1

Contract-oriented programming with timed session types

Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, Maurizio Murgia, Alessandro Sebastian Podda and Livio Pompianu

University of Cagliari, Italy.

Abstract

Contract-oriented programming is a software engineering paradigm which proposes the use of behavioural contracts to discipline the interaction among software components. In a distributed setting, the various components of an application may be developed and run by untrustworthy parties, which could opportunistically diverge from the expected behaviour when they find it convenient. The use of contracts in this setting is essential: by binding the behaviour of each component to a contract, and by sanctioning contract violations, components are incentivized to behave in a correct and cooperative manner.

This chapter is a step-by-step tutorial on programming contract-oriented distributed applications. The glue between components is a middleware which establishes sessions between services with compliant contracts, and monitors sessions to detect and punish violations. Contracts are formalised as timed session types, which describe timed communication protocols between two components at the endpoints of a session. We illustrate some basic primitives of contract-oriented programming: advertising contracts, performing contractual actions, and dealing with violations. We then show how to exploit these primitives to develop some small distributed applications.

Keywords: behavioural contracts, service composition, timed systems.

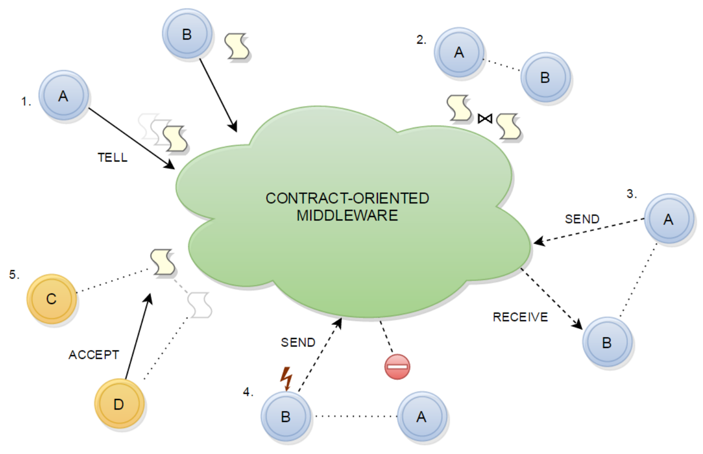

Figure 1.1: Contract-oriented interactions in the CO_2 middleware.

1.1 Introduction

Developing trustworthy distributed applications can be a challenging task. A key issue is that the services that compose a distributed application may be under the governance of different providers, which may compete against each other. Furthermore, services interact through open networks, where competitors and adversaries can try to exploit their vulnerabilities.

A possible countermeasure to these issues is to use *behavioural contracts* to discipline the interaction among services. These are formal descriptions of service behaviour, which can be used at static or dynamic time to discover and bind services, and to guarantee that they interact in a protected manner: namely, when a service does not behave as prescribed by its contract, it can be blamed and sanctioned for a contract breach.

In previous work [7] we presented a middleware that uses behavioural contracts to discipline the interactions among distrusting services. Since it supports the COntract-Oriented paradigm, we called it "CO_2 middleware".

Figure 1.1 illustrates the main features of the CO_2 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*: this means that interactions which respect the contracts are deadlock-free. Upon compliance, the middleware establishes a session through which the two participants can interact. This interaction consists of sending and receiving messages, similarly to a standard message-oriented middleware (MOM): 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 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 its reputation is, the lower the probability of being able to establish new sessions with it.

Item (5) shows another mechanism for establishing sessions: here, the participant C advertises a contract, and D just *accepts* it. This means that the middleware associates D with the *canonical compliant* of the contract of C, and it establishes a session between C and D. The interaction happening in this session then proceeds as described previously.

In this chapter we illustrate how to program contract-oriented distributed applications which run on the CO_2 middleware. A public instance of the middleware is accessible from `co2.unica.it`, together with all examples and experiments we carried out.

1.2 Timed Session Types

The CO_2 middleware currently supports two kinds of contracts:

- first-order binary session types [18];
- timed session types (TSTs) [6].

In this section we illustrate TSTs with the help of a small case study, an online store which receives orders from customers. The use of *un*timed session types in contract-oriented applications is discussed in the literature [3, 4, 8].

Specifying contracts

Timed session types extend binary session types [18, 26] with clocks and timing constraints, similarly to the way timed automata [1] extend (classic) finite state automata. We informally describe the syntax of TSTs below, and we refer to [5, 6] for the full technical development.

