Developing Honest Java Programs with Diogenes

Nicola Atzei and Massimo Bartoletti(✉)

Università Degli Studi di Cagliari, Cagliari, Italy
{atzeinicola,bart}@unica.it

Abstract. Modern distributed applications are typically obtained by integrating new code with legacy (and possibly untrusted) third-party services. Some recent works have proposed to discipline the interaction among these services through *behavioural contracts*. The idea is a dynamic discovery and composition of services, where only those with compliant contracts can interact, and their execution is monitored to detect and sanction contract breaches. In this setting, a service is said *honest* if it always respects the contracts it advertises. Being honest is crucial, because it guarantees a service not to be sanctioned; further, compositions of honest services are deadlock-free. However, developing honest programs is not an easy task, because contracts must be respected even in the presence of failures (whether accidental or malicious) of the context. In this paper we present Diogenes, a suite of tools which supports programmers in writing honest Java programs. Through an Eclipse plugin, programmers can write a specification of the service, verify its honesty, and translate it into a skeletal Java program. Then, they can refine this skeleton into proper Java code, and use the tool to verify that its honesty has not been compromised by the refinement.

1 Introduction

Developing modern distributed 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. Unless these services are implemented in a decentralized manner (e.g., as smart contracts in Ethereum or Contractvm [11,13]), they will be under the governance of mutually distrusting providers, possibly competing among each other. Typically, these providers offer little or no guarantees about the services they control, and in particular they reserve the right to change the service code (if not the Service Level Agreement *tout court*) at their discretion.

Therefore, to guarantee the reliability and security of distributed applications, one cannot directly apply standard analysis techniques for programming languages (like e.g., type systems or model checking). Indeed, these analysis techniques usually need to inspect the code of the *whole* application, while under the given assumptions one can only reason about the services under their control. In particular, compositional verification based on choreographies [1,16] is not suitable in this setting, because to ensure the correctness of the whole application, all its components have to be verified.

E. Albert and I. Lanese (Eds.): FORTE 2016, LNCS 9688, pp. 52–61, 2016.
DOI: 10.1007/978-3-319-39570-8_4

From Service-Oriented to Contract-Oriented Computing. A possible countermeasure to these issues is to use *contracts* to regulate the interaction between services. By advertising a contract, a service commits itself to respect a given behaviour when, after stipulation, it will interact with others. In this setting, a service infrastructure acts as a trusted third party, which collects all the advertised contracts, and establishes sessions between participants with compliant ones. Participants can then interact through sessions, by sending and receiving messages as required by their contracts (or even choosing to violate them, if they find this choice more convenient). An actual implementation of this paradigm is the middleware in [5], which offers a set of APIs through which services can advertise, stipulate, and execute contracts (based on binary session types [14]).

To incentivize honest behaviour, contract-oriented infrastructures monitor all the messages exchanged among services, to sanction those which do not respect their contracts. Sanctions can be of different nature: e.g., pecuniary compensations, adaptations of the service binding [19], or they can decrease the reputation of a service whenever it violates a contract, in order to marginalize dishonest services in the selection phase [5].

Experimental evidence about the programming paradigm and incentive mechanisms of [5] 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 sanction mechanism of contract-oriented services allows a new kind of attacks: 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 an `order` from a client, and then either `ship` the ordered item or `abort` the transaction";
2. wait to receive an `order`;
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 `ship` to the client; otherwise, `abort` both transactions.

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 client: indeed, the store should either `ship` or `abort`, 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. This may happen either unintentionally (because of errors in the service specification, or in its implementation), or even because of malicious behaviour. Therefore, to avoid sanctions a service must be able to respect *all* the contracts it advertises, in *all*

possible contexts — even those populated by adversaries. We call this property *honesty*. Whenever compliance between contracts ensures their deadlock-freedom (as for the relations in [2,3,17,20]), the honesty property can be lifted from contracts to services: systems of honest services are deadlock-free (Theorem 6 in [9]).

