says if a contract admits a compliant one; (3) a function $\mathrm{co}(\cdot)$ that, given a contract p, gives a contract compliant with p (when this exists); (4) a transition relation $\twoheadrightarrow$ between *contract configurations* γ, γ', which makes contracts evolve upon actions and time passing. We denote with $\Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$ the initial configuration of an interaction between A (with contract p) and B (with contract q).

The syntax of TCO$_2$ is defined as follows, where $x, y, \ldots \in \mathcal{V}$ are *variables*, $s, t, \ldots \in \mathcal{N}$ are *names*, and $u, v, \ldots \in \mathcal{V} \cup \mathcal{N}$. Further, we assume a set of *participants* (ranged over by $\mathsf{A}, \mathsf{B}, \ldots$), a set of message labels (ranged over by $\mathtt{a}, \mathtt{b}, \ldots$), and a set of process names (ranged over by $\mathrm{X}, \mathrm{Y}, \ldots$).

$$\begin{array}{rclclclclclcl}
S & ::= & \mathbf{0} & \mid & \mathsf{A}[P] & \mid & s[\gamma] & \mid & (u)S & \mid & S \mid S & \mid & \{\downarrow_u p\}_{\mathsf{A}} \\
P & ::= & \mathbf{0} & \mid & \mathrm{X}(\boldsymbol{u}) & \mid & \pi \,.\, P & \mid & (u)P & \mid & u \rhd \{\mathtt{a}_i \,.\, P_i\}_{i \in I} & & \\
\pi & ::= & \tau & \mid & \mathtt{tell} \downarrow_u p & \mid & \mathtt{send}_u\ \mathtt{a} & \mid & \mathtt{idle}(\delta) & \mid & \mathtt{accept}(x) & \mid & \bar{x}y \mid x(y)
\end{array}$$

Systems $S, S', \ldots$ are the parallel composition of *agents* $\mathsf{A}[P]$, *sessions* $s[\gamma]$, *delimited systems* $(u)S$, and *latent contracts* $\{\downarrow_u p\}_{\mathsf{A}}$. The latter represents a contract p (advertised by A) which has not been stipulated yet; upon stipulation, the variable u will be instantiated to a fresh session name.

Processes $P, Q, \ldots$ are: prefixed processes $\pi \,.\, P$; branching $u \rhd \{\mathtt{a}_i \,.\, P_i\}_{i \in I}$, which behaves as the continuation P_j upon receiving at session u a message $\mathtt{a}_j$; named processes $\mathrm{X}(\boldsymbol{u})$, used e.g., to specify recursive behaviours[1]; delimited processes $(u)P$; and the terminated process $\mathbf{0}$.

The *prefix* τ allows to do some internal actions, $\mathtt{tell}\downarrow_u p$ to advertise a contract p. Intuitively, u is a place-holder for the name of the session where p will be used. $\mathtt{accept}(x)$ allows to accept the contract received at x, $\mathtt{send}_u\ \mathtt{a}$

[1] We denote with $\boldsymbol{u}$ a sequence of names/variables, and we assume each X to have a unique definition $\mathrm{X}(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$, with the free vars of P included in $x_1, \ldots, x_j$.

$$\dfrac{\exists q : p \bowtie q}{\mathsf{A}[\mathtt{tell} \downarrow_u p \,.\, P] \xrightarrow{\mathtt{tell}} \mathsf{A}[P] \;\mid\; \{\downarrow_u p\}_{\mathsf{A}}} \qquad [\textsc{Tell}]$$

$$\dfrac{p \bowtie q \qquad \gamma = \Gamma_0(\mathsf{A} : p, \mathsf{B} : q) \qquad \sigma = \{{}^{s}/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \;\mid\; \{\downarrow_x p\}_{\mathsf{A}} \;\mid\; \{\downarrow_y q\}_{\mathsf{B}}) \xrightarrow{\mathtt{fuse}} (s)(S\sigma \;\mid\; s[\gamma])} \qquad [\textsc{Fuse}]$$

$$\dfrac{\gamma = \Gamma_0(\mathsf{A} : \mathrm{co}(q), \mathsf{B} : q) \qquad \sigma = \{{}^{s}/_{x}\} \qquad s \text{ fresh}}{(x)(\mathsf{A}[\mathtt{accept}(x) \,.\, P] \mid \{\downarrow_x p\}_{\mathsf{B}} \mid S) \xrightarrow{\mathtt{accept}} (s)(\mathsf{A}[P\sigma] \mid s[\gamma] \mid S\sigma)} \qquad [\textsc{Acpt}]$$

$$\dfrac{\gamma \overset{\mathsf{A}:\mathtt{!a}}{\twoheadrightarrow} \gamma'}{\mathsf{A}[\mathtt{send}_s\, \mathtt{a} \,.\, P] \;\mid\; s[\gamma] \xrightarrow{\mathtt{send}} \mathsf{A}[P] \;\mid\; s[\gamma']} \qquad [\textsc{Send}]$$

$$\dfrac{\gamma \overset{\mathsf{A}:\mathtt{?a}}{\twoheadrightarrow} \gamma' \qquad \mathtt{a} = \mathtt{a}_j}{\mathsf{A}[s \rhd \{\mathtt{a}_i \,.\, P_i\}] \;\mid\; s[\gamma] \xrightarrow{\mathtt{receive}} \mathsf{A}[P_j] \;\mid\; s[\gamma']} \qquad [\textsc{Recv}]$$

$$\dfrac{\gamma \overset{\delta}{\twoheadrightarrow} \gamma'}{s[\gamma] \xrightarrow{\delta} s[\gamma']} \; [\textsc{Delay-}\gamma] \qquad\qquad \dfrac{0 < \delta' \leq \delta}{\mathsf{A}[\mathtt{idle}(\delta) \,.\, P] \xrightarrow{\delta'} \mathsf{A}[\mathtt{idle}(\delta - \delta') \,.\, P]} \; [\textsc{Idle}]$$

Fig. 2: Reduction semantics of TCO_2 (full set of rules in Appendix B.

to send a message $\mathtt{a}$ at session u, and $\mathtt{idle}(\delta)$ to delay by a time $\delta \in \mathbb{R}_{\geq 0}$; the prefixes $\bar{x}y$ and $x(y)$ allow for the usual channel-based communication à la π-calculus [27]. Note that the primitive $\mathtt{tell}$ allows process to communicate (when their contracts will be fused), in the absence of any pre-shared name.[2]

The semantics of TCO_2 is summarised in Figure 2 as a reduction relation between systems. The labels are used to separate urgent actions from non-urgent ones. When an urgent label is enabled, time is not allowed to pass (similarly to the `asap` operator in U-LOTOS [30]). This enforces a fairness property: if an urgent action is enabled, the scheduler can not prevent it by letting time pass. In TCO_2, every discrete action is urgent, except for $\mathtt{fuse}$; this formalises the intuition that a session between two compliant contracts can be created at any time by the middleware, independently from the participants' behaviour.

Rule [Tell] adds to the system a latent contract $\{\downarrow_u p\}_{\mathsf{A}}$, if p admits a compliant contract. Rule [Fuse] searches the system for compliant pairs of latent contracts, i.e. $\{\downarrow_x p\}_{\mathsf{A}}$ and $\{\downarrow_y q\}_{\mathsf{B}}$ such that $p \bowtie q$ (and $\mathsf{A} \neq \mathsf{B}$). Then, a fresh session s containing the initial configuration $\gamma = \Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$ is established, and the name s is shared between A and B. Rule [Acpt] allows A to accept a latent contract q, which is passed through the channel x; then, the contract of A at s will be $\mathrm{co}(q)$. Rule [Send] allows A to send a message $\mathtt{!a}$ to the other endpoint of session s. This is only permitted if the contract configuration at s can take a transition on $\mathsf{A} : \mathtt{!a}$, whereas messages not conforming to the contract will make

[2] To avoid confusion between "channel-kinded" variables used in input/output prefixes and "session-kinded" variables, we forbid processes which improperly mix them, like e.g. $\mathtt{tell} \downarrow_y p.y(x)$, where y is used both as a session variable and a channel variable.

A culpable of a violation. Rule [Recv] allows A to receive a message $\mathtt{a}_j$ from the other endpoint of s, and to behave like the continuation P_j. Rule [Delay-γ] allows a session $s[\gamma]$ to idle, if permitted by the contract configuration γ at s (note that idling may make one of the participants culpable). Rule [Idle] is standard [30], and it allows a process to idle for a certain time δ. The other rules for dealing with time (and with the other constructs) are reported in Figure 7.

A simple interaction in TCO_2 is shown in Example 2.

4 System design

In this section we show how the interaction paradigm sketched in Section 2 (and formalised in Section 3) is supported by our middleware, and we illustrate the main design choices.

4.1 Specifying contracts

Although the design of the middleware is mostly contract-agnostic, in this paper we describe and evaluate *timed session types* [6] (TSTs) as a particular instance of contracts. TSTs extend binary session types [37,24] with clocks and timing constraints, similarly to the way timed automata [2] extend (untimed) finite state automata. We give below a brief overview of TSTs, and we refer to [6] for the full technical development. Clocks $x, y, \ldots$ are variables over $\mathbb{R}_{\geq 0}$, which can be reset, and used within *guards* $g, g', \ldots$. Atomic guards are timing constraints of the form $x \circ d$ or $x - y \circ d$, where $d \in \mathbb{N}$ and $\circ \in \{<, \leq, =, \geq, >\}$, and they can be composed with the boolean connectives $\wedge$, $\vee$, $\neg$.

A TST p (Definition 1) describes the behaviour of a single participant involved in an interaction. An *internal choice* $\sum_i \mathtt{!a}_i\{g_i, R_i\} \,.\, p_i$ models the fact that its participant wants to do one of the outputs with label $\mathtt{a}_i$ in a time window where the guard g_i is true; the clocks in R_i will be reset after the output is performed. An *external choice* $\&_i \mathtt{?a}_i\{g_i, R_i\} \,.\, q_i$ models the fact that its participant is available to receive each message $\mathtt{a}_i$ at *any instant* within the time window where the guard g_i is true; furthermore, the clocks in R_i will be reset after the input is received. The term $\mathbf{1}$ denotes *success* (i.e., a terminated interaction). Infinite behaviour can be specified through recursion $\mathrm{rec}\, X \,.\, p$.

Definition 1 (Timed session types [6]). Timed session types $p, q, \ldots$ *are terms of the following grammar:*

$$p \;::=\; \mathbf{1} \;\;\big|\;\; \textstyle\sum_{i \in I} \mathtt{!a}_i\{g_i, R_i\} \,.\, p_i \;\;\big|\;\; \&_{i \in I} \mathtt{?a}_i\{g_i, R_i\} \,.\, p_i \;\;\big|\;\; \mathrm{rec}\, X \,.\, p \;\;\big|\;\; X$$

where (i) the set I is finite and non-empty, (ii) the labels in internal/external choices are pairwise distinct, (iii) recursion is guarded and considered up-to unfolding. True guards, empty resets, and trailing occurrences of $\mathbf{1}$ *can be omitted.*

Message labels are grouped into *contexts*, which can be created and made public through the middleware APIs. Each context defines the labels related to an application domain, and it associates each label with a *type* and a *verification link*. The type (e.g., `int`, `string`) is that of the messages exchanged with that label. The verification link is used by the runtime monitor (Section 4.4) to delegate the verification of messages to a trusted third party. For instance, the middleware supports Paypal as a verification link for online payments (see Section 6.3). The context also specifies the duration of a time unit: the shortest time unit supported by the middleware is that of seconds, which is also the one we use in all the examples in this paper.