Guards. Guards describe timing constraints, and they are conjunctions of simple guards of the form `t` $\circ$ `d`, where `t` is a *clock*, `d` $\in \mathbb{N}$, and $\circ$ is a relation in `<`, `<=`, `=`, `>=`, `>`. For instance, the guard `t<60,u>10` is true whenever the value of clock `t` is less than 60, *and* the value of clock `u` is greater than 10. The value of clocks is in $\mathbb{R}_{\geq 0}$, like for timed automata.

Send and receive. A TST describes the behaviour of a single participant A at the end-point of a session. Participants can perform two kinds of actions:

- a *send action* `!a{g;t1,...,tk}` stipulates that A will output a message with label `a` in a time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the output is performed.
- a *receive action* `?a{g;t1,...,tk}` stipulates that A will be available to receive a message with label `a` at *any instant* within the time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the input is received.

When `g` = `true`, the guard can be omitted.

For instance, consider the contract `store1` between the store and a customer, from the point of view of the store.

```
store1 = "?order{;t} . !price{t<60}"
```

The store declares that it will receive an order at any time. After it has been received, the store will send the corresponding price within 60 seconds.

Internal and external choices. TSTs also feature two forms of choice:

- `!a1{g1;R1} + ...  + !an{gn;Rn}`

  This is an *internal choice*, stipulating that A will decide at run-time which one of the output actions `!ai{gi;Ri}` (with `1` $\leq$ `i` $\leq$ `n`) to perform, and at which time instant. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset.
- `?a1{g1;R1} & ...& ?an{gn;Rn}`

  This is an *external choice*, stipulating that A will be able to receive any of the inputs `!ai{gi;Ri}`, in the declared time windows. The actual choice of the action, and of the instant when it is performed, will be made by the participant at the other endpoint of the session. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset.

With these ingredients, we can refine the contract of our store as follows:

```
store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

This version of the contract deals with the case where the store receives an unknown or invalid product code. In this case, the internal choice allows the store to inform the buyer that the requested item is `unavailable`.

Recursion. The contracts shown so far can only handle a bounded (statically known) number of interactions. We can overcome this limitation by using recursive TSTs. For instance, the contract `store3` below models a store which handles an arbitrary number of orders from a buyer:

```
store3 = "REC 'x' [?addtocart{t<60;t}.'x'
                  & ?checkout{t<60;t}.(
                        !price{t<20;t}.(
                            ?accept{t<10} & ?reject{t<10})
                      + !unavailable{t<20})]"
```

The contract `store3` allows buyers to add some item to the cart, or checkout. When a buyer chooses `addtocart`, the store must allow him to add more items: this is done recursively. After a `checkout`, the store must send the overall `price`, or inform the buyer that the requested items are `unavailable`. If the store sends a price, it must expect a response from the buyer, who can either `accept` or `reject` the price.

Context. Action 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 (described later on in this section) to delegate the verification of messages to a trusted third party. For instance, the middleware supports Paypal as a verification link for online payments [7].

Compliance

Besides being used to specify the interaction protocols between pairs of services, TSTs feature the following primitives:

- a decidable notion of *compliance* between two TSTs;
- an algorithm to detect if a TST admits a compliant one;
- a computable *canonical compliant* construction.

These primitives are exploited by the CO_2 middleware to establish sessions between services: more specifically, the middleware only allows interactions between services with compliant contracts. Intuitively, compliance guarantees that, if *all* services respect *all* their contracts, then the overall distributed application (obtained by composing the services) will not deadlock.

Below we illustrate the primitives of TSTs by examples; a comprehensive formal treatment is in [5].

Informally, two TSTs are *compliant* if, in the interactions where both participants respect their contract, the deadlock state is not reachable (see [5] for details). For instance, recall the simple version of the store contract:

```
store1 = "?order{;t} . !price{t<60}"
```

and consider the following buyer contracts:

