Developing honest Java programs with Diogenes

Nicola Atzei and Massimo Bartoletti

Università degli Studi di Cagliari, Italy

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 contract it advertises. Being honest is crucial, because it guarantees a service not to be sanctioned; further, compositions of honest services enjoy deadlock-freedom. 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 CO_2 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, and may be subject to failures and attacks. These services may be under the governance of mutually distrusting providers (possibly competing among each other), and interact through open networks, where attackers can try to exploit their vulnerabilities. To guarantee the reliability and security of these applications, one cannot directly apply standard analysis techniques for programming languages (like e.g., type systems), since they usually need to inspect the code of the whole application, while under the given assumptions one can only reason about the services under their control.

A possible countermeasure to these issues is to regulate the interaction between services through *contracts*. By advertising a contract, a service commits itself to stick to a given behavior when he will interact with other services. 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. To incentivize honest behaviour, the infrastructure monitors all the messages exchanged among services, and it sanctions those which do not

respect their contracts. These sanctions can be pecuniary compensations, adaptations of the service binding [15], or they can decrease the service reputation [5] in order to marginalize dishonest services in the binding phase.

The sanction mechanism of contract-oriented services allows for a new kind of attacks: malicious users can try to exploit possible discrepancies between the promised and the actual behaviour of a service in order to make it sanctioned. Since these attacks may compromise the service and cause economic damage to its provider, it is important to detect these vulnerabilities *before* deployment. Intuitively, a service is vulnerable if, in *some* execution context, it does *not* respect 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 compliance relations in [3,4,13,16]), then the honesty property is lifted from contracts to services: systems of honest services are deadlock-free [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 modelling 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 of implementation are therefore needed in order to develop reliable contract-oriented applications.

Contributions. To support programmers in the development of contract-oriented applications, we provid a suite of tools (named *Diogenes*) with the following features: (i) writing CO_2 specification of services within an Eclipse plugin; (ii) verifying honesty of CO_2 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 have validated our tools by applying them to all the case studies in [6]: an online store with bank, a voucher distribution system, a car purchase financed with a loan, an online casino featuring blackjack, and a travel agency. We have specified each of these case studies in CO_2, and we have successfully verified the honesty of both the specifications and of 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 they will enjoy deadlock-freedom when interacting with other honest services. Our verification tools, the Eclipse plugin, and an online tutorial about Diogenes 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` to interact with customers, and `D` to interact 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`est, and then answers `ok` or `no`. We write these two contracts as the following binary session types [12]:

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

Receive actions are denoted by the question mark (`?`) and grouped with the symbol `+`; similarly, we use the bang (`!`) and the symbol `(+)` for send actions. We specify the sort of a message (`int`, `string` or `unit`) after the action label; when the sort is omitted, we assume it is `unit`.

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

```
1  specification StoreDishonest {
2      tellAndWait x C .
3      receive x [
4          order? v:string . (
5              tellAndWait y D . (
6                  send y req! v .
7                  receive y [
8                      ok? . send x amount! 100
9                      + no? . send x abort!
10                     + t . send x abort!]))]
11 }
```

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. Then, the store waits to receive an `order`, binding it to the variable `v` (lines 3-4). At this point the store advertises the contract `D` to establish a session `y` with a distributor (line 5); after that, it sends a `req`uest (line 6) 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 7-9). The internal action `t` models a timeout, fired in case no response is received from the distributor (line 10).

Our tool correctly detects that the above specification is *dishonest*, by saying:

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

The tool output hints the two causes of dishonesty. First, if the session `y` is never established (which may happen when no distributor is available), the store is stuck at line 5 and cannot fulfil the contract `C` at session `x` (`$0` in the output above). Second, if the distributor response arrives after the timeout (line 10), the store does not consume the input and so it does not respect the contract `D`.

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

```
1  specification StoreHonest {
2      tellAndWait x C .
3      receive x [
4          order? v:string . (
5              tellRetract y D . (
6                  send y req! v .
7                  receive y [
8                      ok? . send x amount! 100
9                      + no? . send x abort!
10                     + t . (send x abort! | receive y [ok? + no?])])
11             : send x abort!)]
12 }
```

The primitive `tellRetract` at line 5 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 11. Further, in the timeout branch we have added a parallel execution of a `receive` to consume possible inputs. Now the tool correctly detects that the revised specification is honest.