Some recent works have studied honesty at the specification level, using the process calculus CO_2 for modelling contract-oriented services [6,8,9]. Practical experience with CO_2 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 all the possible moves of the context, but at design time he does not yet know in which context his service will be run. Hence, tools to automate the verification of honesty in CO_2 may be of great help.

A further obstacle to the development of honest services is that, even if we start from an honest CO_2 specification, it is still possible that honesty is not preserved when refining the specification into an actual implementation. Analysis techniques for checking honesty at the level of the implementation are therefore needed in order to develop reliable contract-oriented applications.

Contributions. To support programmers in the development of contract-oriented applications, we provide a suite of tools (named *Diogenes*) with the following features: (i) writing CO_2 specifications of services within an Eclipse plugin; (ii) verifying honesty of these specifications; (iii) generating from them skeletal Java programs which use the contract-oriented APIs of the middleware in [5]; (iv) verifying the honesty of Java programs upon refinement. We validate our tools by applying them to all the case studies in [6]: in particular, we specify each of these case studies in CO_2, and we successfully verify the honesty of both the specifications and their Java refinements. Overall, we can execute these verified services using the middleware in [5], being guaranteed that they will not incur in sanctions, and that interactions with other honest services will be deadlock-free. Our tools, the case studies, and a tutorial are available at co2.unica.it/diogenes.

2 Diogenes in a Nutshell

In this section we show the main features of our tools with the help of a small example. Suppose we want to implement an online store which receives orders from customers, and relies upon external distributors to retrieve items.

Contracts. The store has two contracts: `C` specifies the interaction with customers, and `D` with distributors. In `C`, the store declares that it will wait for an `order`, and then send either the corresponding `amount` or an `abort` message (e.g., in case the ordered item is not available). The answer may depend on an external distributor service, which waits for a `req`uest, and then answers `ok` or `no`. We specify these two contracts as the following binary session types [15]:

```
contract C { order? string . ( amount! int (+) abort! ) }
contract D { req! string . ( ok? + no? ) }
```

Receive actions are marked with the symbol `?` and grouped by `+`; instead, send actions are marked with `!` and grouped by `(+)`. The symbol `.` denotes sequential composition. We specify the sort of a message (`int`, `string`, or `unit`) after the action label; sort `unit` is used for pure synchronization, and it can be omitted.

Specification. A naïve CO_2 specification of our store is the following:

```
1  specification StoreDishonest {
2      tellAndWait x C .
3      receive x order? v:string . (
4          tellAndWait y D . (
5              send y req! *:string .
6              receive [
7                  y <- ok? . send x amount! *:int
8                  y <- no? . send x abort!
9              ] + t . send x abort!))
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 `req`uest with the value `v`. Finally, the store waits to receive a response `ok` or `no` from the distributor, and accordingly responds `amount` or `abort` to the customer (lines $6-8$). The sent amount `*:int` is placeholder, to be replaced upon refinement. The internal action `t` at line 9 models a timeout, fired in case no response is received from the distributor.

Our tool correctly detects that this specification is *dishonest*, outputting:

```
result: ("y",$ 0)(
    StoreDishonest[tell "y" D . ask "y" True . (...)]
    | $ 0["abort" ! unit . 0 (+) "amount" ! int . 0])
result: ($ 0,$ 1)(
    StoreDishonest[do $ 0 "abort" ! unit . (0).Sum]
    | $ 0["abort" ! unit . 0 (+) "amount" ! int . 0]
    | $ 1[ready "no" ? unit . 0])
honesty: false
```

There are two causes for dishonesty. First, if the session `y` is never established (e.g., because no distributor is available), the store is stuck at line 4 and cannot fulfil `C` at session `x` (`$0` in the output message). Second, if the distributor response arrives after the timeout (line 9), the store does not consume its input, and so it does not respect the contract `D` at session `y` (`$1` in the output message).

A possible way to fix the previous specification is the following:

