1

Contract-oriented design of distributed applications: a tutorial

Nicola Atzei[1], Massimo Bartoletti[1], Maurizio Murgia[1], Emilio Tuosto[2], Roberto Zunino[3]

[1] *Università degli Studi di Cagliari, Italy*
[2] *University of Leicester, UK*
[3] *Università degli Studi di Trento, Italy*

Abstract

Modern distributed applications typically blend new code with legacy (and possibly untrusted) third-party services. Behavioural contracts can be used to discipline the interaction among these services. Contract-oriented design advocates that composition is possible only among services with compliant contracts, and execution is monitored to detect (and possibly sanction) contract breaches.

In this tutorial we illustrate a contract-oriented design methodology consisting of five phases: specification writing, specification analysis, code generation, code refinement, and code analysis. Specifications are written in CO_2, a process calculus whose primitives include contract advertisement, stipulation, and contractual actions. Our analysis verifies a property called honesty: intuitively, a process is honest if it always honors its contracts upon stipulation, so being guaranteed to never be sanctioned at run-time. We automatically translate a given honest specification into a skeletal Java program which renders the contract-oriented interactions, to be completed with the application logic. Then, programmers can refine this skeleton into the actual Java application: however, doing so they could accidentally break its honesty.

An Edited Volume, 1–24.

The last phase is an automated code analysis to verify that honesty has not been compromised by the refinement.

All the phases of our methodology are supported by a toolchain, called Diogenes. We guide the reader through Diogenes to design small contract-oriented applications.

Keywords: behavioural contracts, service composition, verification.

1.1 Introduction

Developing service-oriented applications is a challenging task: programmers have to reliably compose loosely-coupled services which can dynamically discover and invoke other services through open networks, and may be subject to failures and attacks. Usually, services live in a world of mutually distrusting providers, possibly competing among each other. Typically, these providers offer little guarantees about the services they control, and in particular they might arbitrarily change the service code (if not the Service Level Agreement *tout court*) at any time.

Therefore, to guarantee the reliability and security of service-oriented applications, one must use suitable analysis techniques. Remarkably, most existing techniques to guarantee deadlock-freedom of service-oriented applications (e.g., compositional verification based on choreographies [35, 21]) need to inspect the code of *all* its components. Instead, under the given assumptions of mutual distrust between services, one can only analyse those under their control.

From service-oriented to contract-oriented computing

A possible countermeasure to these issues is to use *behavioural contracts* to regulate the interaction between services. In this setting, a service infrastructure acts as a trusted third party, which collects all the contracts advertised by services, and establishes sessions between services with compliant contracts. Unlike the usual service-oriented paradigm, here services are responsible for respecting their contracts. To incentivize such honest behaviour, the service infrastructure monitors all the messages exchanged among services, and sanctions those which do not respect their contracts.

Sanctions can be of different nature: e.g., pecuniary compensations, adaptations of the service binding [29], or reputation penalties which marginalize dishonest services in the selection phase [3]. Experimental evidence [3] shows that contract-orientation can mitigate the effort of handling potential misbehaviour of external services, at the cost of a tolerable loss in efficiency due to the contract-based service selection and monitoring.

Honesty attacks

The sanctioning mechanism of contract-oriented infrastructures protects honest services against malicious behaviours of the other services: indeed, if a malevolent service attempts to break the protocol (e.g. by prematurely terminating the interaction), it is punished by the infrastructure. At the same time, a new kind of attacks becomes possible: adversaries can try to exploit possible discrepancies between the promised and the actual behaviour of a service, in order to make it sanctioned. For instance, consider a naïve online store with the following behaviour:

1. advertise a contract to "receive a `request` from a buyer, and then either send the `price` of the ordered item, or notify that the item is `unavailable`";
2. wait to receive a `request`;
3. advertise a contract to "receive a `quote` from a package delivery service, and then either `confirm` or `abort`";
4. wait to receive a quote from the delivery service;
5. if the quote is below a certain threshold, then `confirm` the delivery and send the `price` to the buyer; otherwise, send `abort` to the delivery service, and notify `unavailable` to the buyer.

Now, assume an adversary which plays the role of a delivery service, and never sends the `quote`. This makes the store violate its contract with the buyer: indeed, the store should either send `price` or `unavailable` to the buyer, but these actions can only be performed after the delivery service has sent a `quote`. Therefore, the store can be sanctioned.

Since these *honesty attacks* may compromise the service and cause economic damage to its provider, it is important to detect the underlying vulnerabilities *before* deployment. Intuitively, a service is vulnerable if, in *some* execution context, it does *not* respect some of the contracts it advertises. Therefore, to avoid sanctions a service must be able to respect *all* the contracts it advertises, in *all* possible contexts — even in those populated by adversaries. We call this property *honesty*.