4.2 Advertising contracts

Once a contract has been created, a participant can advertise it to the middleware. At that point, the contract stays *latent* until the middleware finds a *compliant* one, i.e. another latent contract with whom the interaction is guaranteed not to get stuck. When this is found, the middleware creates a *session* between the two participants: the session consists of a private channel name and a *contract configuration*, which keeps track of the state of the contract execution.

The notion of compliance between TSTs (Definition 6 in [6]) is based on a transition system over contract configurations (Definition 5 in [6]). Contract configurations have the form $(p, \nu) \mid (q, \eta)$, where p, q are TSTs, and ν, η are *clock evaluations* (i.e., functions from clocks to $\mathbb{R}_{\geq 0}$); in the initial configuration $\Gamma_0(\mathsf{A}: p, \mathsf{B}: q)$, the clock evaluations map each clock to 0. Intuitively, p and q are compliant (in symbols, $p \bowtie q$) if, in all reachable configurations, the "required" behaviour of p (i.e., the branches in its internal choice) is "offered" by q in an external choice, while respecting the time constraints.

Example 1. Let $p = \texttt{?a}\{x \leq 2\} \ \& \ \texttt{?b}\{x \leq 5\}$, and consider the following TSTs:

$$q_1 = \texttt{!a}\{y \leq 1\} \qquad q_2 = \texttt{!a}\{y \leq 3\} \qquad q_3 = \texttt{!a}\{y \leq 2\} + \texttt{!c}\{y \leq 2\}$$

We have that $p \bowtie q_1$: indeed, q_1 wants to output $\texttt{a}$ within one time unit, and p is available to input $\texttt{a}$ for two time units; compliance follows because the time window for the input includes that for the output. On the contrary, $p \not\bowtie q_2$, since the time window required by q_2 is larger than the one offered by p. Finally, $p \not\bowtie q_3$: although the timing constraints for label $\texttt{a}$ match, q_3 can also choose to send $\texttt{c}$, which is not among the labels offered by p in its external choice.

Deciding compliance. Compliance between TSTs is decidable (Theorem 1 in [6]). To check if $p \bowtie q$, we use the encoding in [6] to translate p and q into Uppaal timed automata [11], and then we model-check the resulting network for deadlock freedom. This amounts to solve the reachability problem for timed automata, whose theoretical worst-case complexity is exponential (more precisely, the problem is PSPACE-complete [2]). In practice, the overall execution time for compliance checking for the TSTs in our test suite is in the order of milliseconds; e.g., in the experimental setup described in Section 6, it takes approximately 20ms to check compliance between the largest TSTs on our hand, i.e. those modelling

PayPal Protection for Buyers [1]. Since, however, the execution time of compliance checking is non-negligible, we do not perform an exhaustive search when searching the contract store for compliant pairs of contracts; rather, we use the techniques described in the following paragraphs to reduce the search space.

Compliance pre-check. When a TST is advertised, the middleware stores in its database the associated timed automaton (which is then computed only once for each TST), and a *digest* of the TST; this digest comprises its context, and one bit which tells whether its top-level operation is an internal or an external choice (up-to unfolding). When looking for a contract compliant with p, the digests are used to rule out (without invoking the Uppaal model checker) some contracts which are surely *not* compliant with p. In particular, we rule out those q belonging to a context different from that of p, and those with the same top-level operator as p (as internal choices can only be compliant with external ones, and *vice versa*). The remaining contracts are potentially compliant with p, and so we restrict the search space to them. The search also takes into account the reputation of the participants who have advertised these contracts, as described in the following paragraph.

Reputation. The middleware assigns to each participant a *reputation*, which measures its ability to respect contracts. Intuitively, the reputation is increased when the participant successfully completes a session, while it is decreased when it is found culpable of a contract violation (more details about the formulation of the reputation system in Section 4.4). Reputation is used to sort latent contracts when searching for compliant pairs: the higher the participant's reputation, the higher the probability to establish a session with it. When looking for a contract compliant with p, we first construct the list of contracts potentially compliant with it (sorted by descending reputation). Then, we randomly choose one of them, according to the folded normal probability distribution. This causes contracts with high reputation to be chosen with high probability, while giving some chances also to contracts with low reputation. If the chosen contract is not compliant with p, it is discarded, and the algorithm chooses another one.

Checking the existence of a compliant. Not all TSTs admit a compliant one. For instance, no contract can be compliant with $p = \texttt{!a}\{y < 7\}.\texttt{?b}\{y < 5\}$, because if p outputs $\texttt{a}$ at time 6, the counterpart cannot send $\texttt{b}$ in the required time constraint. A sound and complete decision procedure for the existence of a compliant is developed in [6]. When advertising a contract, we use this procedure to rule out those contracts which do not admit a compliant one.

4.3 Accepting contracts

As discussed in Section 2, a participant A can establish a session with B by accepting one of its contracts, whose identifier has been made public by B. Technically, when A declares to accept a contract p, the middleware constructs the *dual* of p, and assigns it to A. The dual of p is the greatest contract compliant with p, according to the subcontract preorder [6]: intuitively, it is the one whose offers match all of p's requests, and whose requests match all p's offers.

Unlike in the untimed case, the naïve construction of the dual of a TST p (i.e., the one which simply swaps inputs with outputs and internal choices with external ones) does not always produce a compliant TST. For instance, the naïve dual of $p = \texttt{?a}\{x \leq 2\}.\,\texttt{?b}\{x \leq 1\}$ is $q = \texttt{!a}\{x \leq 2\}.\,\texttt{!b}\{x \leq 1\}$, which is *not* compliant with p. Indeed, since q can output $\texttt{!a}$ at any time $1 < \delta \leq 2$, the interaction between p and q can become deadlock, and so they are not compliant.

The dual construction used by the middleware is the one defined in [6], which guarantees to obtain a TST compliant with p, if it exists. Roughly, the construction turns all the internal choices into external ones (without changing guards), and it turns external choices into internal ones, updating the guards to preserve future interactions. For instance, in the example above we obtain the TST $\texttt{!a}\{x \leq 1\}.\,\texttt{!b}\{x \leq 1\}$, which is compliant with p.

4.4 Service interaction and runtime monitoring

When a session is established, the participants at the two endpoints can interact by sending and receiving messages. At a more concrete level, sending a message through a session is implemented by posting the message to the middleware, through its RESTful API. The middleware logs the whole interaction history, by recording and timestamping all the messages exchanged in the session. Receiving a message is also implemented by invoking the middleware API; upon a receive request, the middleware inspects the session history to retrieve the first unread message (which is then marked as read). The interaction over the session is asynchronous, as the middleware (similarly to a standard MOM) interprets the session history as two unbounded FIFO buffers containing the messages sent by the two endpoints[3]. However, differently from standard MOMs, our middleware monitors the interaction to verify that contracts are respected.

The runtime monitor processes each message exchanged in a session, by querying the verification link associated to it (to detect whether the message is genuine or not), and by checking that the message is permitted in the current contract configuration. Then, the monitor computes who is in charge of the next move, and, in case of contract violations, it detects which of the two participants is culpable. A participant A can become culpable for different reasons:

1. A sends a message not expected by her contract;
2. A's contract is an internal choice, but A loses time until all the branches become unfeasible (i.e., the time constraints are no longer satisfiable);
3. A sends some action at a valid time, but the trusted third party (associated to the action by the verification link) rejects it. For instance, this can happen if A tries to send a fake payment, but Paypal does not certify it.

[3] Asynchronous communication is possible despite TSTs having a synchronous semantics, as the middleware is delegated to receive messages on behalf of the recipient.

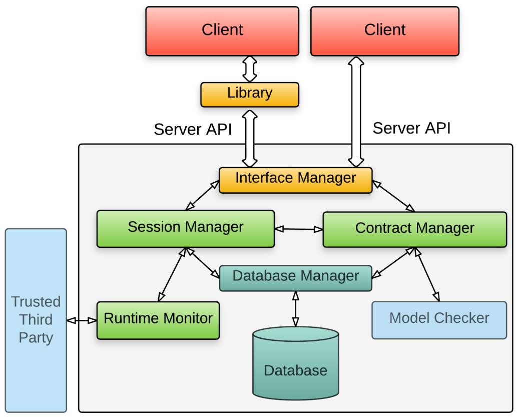


Fig. 3: A diagram of the middleware architecture.

The monitor guarantees that, in all possible states of the interaction, only one of the participants can be in charge of the next action; if no one is in charge nor culpable, then both participants have reached success (Lemma 3 in [6]).

Once a session terminates (either succesfully or not), the reputation of the involved participants is updated. If the session terminates successfully, then the reputation of both participants is increased; otherwise, the reputation of the culpable participant is decreased, while the other participant's reputation is increased. Further, we make participants consume reputation points each time they enter in session, and we use the *fading memories* technique of [35] to calculate the reputation value without recording the whole history of interactions. We weight recent negative behavior more than old positive behaviour, in order to mitigate *self-promoting attacks*, where a malicious participant tries to gain reputation by running successful sessions with himself or with some accomplices [23].

5 System architecture

The middleware is a Java RESTful Web service; the primitives described in Section 4 are organised in components, as shown in Figure 3. We have adopted a 3-tier architecture, consisting of a presentation layer, a business logic layer, and a data storage layer. The Interface Manager, which is the only component in the presentation layer, offers APIs to query the middleware, through HTTP POST requests. APIs can be accessed through language-specific libraries, which allow for an object-oriented programming style (see Appendix A). The data storage layer comprises a relational DB and a Database Manager, which takes care of handling queries, managing the cache, and modelling the data used in the other layers. The business logic layer manages contracts and sessions. More specifically, the Contract Manager performs the contract validation, advertisement (as in Section 4.2), and `accept` requests (Section 4.3); the Session Manager establishes sessions, by allowing clients to send and receive messages, managing the session history, and querying the Runtime Monitor to detect contract violations.

A client advertises a contract p with the `tellContract` API of the Interface Manager, encoding the required data in the JSON data exchange format. The Interface Manager validates p, then it asks the Contract Manager to store it and to find a compliant contract, as outlined in Section 4.2. If no latent contracts are compliant with p, then p is kept latent, otherwise a new session is established. The Interface Manager also provides the `acceptContract` API, which requires the Contract Manager to compute the dual of a latent contract q, whose identifier has been made public by another participant.

When a session is established, participants can query the middleware to get the current time, to send and receive messages, to check culpability, etc. The Interface Manager provides the methods for handling such requests, delegating the internal operations to the Session Manager. When a participant `send`s a message, the Session Manager uses the Runtime Monitor to determine whether the action is permitted (and in case it is not, to assign the blame). If the action is permitted, the message is stored by the Database Manager, and then forwarded to the other participant upon a `receive`. To verify a message, the Runtime Monitor can invoke a trusted third party: if the verification fails, the action is rejected (so, our monitor implements *truncation*, in the terminology of [26]).