```
1  specification StoreHonest {
2      tellAndWait x C .
3      receive x order? v:string . (
4          tellRetract y D . (
5              send y req! *:string .
6              receive [
7                  y <- ok? . send x amount! *:int
8                  y <- no? . send x abort!
9              ] + t . (send x abort! | receive y ok? no?))
10         : send x abort!)
11 }
```

The primitive `tellRetract` at line 4 ensures that if the session `x` is not established within a given deadline (immaterial in the specification) the contract `D` is retracted, and the control passes to line 10. Further, in the timeout branch we have added a parallel execution of a `receive` to consume orphan inputs (line 9). Now the tool correctly detects that the revised specification is honest.

An Execution Context. We now show a possible context wherein to execute our online store. Although the context is not needed for checking the honesty of the store, we use it to complete the presentation of the primitives of CO_2.

```
1  specification Buyer {
2      tellAndReturn x { order! string . ( amount? int + abort? ) } .
3      send x order! *:string .
4      receive [
5          x <- amount? n:int
6          x <- abort?
7      ]
8  }
9
10 specification Distributor {
11     tellRetract x { req? string . ( ok! (+) no! ) } .
12     receive x req? msg:string .
13     if *:boolean then send x ok! else send x no!
14 }
```

The contracts of the `Buyer` (lines 2) and that of the `Distributor` (line 11) are compliant with the contracts `C` and `D` advertised by the store. The `Buyer` uses the statement `tellAndReturn` to advertise its contract: it does not wait the session is established, but postpone the waiting phase until it is strictly required (line 3 in this case), although the session could be already started in the meantime. The `Distributor` uses a conditional statement (with a dummy guard `*:boolean`) to choose whether accepting or not the store request.

Code Generation and Refinement. Diogenes translates CO_2 specifications into Java skeletons, using the APIs of the middleware in [5]. For instance, from the `StoreHonest` specification given above, it generates the following skeleton[1]:

```
1  public class StoreHonest extends Participant {
2    public void run() {
3      Session<TST> x = tellAndWait(C);              // (line 2)
4
5      Message msg = x.waitForReceive("order");      // (line 3)
6      String v = msg.getStringValue();
7
8      try {
9        Session<TST> y = tellAndWait(D, 10000);     // (line 4)
10       y.sendIfAllowed("req", stringP);            // (line 5)
11
12       try {
13         Message msg_1 = y.waitForReceive(10000,"ok","no"); // (line 6)
14         switch (msg_1.getLabel()) {
15         case "ok": x.sendIfAllowed("amount", intP); break; // (line 7)
16         case "no": x.sendIfAllowed("abort"); break;        // (line 8)
17         }
18       }
19       catch (TimeExpiredException e) {                     // (line 9)
```

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