Code generation and refinement. Diogenes translates CO_2 specifications into Java skeletons, using the APIs of the contract-oriented middleware in [5]. From the honest specification given above, it generates the following skeleton:

```
1  public class StoreHonest extends Participant {
2    public void run() {
3      Session<TST> x = tellAndWait(C);          // tellAndWait x C
4
5      Message msg = x.waitForReceive("order");  // receive c [ order? v:string ]
6      String v = msg.getStringValue();
7
8      try {
9        Session<TST> y = tellAndWait(D, 10000); // tellRetract y D
10       y.sendIfAllowed("req", v);
11
12       try {                                   // receive y [ok? + no? + t]
13         Message msg_1 = y.waitForReceive(10000, "ok", "no");
14         switch (msg_1.getLabel()) {
15           case "ok": x.sendIfAllowed("amount", 100); break;
16           case "no": x.sendIfAllowed("abort"); break;
17         }
18       }
19       catch (TimeExpiredException e) {        // send x abort! | receive y [ok? + no?]
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");               // : send x abort (line 11)
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 ten seconds, while the `TimeExpiredException` is thrown by line 13 if a message is not received within (again) ten seconds. The `parallel` method at lines 20-21 starts a new thread that will execute the passed `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 the `amount` to a separated method, e.g.:

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

and change the number `100` 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>")`, that is interpreted by the checker as follows: (i) ignore what the method really does (it imposes that an annotated method does not perform any action directed to the middleware); (ii) consider `<value>` as the returning value on success; (iii) consider the declared exceptions as possible exit points on failure. Diogenes can symbolically consider both the case of a normal termination of the method and all the possible exceptional terminations.

Verification. Diogenes can verify the honesty of Java programs by invoking the static method `HonestyChecker.isHonest(StoreHonest.class)`, which returns one of the following values:

- `HONEST`: the tool has extracted a CO_2 specification and verified its honesty;
- `DISHONEST`: as above, but the CO_2 specification is dishonest;
- `UNKNOWN`: the tool has been unable to extract 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 it.unica.co2.store.Store$Phonest.getOrderAmount(Store.java:166)
    at it.unica.co2.store.Store$Phonest.run(Store.java:129)
    at it.unica.co2.honesty.HonestyChecker.runProcess(HonestyChecker.java:182)
```

This means that if the method `getOrderAmount` fails the program will be terminated 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 Figure 1.

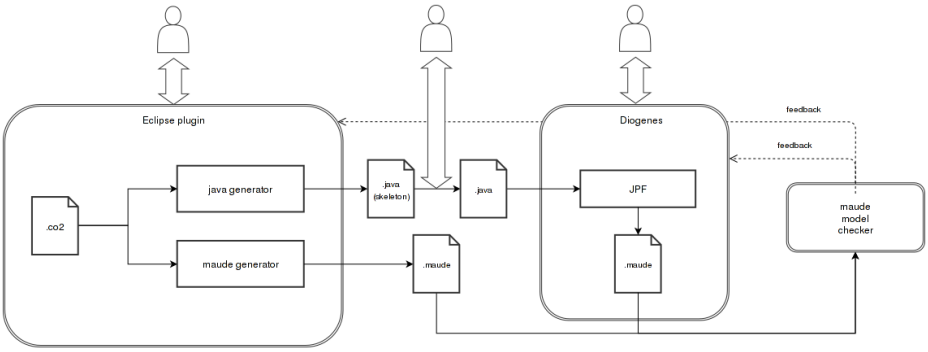

Fig. 1. Data flow schema

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 associate abstraction is 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 algorithm 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 [11] to decide honesty of its abstract semantics.

The Java honesty checker is built on top of *Java PathFinder* (JPF, [14,17]). 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 executes and backtracks the program, and eventually we obtain 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 provides advanced support for writing CO_2 specifications, like syntax highlighting, code auto-completion, outline view, syntax/semantics checks, and static type checking to detect errors in the use of sorts. The plugin relies on Xtext [2], a framework for developing programming languages, and Xsemantics [1], a domain-specific language for writing type systems for Xtext-

based languages. We use Xtext to define the grammar of CO_2 and we extend it to obtain custom semantic checks and suitable scope resolution of variables.

4 Conclusions

Diogenes fills a gap between foundational research on honesty [8,6,9] 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 analysis could be improved e.g., by implementing the type checking technique of [7], which can correctly classify the honesty of processes with delimitation/parallel within recursive processes. For informativeness, it would be helpful for programmers to know which parts of the program make it dishonest.

References

1. Xsemantics – a DSL for writing rules for Xtext languages. http://xsemantics.sourceforge.net/. Accessed: 2016-02.
2. Xtext – a framework for development of programming languages and domain-specific languages. http://www.eclipse.org/Xtext/. Accessed: 2016-02.
3. L. Acciai, M. Boreale, and G. Zavattaro. Behavioural contracts with request-response operations. In *COORDINATION*, 2010.
4. F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, 2010.
5. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A contract-oriented middleware. In *FACS*, 2015. http://co2.unica.it.
6. M. Bartoletti, M. Murgia, A. Scalas, and R. Zunino. Verifiable abstractions for contract-oriented systems. *JLAMP*, 2015. To appear.
7. M. Bartoletti, A. Scalas, E. Tuosto, and R. Zunino. Honesty by typing. In *FMOOD-S/FORTE*, volume 7892 of *LNCS*, pages 305–320, 2013.
8. M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1):5–60, 2012.
9. M. Bartoletti and R. Zunino. On the decidability of honesty and of its variants. In *WSFM/BEAT*, 2015. To appear.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *TCS*, 2001.
11. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. *Electr. Notes Theor. Comput. Sci.*, 71:162–187, 2002.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, 1998.
13. C. Laneve and L. Padovani. The *must* Preorder Revisited. In *CONCUR*, 2007.
14. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN workshop on Model checking of software*, 2001.
15. A. Mukhija, A. Dingwall-Smith, and D. Rosenblum. Qos-aware service composition in Dino. In *ECOWS*, pages 3–12, 2007.
16. A. Rensink and W. Vogler. Fair testing. *Info. and Co.*, 205(2):125–198, 2007.
17. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.