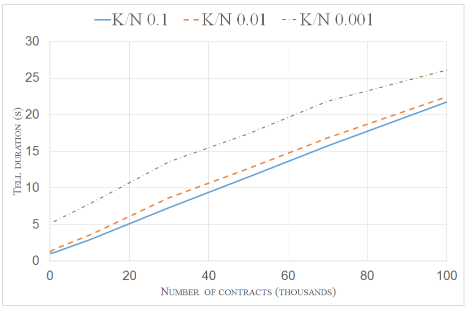
6 Validation

In this section we validate our middleware, mainly focussing on the aspects related to system scalability (Section 6.1), and to the effectiveness of the reputation system to rule out services not respecting contracts (Section 6.2). We also discuss how the middleware has been exploited to implement a large software system for managing online reservations (Section 6.3).

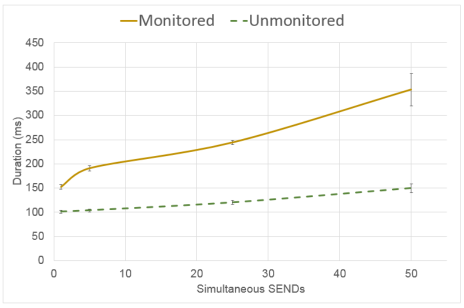
We carry out our experiments using a public instance of the middleware, accessible from the Web at `co2.unica.it`. The instance is a Web service running in a dedicated cloud server, equipped with a quad-core Intel Xeon CPU @ 2.27GHz, 16GB of RAM and a 50GB SSD hard drive; the server runs Ubuntu 14.04 LTS, with Apache Tomcat and Oracle MySQL. Clients are tested in standard desktop PCs and laptops, while the multi-threaded simulations are executed in a high-level desktop configuration, with an octa-core Intel Core i7 @ 4.00GHz and 16Gb of memory, running Microsoft Windows 7 and Oracle JRE 1.7.

6.1 Scalability

In this section we assess the scalability of our middleware. We start by benchmarking the `tell` primitive, which triggers a search for compliant pairs of TSTs in the contract store. This is the most computationally expensive operation in the middleware: although the heuristics discussed in Section 4.2 allow for limiting the number of calls to the Uppaal model checker, the execution time of



(a) Duration of `tell` p (in seconds).



(b) Duration of `send` (in milliseconds).

Fig. 4: Results of the scalability tests. In (a), K is the number of contracts compliant with p, and N is the total number of contracts.

a `tell` could be non-negligible for a high number of latent contracts. So, we measure the execution time of `tell` p as a function of the number of TSTs in the contract store, and of the number of latent TSTs compliant with p.

Our second experiment concerns the performance of the runtime monitor. As described in Section 4.4, this component processes all the messages exchanged in sessions, to check if contracts are respected. Potentially, this could introduce a relevant computational overhead, so we measure the execution time of `send` in case the runtime monitor is turned on, or off. Note that, while the duration of `tell` does not affect the interaction between the participants once a session is established, a slowdown of the `send` can make an otherwise-honest participant culpable for not respecting some deadline. So, it is important that the overhead of the runtime monitor is negligible, w.r.t. the time scale of temporal constraints.

We build our scalability tests upon the discrete-event simulator DESMO-J [19], and the statistical model-checker MultiVeStA [34]. In particular, we use DESMO-J to define a single instance of the simulation, and MultiVeStA to run sequences of simulations until reaching a given confidence interval.

Tell. We test the execution time of `tell` p as a function of the number N of contracts stored in the middleware. The contract p used in our experiments is a simplified version of the Paypal Protection for Buyers (Example 1 in [6]). We assume that, among the N contracts, only $K \ll N$ are compliant with p, while the remaining $N - K$ are not, but they still pass the pre-check discussed in Section 4.2 (so, we are considering a worst-case scenario, because in the average case we expect that only a fraction of the contracts would pass the pre-check). We populate the contract store by choosing at each step whether to insert a contract compliant with p or a non-compliant one, according to a random weighted probability. Then, with DESMO-J we execute `tell` p, and we measure its execution time. MultiVeStA makes DESMO-J execute this simulation for several times, each time collecting the new `tell` duration and updating the average and the standard deviation; the simulations stop when the average fits into the confidence interval.

The results of our experiments are shown in Figure 4. As we can see, the `tell` duration grows linearly with N, and it increases by a constant when the percentage K/N of contracts compliant with p decreases; note that the slope of the curves does not seem to be significantly affected by K/N.

Runtime monitor. The goal of this experiment is to quantify how the execution of a large number of simultaneous `send` affects the performance of the middleware. To achieve this goal, we use a multi-threaded simulation, where all the threads advertise a contract with an internal sum, wait the session to be established, and then simultaneously perform the `send`. We repeat the measure of the `send` duration until its standard deviation fits into the confidence interval. The results of this experiment are reported in Figure 4b, which shows that the execution of a large number of simultaneous `send`s penalizes the duration of the request, compared to the situation where the runtime monitor is switched off. However, the performance degradation seem to grow sub-linearly in the number of simultaneous requests, and in any case it is negligible w.r.t. the time scale of temporal constraints (1 time unit = 1 second).

6.2 A distributed experiment: RSA cracking

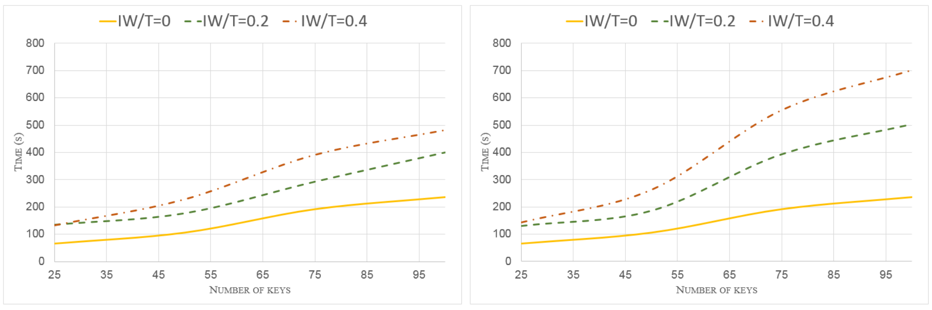
Consider a service (hereafter referred to as *master*, or just M) who wants to solve a cryptographic problem by exploiting the computational resources of external nodes (hereafter called *workers*, or W) distributed over the network. In particular, M wants to crack a set of public RSA keys, in order to get the corresponding private keys. However, the master does not know the network structure (i.e., how many workers are available, where they are located, and how they are connected), and it does not have any pre-shared channel for communicating with them. Furthermore, the master does not trust the workers: they are not bound to run any particular cracking algorithm, they can return wrong/incomplete results, or they can fail to answer within the expected deadline.

To cope with these issues, the master exploits our middleware to automatically discover and invoke suitable workers. For each public key in its set, the master spawns a process which advertises the contract:

$$p_{\mathsf{M}} \;=\; \texttt{!pubkey}\{;x\}.\,(\texttt{?confirm}\{x < 15\}.\,\texttt{?result}\{x < 90\}.\,\texttt{!pay1xbt}\{x < 120\} \\ \quad \& \ \texttt{?abort}\{x < 15\})$$

Here, M is promising to send a public key (`pubkey`); doing so triggers a reset of the clock x. Then, the worker has 15 seconds to either `confirm` that he will carry on the task, or `abort` (e.g., if the key is considered too strong). If the worker confirms, it must return the corresponding `result` (a private key) within 90 seconds since the public key was sent (the correctness of the result is checked by a trusted third party,[4] specified by the context of p_{W}); finally, M rewards the worker

[4] Note that verifying the correctness of private keys has a polynomial complexity in the number of bits of the public key, while the problem of cracking RSA keys is considered to be exponentially hard.



(a) OET with reputation enabled. (b) OET with reputation disabled.

Fig. 5: Overall Execution Time as a function of the number of keys to be broken. IW is the number of inefficient workers, and T is the total number of workers.

with 1 bitcoin (pay1xbt). At runtime, the master behaves as prescribed by its contract; if the worker accepts the public key and it returns the corresponding private key, then M removes that public key from the list; otherwise, it advertises another instance of p_{M}, and when the session is established it sends the same public key to another worker.

The advantage offered by the middleware in terms of code succinctness is clear, as the search of workers, the establishment of sessions, and the runtime monitoring is completely transparent to programmers. So, we assess below the reputation system implemented in the middleware (Sections 4.2 and 4.4). In particular, we measure the time taken by the master for cracking all the public keys in its list (*Overall Execution Time, OET*). We do this in two configurations of the middleware: the one where the reputation system is turned on, and the one where it is turned off. Our conjecture is that turning the reputation system on will reduce the OET, because it increases the probability of establishing sessions with *honest* workers which produce correct results while respecting deadlines.

In our experiments, we assume that workers are drawn from two different classes: those using an efficient cracking algorithm, which always return the correct result within the deadline; and those using an inefficient algorithm, which sometimes may miss the deadline, because the computation takes too long. We also assume that the number of public keys is bigger than the number of workers, so each of them may receive many keys to break. Each worker iteratively advertises its contract (the dual of p_{M}), then waits for a public key, runs the cracking algorithm, and finally return the private key to the master.

The results of our experiment are shown in Figure 5, where we measure the OET as a function of the number of keys to be broken, and of the ratio between efficient and inefficient workers. The solid curve is identical in the two figures, since the reputation system does not affect the selection of workers when there are only efficient ones. In the dashed curve and in the dot-dashed one the percentage of inefficient workers grows (to 20% and 40%, respectively), and we see that the OET grows accordingly when the reputation system is turned off.

This is because the reputation system penalizes inefficient workers, by reducing the probability they can establish sessions with the master.

6.3 Case study: a contract-oriented reservation marketplace

To test the effectiveness and versatility of our middleware for the development of real distributed applications, we have exploited it as a *contract layer* in a software infrastructure for online reservations [5]. The infrastructure acts as a marketplace wherein service providers make available their resources, which can then be searched, reserved, and used by clients. These reservations can be of arbitrary nature, as the infrastructure features an abstract model of resources, which can be suitably instantiated by service providers. The infrastructure has been tested with various instances of providers, offering e.g. car sharing facilities, medical appointments, and hotel accommodations.