```
buyer1 = "!order{;u} . ?price{u<70}"
buyer2 = "!order{;u} . (?price{u<70} & ?unavailable)"
buyer3 = "!order{;u} . (?price{u<30} & ?unavailable)"
buyer4 = "!order{u<20} . ?price{u<70}"
```

We have that:

- `store1` and `buyer1` are compliant: indeed, the time frame where `buyer1` is available to receive `price` is larger than the one where the store can send;
- `store1` and `buyer2` are compliant: although the action `?unavailable` enables a further interaction, this is never chosen by the store `store1`.
- `store1` and `buyer3` are *not* compliant, because the store may choose to send `price` 60 seconds after he got the order, while `buyer2` is only able to receive within 30 seconds.

- store1 and buyer4 are *not* compliant. Here the reason is more subtle: assume that the buyer sends the order at time 19: at that point, the store receives the order and resets the clock t; after that, the store has 60 seconds more to send price. Now, assume that the store chooses to send price after 59 seconds (which fits within the declared time window of 60 seconds). The total elapsed time is 19+59=78 seconds, but the buyer is only able to receive before 70 seconds.

We can check if two contracts are compliant through the middleware Java APIs[1]. We show how to do this through the Groovy[2] interactive shell[3].

```
cS1 = new TST(store1)
cS1.isCompliantWith(new TST(buyer1))
>>> true
cS1.isCompliantWith(new TST(buyer3))
>>> false
```

Consider now the second version of the store contract:

```
store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

The contract store2 is compliant with the buyer contract buyer2 discussed before, while it is *not* compliant with:

```
buyer5 = "!order{;u} . (?price{u<90})"
buyer6 = "!order{;u} . (?price{u<90} + ?unavailable{u>5,u<12})"
```

The problem with buyer5 is that the buyer is only accepting a message labelled price, while store2 can also choose to send unavailable. Although this option is present in buyer6, the latter contract is not compliant with store2 as well. In this case the reason is that the time window for receiving unavailable does not include that for sending it (recall that the sender can choose any instant satisfying the guard in its output action). To illustrate some less obvious aspects of compliance, consider the following buyer contract:

```
buyer7 = "!order{u<100} . ?price{u<70}"
```

[1] co2.unica.it/downloads/co2api/
[2] groovy-lang.org/download.html
[3] On Unix-like systems, copy the API's jar in $HOME/.groovy/lib/. Then, add import co2api.* to $HOME/.groovy/groovysh.rc, and run groovysh.

This contract stipulates that the buyer can wait up to 100 seconds for sending an order, and then she can wait until 60 seconds (from the *start* of the session), to receive the price from the store.

Now, assume that some store contract is compliant with `buyer7`. Then, the store must be able to receive the `order` at least until time 100. If the buyer chooses to send the `order` at time 90 (which is allowed by contract `buyer7`), then the store would never be able to send `price` before time 70. Therefore, no contract can be compliant with `buyer7`.

The issue highlighted by the previous example must be dealt with care: if one publishes a service whose contract does not admit a compliant one, then the middleware will never connect that service with others. To check whether a contract admits a compliant one, we can query the middleware APIs:

```
cB7 = new TST(buyer7)
>>> !order{u<100} . ?price{u<70}

cB7.hasCompliant()
>>> false
```

Recall from Section 1.1 that the CO_2 middleware also allows a service to *accept* another service's contract, as per item (5) in Figure 1.1. E.g., assume that the store has advertised the contract `store2` above. When the buyer uses the primitive `accept`, the middleware associates the buyer with the *canonical compliant* of `store2`, constructed through the method `dualOf`, i.e.:

```
cS2 = new TST(store2)
>>> ?order{;t} . (!price{t<60} + !unavailable{t<10})

cB2 = cS2.dualOf()
>>> !order{;t} . (?price{t<60} & ?unavailable{t<10})
```

Intuitively, if a TST admits a compliant one, then its canonical compliant is constructed as follows:

1. output labels `!a` are translated into input labels `?a`, and *vice versa*;
2. internal choices are translated into external choices, and *vice versa*;
3. prefixes and recursive calls are preserved;
4. guards are suitably adjusted in order to ensure compliance.

Consider now the following contract of a store which receives an order and a coupon, and then sends a discounted price to the buyer:

```
store4 = "?order{t<60} . ?coupon{t<30;t} . !price{t<60}"
```

In this case `store4` admits a compliant one, but this cannot be obtained by simply swapping input/output actions and internal/external choices.

```
cS4 = new TST(store4)
cB4 = new TST("!order{t<60} . !coupon{t<30;t} . ?price{t<60})")
cS4.isCompliantWith(cB4)
>>> false
```

Indeed, the canonical compliant construction gives:

```
cB5 = cS4.dualOf()
>>> !order{t<30} . ?coupon{t<30;t} . ?price{t<60}
```

Run-time monitoring of contracts

In order to detect (and sanction) contract violations, the CO_2 middleware monitors all the interactions that happen through sessions. The monitor guarantees that, in each reachable configuration, only one participant can be "on duty" (i.e., she has to perform some actions); and if no one is on duty nor culpable, then both participants have reached success. Here we illustrate how runtime monitoring works, by making a store and a buyer interact.