Some recent works have studied honesty at the specification level, using the process calculus CO_2 for modelling contract-oriented services [6, 7, 8, 9], whose primitives include contract advertisement, stipulation, and contractual actions. Practical experience has shown that writing honest specifications is not an easy task, especially when a service has to juggle with multiple sessions. The reason of this difficulty lies in the fact that, to devise an honest specification, a designer has to anticipate the possible behaviour of the context, but at design time he does not yet know in which context his service will be run. Tools to automate the verification of honesty may be of great help.

Diogenes

In this paper we illustrate the Diogenes toolchain [1], which supports the correct design of contract-oriented services as follows:

Specification Designers can specify services in the process calculus CO_2. An Eclipse plugin supports writing such specifications, providing syntax highlighting, code auto-completion, syntactic and semantic checks, and basic static type checking.

Honesty checking of specifications Our tool can statically verify the honesty of specifications. When the specification is dishonest, the tool provides a counterexample, in the form of a reachable abstract state of the service which violates some contract.

Translation into Java The tool automatically translates specifications into skeletal Java programs, implementing the required contract-oriented interactions (while leaving the actual application logic to be implemented in a subsequent step). The obtained skeleton is honest when the specification is such.

Honesty checking of refined Java code Programmers can refine the skeleton by implementing the actual application logic. This is a potentially dangerous operation, since honesty can be accidentally lost in the manual refinement. The tool supports this step, by providing an honesty checker for refined Java code.

1.2 Specifying contract-oriented services in CO2

A service in our modelling language consists of a CO_2 process. CO_2 is a process algebra inspired from CCS [28], and equipped with contract-oriented primitives: contract advertisement, stipulation, and contractual actions. Contracts are meant to model the promised behaviour of services, and they are expressed as session types ([34]).

We show the main features of our language with the help of a small case study, an online store which receives orders from customers.

Contracts

We first specify the contract `C` between the store and a customer, from the point of view of the store. The store declares that it will receive an `order`, and then send either the corresponding `price`, or declare that the item is `unavailable`. We formalise this contract as the following first-order binary session type [19]:

```
contract C { order? string . ( price! int (+) unavailable! ) }
```

Receive actions are marked with the symbol `?`, while send actions are marked with `!`. The sort of a message (`int`, `string`, or `unit`) is specified next to the action label; the sort `unit` is used for pure synchronizations, and it can be omitted. The symbol `_._` denotes prefixing. The symbol `_(+)_` is used to group send actions, and it denotes an *internal* choice made by the store.

Processes

Note that contracts only formalise the interaction protocol between two services, while they do not specify *how* these services advertise and realise the contracts. This behaviour is formalised in CO_2 [6, 7], a specification language for contract-oriented services. For instance, a possible CO_2 specification of our store is the following:

```
1 specification Store {
2     tell x C .        // wait until session x is created
3     receive@x order?[v:string] . (
4         if *          // checks if the item is in stock
5         then send@x price![*:int]
6         else send@x unavailable! ) }
```

At line 2, the store *advertises* the contract `C`, waiting for the service infrastructure to find some other service with a *compliant* contract. Intuitively, two contracts are compliant if they fulfil each other expectations[1]. When the infrastructure finds a contract compliant with `C`, a new session is created

[1] More precisely, the notion of compliance we use here is *progress*, that relates two processes whenever their interaction never reaches a deadlock [4].

between the respective services, and the variable `x` is bound to the session name.

At line 3 of the snippet above the store waits to receive an `order`, binding it to the variable `v` of sort `string`. At line 4, the store checks whether the ordered item is in stock (the actual condition is not given in the specification). If the item is in stock, then the store sends the `price` to the customer; otherwise it notifies that the item is `unavailable` (lines 5-6). The sent price `*:int` is a placeholder, to be replaced with an actual price upon refinement of the specification into an actual implementation of the service.

An execution context

We now show a possible context wherein to execute our store. Although the context is not needed for verifying the store specification, we use it to complete the presentation of the primitives of our modelling language.

```
1  specification BuyerA {
2      tell y { order! string . price? int } .
3      send@y order![*:string] .
4      receive@y price?[n:int]
5  }
6
7  specification BuyerB {
8      tell y { order! string . ( price? int + unavailable?
                                 + availablefrom? string) } .
9      send@y order![*:string] .
10     receive {
11         @y price?[n:int]
12         @y unavailable?
13         @y availablefrom?[date:string]}
14 }
```

The contract advertised by `BuyerA` at line 2 is *not* compliant with the contract `C` advertised by the store: indeed, after sending the `order`, `BuyerA` only expects to receive the `price` — while the store can also choose to send `unavailable`. Therefore, any service implementing `BuyerA` will never be put in a session with the `Store`. Instead, the contract advertised at line 8 by `BuyerB` is compliant with `C`. Note that this is true also if the two contracts are not one dual of each other: indeed, `BuyerB` accepts all the messages that the store may send (i.e., `price` and `unavailable`), and it also allows for a further message (`availablefrom`), to be used e.g. to notify when the item will be available. Although this message will never be used by the `Store`, it could allow `BuyerB` to establish sessions with more advanced stores. The symbol + is used to group receive actions, and it denotes an *external choice*, one which is not made by the buyer. At lines 11-13, `BuyerB` waits to receive at session `y` one of the messages declared in the contract.