The reservation marketplace adds a search layer to that of the middleware: clients can search among the resources, and when they find a suitable one they can `accept` its contract. Contracts are constructed by service providers through a GUI, starting from a template and then selecting among various options and parameters. For instance, a simple contract for a service provider is the following:

$$\begin{aligned}
p &= \texttt{?pay}\{t < d_{pay}\}.\,\texttt{?details}\{t < d_{pay} + 60, t\}.\,p' \quad \& \quad \texttt{?cancel}\{t \leq d_{cc}\} \\
p' &= \mathrm{rec}\, X.\, \big(\texttt{?feedback}\{t < d_{fb}\} \;\; \& \; \texttt{?cancel}\{t < d_{cc}\}.\,\texttt{!refund}\{t \leq d_{rf}\} \;\; \& \; p''\big) \\
p'' &= \texttt{?move}\{t < d_{mv}\}.\,\big(\texttt{!ok}\{t < d_{ok}\}.\,\texttt{?feedback}\{t < d_{fb}\} + \texttt{!no}\{t < d_{no}\}.\,X\big)
\end{aligned}$$

The provider waits for a `pay`ment and some `details` about the reservation; then, it gives the client a choice among three actions: accept the reservation (and leave a `feedback`), `cancel` it (which involves a `refund`), or `move` it to another date. Moving reservations is not always permitted (e.g., because the new date is not available), so when the provider notifies `no`, it allows the client to try again.

Contracts are enforced by the runtime monitor of the middleware, which delegates the verification of `pay`ments and `refund`s to PayPal. Clients and providers can check the state of their contracts through the GUI, which at any time also highlights the permitted actions and their deadlines.

7 Related work

Our middleware builds upon CO_2 [10,9], a core calculus for contract-oriented computing; in particular, the middleware implements all the main primitives of CO_2 (`tell`, `send`, `receive`), and it introduces new concepts, like e.g. the `accept` primitive, time constraints, and reputation.

From the theoretical viewpoint, the idea of constraint-based interactions has been investigated in other process calculi, e.g. Concurrent Constraint Programming (CCP [33]), and cc-pi [18], albeit the kind of interactions they induce is quite different from ours. In CCP, there is a global constraint store through which processes can interact by telling/asking constraints. In cc-pi, interaction is a mix of name communication *à la* π-calculus [27] and `tell` *à la* CCP (which is used to put constraints on names). E.g., $\bar{x}\langle z\rangle$ and $y\langle w\rangle$ can synchronise iff the constraint store entails $x = y$; when this happens, the equality $z = w$ is added to the store, unless making it inconsistent. In cc-pi consistency plays a crucial role: `tell`s *restricts* the future interactions with other processes, since adding constraints can lead to more inconsistencies; by contrast, in our middleware telling a contract *enables* interaction with other services, so consistency is immaterial.

The notion of time in behavioural contracts has been studied in [16], which addresses a timed extension of multi-party asynchronous session types [25]; however, the goals of [16] are quite different from ours. The approach pursued in [16] is top-down: a *global type* (specifying the overall communication protocol of a set of services, and satisfying some safety properties, e.g. deadlock-freedom) is projected into a set of *local types*; then, a composition of services preserves the properties of the global type if each service type-checks against the associated local type. Our middleware fosters a different approach to service composition: a distributed application is built bottom-up, by advertising contracts to delegate work to external (unknown and untrusted) services. Both our approach and [16,29] use runtime monitoring to detect contract violations and assign the blame; additionally, in our middleware these data are exploited as an automatic source of information for the reputation system. Another formalism for communication protocols with time constraints is proposed in [20], where live sequence charts are extended with a global clock. The approaches in [16,20] cannot be directly used in our middleware, because they do not provide algorithms to decide compliance, or to construct a contract compliant with a given one.

From the application viewpoint, several works have investigated the problem of *service selection* in open dynamic environments [3,28,39,40]. This problem consists in matching client requests with service offers, in a way that, among the services respecting the given functional constraints, the one which maximises some *non-functional* constraints is selected. These non-functional constraints are often based on quality of service (QoS) metrics, e.g. cost, reputation, guaranteed throughput or availability, etc. The selection mechanism featured by our middleware does not search for the "best" contract compliant with a given one (actually, typical compliance relations in behavioural contracts are qualitative, rather than quantitative); the only QoS parameter we take into account is the reputation of services (see Section 4.2). In [40,3] clients can require a sequence of tasks together with a set of non-functional constraints, and the goal is to find an assignment of tasks to services which optimises all the given constraints. There are two main differences between these approaches and ours. First, unlike behavioural contracts, tasks are considered as atomic activities, not requiring any interaction between clients and services. Second, unlike ours, these approaches do not consider the possibility that a service may not fulfil the required task.

In the work [28], a service selection mechanism is implemented where functional constraints can be required in addition to QoS constraints: the first are described in a web service ontology, while the others are defined as requested and offered ranges of basic QoS attributes. A runtime monitor and a reputation system are also implemented, which, similarly to ours, help to marginalise those services which do not respect the advertised QoS constraints. Some kinds of QoS constraints cannot be verified by the service broker, so their verification is delegated to clients. This can be easily exploited by malicious participants to carry on *slandering attacks* to the reputation system [23]: an attacker could destroy another participant's reputation by involving it in many sessions, and each time declare that the required QoS constraints have been violated. In our middleware there is no need to assume participants trusted, as the verification of contracts is delegated to the middleware itself and to trusted third parties.

8 Conclusions

We have explored a new application domain for behavioural contracts, i.e. their use as interaction protocols in MOMs. In particular, we have developed a middleware where services can advertise contracts (in the form of timed session types, TSTs), and interact through sessions, which are created only between services with compliant contracts. To implement the middleware primitives, we have exploited much of the theory of TSTs in [6]: in particular, a decidable notion of compliance between TSTs, a decidable procedure to detect when a TST admits a compliant one (and, if so, to construct it), and a decidable runtime monitoring.

We have validated our middleware through a series of experiments. The scalability tests (Section 6.1) seem to suggest that the performance of middleware is acceptable for up to $100K$ latent contracts. However, we feel that good performance can be obtained also for larger contract stores, for two reasons. First, in our experiments we have considered the pessimistic scenario where *all* latent contracts in the store are potentially compliant with a newly advertised one. Second, the current prototype of the middleware is sequential and centralised: parallelising the instances of the compliance checker, or distributing those of the middleware, would result in a performance boost. The experiments about the reputation system (Section 6.2) show that the middleware can relieve developers from dealing with misbehaviour of external services, and still obtain efficient distributed applications, which dynamically reconfigure themselves to foster the interaction among trustworthy services.

Although in this paper we have focussed on TSTs, the middleware only makes mild assumptions about the nature of contracts, e.g., that their observable actions are `send` and `receive`, and that they feature some notion of compliance with a sound (but not necessarily complete) verification algorithm. Hence, with minor efforts it would be possible to extend the middleware to support other contract models. For instance, communicating timed automata [15] (which are timed automata with unbounded communication channels) would allow for multi-party sessions, while session types with assertions [14], would allow for an explicit specification of the constraints among the values exchanged in sessions.

Besides the issues related to the expressiveness of contracts and to the scalability of their primitives (e.g., service binding and composition, runtime monitoring, *etc.*), we believe that also security issues should be taken into account:

indeed, attackers could make a service sanctioned by exploiting discrepancies between its contracts and its actual behaviour. These mismatches are not always easy to spot (see e.g. the online bookstore example in Appendix A.4); analysis techniques are therefore needed to ensure that a service will not be susceptible to this kind of attacks.

Acknowledgments

The authors thank Maria Grazia Patteri, Mirko Joshua Mascia and Stefano Lande for their assistance in setting up the evalution and the case studies, and Alceste Scalas for the discussion about Java APIs. This work is partially supported by Aut. Reg. of Sardinia grants L.R.7/2007 CRP-17285 (TRICS), P.I.A. 2010 ("Social Glue"), P.O.R. F.S.E. Operational Programme of the Aut. Reg. of Sardinia, EU Social Fund 2007-13 – Axis IV Human Resources, Objective l.3, Line of Activity l.3.1), by MIUR PRIN 2010-11 project "Security Horizons", and by EU COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems" (BETTY).

References

1. PayPal buyer protection. `https://www.paypal.com/us/webapps/mpp/ua/useragreement-full#13`. Accessed: July 8, 2015.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
3. D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Software Eng.*, 33(6):369–384, 2007.
4. G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proc. Distributed Computing*, pages 1–18, 1999.
5. M. Bartoletti, T. Cimoli, M. Murgia, M. G. Patteri, M. J. Mascia, A. S. Podda, and L. Pompianu. COREserve: a contract-oriented reservation marketplace. `http://coreserve.unica.it`, 2015.
6. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. Compliance and subtyping in timed session types. In *FORTE 2015*, pages 161–177, 2015.
7. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A contract-oriented middleware, 2015. `http://co2.unica.it`.
8. M. Bartoletti, T. Cimoli, and R. Zunino. Compliance in behavioural contracts: a brief survey. In *Programming Languages with Applications to Biology and Security*, volume 9465 of *LNCS*. Springer, 2015.
9. M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO$_2$. *Sci. Ann. Comp. Sci.*, 22(1), 2012.
10. M. Bartoletti and R. Zunino. A calculus of contracting processes. In *LICS*, 2010.
11. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
12. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *ACPN*, pages 87–124, 2003.
13. J. O. Blech, Y. Falcone, H. Rueß, and B. Schätz. Behavioral specification based runtime monitors for OSGi services. In *ISoLA*, pages 405–419, 2012.

14. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, 2010.
15. L. Bocchi, J. Lange, and N. Yoshida. Meeting deadlines together. In *CONCUR*, 2015. To appear.
16. L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *CONCUR*, pages 419–434, 2014.
17. A. Brogi, C. Canal, and E. Pimentel. Behavioural types for service integration: Achievements and challenges. *ENTCS*, 180(2):41–54, 2007.
18. M. G. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP*, pages 18–32, 2007.
19. J. Göbel, P. Joschko, A. Koors, and B. Page. The discrete event simulation framework DESMO-J: review, comparison to other frameworks and latest development. In *Proc. ECMS*, pages 100–109, 2013.
20. D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *MASCOTS*, pages 193–202, 2002.
21. R. Heckel and M. Lohmann. Towards contract-based testing of Web services. *Electr. Notes Theor. Comput. Sci.*, 116:145–156, 2005.
22. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.
23. K. J. Hoffman, D. Zage, and C. Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42(1), 2009.
24. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, 1998.
25. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, 2008.
26. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.
27. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1), 1992.
28. A. Mukhija, A. Dingwall-Smith, and D. Rosenblum. QoS-aware service composition in Dino. In *ECOWS*, pages 3–12, 2007.
29. R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*, pages 19–26, 2014.
30. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *CAV*, pages 376–398, 1991.
31. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
32. A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA monitoring for Web services. In *DSOM*, pages 28–41, 2002.
33. V. A. Saraswat and M. C. Rinard. Concurrent constraint programming. In *POPL*, pages 232–245, 1990.
34. S. Sebastio and A. Vandin. MultiVeStA: statistical model checking for discrete event simulators. In *Proc. ValueTools*, pages 310–315, 2013.
35. M. Srivatsa, L. Xiong, and L. Liu. TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks. In *WWW*, pages 422–431, 2005.
36. A. Strunk. QoS-aware service composition: A survey. In *ECOWS*, pages 67–74. IEEE, 2010.
37. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.