To this purpose, we split the paper in two columns: in the left column we show the store behaviour, while in the right column we show the buyer. We assume that both participants call the middleware APIs through the Groovy shell, as shown before. Note that the interaction between the two participants is asynchronous: when needed, we will highlight the points where one of the participants performs a time delay.

Both participants start by creating a connection `co2` with the middleware:

```
usr = "testuser1@gmail.com"
pwd = "testuser1"
co2 = new CO2ServerConnection(
    usr,pwd)
```

```
usr = "testuser2@gmail.com"
pwd = "testuser2"
co2 = new CO2ServerConnection(
    usr,pwd)
```

Then, the participants create their contracts, and advertise them to the middleware through the primitive `tell`. The variables `pS` and `pB` are the handles to the published contracts.

```
cS = new TST(store2)
pS = cS.toPrivate(co2).tell()
```

```
cB = new TST(buyer2)
pB = cB.toPrivate(co2).tell()
```

Now the middleware has two compliant contracts in its collection, hence it can establish a session between the store and the buyer. To obtain a handle to the session, both participants use the blocking primitive `waitForSession`:

```
sS = pS.waitForSession()
```

```
sB = pB.waitForSession()
```

At this point, participants can query the session to see who is “on duty” (namely, one is on duty if the contract prescribes her to perform the next action), and to check if they have violated the contract:

```
sS.amIOnDuty()
>>> false
sS.amICulpable()
>>> false
```

```
sB.amIOnDuty()
>>> true
sB.amICulpable()
>>> false
```

Note that the first action must be performed by the buyer, who must send the `order`. This is accomplished by the `send` primitive. Dually, the store waits for the receipt of the message, using the `waitForReceive` primitive:

```
msg = sS.waitForReceive()
msg.getStringValue()
>>> 0123
sS.amIOnDuty()
>>> true
```

```
// send at an arbitrary time
sB.send("order", "0123")

sB.amIOnDuty()
>>> false
```

Since there are no time constraints on sending `order`, this action can be successfully performed at any time; once this is done, the `waitForReceive` unlocks the store. The store is now on duty, and it must send `price` within 60 seconds, or `unavailable` within 10 seconds. Now, assume that the store tries to send `unavailable` after the deadline:

```
// wait more than 10 seconds

sS.send("unavailable")
>>> ContractException
```

```
msg = sB.waitForReceive()

>>> ContractViolationException:
"The other participant is culpable"
```

On the store’s side, the `send` throws a `ContractException`; on the buyer side, the `waitForReceive` throws an exception which reports the violation of the store. At this point, if the two participants check the state of the session, they find that none of them is still on duty, and that the store is culpable:

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> true
```

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> false
```

At this point, the session is terminated, and the reputation of the store is suitably decreased.

1.3 Contract-oriented programming

In this section we develop some simple contract-oriented services, using the middleware APIs via their Java binding[4].

A simple store

We start with a basic store service, which advertises the contract `store2`:

```
1   String store2 ="?order{;t}.(!price{t<60} + !unavailable{t<10})";
2   TST c = new TST(store2);

4   CO2ServerConnection co2 =
5     new CO2ServerConnection("testuser@co2.unica.it", "pa55w0rd");
6   Private r = c.toPrivate(co2);
7   Public  p = r.tell();          //advertises the contract store2

9   Session s = p.waitForSession();//blocks until session is created
10  String id = s.waitForReceive().getStringValue();

12  if(isAvailable(id)) { s.send("price", getPrice(id)); }
13  else { s.send("unavailable"); }
```

At lines 1-2, the store constructs a TST `c` for contract `store2`. At lines 4-5, the store connects to the middleware, providing its credentials. At line 6, the `Private` object represents the contract in a state where it has not been advertised to the middleware yet. To advertise the contract, we invoke the `tell` method at line 7. This call returns a `Public` object, modelling a latent contract that can be "fused" with a compliant one to establish a new session. At line 9, the store waits for a session to be established; the returned `Session` object allows the store to interact with a buyer. At line 10, the store waits for the receipt of a message, containing the code of the product requested by the buyer. At

[4] Full code listings are available at `co2.unica.it`

lines 12-13, the store sends the message `price` (with the corresponding value) if the item is available, otherwise it sends `unavailable`.