Adding recursion

Note that our `Store` can only manage the `order` of a single item: if some buyer wants to order two or more items, she has to use distinct instances of the store. We now extend the store so that it can receive several orders in the same session, adding all the items to a cart.

We start by refining our contract as follows:

```
contract Crec {
    addToCart? string . Crec
    + checkout? . (
           price! int . (accept? + reject?)
           (+) unavailable!
    )
}
```

The contract `Crec` requires the store to accept from buyers two kinds of messages: `addToCart` and `checkout`. When a buyer chooses `addToCart`, the store must allow the buyer to order more items. This is done by recursively calling `Crec` in the `addToCart` branch. When a buyer stops adding items to the cart (by choosing `checkout`), the store must either send a `price` or state that the items are `unavailable`. In the first case, the store allows the buyer to `accept` the quotation and finalise the order, or to `reject` it and abort.

A possible specification of the store using the contract `Crec` is as follows:

```
specification StoreRec { tell x Crec . Loop(x) }
specification Loop(x:session) {
    receive {
        @x addToCart?[item:string] -> Loop(x)
        @x checkout? -> Checkout(x)
    }
}
specification Checkout(x:session) {
    if *     // checks whether the items are available
    then
        send@x price![*:int] .
        receive {
            @x accept?
            @x reject?
        }
    else send@x unavailable!
}
```

The store `StoreRec` advertises the contract `Crec`, and then continues as the process `Loop(x)`, where `x` is the handle to the new session. The process `Loop(x)` receives messages from buyers through session `x`. When it receives `addToCart`, it just calls itself recursively; instead, when it receives `checkout`, it calls the process `Checkout`. This process internally chooses whether to send the buyer a `price`, or to notify that the requested items are `unavailable`. In the first case, it receives from the client a confirmation, that can be either `accept` or `reject`.

A possible buyer interacting with `StoreRec` is the following:

```
specification BuyerC {
    tell y { addToCart! string . addToCart! string . checkout! .
         (price? int . (accept! (+) reject!) + unavailable?) } .
    send@y addToCart![*:string] .
    send@y addToCart![*:string] .
    send@y checkout! .
    receive {
        @y price?[n:int] ->
            if * then send@y accept! else send@y reject!
        @y unavailable? -> nil
    }
}
```

Note that the buyer's contract is compliant with `Crec`, even though the store contract is recursive, while the buyer's one is not.

1.3 Honesty

In an ideal world, one would expect that services respect the contracts they advertise, in *all* execution contexts: we call *honest* such services. In this section we illustrate, through a series of examples, that writing honest services may be difficult and error-prone. Further, we show how our tools may help service designers in specifying and implementing honest services.

A simple dishonest store

Our first example is a naïve CO_2 specification of the store advertising the contract `C` at page 5:

```
specification StoreDishonest1 {
    tell x C .
    receive@x order?[v:string] . (
        if *
        then send@x price![*:int]) }
```

The store above waits for an order of some item `v`. Then, it checks whether `v` is in stock (the actual test is abstracted by the `*:boolean` guard). If the item is in stock, the store sends a price quotation to the buyer (again, the price is abstracted in the specification).

Note that the store does nothing when the ordered item is not in stock. In this way, the store fails to respect its advertised contract `C`, which prescribes to always respond to the buyer by sending either `price` or `unavailable`. Therefore, we classify this specification of the store as *dishonest*.

In this paper we give an intuitive description of honesty, referring the reader to the literature [6, 7] for a formal definition. A specification `A` is honest when, in all possible executions, if a contract at some session requires `A` to do some action, then `A` actually performs it. Basically, this boils down to say that when `A` is required to send a message, then it does so. Likewise, when `A` is required to receive a message, then `A` is ready to accept any message that its partner may be willing to send. More in detail:

- if the contract is an *internal* choice `a1!S1 (+) ... (+) an!Sn`, then `A` must `send` a message having sort `Si`, and labelled `ai`, for some `i`;
- if the contract is an *external* choice `a1?S1 + ... + an?Sn`, then `A` must be able to `receive` messages labelled with *any* labels `ai` in the choice (with the corresponding sorts `Si`).