38. E. Tuosto. Contract-oriented services. In *WS-FM*, pages 16–29, 2012.
39. T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6, 2007.
40. L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

```
 1 CO2ServerConnection co2 =
 2         new CO2ServerConnection("alice@test.com", "alice");

 3 TST tst = new TST("!greet{;t}.?planet{t<7}");
 4 tst.setContext("helloworld");
 5 Private<TST> privateA = tst.toPrivate(co2);
 6 Public<TST> publicA = privateA.tell();

 7 Session<TST> s = publicA.waitForSession();
 8 s.send("greet", "hello");

 9 try {
10         Message response = s.waitForReceive();
11         System.out.println("hello " + response.getStringValue());
12 } catch (ContractException e) {
13         System.out.println("The other participant is culpable.");
14 }
```

Listing 1: A simple "Hello world" service.

A Contract-oriented programming

In this appendix we show how to write simple contract-oriented services, using the middleware APIs via their Java binding; full code listings are available at [7].

A.1 Hello world

Listing 1 shows a simple "Hello world" example. At lines 1-2 participant A establishes a connection with the middleware. The contract of A (constructed at line 3) is the TST $p_{\mathsf{A}} = \texttt{!greet}\{t\}.\texttt{?planet}\{t < 7\}$, i.e. A wants to send a greeting message, then resets the clock t, and waits for 7 time units to receive the name of the greeted planet. The messages belong to the `helloworld` context (set at line 4), where they have string type, and their verification link always returns true. At line 5, we construct a `Private` object, which represents the contract p_{A} in a state where it has not been advertised to the middleware, yet. As soon as it is advertised by invoking the `tell` method at line 6, its state is changed into `Public`. At line 7, A waits for a session to be established; so that a `Session` object is created, through which A can interact with the participant (say, B) at the other endpoint. At line 8, A says `hello`, by `sending` a message with label `greet`. At line 10, A waits to receive a `Message` from B. If B respects its contract, `getStringValue` at line 11 gets the string associated to the `planet` action, and the session terminates successfully. Otherwise, the `waitForReceive` of A is unblocked, and a `ContractException` is caught at line 12. In both cases, the middleware updates the participants' reputation as discussed in Section 4.4.

```
1  TST tst = new TST("!x.!eps{;t}.?y{t<5}.(!ok + !no{;t}.?culpable{t<5})");
2  tst.setContext("arithmetic");
3  Private<TST> privateA = tst.toPrivate(co2);
4  Public<TST> publicA = privateA.tell();

5  Session<TST> s = publicA.waitForSession();
6  s.send("x", Double.toString(x));
7  s.send("eps", Double.toString(eps));

8  y = Double.parseDouble(s.waitForReceive().getStringValue());
9  if(Math.abs(x - y*y) < eps) {
10         s.send("ok");  // Value accepted
11 } else {
12         s.send("no"); // Value rejected
13         s.waitForReceive();
14 }
```

Listing 2: Computing square roots.

A.2 Square roots

In Listing 2, A computes the square root of `x` (with a tolerance `eps`) with the help of an external service. The contract of A, defined at lines 1-2, requires to first send `x` and `eps`, and then to wait for an answer `y` (the square root of `x`, up-to tolerance `eps`), which must be received within 5 seconds from the time `!eps` is sent. Upon receiving `y` at line 8, A checks whether the distance between `x` and the square of `y` is less then `eps`: we assume that A cannot perform the onerous math operation *square root*, but it can perform simpler ones such as multiplications and subtractions. If the check succeeds, then A sends a confirmation message `ok`, otherwise it sends `no`, after which the other participant must produce the message with label `culpable` within 5 seconds. The label `culpable` is a special one: it has a verification link which always returns false. Hence, both in case the participant sends it, or if it waits for more than 5 seconds, it eventually becomes culpable. This scenario models a situation where participants trust A for verifying their messages (at lines 9-14); in Appendix A.3 we will show how to delegate the verification of messages to a trusted third party.

A.3 Square roots with a trusted third party

When participants do not trust each other for the verification of messages (e.g., for online payments), they can delegate this operation to a trusted third party. In Listing 3 we rework the example in Appendix A.2, but now we resort to a trusted third party to check that the returned square root respects the tolerance `eps`. The contract of A is simplified accordingly, by removing the part of interaction after the reception of `y` (line 1). The context `trustedsqrt` (line 2) defines the verification link associated to `y`. This is a process which first stores `x`

```
1  TST tst = new TST("!x.!eps{;t}.?y{t<5}");
2  tst.setContext("trustedsqrt");
3  Private<TST> privateA = tst.toPrivate(co2);
4  Public<TST> publicA = privateA.tell();

5  Session<TST> s = publicA.waitForSession();
6  s.send("x", Double.toString(x));
7  s.send("eps", Double.toString(eps));
8  try {
9          System.out.println("y: " + s.waitForReceive().getStringValue());
10 } catch (ContractException e) {
11         // Verification failed
12 }
```

Listing 3: Computing square roots with a trusted third party.

and `eps`, and, upon receiving `y`, performs the check `Math.abs(x - y*y) < eps`. If the check succeeds, A receives the square root `y` and prints it (line 9); otherwise, an exception is raised (line 10).

A.4 A simple book store

Consider an online book store taking orders from clients. If the requested book is in stock, then the store confirms the order and accepts a payment from the client. Otherwise, the store contacts some external distributors, to check if the requested book is available; if so, then the store can confirm the order to the client, and upon payment from the client it orders the book from the distributor.

Listing 4 shows the code of a book store S which respects these requirements. At lines 1-4 we define two contracts: `tstB` for interacting with buyers, and `tstD` for interacting with distributors. In a session with a buyer B, the store S waits for a book identifier (e.g., an *ISBN* code) at line 7; if the book is in stock or available from some distributors, it will send `confirm` to the buyer, otherwise it will send `abort`. The confirmation message contains the price of the book, so the buyer can choose whether to perform the payment (with action `pay`), or just to terminate the session without finalizing the purchase (with action `quit`). The contract between the book store and a distributor is essentially the same, except that the roles are swapped with respect to the store-buyer case: in `tsbD`, the participant S acts as a buyer, while the distributor acts as a seller. Note also that the deadlines in `tsbB` are slightly stricter than those in `tsbD`, because a distributor could have to handle a greater load of requests than a single store.

At lines 5-7, the store tells its *buyer-side* contract and waits for B to join the session; then, it receives the book identifier, and checks if the book is in stock (line 8). If so, the store sends a confirmation message (with the price of the book) to B (line 9); then, it waits for a decision from B (line 10). If B does not `quit` the session, the payment is taken in charge (PayPal is delegated to certify the payment, as specified in the verification link associated to action `pay`).

```java
TST tstB = new TST("?book{;x}.(!confirm{x<60}.(?pay{x<120} & ?quit{x<120}) +
                              !abort{x<60})", "bookstorebuyer"); // buyer
TST tstD = new TST("!book{;x}.(?confirm{x<10}.(!pay{x<150} + !quit{x<150}) &
                              ?abort{x<10})", "bookstoredist"); // distr

pB = tstB.toPrivate(co2).tell();
sB = pB.waitForSession();
String chosenBook = sB.waitForReceive().getStringValue();

if (isInStock(chosenBook)) { // handled internally
  sB.send("confirm");
  mB = sB.waitForReceive();
  switch(mB.getLabel()) {
  case "quit" : return;
  case "pay" : handlePayment(mB.getStringValue());
  }
}
else { // handled with the distributor
  pD = tstD.toPrivate(co2).tell(30 * 1000);

  try {
    sD = pD.waitForSession(30 * 1000); // 30s
    sD.send("book", chosenBook);
    mD = sD.waitForReceive(10 * 1000);
    switch (mD.getLabel()) {
    case "abort" :
      sB.send("abort");
      return;
    case "confirm" :
      sB.send("confirm");
      try {
        mB = sB.waitForReceive(120 * 1000); // waiting buyer

        switch(mB.getLabel()) {
        case "quit" : sD.send("quit"); return;
        case "pay" :
          handlePayment(mB.getStringValue());
          sD.send("pay", bookPrice(chosenBook));
        }
      } catch (TimeExpiredException | ContractViolationException e) {
        sD.send("quit");
      }
    }
  }
  catch (TimeExpiredException | ContractViolationException e) {
    sB.send("abort");
  }
}
```

Listing 4: A simple book store interacting with a buyer and a book distributor.

If the book is not in stock, the store advertises the contract `tsbD`, in order to redirect the order to a book distributor (line 17). Note that the contract `tsbD` will expire in 30 seconds after its advertisement: if there are no available distributors, a `TimeExpiredException` is thrown and an `abort` message is sent to B (line 42). If a session with some distributor D is established (line 19), the book identifier is forwarded to D. Then, D can either send `abort` if the book is not available (line 23), or `confirm` otherwise (line 26). In the first case, the store sends `abort` to B (line 24), while in the second case it sends `confirm` (line 27). At line 29, the store S waits for the final decision of B: if the buyer chooses to `pay`, then the store will settle the bill with the distributor, otherwise both sessions are terminated. Catching the `TimeExpiredException` and the `ContractViolationException` at line 36 allows the store to be protected against an unspecified behavior of B (for instance, if B does not send `pay` or `quit` within the given deadline): in this case, S will `quit` the session with the distributor, before becoming culpable at session `sD`.

B Timed CO$_2$

In this appendix we provide a more detailed account of the specification of contract-oriented services sketched in Section 3.

B.1 Syntax

Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of *variables* (ranged over by $x, y, \ldots$) and *names* (ranged over by $s, t, \ldots$). We use $\boldsymbol{u}, \boldsymbol{v}, \ldots$ to range over sequences of variables.

Definition 2. *The syntax of TCO$_2$ is defined as follows:*

$$\begin{array}{rcl|c|c|c|c|c|c}
S & ::= & \mathbf{0} & \mathsf{A}[P] & s[\gamma] & (u)S & S \mid S & \{\downarrow_u p\}_{\mathsf{A}} & \\
P & ::= & \mathbf{0} & \mathrm{X}(\boldsymbol{u}) & \pi \,.\, P & (u)P & u \rhd \{\mathsf{a}_i \,.\, P_i\}_{i \in I} & & \\
\pi & ::= & \tau & \mathtt{tell} \downarrow_u p & \mathtt{send}_u\ \mathsf{a} & \mathtt{idle}(\delta) & \mathtt{accept}(x) & \bar{x}y & x(y)
\end{array}$$

If $\boldsymbol{u} = u_0, \ldots, u_n$, we will use $(\boldsymbol{u})S$ and $(\boldsymbol{u})P$ as shorthands for $(u_0) \cdots (u_n)S$ and $(u_0) \cdots (u_n)P$, respectively. We also assume the following syntactic constraints on processes and systems:

1. *each occurrence of named processes is prefix-guarded;*
2. *in $(\boldsymbol{u})(\mathsf{A}[P] \mid \mathsf{B}[Q] \mid \cdots)$, it must be $\mathsf{A} \neq \mathsf{B}$;*
3. *in $(\boldsymbol{u})(s[\gamma] \mid t[\gamma'] \mid \cdots)$, it must be $s \neq t$.*
4. *each variable used in contract primitives can not be used as input/output channel (and vice-versa).*

Systems $S, S', \ldots$ are the parallel composition of *participants* $\mathsf{A}[P]$, *sessions* $s[\gamma]$, *delimited systems* $(u)S$, and *latent contracts* $\{\downarrow_u p\}_{\mathsf{A}}$. A latent contract commutative monoidal laws for $|$ on processes and systems

$$\mathsf{A}[(v)P] \equiv (v)\,\mathsf{A}[P] \qquad Z \mid (u)Z' \equiv (u)(Z \mid Z') \ \text{ if } u \not\in \mathtt{fv}(Z) \cup \mathrm{fn}(Z)$$
$$(u)(v)Z \equiv (v)(u)Z \qquad (u)Z \equiv Z \ \text{ if } u \not\in \mathtt{fv}(Z) \cup \mathrm{fn}(Z)$$
$$\{\downarrow_s p\}_{\mathsf{A}} \equiv \mathbf{0} \qquad \mathtt{idle}(0).P \equiv P$$

Fig. 6: Structural equivalence for CO_2 (Z, Z' range over systems or processes).

