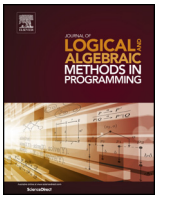
Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Verifiable abstractions for contract-oriented systems ☆

Massimo Bartoletti [a,*], Maurizio Murgia [a], Alceste Scalas [a], Roberto Zunino [b]

[a] *Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy*
[b] *Dipartimento di Matematica, Università degli Studi di Trento, Italy*

ARTICLE INFO

Article history:
Received 19 September 2014
Received in revised form 13 October 2015
Accepted 14 October 2015
Available online xxxx

Keywords:
Contract-oriented computing
Verification
Rewriting logic
Session types

ABSTRACT

We address the problem of modelling and verifying contract-oriented systems, wherein distributed agents may advertise and stipulate contracts, but — differently from most other approaches to distributed agents — are not assumed to always respect them. A key issue is that the *honesty* property, which characterises those agents which respect their contracts in *all* possible execution contexts, is undecidable in general. The main contribution of this paper is a sound verification technique for honesty, targeted at agents modelled in a value-passing version of the calculus CO_2. To do that, we safely over-approximate the honesty property by abstracting from the actual values and from the contexts a process may be engaged with. Then, we develop a model-checking technique for this abstraction, we describe its implementation in Maude, and we discuss some experiments with it.

1. Introduction

Contract-oriented computing [1] is a design paradigm for distributed systems wherein the interaction between services is disciplined at run-time through *contracts*. A contract specifies an abstraction of the intended behaviour of a service, both from the point of view of what it offers to the other services, and of what it requires in exchange. Services *advertise* contracts when they want to offer (or sell) some features to clients over the network, or when they want to delegate the implementation of some features to some other services. New *sessions* are established between services whose advertised contracts are *compliant*; such contracts are then used to monitor their interaction in the sessions. When a service diverges from its contract, it can be sanctioned by the runtime monitor (e.g., by decreasing the service reputation, as in [2]).

For instance, consider an online store that wants to allow clients to order items, and wants to delegate to a bank the activity of checking payments. Both these behaviours (ordering items, checking payments) can be formalised as contracts (see e.g. Example 3.9 later on). If other services advertise contracts which are compliant with those of the store (e.g., a client advertises its interest in ordering one of the available items), then the store can establish new sessions with such services.

When services behave in the "right way" for all their advertised contracts, they are called *honest*. Instead, when services are *not* honest, they do *not* always respect the contracts they advertise, at least in some execution context. This may happen either unintentionally (because of errors in the service specification or in its implementation), or even because of malicious behaviour. Since discrepancies between the advertised and the actual behaviour can be sanctioned, a new kind of attacks becomes possible: if a service does not behave as promised, an attacker can induce it to a situation where the service is

☆ Work partially supported by Aut. Region of Sardinia under grants L.R.7/2007 CRP-17285 (TRICS), P.I.A. 2010 Project "Social Glue", by MIUR PRIN 2010-11 project "Security Horizons", and by EU COST Action IC1201 (BETTY).

* Corresponding author at: Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, via Ospedale 72, 09124 Cagliari, Italy.
E-mail address: bart@unica.it (M. Bartoletti).

http://dx.doi.org/10.1016/j.jlamp.2015.10.005
2352-2208/

sanctioned, while the attacker is not. A crucial problem is then how to ensure that a service will *never* result responsible of a contract violation, before deploying it in an unknown (and possibly adversarial) environment.

Contract-oriented computing in CO_2. The distributed, contract-oriented systems outlined in the previous paragraphs can be formally modelled and studied in CO_2, a core process calculus for contract-oriented computing [1,3]. CO_2 is not tied to a specific language or semantics for contracts. This flexibility allows to adopt one of the many different contract formalisms available in literature: these include behavioural types [4–7], Petri nets [8–10], multi-player games [11,12], logics [1,3], *etc.* Among behavioural types, *session types* [13,5] have been devoted a lot of attention in the last few years, both at the foundational level [6,14–23] and at the application level [24–28]. In their simplest incarnation, session types are terms of a process algebra featuring a *selection* construct (i.e., an internal choice among a set of branches), a *branching* construct (i.e., an external choice offered to the environment), and recursion.