The honesty property discussed above can be automatically verified using the Diogenes honesty checker, which uses the verification technique described and implemented in [6]. This technique is built upon an abstract semantics of CO_2 which approximates both values (sent, received, and in conditional expressions) and the actual *context* wherein a specification is executed. Basically, the tool checks, through an exaustive exploration, that in every reachable state of the abstract semantics a participant is always able to perform some of the actions prescribed in each of her stipulated contracts. Since this is a branching-time property, a natural approach to verify it is by model checking. To this purpose we exploit a rewriting logic specification of the CO_2 abstract semantics and the Maude [12] search capabilities. This abstraction is a *sound* over-approximation of honesty: namely, if the abstraction of a specification is honest, then also the concrete one is honest. Further, the analysis is *complete* for specifications without conditional statements: i.e., if an abstracted specification is dishonest, then also its concrete counterpart is dishonest. If the abstractions are finite-state, we can verify their honesty by model checking a (finite) state space[2]. Our implementation first translates a CO_2 specification into a Maude term [12], and then uses the Maude model checker to decide the honesty of its abstract semantics.

The honesty checker outputs the message below, that reports that the specification `StoreDishonest1` is *dishonest*. The reason for its dishonesty can be inferred from the following output:

```
result:  ($ 0)(
  StoreDishonest1[if exp then do $ 0 "price" ! int . 0 else 0]
  | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
)
honesty: false
```

This shows a reachable (abstract) state of the specification, where `$ 0` denotes an open session between the store and a buyer.

The state consists of two parallel components: the state of the store

```
StoreDishonest1[if exp then do $ 0 "price" ! int . 0 else 0]
```

and the state of the contract at session `$ 0`, from the point of view of the store:

```
$ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
```

Such contract requires the store to send either `price` or `unavailable` to the buyer. However, if the guard `exp` of the conditional (within the state of the store) evaluates to false, the store will not send any message to the buyer, so violating the contract `C`. Therefore, the honesty checker correctly classifies `StoreDishonest1` as dishonest.

A more complex dishonest store

We now consider a more evolved specification of the store, which relies on external distributors to retrieve items. The contract `D` specifies the interaction between the store and distributors:

[2] Abstractions are finite-state in the fragment of CO_2 without delimitation/parallel under process definitions. For specifications outside this fragment the analysis is still correct, but it may diverge; indeed, a negative result [9] excludes the existence of algorithms for honesty that are at the same time sound, complete, and terminating in full CO_2.

```
contract D { req! string . ( ok? + no? ) }
```

Namely, the store first sends a request to the distributor for some item, and then waits for an `ok` or `no` answer, according to whether the distributor is able to provide the requested item or not.

Our first attempt to specify a store interacting with customers and distributors is the following:

```
1  specification StoreDishonest2 {
2      tell x C .
3      receive@x order?[v:string] .
4      tell y D .
5      send@y req![v] .
6      receive {
7          @y ok? -> send@x price![*:int]
8          @y no? -> send@x unavailable!
9      }
10 }
```

At line 2, the store advertises the contract `C`, and then waits until a session is established with some customer; when this happens, the variable `x` is bound to the session name. At line 3 the store waits to receive an `order`, binding it to the variable `v`. At line 4 the store advertises the contract `D` to establish a session `y` with a distributor; at line 5, it sends a `request` with the value `v`. Finally, the store waits to receive a response `ok` or `no` from the distributor, and accordingly responds `price` or `unavailable` to the customer (lines 6-9). The price `*:int` is a placeholder, to be replaced upon refinement.

The honesty checker classifies `StoreDishonest2` as *dishonest*. The reason for its dishonesty can be inferred from the following output:

```
result: ("y",$ 0)(
    StoreDishonest2[tell "y" D. (...)]
    | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0])
honesty: false
```

This output shows a possible (abstract) state which could be reached by `StoreDishonest2`. There, `$ 0` denotes an open session between the store and a buyer, while `"y"` indicates that no session between the store and a distributor is established, yet. The contract at session `$ 0` requires the store to send either a price or an unavailability message. However, in the given state there is no guarantee to find a distributor, hence the store might be stuck in the `tell`, never performing the required actions at session `$ 0`. Because of this, the store does not fulfil the contract `C`, hence it is correctly classified as dishonest.

Handling failures

We try to fix the specification `StoreDishonest2` by adapting it so to consider the case where the distributor is not available. Let us refine the specification `StoreDishonest2` as follows:

```
1  specification StoreDishonest3 {
2      tell x C .
3      receive@x order?[v:string] . (
4          tell y D .
5              send@y req![v] .
6              receive {
7                  @y ok? -> send@x price![*:int]
8                  @y no? -> send@x unavailable!
9              }
10         after * -> send@x unavailable!
11     )
12 }
```

Note that `StoreDishonest3` uses the construct **tell** ··· **after** ··· at lines 4-10. This ensures that, if no session is established within a given deadline, then the contract is *retracted* (i.e., removed from the registry of available contracts), and the control passes to the **after** process. In particular, in our `StoreDishonest3`, if no distributor is found, then `D` is retracted, and the store performs its duties with the buyer by sending him `unavailable`. Since the actual deadline is immaterial in this specification, it is abstracted here as `*`.