$\{\downarrow_x p\}_{\mathsf{A}}$ represents a contract p (advertised by A) which has not been stipulated yet; upon stipulation, the variable x will be instantiated to a fresh session name.

Processes $P, Q, \ldots$ are prefix-guarded processes; the branching construct $u \rhd \{\mathtt{a}_i \,.\, P_i\}$, which waits for input one of the atoms $\mathtt{a}_i$ and the behave as the corresponding P_i; named processes $\mathrm{X}(\boldsymbol{u})$ (used e.g. to specify recursive behaviours); delimited processes $(u)P$; and the nil process $\mathbf{0}$.

Prefixes π include contract advertisement $\mathtt{tell}{\downarrow_u}\, p$, contractual output $\mathtt{send}_u\ \mathtt{a}$, delay $\mathtt{idle}(\delta)$, the accept primitive $\mathtt{accept}(x)$, and channel input/output $\bar{x}y$ and $x(y)$. In each prefix $\pi \neq \tau$, the index u refers to the target session involved in the execution of π.

The only binder for names is the delimitation (u), both in systems and processes. Instead, variables have two binders: delimitations (x) (both in systems and processes), and input actions.

We stipulate that each process identifier X has a unique defining equation $\mathrm{X}(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$ such that $\mathtt{fv}(P) \subseteq \{x_1, \ldots, x_j\} \subseteq \mathcal{V}$. We will sometimes omit the arguments of $\mathrm{X}(\boldsymbol{u})$ when they are clear from the context. As usual, we omit trailing occurrences of $\mathbf{0}$ in processes.

B.2 Semantics

Definition 3. *The semantics of TCO_2 is the least relation closed under the rules in Figure 2.*

We now comment only the rules of Figure 7 not already discussed in Section 3. First, notice that in rules [Fuse] and [Acpt] we have instantiated the contract model to use TSTs: in particular, the initial contract configuration $\Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$ is defined as $\mathsf{A} : p, \nu_0 \mid \mathsf{B} : q, \nu_0$. Rules [Def], [Par-Act] and [Del] are mostly standard. Rule [Delay-K] allows a latent contract to idle indefinitely. Rules [Delay-P] model time elapsing for processes without timed constructs at the top level of the syntax. Roughly, time passes only when urgent actions are not possible. Rule [Delay-Par] allows parallel composition of systems to idle if all the components can, and no urgent transitions are possible.

$$\frac{\exists q : p \bowtie q}{\mathsf{A}[\mathtt{tell} \downarrow_u p \,.\, P] \xrightarrow{\mathtt{tell}} \mathsf{A}[P] \;\mid\; \{\downarrow_u p\}_{\mathsf{A}}} \;\text{[Tell]}$$

$$\frac{p \bowtie q \qquad \gamma = \mathsf{A} : p, \nu_0 \mid \mathsf{B} : q, \eta_0 \qquad \sigma = \{{}^{s}/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \;\mid\; \{\downarrow_x p\}_{\mathsf{A}} \;\mid\; \{\downarrow_y q\}_{\mathsf{B}}) \xrightarrow{\mathtt{fuse}} (s)(S\sigma \;\mid\; s[\gamma])} \;\text{[Fuse]}$$

$$\frac{\gamma = \mathsf{A} : \mathrm{co}(p)\,, \nu_0 \mid \mathsf{B} : p, \nu_0 \qquad \sigma = \{{}^{s}/_{x}\} \qquad s \text{ fresh}}{(x)(\mathsf{A}[\mathtt{accept}(x) \,.\, P] \mid \{\downarrow_x p\}_{\mathsf{B}} \mid S) \xrightarrow{\mathtt{accept}} (s)(\mathsf{A}[P\sigma] \mid s[\gamma] \mid S\sigma)} \;\text{[Acpt]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:\mathtt{!a}}\!\!\!\!\rightarrow \gamma'}{\mathsf{A}[\mathtt{send}_s\, \mathtt{a} \,.\, P] \;\mid\; s[\gamma] \xrightarrow{\mathtt{send}} \mathsf{A}[P] \;\mid\; s[\gamma']} \;\text{[Send]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:\mathtt{?a}}\!\!\!\!\rightarrow \gamma' \qquad \mathtt{a} = \mathtt{a}_j}{\mathsf{A}[s \rhd \{\mathtt{a}_i \,.\, P_i\}] \;\mid\; s[\gamma] \xrightarrow{\mathtt{receive}} \mathsf{A}[P_j] \;\mid\; s[\gamma']} \;\text{[Recv]}$$

$$\frac{\mathrm{X}(\boldsymbol{x}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{{}^{\boldsymbol{u}}/_{\boldsymbol{x}}\}] \mid S \xrightarrow{\mu} S'}{\mathsf{A}[\mathrm{X}(\boldsymbol{u})] \mid S \xrightarrow{\mu} S'} \;\text{[Def]} \qquad \frac{S \xrightarrow{\mu} S' \qquad \mu \neq \delta}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \;\text{[Par-Act]}$$

$$\frac{S \xrightarrow{\mu} S'}{(u)S \xrightarrow{\mu} (u)S'} \;\text{[Del]} \qquad \{\downarrow_u p\}_{\mathsf{A}} \xrightarrow{\delta} \{\downarrow_u p\}_{\mathsf{A}} \;\text{[Delay-K]} \qquad \frac{\gamma \xrightarrow{\delta}\!\!\!\!\rightarrow \gamma'}{s[\gamma] \xrightarrow{\delta} s[\gamma']} \;\text{[Delay-}\gamma\text{]}$$

$$\frac{P \neq \mathtt{idle}(\delta') \,.\, P' \qquad \forall \mu \in \mathrm{Urg} : \big(\mathsf{A}[P] \stackrel{\mu}{\not\rightarrow} \;\wedge\; \forall \delta' \leq \delta :\; \mathsf{A}[P] \xrightarrow{\delta'} S \implies S \stackrel{\mu}{\not\rightarrow}\big)}{\mathsf{A}[P] \xrightarrow{\delta} \mathsf{A}[P]} \;\text{[Delay-P]}$$

$$\frac{\delta' \leq \delta}{\mathsf{A}[\mathtt{idle}(\delta) \,.\, P] \xrightarrow{\delta'} \mathsf{A}[\mathtt{idle}(\delta - \delta') \,.\, P]} \;\text{[Idle]}$$

$$\frac{\begin{array}{c} S_0 \xrightarrow{\delta} S_0' \qquad S_1 \xrightarrow{\delta} S_1' \\ \forall \mu \in \mathrm{Urg} : \big(S_0 \mid S_1 \stackrel{\mu}{\not\rightarrow} \;\wedge\; \forall \delta' \leq \delta :\; S_0 \mid S_1 \xrightarrow{\delta'} S \implies S \stackrel{\mu}{\not\rightarrow}\big) \end{array}}{S_0 \mid S_1 \xrightarrow{\delta} S_0' \mid S_1'} \;\text{[Delay-Par]}$$

$$\mathsf{A}[\bar{x}y \,.\, P] \mid \mathsf{B}[x(z) \,.\, Q] \xrightarrow{\tau} \mathsf{A}[P] \mid \mathsf{B}[Q\{{}^{y}/_{z}\}] \;\text{[Comm]}$$

Fig. 7: Reduction semantics of TCO$_2$ (full set of rules).

Example 2. We now specify in TCO$_2$ the "Hello world" service in Listing 1. Let:

$$P = (x)\,\texttt{tell}\downarrow_x p\,.\,\texttt{send}_x\,\texttt{g}\,.\,x \rhd \{\texttt{p}\,.\,\mathbf{0}\} \qquad p = \texttt{!g}\{\{t\}\}.\,\texttt{?p}\{t < 7\}$$
$$Q = (y)\,\texttt{tell}\downarrow_y q\,.\,y \rhd \{\texttt{g}\,.\,\texttt{idle}(10)\,.\,\texttt{send}_y\,\texttt{p}\} \quad q = \mathrm{co}(p) = \texttt{?g}\{\{t\}\}.\,\texttt{!p}\{t < 7\}$$

The process P is basically a translation in TCO$_2$ of the Java code in Listing 1: it tells the TST p, after a session is opened sends $\texttt{g}$, and then waits for $\texttt{p}$. The process Q tells the dual of p, reads $\texttt{g}$, and then tries to send $\texttt{p}$ after idling for 10 time units. A possible reduction of $S = \mathsf{A}[P] \mid \mathsf{B}[Q]$ is the following:

$$
\begin{aligned}
S &\rightarrow^* (x,y)\,\mathsf{A}[\texttt{send}_x\,\texttt{g}\ldots] \mid \mathsf{B}[y \rhd \{\texttt{g}\ldots\}] \mid \{\downarrow_x p\}_{\mathsf{A}} \mid \{\downarrow_y q\}_{\mathsf{B}} \\
&\rightarrow (s)\,\mathsf{A}[\texttt{send}_s\,\texttt{g}\ldots] \mid \mathsf{B}[s \rhd \{\texttt{g}\ldots\}] \mid s[\mathsf{A} : p, \epsilon, \nu_0 \mid \mathsf{B} : q, \epsilon, \nu_0] \\
&\rightarrow (s)\,\mathsf{A}[s \rhd \{\texttt{p}\}] \mid \mathsf{B}[s \rhd \{\texttt{g}\ldots\}] \mid s[\mathsf{A} : \texttt{?p}\{t < 7\}, \texttt{!g}, \nu_0 \mid \mathsf{B} : \texttt{!p}\{t < 7\}, \epsilon, \nu_0] \\
&\rightarrow (s)\,\mathsf{A}[s \rhd \{\texttt{p}\}] \mid \mathsf{B}[\texttt{idle}(10)\ldots] \mid s[\mathsf{A} : \texttt{?p}\{t < 7\}, \epsilon, \nu_0 \mid \mathsf{B} : \texttt{!p}\{t < 7\}, \epsilon, \nu_0] \\
&\xrightarrow{10} (s)\,\mathsf{A}[s \rhd \{\texttt{p}\}] \mid \mathsf{B}[\texttt{send}_s\,\texttt{p}] \mid s[\mathsf{A} : \texttt{?p}\{t < 7\}, \epsilon, \nu_0 + 10 \mid \mathsf{B} : \mathbf{0}, \epsilon, \nu_0 + 10]
\end{aligned}
$$

As expected, the computation reaches a state in which B is culpable.