A simple buyer

We now show a buyer that can interact with the store. This buyer just accepts the already published contract `store2`. The contract is identified by its hash, which is obtained from `Public.getContractID()`.

```
 1  CO2ServerConnection co2 = new CO2ServerConnection(...);

 3  String storeCID = "0x...";
 4  Integer desiredPrice = 10;

 6  Public  p = Public.accept(co2, storeCID, TST.class);
 7  Session s = p.waitForSession();

 9  s.send("order", "11235811");

11  try {
12      Message m = s.waitForReceive();
13      switch (m.getLabel()) {
14      case "unavailable": break;
15      case "price":
16          Integer price = Integer.parseInt(m.getStringValue());
17          if (price > desiredPrice) { /*abort the purchase*/ }
18          else { /*proceed with the purchase*/ }
19      }
20  } catch(ContractViolationException e){/*The store is culpable*/}
```

At line 6, the buyer accepts the store's contract, identified by `storeCID`. The call to `Public.accept` returns a `Public` object. At this point a session with the store is already established, and `waitForSession` just returns the corresponding `Session` object (line 7). Now, the buyer sends the item code (line 9), waits for the store response (line 12), and finally in the `try-catch` statement it handles the messages `price` and `unavailable`.

Note that the `accept` primitive allows a participant to establish sessions with a chosen counterpart; instead, this is not allowed by the `tell` primitive, which can establish a session whenever two contracts are compliant.

A dishonest store

Consider now a more complex store, which relies on external distributors to retrieve items. As before, the store takes an order from the buyer; however, now it invokes an external distributor if the requested item is not in stock. If the distributor can provide the item, then the store confirms the order to the buyer; otherwise, it informs the buyer that the item is unavailable.

Our first attempt to implement this refined store is the following.

```
1  TST cB = new TST(store2);
2  TST cD = new TST("!req{;t}.(?ok{t<10} & ?no{t<10})");

4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();

8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with a distributor
12     Public  pD = cD.toPrivate(co2).tell();
13     Session sD = pD.waitForSession();

15     sD.send("req", id);
16     Message mD = sD.waitForReceive();

18     switch (mD.getLabel()) {
19     case "no" : sB.send("unavailable"); break;
20     case "ok" : sB.send("price", getPrice(id)); break;
21   }
22 }
```

At lines 1-2 we construct two TSTs: `cB` for interacting with buyers, and `cD` for interacting with distributors. In `cD`, the store first sends a request for some item to the distributor, and then waits for an `ok` or `no` answer, according to whether the distributor is able to provide the requested item or not. At lines 4-6, the store advertises `cB`, and it waits for a buyer to join the session; then, it receives the order, and checks if the requested item is in stock (line 8). If so, the store sends the price of the item to the buyer (line 9).

If the item is not in stock, the store advertises `cD` to find a distributor (lines 12-13). When a session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then it waits for a reply. If the reply is `no`, the store sends `unavailable` to the buyer, otherwise it sends a `price`.

Note that this implementation of the store is *dishonest*, namely it may violate contracts [11]. This happens in the following two cases:

1. Assume that the store has received the buyer's order, but the requested item is not in stock. Then, the store advertises the contract `cD` to find a distributor. Note that there is no guarantee that the session `sD` will be established within a given deadline, nor that it will be established at all. If more than 60 seconds pass on the `waitForSession` at line 13,

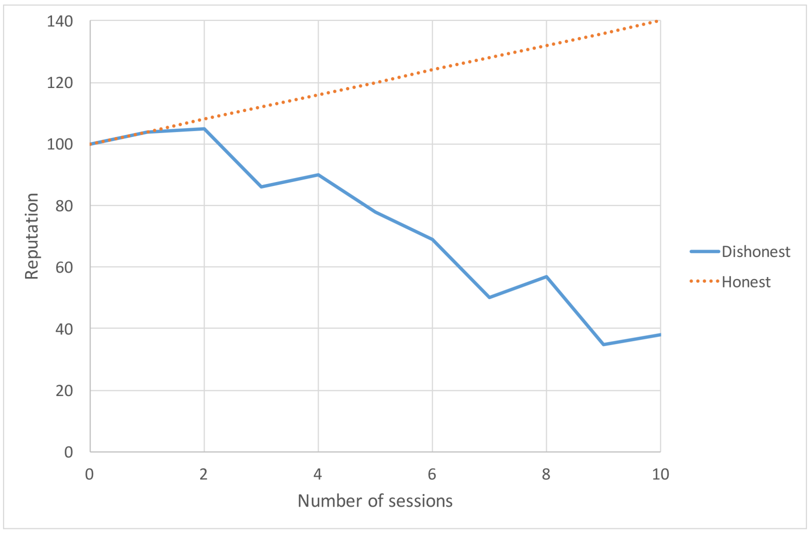

Figure 1.2: Reputation of the dishonest and honest stores as a function of the number of sessions with malicious distributors.