By running the honesty checker on the amended specification, we obtain:

```
result: ($ 0,$ 1)(
    StoreDishonest3
        [ retract $ 1 . ( ... )
        + ask $ 1 True . do $ 1 "req" ! string .
            ( do $ 1 "no" ? unit . do $ 0 "unavailable" ! unit .
                (...)
            + do $ 1 "ok" ? unit . do $ 0 "price" ! int . (...)
            )]
    | $ 0["price" ! int . 0 (+) "unavailable" ! unit . 0]
    | $ 1["req" ! string . ("no" ? unit . (0).Id + "ok" ? unit .
         (0).Id)]
    )
honesty: false
```

Note that `StoreDishonest3` is still dishonest. The output above shows a reachable (abstract) state where the store has opened two sessions, `$ 0` and `$ 1`, with a buyer and a distributor, respectively. At session `$ 0` the store is expected to send either `price` or `unavailable` to the buyer. Now, the store can perform `do $ 0 "price" ! int` only *after* receiving the input from the distributor, i.e. after performing `do $ 1 "ok" ? unit`. Similarly, the store can only perform the action `do $ 0 "unavailable" ! unit` after the action `do $ 1 "no" ? unit`. Should the distributor fail to send either of these messages, then the store would fail to honour its contract `C` with the buyer. Therefore, the honesty checker correctly classifies `StoreDishonest3` as dishonest. Note that, even if in this case the distributor would be dishonest as well, (since it violates the contract `D` with the store), this does not excuse the store from violating the contract `C` with the buyer.

An honest store, finally

In order to address the dishonesty issues in the previous specification, we revise the store as follows:

```
1  specification StoreHonest {
2      tell x C .
3      receive@x order?[v:string] . (
4          tell y D .
5              send@y req![v] .
6              receive {
7                  @y ok?  -> send@x price![*:int]
8                  @y no?  -> send@x unavailable!
9                  after * -> (
10                     send@x unavailable!
11                     | receive {
12                           @y ok? -> nil
13                           @y no? -> nil
14                       }
15                 )
16             }
17         after * -> send@x unavailable!
18     )
19 }
```

The main difference between this specification and the previous one is related to the `receive` at session `y`. At line 9, `after *` represents the case in which no messages are received within a given timeout (immaterial in this specification). In such case, the store fulfils its contract at session `x`, by sending `unavailable` to the buyer. Further, the store also fulfils its contract at session `y`, by receiving any message that could still be sent from the distributor after the timeout.

Now the honesty checker correctly detects that the revised specification `StoreHonest` is honest.

A recursive honest store

We reprise the specification of `StoreRec` in Section 1.2, by providing a recursive store which interacts with buyers (via contract `Crec` at page 7) and with distributors (via contract `D`).

```
specification StoreHonestRec {
    tell x Crec . Loop(x)
}

specification Loop(x:session) {
    receive {
        @x addToCart?[item:string] -> Loop(x)
        @x checkout? -> Checkout(x)
    }
}

specification Checkout(x:session) {
    tell y D .
        send@y req![*:string] .
        receive {
            @y ok?  -> send@x price![*:int] .
                receive {
                    @x accept?
                    @x reject?
                }
            @y no?  -> send@x unavailable!
            after * -> (
                send@x unavailable! |
                receive {
                    @y ok?
                    @y no?
                }
            )
        }
    after * -> send@x unavailable!
}
```

The specification `StoreHonestRec` handles the checkout of buyers in the process `Checkout`, which is identical to lines 4-14 in `StoreHonest`. The main difference with respect to `StoreHonest` is that `StoreHonestRec` can receive multiple requests from a buyer, via the recursive process `Loop(x)`. Despite this complication, the specification is still verified as honest by Diogenes.

1.4 Refining CO2 specifications in Java programs

Diogenes translates CO_2 specifications into Java skeletons, using the APIs of the contract-oriented middleware in [3]. This middleware collects the contracts advertised by services, establishes sessions between those with compliant contracts, and it allows services to send/receive messages through sessions, while monitoring this activity to detect and punish violations. More specifically, 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.

Compilation of CO2 specifications into Java skeletons

We illustrate the translation of CO_2 specifications into Java through an example, the `StoreHonest` given in the previous section. From it, we obtain the following Java skeleton[3]:

```java
public class StoreHonest extends Participant {
  public void run() {
    Session x = tellAndWait(C);

    Message msg = x.waitForReceive("order");
    String v = msg.getStringValue();

    try {
      Session y = tellAndWait(D, timeoutP);
      y.sendIfAllowed("req", v);

      try {
        Message msg_1 = y.waitForReceive(timeoutP,"ok","no");
        switch (msg_1.getLabel()) {
        case "ok": x.sendIfAllowed("price", intP); break;
        case "no": x.sendIfAllowed("unavailable"); break;
        }
      }
      catch (TimeExpiredException e) {
        parallel(()->{x.sendIfAllowed("unavailable");});
        parallel(()->{y.waitForReceive("ok","no");});
      }
    }
    catch(ContractExpiredException e) {
      //contract D retracted
      x.sendIfAllowed("unavailable");
    }
  }
}
```

[3] Minor cosmetic changes are applied to improve readability.