C Timed Session Types

In this appendix we recall from [6] some useful notions about the theory of timed session types.

C.1 Semantics of Timed Session Types

We use *clock valuations*, which associate each clock with its value. The state of the interaction between two TSTs is described by a *configuration* $(p, \nu) \mid (q, \eta)$, where the clock valuations ν and η record (keeping the same pace) the time of the clocks in p and q, respectively. The dynamics of the interaction is formalised as a transition relation between configurations (Definition 7). This relation describes all and only the *correct* interactions: for instance, we do not allow time passing to make unsatisfiable all the guards in an internal choice, since doing so would prevent a participant from respecting her protocol.

We denote with $\mathbb{V} = \mathbb{C} \to \mathbb{R}_{\geq 0}$ the set of clock valuations (ranged over by $\nu, \eta, \ldots$), and with ν_0 the valuation mapping each clock to zero. We write $\nu + \delta$ for the valuation which increases ν by δ, i.e., $(\nu + \delta)(t) = \nu(t) + \delta$ for all $t \in \mathbb{C}$. For a set $R \subseteq \mathbb{C}$, we write $\nu[R]$ for the *reset* of the clocks in R, i.e., $\nu[R](t) = 0$ if $t \in R$, and $\nu[R](t) = \nu(t)$ otherwise.

Definition 4 (Semantics of guards). *For all guards g, we define the set of clock valuations $[\![g]\!]$ inductively as follows, where $\circ \in \{<, \leq, =, \geq, >\}$:*

$$[\![\texttt{true}]\!] = \mathbb{V} \qquad [\![\neg g]\!] = \mathbb{V} \setminus [\![g]\!] \qquad [\![g_1 \wedge g_2]\!] = [\![g_1]\!] \cap [\![g_2]\!]$$
$$[\![t \circ d]\!] = \{\nu \mid \nu(t) \circ d\} \qquad [\![t - t' \circ d]\!] = \{\nu \mid \nu(t) - \nu(t') \circ d\}$$

$$(\texttt{!a}\{g,R\}.\,p{+}p',\ \nu) \xrightarrow{\tau} ([\texttt{!a}\{g,R\}]\,p,\ \nu) \qquad \text{if } \nu \in [\![g]\!] \qquad [+]$$

$$([\texttt{!a}\{g,R\}]\,p,\ \nu) \xrightarrow{\texttt{!a}} (p,\ \nu[R]) \qquad [!]$$

$$(\texttt{?a}\{g,R\}.\,p + p',\ \nu) \xrightarrow{\texttt{?a}} (p,\ \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad [?]$$

$$(p,\ \nu) \xrightarrow{\delta} (p,\ \nu + \delta) \qquad \text{if } \delta > 0\ \wedge\ \nu + \delta \in \texttt{rdy}(p) \qquad [\textsc{Del}]$$

$$\frac{(p,\nu) \xrightarrow{\tau} (p',\nu')}{(p,\nu) \mid (q,\eta) \xrightarrow{\tau} (p',\nu') \mid (q,\eta)}\ [\text{S-}+] \qquad \frac{(p,\nu) \xrightarrow{\delta} (p,\nu') \quad (q,\eta) \xrightarrow{\delta} (q,\eta')}{(p,\nu) \mid (q,\eta) \xrightarrow{\delta} (p,\nu') \mid (q,\eta')}\ [\text{S-}\textsc{Del}]$$

$$\frac{(p,\nu) \xrightarrow{\texttt{!a}} (p',\nu') \quad (q,\eta) \xrightarrow{\texttt{?a}} (q',\eta')}{(p,\nu) \mid (q,\eta) \xrightarrow{\tau} (p',\nu') \mid (q',\eta')}\ [\text{S-}\tau]$$

$$\texttt{rdy}(\textstyle\sum \texttt{!a}_i\{g_i,R_i\}\,.\,p_i) = \downarrow \bigcup [\![g_i]\!] \qquad \texttt{rdy}(\&\cdots) = \texttt{rdy}(\mathbf{1}) = \mathbb{V} \qquad \texttt{rdy}([\texttt{!a}\{g,R\}]\,p) = \emptyset$$

Fig. 8: Semantics of timed session types (symmetric rules omitted).

Before defining the semantics of TSTs, we recall from [12] some basic operations on *sets* of clock valuations (ranged over by $\mathcal{K}, \mathcal{K}', \ldots \subseteq \mathbb{V}$).

Definition 5 (Past and inverse reset). *For all sets $\mathcal{K}$ of clock valuations, the set of clock valuations $\downarrow\mathcal{K}$ (the* past *of $\mathcal{K}$) and $\mathcal{K}[T]^{-1}$ (the* inverse reset *of $\mathcal{K}$) are defined as: $\downarrow\mathcal{K} = \{\nu \mid \exists \delta \geq 0 : \nu + \delta \in \mathcal{K}\}$, $\mathcal{K}[T]^{-1} = \{\nu \mid \nu[T] \in \mathcal{K}\}$.*

Definition 6. *For all TSTs p, we define the set of clock valuations $\texttt{rdy}(p)$ as:*

$$\texttt{rdy}(p) = \begin{cases} \downarrow\bigcup [\![g_i]\!] & \text{if } p = \sum_{i\in I} \texttt{!a}_i\{g_i, R_i\}\,.\,p_i \\ \mathbb{V} & \text{if } p = \&\cdots \text{ or } p = \mathbf{1} \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 7 (Semantics of TSTs). *A* configuration *is a term of the form $(p,\nu) \mid (q,\eta)$, where p,q are TSTs extended with* committed choices *$[\texttt{!a}\{g,R\}]\,p$. The semantics of TSTs is defined as a labelled relation $\rightarrow$ over configurations, whose labels are either silent actions τ, delays δ, or branch labels.*

We now comment the rules in Figure 8. The first four rules are auxiliary, as they describe the behaviour of a TST in isolation. Rule [+] allows a TST to commit to the branch `!a` of her internal choice, provided that the corresponding guard is satisfied in the clock valuation ν. This results in the term $[\texttt{!a}\{g,R\}]\,p$, which represents the fact that the endpoint has committed to branch `!a` in a specific time instant: actually, it can only fire `!a` through rule [!] (which also resets the clocks in R), while time cannot pass. Rule [?] allows an external choice to fire any of its input actions whose guard is satisfied. Rule [Del] allows time to pass; this is always possible for external choices and success term, while for an internal choice we require that at least one of the guards remains satisfiable; this is obtained through the function `rdy` in Figure 8. The last three rules deal with configurations of two TSTs. Rule [S-$+$] allows a TSTs to commit in an internal choice. Rule [S-τ] is the standard synchronisation rule *à la* CCS; note that B is assumed to read a message as soon as it is sent, so A never blocks on internal choices. Rule [S-Del] allows time to pass, equally for both endpoints.

Example 3. Let $p = \texttt{!a}+\texttt{!b}\{t > 2\}$, let $q = \texttt{?b}\{t > 5\}$, and consider the following computations:

$$\begin{aligned}
(p, \nu_0) \mid (q, \eta_0) &\xrightarrow{7}\xrightarrow{\tau} ([\texttt{!b}\{t > 2\}], \nu_0 + 7) \mid (q, \eta_0 + 7) \\
&\xrightarrow{\tau} (\mathbf{1}, \nu_0 + 7) \mid (\mathbf{1}, \eta_0 + 7) &(1)\\
(p, \nu_0) \mid (q, \eta_0) &\xrightarrow{\delta}\xrightarrow{\tau} ([\texttt{!a}], \nu_0 + \delta) \mid (q, \eta_0 + \delta) &(2)\\
(p, \nu_0) \mid (q, \eta_0) &\xrightarrow{3}\xrightarrow{\tau} ([\texttt{!b}\{t > 2\}], \nu_0 + 3) \mid (q, \eta_0 + 3) &(3)
\end{aligned}$$

The computation in (1) reaches success, while the other two computations reach the deadlock state. In (2), p commits to the choice `!a` after some delay δ; at this point, time cannot pass (because the leftmost endpoint is a committed choice), and no synchronisation is possible (because the other endpoint is not offering `?a`). In (3), p commits to `!b` after 3 time units; here, the rightmost endpoint would offer `?b`, — but not in the time chosen by the leftmost endpoint. Note that, were we allowing time to pass in committed choices, then we would have obtained e.g. that $(\texttt{!b}\{t > 2\}, \nu_0) \mid (q, \eta_0)$ never reaches deadlock — contradicting our intuition that these endpoints should not be considered compliant.

Definition 8 (Compliance [6]). *We say that* $(p, \nu) \mid (q, \eta)$ *is* deadlock *whenever* (i) *it is not the case that both* p *and* q *are* $\mathbf{1}$, *and* (ii) *there is no* δ *such that* $(p, \nu + \delta) \mid (q, \eta + \delta) \xrightarrow{\tau}$. *We then write* $(p, \nu) \bowtie (q, \eta)$ *whenever the labels of* p *and* q *belong to the same context, and:*

$$(p, \nu) \mid (q, \eta) \rightarrow^* (p', \nu') \mid (q', \eta') \quad \textit{implies} \quad (p', \nu') \mid (q', \eta') \textit{ not deadlock}$$

We say that p *and* q *are* compliant *whenever* $(p, \nu_0) \bowtie (q, \eta_0)$ *(in short,* $p \bowtie q$*).*

C.2 Dual construction

The dual construction makes sense only for those TSTs for which a compliant exists. To this purpose, we define a procedure (more precisely, a kind system) which computes the set of clock valuations $\mathcal{K}$ (called *kinds*) such that p admits a compliant TST in all $\nu \in \mathcal{K}$.

Definition 9 (Kind system). *Kind judgements* $\Gamma \vdash p : \mathcal{K}$ *are defined in Figure 9. where* Γ *is a partial function which associates kinds to recursion variables.*

$$\Gamma \vdash \mathbf{1} : \mathbb{V} \;\; [\text{T-}\mathbf{1}] \qquad \frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \text{for } i \in I}{\Gamma \vdash \&_{i \in I} \mathtt{?a}_i\{g_i, T_i\} \,.\, p_i : \bigcup_{i \in I} \downarrow \left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1} \right)} \; [\text{T-}\&]$$

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \text{for } i \in I}{\Gamma \vdash \sum_{i \in I} \mathtt{!a}_i\{g_i, T_i\} \,.\, p_i : \left(\bigcup_{i \in I} \downarrow [\![g_i]\!] \right) \setminus \left(\bigcup_{i \in I} \downarrow \left([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1} \right) \right)} \; [\text{T-}+]$$

$$\Gamma, X : \mathcal{K} \vdash X : \mathcal{K} \;\; [\text{T-Var}] \qquad \frac{\exists \mathcal{K}, \mathcal{K}' : \;\; \Gamma\{^{\mathcal{K}}/_{x}\} \vdash p : \mathcal{K}'}{\Gamma \vdash \text{rec}\, X \,.\, p : \bigcup \{ \mathcal{K} \mid \Gamma\{^{\mathcal{K}}/_{x}\} \vdash p : \mathcal{K}' \land \mathcal{K} \subseteq \mathcal{K}' \}} \; [\text{T-Rec}]$$

Fig. 9: Kind system for TSTs.