the store becomes culpable with respect to the contract `cB`. Indeed, such contract requires the store to perform an action before 60 seconds (10 seconds if the action is `unavailable`).

2. Moreover, if the session `sD` is established in timely fashion, a slow or unresponsive distributor could make the store violate the contract `cB`. For instance, assume that the distributor sends message `no` after nearly 10 seconds. In this case, the store may not have enough time to send `unavailable` to the buyer within 10 seconds, and so it becomes culpable at session `sB`.

We have simulated the scenario described in Item 1, by making the store interact with slow or unresponsive distributors (see Figure 1.2). The experimental results show that, although the store is not culpable in all the sessions, its reputation decreases over time. Recovering from such situation is not straightforward, since the reputation system of the CO_2 middleware features defensive techniques against self-promoting attacks [25].

An honest store

In order to implement an honest store, we must address the fact that, if the distributor delays its message to the maximum allowed time, the store may not have enough time to respond to the buyer. To cope with this scenario, we adjust the timing constraints in the contract between the store and the distributor, and we implement a revised version of the store as follows.

```
1  TST cB = new TST(store2);
2  TST cD = new TST("!req{;t} . (?ok{t<5} & ?no{t<5})");

4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();

8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with the distributor
12     Public pD = cD.toPrivate(co2).tell(3 * 1000);
13     try {
14         Session sD = pD.waitForSession();
15         sD.send("req", id);

17         try{
18             Message mD = sD.waitForReceive();

20             switch (mD.getLabel()) {
21             case "no": sB.send("unavailable"); break;
22             case "ok": sB.send("price", getPrice(id)); break;
23             }
24         } catch(ContractViolationException e){
25             //the distributor did not respect its contract
26             sB.send("unavailable");
27         }
28     } catch(ContractExpiredException e) {
29         //no distributor found
30         sB.send("unavailable");
31     }
32 }
```

The parameter in the `tell` at line 12 specifies a deadline of 3 seconds: if the session `sD` is not established within the deadline, the contract `cD` is retracted from the middleware, and a `ContractExpiredException` is thrown. The store catches the exception at line 28, sending `unavailable` to the buyer.

Instead, if the session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then waits for the receipt of a response from it. If the distributor sends neither `ok` nor `no` within the deadline specified in `cD` (5 seconds), the middleware assigns the blame to the distributor for a contract breach, and unblocks the `waitForReceive` in the store with a `ContractViolationException` (line 24). In the exception handler, the store fulfils the contract `cB` by sending `unavailable` to the buyer.

A recursive honest store

We now present another version of the store, which uses the recursive contract `store3` on page 5. As in the previous version, if the buyer requests an item that is not in stock, the store resorts to an external distributor.

```
TST cB = new TST(store3);
TST cD = new TST("!req{;t}.(?ok{t<5} & ?no{t<5})");

Public  pB = cB.toPrivate(co2).tell();
Session sB = pB.waitForSession();
List<String> orders = new ArrayList<>();
Message mB;

try {
    do {
        mB = sB.waitForReceive();
        if (mB.getLabel().equals("addtocart")){
            orders.add(mB.getStringValue());
        }
    } while(!mB.getLabel().equals("checkout"));

    if (isAvailable(orders)) { // handled internally
        sB.send("price", getPrice(orders));
        String res = sB.waitForReceive().getLabel();
        switch (res){
            case "accept": // handle the order
            case "reject": // terminate
        }
    }
    else { // handled with the distributor
        Public pD = cD.toPrivate(co2).tell(5 * 1000);
        try {
            Session sD = pD.waitForSession();
            sD.send("req", getOutOfStockItems(orders));
            try{
                switch (sD.waitForReceive().getLabel()) {
                case "no": sB.send("unavailable"); break;
                case "ok":
                    sB.send("price", getPrice(orders));
                    try{
                        String res =
                          sB.waitForReceive().getLabel();
                        switch (res) {
                        case "accept": // handle the order
                        case "reject": // terminate
41                          }
42                      }
43                      catch (ContractViolationException e) {
44                          //the buyer is culpable, terminate
45                      }
46                  }
47              } catch (ContractViolationException e){
48                  //the distributor did not respect its contract
49                  sB.send("unavailable");
50              }
51          }
52          catch (ContractExpiredException e) {
53              //no distributor found
54              sB.send("unavailable");
55          }
56      }
57  } catch(ContractViolationException e){/*the buyer is culpable*/}
```