We comment below how the specification of `StoreHonest` at page 13 is rendered in Java.

- **`tell`** `x C` (at line 2) is translated into the assignment

  ```
  Session x = tellAndWait(C)
  ```

  The API method `tellAndWait` advertises the contract `C` to the middleware, and blocks until a compliant buyer contract is found. Then, it returns a new object, representing the newly established session between the store and the buyer.

- **`receive`** `@x order?[v:string]` (at line 3) is translated into

  ```
  Message msg = x.waitForReceive("order");
  String v = msg.getStringValue();
  ```

  where the call to `waitForReceive` blocks until the store receives a message labelled `order` on session `x`.

- The block **`tell`** `y D ...` **`after`** `* ...` (at lines 4-17) is translated in Java as the try-catch statement:

  ```
  try {
      Session y = tellAndWait(D, timeoutP);
      ...
  }
  catch(ContractExpiredException e) {
  ...
  }
  ```

  The call `tellAndWait(D, timeoutP)` advertises the contract `D`; the second parameter specifies a timeout (in milliseconds) to find a compliant contract. If the timeout expires, the contract `D` is retracted, and an exception is thrown. Then, the exception handler performs the recovery action specified in the **`after`** clause by sending `unavailable` to the client.

- **`send`** `@y req![*:string]` (at line 5) is translated as

  ```
  y.sendIfAllowed("req", stringP)
  ```

  This method call sends a message labelled `req` at session `y`, blocking until this action is permitted by the contract.

- The `receive` block at lines 6-16 is translated into a try-catch statement

```
try {
    Message msg_1 = y.waitForReceive(timeoutP,"ok","no");
    ...
}
catch (TimeExpiredException e) {
        parallel(()->{x.sendIfAllowed("unavailable");});
        parallel(()->{y.waitForReceive("ok","no");});
}
```

  The `waitForReceive` waits (until the given timeout) to receive on session `y` a message labelled either `yes` or `no`, throwing an exception in case the timeout expires. In such case, the `catch` block performs the recovery actions in the `after` clause of the specification. Namely, the service spawns two parallel processes, which send `unavailable` to the buyer, and receives late replies from the distributor.

Note that the timeout values `timeoutP`, as well as the order price `intP`, are just placeholders. Further, in an actual implementation of the store service, we may want e.g. to read the order price from a file or a database. This can be done by refining the skeleton, introducing the needed code to make the service actually implement the desired functionality.

Checking honesty of refined Java programs

Note that when refining the skeleton into the actual Java application, programmers could accidentally break its honesty. In general, this happens when the refinement alters the interaction behaviour of the service. For instance, in an actual implementation of our store service, we may want to delegate the computation of `price` to a separated method, as follows:

```
public int getOrderPrice(String order) throws MyException {...}
```

and change the placeholder `intP` at line 15 of the generated code with an invocation `getOrderPrice(v)`. The method could read the order price from a file or a database, and suppose that, in that method, each possible exception is either handled or re-thrown as `MyException`. If `getOrderPrice` throws an exception, then the `sendIfAllowed()` at line 15 is not performed. Unless the store performs it while handling `MyException`, the store violates the contract with the buyer, and so it becomes dishonest.

To address this issue, the Diogenes toolchain includes an *honesty checker* for Java programs, to be used after refinement. This honesty checker is built on top of *Java PathFinder* (JPF [27, 37]). We define suitable *listeners* for JPF, to intercept the requests to the contract-oriented middleware, and to simulate *all* the possible responses that the application can receive from it. Through JPF we symbolically execute the program, in order to infer a CO_2 specification that abstracts its behaviour, preserving dishonesty. Once a specification is constructed in this way, we apply the CO_2 honesty checker discussed in Section 1.3 to establish the honesty of the Java program.

We can check the honesty of a Java program through the static method `HonestyChecker.isHonest(StoreHonest.class)`, which returns one of the following values:

- `HONEST`: the tool has inferred a CO_2 specification and verified its honesty;
- `DISHONEST`: as above, but the inferred CO_2 specification is inferred, but it is dishonest;
- `UNKNOWN`: the tool has been unable to infer a CO_2 specification, e.g. because of unhandled exceptions within the class under test.