Rule [T-1] says that the success TST $\mathbf{1}$ admits compliant in every ν: indeed, $\mathbf{1}$ is compliant with itself. The kind of an exernal choice is the union of the kinds of its branches (rule [T-&]), where the kind of a branch is the past of those clock valuations which satisfy both the guard and, after the reset, the kind of their continuation. Internal choices are dealt with by rule [T-+], which computes the difference between the union of the past of the guards and a set of error clock valuations. The error clock valuations are those which can satisfy a guard but not the kind of its continuation. Rule [T-Var] is standard. Rule [T-Rec] looks for a kind which is preserved by unfolding of recursion (hence a fixed point). In order to obtain completeness of the kind system we need the greatest fixed point.

The following theorem states that every TST is kindable. We stress the fact that being kindable does not imply admitting a compliant. This holds if and only if ν_0 belongs to the kind (see Theorems 3 and 4).

Theorem 1. *For all closed* p*, there exists some* $\mathcal{K}$ *such that* $\vdash p : \mathcal{K}$.

The following theorem states that the problem of determining the kind of a TST is decidable. This might seem surprising, as the cardinality of kinds is $2^{2^{\aleph_0}}$. However, the kinds constructed by our inference rules can always be represented syntactically by guards [22].

Theorem 2. *Kind inference is decidable.*

Definition 10 (Dual of a TST [6]). *For all kindable* p *kindable and kinding environments* Γ*, we define the TST* $\text{co}_\Gamma(p)$ *(in short,* $\text{co}(p)$ *when* $\Gamma = \emptyset$*)*

$$\begin{aligned}
\text{co}_\Gamma(\mathbf{1}) &= \mathbf{1} \\
\text{co}_\Gamma\left(\&_{i \in I} \mathtt{?a}_i\{g_i, T_i\} \,.\, p_i\right) &= \textstyle\sum_{i \in I} \mathtt{!a}_i\{g_i \land \mathcal{K}_i[T_i]^{-1}, T_i\} \,.\, \text{co}_\Gamma(p_i) && \textit{if } \Gamma \vdash p_i : \mathcal{K}_i \\
\text{co}_\Gamma\left(\textstyle\sum_{i \in I} \mathtt{!a}_i\{g_i, T_i\} \,.\, p_i\right) &= \&_{i \in I} \mathtt{?a}_i\{g_i, T_i\} \,.\, \text{co}_\Gamma(p_i) \\
\text{co}_\Gamma(X) &= X && \textit{if } \Gamma(X) \textit{ defined} \\
\text{co}_\Gamma(\text{rec}\, X \,.\, p) &= \text{rec}\, X \,.\, \text{co}_{\Gamma\{^{\mathcal{K}}/_{X}\}}(p) && \textit{if } \Gamma \vdash \text{rec}\, X \,.\, p : \mathcal{K}
\end{aligned}$$

The following theorem states the soundness of the kind system: is particular, if the clock valuation ν_0 belongs to the kind of p, then p admits a compliant.

Theorem 3 (Soundness). *If* $\vdash p : \mathcal{K}$ *and* $\nu \in \mathcal{K}$*, then* $(p, \nu) \bowtie (\text{co}(p), \nu)$.

The following theorem states the kind system is also complete: in particular, if p admits a compliant, then the clock valuation ν_0 belongs to the kind of p.

Theorem 4 (Completeness). *If* $\vdash p : \mathcal{K}$ *and* $\exists q, \eta.\ (p, \nu) \bowtie (q, \eta)$*, then* $\nu \in \mathcal{K}$*.*

C.3 Runtime monitoring

We now define the semantics of the runtime monitor of TSTs, which is the one used in the premises of rules [Send] and [Recv] in Figure 7. Note that the semantics in Figure 8 cannot be directly exploited to define such a runtime monitor, for two reasons. First, the synchronisation rule is symmetric and synchronous, while the middleware assumes an asymmetry between internal and external choices and an asynchronous semantics. Second, the semantics in Figure 8 does not have transitions (either messages or delays) not allowed by the TSTs, while the monitoring semantics must also consider illegal moves attempted by participants.

The monitoring semantics is defined on two levels. The first level, specified by the relation $\rightarrow$ (which overloads the transition relation used in Appendix C.1) deals with the case of honest participants; however, unlike the semantics in Appendix C.1, here we decouple the action of sending from that of receiving. More precisely, if A has an internal choice and B has an external choice, then we postulate that A must move first, by doing one of the outputs in her choice, and then B must be ready to do the corresponding input. The second level, called *monitoring semantics* and specified by the relation $\twoheadrightarrow$, builds upon the first one to allow for synchronisation and delay. Additionally, the monitoring semantics defines transitions for actions not accepted by the first level, e.g. unexpected input/output actions. In these cases, the monitoring semantics assigns the blame to the culpable participant, by setting its state to $\mathbf{0}$.

Definition 11 (Monitoring semantics of TSTs). Monitoring configurations $\gamma, \gamma', \ldots$ *are terms of the form* $P \parallel Q$*,* P *and* Q *are triples* (p, c, ν)*, where* p *is either a TST or* $\mathbf{0}$*, and* c *is a sequence of output labels (possibly empty). The transition relations* $\rightarrow$ *and* $\twoheadrightarrow$ *over monitoring configurations, with labels* $\lambda, \lambda', \ldots \in (\{A, B\} \times L) \cup \mathbb{R}_{\geq 0}$*, is defined in Figure 10.*

In the rules in Figure 10, we always assume that the leftmost TST is governed by A, while the rightmost one is governed by B. In rule [M-+], A has an internal choice, and she can fire one of her outputs !a, provided that the guard g is satisfied. When this happens, the message !a is written to the buffer, and the clocks in R are reset. In rule [M-&], B can enable an input ?a in an external choice; this is permitted when the guard g of the selected branch is satisfied. Rules [M-Del] and [M-DelFail] allow time to pass, making A culpable when she definitively disables all the branches in an internal choice. The last four rules specify the runtime monitor. Rule [M-Sync] allows two triples to synchronise; this makes the buffer of A grow (!a is enqueued, according to rule [M-+]), while B just consumes the input prefix ?a. Rule [M-SyncDel] lets some time δ to pass, provided that the delay is the same for both triples. Rule [M-Read] allows B to

$$(\mathtt{!a}\{g, R\}.\, p + p', c, \nu) \xrightarrow{\mathtt{!a}} (p, c \cdot \mathtt{!a}, \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-+]}$$

$$(\mathtt{?a}\{g, R\}.\, p \mathbin{\&} p', c, \nu) \xrightarrow{\mathtt{?a}} (p, c, \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-\&]}$$

$$\frac{\nu + \delta \in \mathtt{rdy}(p)}{(p, c, \nu) \xrightarrow{\delta} (p, c, \nu + \delta)} \qquad \text{[M-Del]}$$

$$\frac{\nu + \delta \notin \mathtt{rdy}(p)}{(p, c, \nu) \xrightarrow{\delta} (\mathbf{0}, c, \nu + \delta)} \qquad \text{[M-DelFail]}$$

$$\frac{(p, c, \nu) \xrightarrow{\mathtt{!a}} (p', c', \nu') \qquad (q, d, \eta) \xrightarrow{\mathtt{?a}} (q', d', \eta')}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\mathsf{A}:\mathtt{!a}}\!\!\!\!\rightarrow (p', c', \nu') \parallel (q', d', \eta')} \qquad \text{[M-Sync]}$$

$$\frac{(p, c, \nu) \xrightarrow{\delta} (p', c', \nu') \qquad (q, d, \eta) \xrightarrow{\delta} (q', d', \eta')}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\delta}\!\!\!\!\rightarrow (p', c', \nu') \parallel (q', d', \eta')} \qquad \text{[M-SyncDel]}$$

$$(p, \mathtt{!a} \cdot c, \nu) \parallel (q, d, \eta) \xrightarrow{\mathsf{B}:\mathtt{?a}}\!\!\!\!\rightarrow (p, c, \nu) \parallel (q, d, \eta) \qquad \text{[M-Read]}$$

$$\frac{(p, c, \nu) \not\xrightarrow{\mathtt{!a}}}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\mathsf{A}:\mathtt{!a}}\!\!\!\!\rightarrow (\mathbf{0}, c, \nu) \parallel (q, d, \eta)} \qquad \text{[M-Fail]}$$

Fig. 10: Monitoring semantics (symmetric rules omitted).

read a message in the buffer; note that the state of the recipient is not updated, since the input prefix was already consumed by rule [M-Sync]. Finally, rule [M-Fail] is used when A attempts to do an action not permitted by $\rightarrow$: this makes the monitor evolve to a configuration where A is culpable (denoted by the term $\mathbf{0}$).

Formally, the runtime monitor can be seen as a *deterministic* automaton, which reads a *timed trace* (a sequence of actions and time delays) and it reaches a unique state γ, which can be inspected to find which of the two participants (if any) is culpable.

Definition 12 (Duties & culpability). *Let $\gamma = (p, c, \nu) \parallel (q, d, \eta)$. We say that* A *is* culpable *in γ iff $p = \mathbf{0}$. We say that* A *is* on duty *in γ if (i)* A *is not culpable in γ, and (ii) either p is an internal choice, or d is not empty.*

When both participants behave honestly, i.e., they never take [*Fail*] moves, the monitoring semantics preserves compliance. This can be proved similarly to Theorem 9 in [6].

Example 4. Let $p = \mathtt{!a}\{2 < t < 4\}$ be the TST of participant A, and let $q = \mathtt{?a}\{2 < t < 5\} + \mathtt{?b}\{2 < t < 5\}$ be that of B. We have that $p \bowtie q$. Let $\gamma_0 = (p, [], \nu_0) \parallel (q, [], \nu_0)$. A correct interaction is given by the timed trace $\eta = \langle 1.2,\ \mathsf{A} : \mathtt{!a},\ \mathsf{B} : \mathtt{?a} \rangle$. Indeed, $\gamma_0 \xrightarrow{\eta}\!\!\!\!\rightarrow (\mathbf{1}, [], \nu_0) \parallel (\mathbf{1}, [], \nu_0)$. On the contrary, things may go wrong in the following two cases:

(i) a participant does something not permitted. E.g., if A fires a at 1 t.u., by [M-FailA]: $\gamma_0 \xrightarrow{1}\!\!\!\!\rightarrow \xrightarrow{\mathsf{A}:\mathtt{!a}}\!\!\!\!\rightarrow (\mathbf{0}, [], \nu_0 + 1) \parallel (q, [], \eta_0 + 1)$, where A is culpable.

(ii) a participant avoids to do something she is supposed to do. E.g., assume that after 6 t.u., A has not yet fired a. By rule [M-SyncDel], we obtain $\gamma_0 \xrightarrow{6}\!\!\!\!\rightarrow (\mathbf{0}, [], \nu_0 + 6) \parallel (q, [], \eta_0 + 6)$, where A is culpable.