After advertising the contract `cB`, the store waits for a session `sB` with the buyer (lines 4-5). After the session is established, the store can receive `addtocart` multiple times: for each `addtocart`, it saves the corresponding item identifier in a list. The loop terminates when the buyer selects `checkout`. If all requested items are available, the store sends the total `price` to the buyer (line 18). After that, the store expects either `accept` or `reject` from the buyer. If the buyer does not respect his deadlines, an exception is thrown, and it is caught at line 57. If the buyer replies on time, the store advertises the contract `cD`, and waits for a session `sD` with the distributor (lines 26-28). If the session is not established within 5 seconds, an exception is thrown. The store handles the exception at line 52, by sending `unavailable` to the buyer. If a session with the distributor is established within the deadline, the store requests the unavailable items, and waits for a response (line 31). If the distributor sends `no`, the store answers `unavailable` to the buyer (line 32). If the distributor sends `ok`, then the interaction between store and buyer proceeds as if the items were in stock. If the distributor does not reply within the deadline, an exception is thrown. The store handles it at line 47, by sending `unavailable` to the buyer. An untimed specification of this store is proved honest in [4]. We conjecture that also this timed version of the store respects contracts in all possible contexts.

1.4 Conclusions

We have explored the use of behavioural contracts as service-level agreements among the components of a distributed application. In particular, we have considered a middleware where services can advertise contracts (in the form of timed session types, TSTs), and interact through sessions, which are only created between services with compliant contracts. The primitives of the middleware exploit the theory of TSTs: in particular, a decidable notion of compliance between TSTs, a decidable procedure to detect when a TST admits a compliant one, and a decidable runtime monitoring. The middleware has been validated in [7] through a series of experiments, which measure the scalability of the approach when the number of exchanged contracts grows, and the effectiveness of the reputation system.

Although the current version of the middleware only features binary (either timed or untimed) session types as contracts, the underlying idea can be extended to other contract models. Indeed, the middleware only makes mild assumptions about the nature of contracts, e.g., that they feature: (i) monitorable send and receive actions, (ii) some notion of accepting a contract or a role, or (iii) some notion of compliance with a sound (but not necessarily complete) verification algorithm. Other timed models of contracts would be ideal candidates for extensions of the middleware. For instance, communicating timed automata [13] (which are timed automata with unbounded communication channels) would allow for multi-party sessions.

Security issues should be seriously taken into account when developing contract-oriented applications. As we have shown for the dishonest online store in Section 1.3, adversaries could make a service sanctioned by exploiting discrepancies between its contracts and its actual behaviour. Since these mismatches are not always easy to spot, analysis techniques are needed in order to ensure that a service will not be susceptible to this kind of attacks. A starting point could be the analyses in [8, 9], that can detect whether a contract-oriented specification is honest; the Diogenes toolchain [3] extends this check to Java code. Since these analyses do not take into account time constraints, further work is needed to extend these techniques to timed applications.

Related work

The theoretical foundations of our middleware are timed session types and CO_2 [12, 10], a specification language for contract-oriented services. The middleware implements the main primitives of CO_2 (`tell`, `send`, `receive`), and it introduces new concepts, such as the `accept` primitive, time constraints, and reputation.

From the theoretical viewpoint, the idea of constraint-based interactions has been investigated in other process calculi, such as Concurrent Constraint Programming (CCP) [24], and cc-pi [16]. The kind of interactions they induce is quite different from ours. In CCP, processes can interact by telling and asking for the validity of constraints on a global constraint store. In cc-pi, interaction is a mix of name communication *à la* π-calculus [21] and `tell` *à la* CCP (which is used to put constraints on names). In cc-pi consistency plays a crucial role: `tell`s *restrict* the future interactions with other processes, since adding constraints can lead to more inconsistencies; by contrast, in our middleware advertising a contract *enables* interaction with other services, so consistency is immaterial, but compliance is a key notion.