In our example, we just provide the following stub implementation of the method `getOrderPrice`:

```
@SkipMethod
public int getOrderAmount(String order) throws MyException {
    return 42; }
```

where the annotation `@SkipMethod` is interpreted by the honesty checker as follows: assume that the method terminates (possibly throwing one of the declared exceptions), and it does not interact with the contract-oriented middleware. For our refined store, the honesty checker returns `UNKNOWN`, outputting:

```
error details: MyException:
  This exception is thrown by the honesty checker.
      Please catch it!
  at i.u.c.store.StoreHonest.getOrderPrice(Store.java:30)
  at i.u.c.store.StoreHonest.run(Store.java:15)
  at i.u.c.honesty.HonestyChecker.runProcess(HonestyChecker.java
      :182)
```

As anticipated above, this output remarks that if `getOrderAmount` throws an exception, then the store is dishonest.

As a first (naïve) attempt to recover honesty, we further refine the store by catching `MyException`, and just logging the error in the exception handler:

```
try {
    ...
    case "ok": x.sendIfAllowed("price",getOrderPrice(v)); break;
    ...
```

```
}
catch (TimeExpiredException e) { ... }
catch (MyException e) { System.out.println("failed"); }
```

In this case, the honesty checker correctly classifies the store as `DISHONEST`, producing the following output:

```
result ($ 0,$ 1)(
   StoreHonest[0] |
   $ 0["price" ! unit . 0 (+) "unavailable" ! unit . 0] |
   $ 1[0])
honesty: DISHONEST
```

This output highlights the reason for dishonesty: `StoreHonest[0]` means that the store does nothing, while at session `$ 0`, it should send either `price` or `unavailable` to the buyer.

To recover honesty, rather than just logging the error, we also perform `x.sendIfAllowed("unavailable")` in the exception handler, in order to fulfil the contract with the buyer:

```
catch (MyException e) {
    System.out.println("failed");
    x.sendIfAllowed("unavailable");
}
```

With this modification, the Java honesty checker correctly outputs `HONEST`.

1.5 Conclusions

We have presented Diogenes, a toolchain for the specification and verification of contract-oriented services. Diogenes fills a gap between foundational research on honesty [6, 7, 8, 9] and more practical research on contract-oriented programming [3]. Our tools can help service designers to write specifications, check their adherence to contracts (i.e., their honesty), generate Java skeletons, and refine them while preserving honesty. We have experimented Diogenes with a set of case studies (more complex than the ones presented in this tutorial); our case studies are available at `co2.unica.it/diogenes`.

The effectiveness of our tools could be improved in several ways, ranging from the precision of the analysis, to the informative quality of output messages provided by the honesty checkers.

The precision of the honesty analysis could be improved e.g., by implementing the type checking technique of [7], which extends the class of infinite-state processes for which honesty can be verified. More specifically, the type system in [7] can also handle some processes with delimitation and parallel composition under recursion.

Another form of improvement would be to extend the formalism and the analysis to deal with timing constraints. This could be done e.g. by exploiting the timed version of CO_2 [3] and timed session types [2]. Although the current analysis for honesty does not consider timing constraints (and therefore is unsound in such scenario), it can still give useful feedback when applied to timed specifications. For instance, it could detect that some prescribed actions cannot be performed because the actions they depend on may be blocked by an unresponsive context.

When a specification/program is found dishonest, it would be helpful for programmers to know which parts of it is responsible for contract violations. The error reporting facilities of Diogenes could be improved to this purpose: this would require e.g., to signal what are the contract obligations that are not fulfilled, and in what session, and in particular which part of the specification/program should be fixed. Further, it would be useful to suggest possible corrections to the designer.

Another direction for future work is to formally establish relations between the original CO_2 specification and the refined Java code. In fact, our tools can only check that the user-refined Java code obtained from an honest CO_2 specification is honest, but this does not imply that the refined Java code still "adheres" to the specification. Indeed, improper refinements could drastically modify the interaction behaviour of a service, e.g. by removing some contract advertisements — while preserving honesty. An additional static analysis could establish that the CO_2 process inferred from the user-refined Java code is behaviourally related to the original specification. An alternative way to cope with this issue would be to enhance the generation of the skeletal Java program, by providing a more structured class hierarchy. More precisely, we could avoid accidental breaches of honesty by separating, in the generated skeleton, the part that handles the interactions from the parts to be refined. This could be done e.g. by inserting entry points to invoke classes/interfaces whose behaviour is defined apart, so separating the application logic and simplifying possible updates in the specifications.

Related work

In recent years many works have addressed the safe design of service-oriented applications. A notable approach is to specify the overall communication behaviour of an application through a *choreography*, which validates some global properties of the application (e.g. safety, deadlock-freedom, *etc.*). To ensure that the application enjoys such properties, all the components forming the application have to be verified; this can be done e.g. by projecting the choreography to end-point views, against which these components are verified [35, 21]. Examples of how to embody such approach in existing programming languages and models are presented for `C` [33], for `Python` [30], and for the actor model [31]. All those approaches are based on `Scribble` [38], a protocol description language featuring multiparty session types [21]. The strict relations between multiparty session types and actor-based models such as communicating machines [15] has been used to develop a framework to monitor `Erlang` applications [18].