```
20            parallel(()->{x.sendIfAllowed("abort");});
21            parallel(()->{y.waitForReceive("ok","no");});
22          }
23        }
24        catch(ContractExpiredException e) {
25          //contract D retracted
26          x.sendIfAllowed("abort");                    // (line 10)
27        }
28      }
29    }
```

We use Java exceptions to deal with both the `tellRetract` and `receive` primitives: the `ContractExpiredException` is thrown by line 9 if the session `y` is not established within a given timeout (10 s), while the `TimeExpiredException` is thrown by line 13 if a message is not received within the timeout. The `parallel` method at lines 20-21 starts a new thread which executes the given `Runnable` instance. The timeout values, as well as the order amount at line 15 , are just placeholders; in an actual implementation of the store service, we may want to delegate the computation of `amount` to a separated method, e.g.:

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

and change the placeholder `intP` at line 15 with `getOrderAmount(v)`. The method could read the order amount from a file or a database, and suppose that each possible exception is caught and hidden behind `MyException`. The failure of this method can be considered non-deterministic, so we need to "instruct" our verification tool in order to consider all the possible ways the method can terminate. To this purpose, we provide the annotation `@SkipMethod(value="<value>")`, interpreted by the checker as follows: (i) assumes that the method does not perform any action directed to the middleware; (ii) considers `<value>` as the returning value on success; (iii) considers the declared exceptions as possible exit points on failure. Diogenes can symbolically consider both the case of a normal method termination and all the possible exceptional terminations.

Verification. 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.

For our refined store, the Java honesty checker returns `UNKNOWN` and outputs:

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

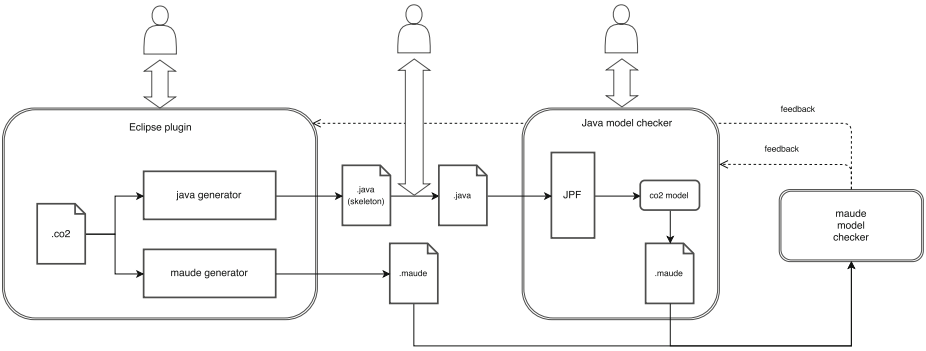

Fig. 1. Data flow schema

This means that if the method `getOrderAmount` fails, then the program will terminate abruptly, and so the store may violate the contract. We can recover honesty by catching `MyException` with `x.sendIfAllowed("abort")`. With this modification, the Java honesty checker correctly outputs `HONEST`.

3 Architecture

Diogenes has three main components: an honesty checker for CO_2, an honesty checker for Java, and an Eclipse plugin which integrates the two checkers with an editor of CO_2 specifications. We sketch the architecture of our tools in Fig. 1.

The CO_2 honesty checker implements the verification technique introduced 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 process is executed. This abstraction is a *sound* over-approximation of honesty: namely, if the abstraction of a process is honest, then also the concrete one is honest. Further, in the fragment of CO_2 without conditional statements the analysis is also *complete*, i.e. if a concrete process is honest, then also its abstraction is honest. For processes without delimitation/parallel under process definitions, the associated abstractions are finite-state, hence we can verify their honesty by model checking a (finite) state space. For processes outside this class the analysis is still correct, but it may not terminate; indeed, a negative result in [9] excludes the existence of algorithms for honesty that are at the same time sound, complete, and terminating in full CO_2. Our implementation first translates a CO_2 process into a Maude term [10], and then uses the Maude LTL model checker [12] to decide honesty of its abstract semantics.

The Java honesty checker is built on top of *Java PathFinder* (JPF, [18,21]). The JPF core is a virtual machine for Java bytecode that can be configured to act as a model checker. We define suitable *listeners* to catch the requests to the contract-oriented middleware [5], and to simulate *all* the possible responses that the application can receive from it. Through JPF we symbolically execute and backtrack the program, and eventually we infer a CO_2 specification that

over-approximates its behaviour. Then, we apply the CO_2 honesty checker to establish the honesty of the Java program. The accuracy of the inferred CO_2 specification partially relies on the programmer: the methods involved in the application logic have to be correctly annotated, and in particular they have to declare all possible exceptions that can be thrown at runtime.