Several formalisms for expressing timed communication protocols have been proposed over the years. The work [14] addresses a timed extension of multi-party asynchronous session types [19]. Unlike ours, the approach pursued in [14] is top-down: a *global type*, specifying the overall communication protocol of a set of services, is projected onto a set of *local types*. Then, a composition of services preserves the properties of the global type (e.g., deadlock-freedom) if each service type-checks against the associated local type. The CO_2 middleware, instead, fosters a bottom-up approach to service composition. Both our approach and [14, 23] use runtime monitoring to detect contract violations and assign the blame to the party that is responsible for a contract violation. The CO_2 middleware also exploits these data in its reputation system.

The work [13] studies *communicating timed automata*, a timed version of communicating finite-state machines [15]. In this model, participants in a network communicate asynchronously through bi-directional FIFO channels; similarly to [14], clocks, guards and resets are used to impose time constraints on when communications can happen. An approximate (sound, but not complete) decidable technique allows one to check when a system of automata enjoys progress. This technique is based on *multiparty compatibility*, a condition that guarantees deadlock-freedom of untimed systems [20].

From the application viewpoint, several works have investigated the problem of *service selection* in open dynamic environments [2, 22, 27, 28]. This problem consists in matching client requests with service offers, in a way that, among the services respecting the given functional constraints, the one that 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 in our middleware does not search for the "best" contract that is 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. In some approaches [2, 28] 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 that 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.

Some works have explored service selection mechanisms where functional constraints can be required in addition to QoS constraints [22]: the first are described by a web service ontology, while the others are defined as requested and offered ranges of basic QoS attributes. Runtime monitor and reputation systems are also implemented, which, similarly to ours, help to marginalise those services that 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 [17]: 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 that participants are trusted, as the verification of contracts is delegated to the middleware itself and to trusted third parties.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia grants L.R.7/2007 CRP-17285 (TRICS), P.I.A. 2013 ("NOMAD"), and by EU COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems" (BETTY). Alessandro Sebastian Podda gratefully acknowledges Sardinia Regional Government for the financial support of her PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1).

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[2] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Software Eng.*, 33(6):369–384, 2007.

[3] Nicola Atzei and Massimo Bartoletti. Developing honest Java programs with Diogenes. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume 9688 of *LNCS*, pages 52–61. Springer, 2016.

[4] Nicola Atzei, Massimo Bartoletti, Maurizio Murgia, Emilio Tuosto, and Roberto Zunino. Contract-oriented design of distributed applications: a tutorial. `tcs.unica.it/papers/diogenes-tutorial.pdf`, 2016.

[5] Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. Timed session types, 2015. Pre-print available at `tcs.unica.it/papers/tst.pdf`.

[6] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume 9039 of *LNCS*, pages 161–177. Springer, 2015.

[7] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. A contract-oriented middleware. In *Formal Aspects of Component Software (FACS)*, volume 9539 of *LNCS*, pages 86–104. Springer, 2015. `co2.unica.it`.

[8] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, and Roberto Zunino. Verifiable abstractions for contract-oriented systems. *Journal of Logical and Algebraic Methods in Programming (JLAMP)*, 86:159–207, 2017.

[9] Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. Honesty by typing. *Logical Methods in Computer Science*, 12(4), 2016. Pre-print available at: `arxiv.org/abs/1211.2609`.

[10] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1):5–60, 2012.

[11] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION*, volume 7274 of *LNCS*, pages 245–260. Springer, 2012.

[12] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 332–341. IEEE Computer Society, 2010.

[13] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[14] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

[15] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[16] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[17] Kevin J. Hoffman, David Zage, and Cristina Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42(1), 2009.

[18] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP)*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

[20] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 221–232. ACM, 2015.

[21] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[22] A. Mukhija, Andrew Dingwall-Smith, and D.S. Rosenblum. QoS-aware service composition in Dino. In *ECOWS*, volume 5900 of *LNCS*, pages 3–12. Springer, 2007.

[23] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*, volume 162 of *EPTCS*, pages 19–26, 2014.

[24] Vijay A. Saraswat and Martin C. Rinard. Concurrent constraint programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 232–245. ACM, 1990.

[25] Mudhakar Srivatsa, Li Xiong, and Ling Liu. TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks. In *International Conference on World Wide Web (WWW)*, pages 422–431. ACM, 2005.

[26] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.

[27] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[28] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.