In the present work, we adopt CO_2 with binary session types as our contract model of choice. Therefore, once formalised in CO_2, a service will be represented as an agent $\mathsf{A}[P]$ that can offer (or require) some behaviour by advertising it in the form of a session type c. In order to establish a session, another compliant session type needs to be advertised by another agent: intuitively, compliance is based on the standard notions of duality and subtyping [29–31], and ensures that, when run in parallel, the two session types enjoy progress. Thus, when an agent $\mathsf{B}[Q]$ advertises a session type d which is compliant with c, a new session s between $\mathsf{A}[P]$ and $\mathsf{B}[Q]$ is created. Then, $\mathsf{A}[P]$ and $\mathsf{B}[Q]$ can start interacting through s, by performing the actions prescribed by c and d, respectively — or even by choosing not to do so.

The problem of verifying honesty, even with this simplistic contract model, and in the most basic version of CO_2, is not trivial: the honesty of an agent turns out to be undecidable (the proof in [32] exploits the fact that the value-free fragment of CO_2 is Turing-powerful). Some preliminary research on the static verification of honesty uses a *type system*: in [33], it is shown that type safety guarantees honesty, but no type inference algorithm is provided; moreover, only a simple version of CO_2 (e.g., without value passing) is addressed. Effective techniques to safely *approximate* honesty of contract-oriented services are therefore in order.

Contributions. In this paper we devise and implement a sound verification technique for honesty in an extended version of CO_2, featuring expressions, value-passing, and conditionals. The main technical insight is an abstract semantics of CO_2 which preserves the transitions of an agent $\mathsf{A}[P]$, while abstracting from values and from the context wherein $\mathsf{A}[P]$ is run. Building upon this abstract semantics, we then devise an abstract notion of honesty (α-*honesty*, Definition 4.10), which approximates the execution context. The main technical result is Theorem 4.12, which states that our approximation is correct (i.e., α-honesty implies honesty), and that — under certain hypotheses on the syntax of processes — it is also *complete* (i.e., honesty implies α-honesty). We then propose a model-checking approach for verifying α-honesty, and we provide an implementation in Maude. A relevant fact about our theoretical work is that, although in this paper we have focused on binary session types, our verification technique appears to be directly reusable to deal with different contract models, e.g. all models satisfying Theorem 4.5. We have validated our technique through a set of case studies; quite notably, our implementation has allowed us to determine the dishonesty of a supposedly-honest CO_2 process appeared in [32] (see Section 5.2). Throughout the paper we shall use a running example (a simple online store), to clarify the different notions as they are introduced.

Structure of the paper. We start in Section 2 by introducing a model for contracts based on binary session types [5]. We define and implement in Maude two crucial primes on these contracts, i.e. *compliance* and *culpability testing*, and we study some relevant properties of them. In Section 3 we present a value-passing version of the calculus CO_2, and we formalise the honesty property. The main technical results follow in Section 4, where we deal with the problem of checking honesty, and in Section 5, where we carry some experiments and benchmarks. The most relevant parts of the Maude implementation are discussed throughout the text.

In Section 6 we discuss related work in the area, and finally in Section 7 we draw some conclusions. Appendices A and B contain the proofs of all our statements. The code of our verification tool and that of all the experiments is available online [34].

2. Session types as contracts

Among the various formalisms for contracts appeared in the literature, in this paper we use binary session types [5]. These are terms of a process algebra featuring internal/external choice, and recursion. Hereafter, the term *contract* will always be used as a shorthand for binary session type. Compliance between contracts (Definition 2.3) ensures their progress, until a successful state is reached. We show that compliance can be decided by model-checking finite-state systems (Lemma 2.6), and we provide an implementation in Maude. We prove that in each non-final state of a contract there is exactly one participant who is *culpable*, i.e., expected to make the next move (Theorem 2.9). Furthermore, a participant can always recover from culpability in a bounded number of steps (Theorem 2.10).

Please cite this article in press as: M. Bartoletti et al., Verifiable abstractions for contract-oriented systems, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.10.005