The Eclipse plugin supports writing CO_2 specifications, providing syntax highlighting, code auto-completion, syntactic/semantic checks, and static type checking on sort usage. It relies on Xtext (www.eclipse.org/Xtext), a framework for developing programming languages, and on Xsemantics (xsemantics.sourceforge.net), a domain-specific language for writing type systems.

4 Conclusions

Diogenes fills a gap between foundational research on honesty [6,8,9] and more practical research on contract-oriented programming [5]. Its effectiveness can 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 accuracy of the honesty analysis could be improved e.g., by implementing the type checking technique of [7], which can correctly classify the honesty of some forms of infinite-state of processes (while the current honesty checker is only guaranteed to terminate for processes without delimitation/parallel within recursion). Another form of improvement would be to extend the analysis to deal with timing constraints. This could be done e.g. by exploiting the timed version of CO_2 proposed in [5] and the timed session types of [4]. Although the current analysis for honesty does not consider timing constraints, it may give useful feedback also 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.

The error reporting facilities could be improved in several directions: e.g., it would be helpful for programmers to know which parts of the program make it dishonest, what are the contract obligations that are not fulfilled, and in what session. Further, it would be useful to suggest possible corrections to the designer.

Another direction for future work concerns relating the original CO_2 specification with the refined Java code. In fact, our tools only guarantee that the Java skeleton generated from an (honest) CO_2 specification is (honest and) coherent with the specification. If the programmer further refines the Java code, e.g. by removing some contract advertisements, then the Java honesty checker can still check that the resulting code is honest, but the coherence with the original specification may be lost. An additional static analysis could establish that the CO_2 process inferred from the refined Java code by JPF is a (sort of) *subtype* of the original specification, and so that it can be safely used in the same contexts.

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. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: agreeing and implementing interorganizational processes. Comput. J. **53**(1), 90–106 (2010)
2. Acciai, L., Boreale, M., Zavattaro, G.: Behavioural contracts with request-response operations. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 16–30. Springer, Heidelberg (2010)
3. Barbanera, F., de'Liguoro, U.: Two notions of sub-behaviour for session-based client/server systems. In: PPDP, pp. 155–164. ACM (2010)
4. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A.S., Pompianu, L.: Compliance and subtyping in timed session types. In: Graf, S., Viswanathan, M. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. LNCS, vol. 9039, pp. 161–177. Springer, Heidelberg (2015)
5. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A.S., Pompianu, L.: A contract-oriented middleware. In: Braga, C., et al. (eds.) FACS 2015. LNCS, vol. 9539, pp. 86–104. Springer, Heidelberg (2016). doi:10.1007/978-3-319-28934-2_5. http://co2.unica.it
6. Bartoletti, M., Murgia, M., Scalas, A., Zunino, R.: Verifiable abstractions for contract-oriented systems. JLAMP (2015, to appear)
7. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 305–320. Springer, Heidelberg (2013). http://tcs.unica.it/papers/HbT.pdf
8. Bartoletti, M., Tuosto, E., Zunino, R.: Contract-oriented computing in CO_2. Sci. Ann. Comp. Sci. **22**(1), 5–60 (2012)
9. Bartoletti, M., Zunino, R.: On the decidability of honesty and of its variants. In: Hildebrandt, T., Ravara, A., van der Werf, J.M., Weidlich, M. (eds.) WS-FM 2014 + WS-FM 2015. LNCS, vol. 9421, pp. 143–166. Springer, Heidelberg (2016). doi:10.1007/978-3-319-33612-1_9
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. In: TCS (2001)
11. Contractvm. https://github.com/contractvm
12. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude LTL model checker. Electr. Notes Theor. Comput. Sci. **71**, 162–187 (2002)
13. Ethereum. https://github.com/ethereum/
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
17. Laneve, C., Padovani, L.: The *must* preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
18. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)

19. Mukhija, A., Dingwall-Smith, A., Rosenblum, D.: QoS-aware service composition in Dino. In: ECOWS, LNCS, vol. 5900, pp. 3–12. Springer (2007)
20. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. **205**(2), 125–198 (2007)
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)