This top-down approach assumes that designers control the whole application, e.g., they develop all the needed components. However, in many real-world scenarios several components are developed independently, without knowing at design time which other components they will be integrated with. In these scenarios, the compositional verification pursued by the top-down approach is not immediately applicable, because the choreography is usually unknown, and even if it were known, only a subset of the needed components is available for verification. However, this issue can be mitigated when the communication pattern of each component is available. In fact, in such case if the set of components is compatible, it is possible to synthesise a faithful choreography [26] with a suitable tool [24]. Such choreography can then be used to distil monitors for the components that are not trusted (if any). The ideas pursued in this paper depart from the top-down approach, because designers can advertise contracts to discover the needed components (and so ours can be considered a *bottom-up* approach). Coherently, the main property we are interested in is *honesty*, which is a property of components, and not of global applications. Some works mixing top-down and bottom-up composition have been proposed in the past few years [5, 16, 25]. Recent works [32] have explored how to integrate the bottom-up approach with inference of multiparty session types from `GO` programs.

The problem of ensuring safe interactions in session-based systems has been addressed by many authors [10, 11, 13, 14, 17, 19, 21, 22, 23, 36]. When processes have a single session, our notion of honesty is close (yet different) to session typeability. A technical difference is that we admit processes to attempt interactions which are not mandated by the contract. E.g., the process:

```
specification P {
  tell x { a! . b! } . (send @x a! | send @x b!)
}
```

is honest, while it would *not* be typeable according to most works on session types, because the action b is not immediately mandated by the contract.

Other, more substantial, differences between honesty and session typing arise when processes have more than one session. More specifically, we consider a process to be honest when it enjoys progress in *all* possible contexts, while most works on session typing guarantee progress in a given context. For instance, consider the process:

```
specification Q {
  tell x { a! } . tell y { b? } . receive @y b? . send @x a!
}
```

We have that Q is *not* honest, because the action at session x is not possible if the participant at the other endpoint of session y does not send b. Note instead that Q would be well-typed in [20], even if some contexts R can lead Q to a deadlock. The interaction type system in [14] would allow to check the progress of Q|R, given a context R.

Acknowledgments. This work has been partially supported by Aut. Reg. of Sardinia P.I.A. 2013 "NOMAD", and by EU COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems" (BETTY).

References

[1] 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.

[2] 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.

[3] 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.

[4] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Compliance in behavioural contracts: a brief survey. In *Programming Languages with Applications to Biology and Security*, volume 9465 of *LNCS*, pages 103–121. Springer, 2015.

[5] Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. Choreographies in the wild. *Science of Computer Programming*, 109:36 – 60, 2015.

[6] 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.

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

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

[9] Massimo Bartoletti and Roberto Zunino. On the decidability of honesty and of its variants. In *Web Services, Formal Methods, and Behavioral Types*, volume 9421 of *LNCS*, pages 143–166. Springer, 2015.

[10] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[11] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of session types. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 219–230. ACM, 2009.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *TCS*, 2001.

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION*, volume 7890 of *LNCS*, pages 45–59. Springer, 2013.

[14] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016.

[15] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *European Symposium on Programming (ESOP)*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[16] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.

[17] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *Trustworthy Global Computing (TGC)*, volume 4912 of *LNCS*, pages 257–275. Springer, 2007.

[18] Simon Fowler. An Erlang implementation of multiparty session actors. In *Interaction and Concurrency Experience*, volume 223 of *EPTCS*, pages 36–50, 2016.

[19] 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.

[20] 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.

[21] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[22] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[23] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[24] Julien Lange and Emilio Tuosto. A toolchain for choreography-based analysis of application level protocols. Available at `https://bitbucket.org/emlio_tuosto/gmc-synthesis-v0.2`.

[25] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In *CONCUR*, volume 7454 of *LNCS*, pages 225–239. Springer, 2012.

[26] 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, 2015.

[27] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN workshop on Model checking of software*, pages 80–102, 2001.

[28] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[29] 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.

[30] Rumyana Neykova. Session types go dynamic or how to verify your Python conversations. In *Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, volume 137 of *EPTCS*, pages 95–102, 2013.

[31] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.

[32] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *International Conference on Compiler Construction (CC)*, pages 174–184. ACM, 2016.

[33] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns (TOOLS)*, pages 202–218, 2012.

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

[35] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106, 2010.

[36] V. T. Vasconcelos. Fundamentals of Session Types. *Information and Computation*, 217:52–70, 2012.

[37] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[38] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *Trustworthy Global Computing (TGC)*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.