Modelling and verifying contract-oriented systems in Maude

Massimo Bartoletti Maurizio Murgia Alceste Scalas

Università degli Studi di Cagliari, Italy

Roberto Zunino

Università di Trento, Italy

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 behave "honestly". We describe an executable specification in Maude of the semantics of CO_2, a calculus for contract-oriented systems [5]. The *honesty* property [4] characterises those agents which always respect their contracts, in *all* possible execution contexts. Since there is an infinite number of such contexts, honesty cannot be directly verified by model-checking the state space of an agent (indeed, honesty is an undecidable property in general [4]). The main contribution of this paper is a sound verification technique for honesty. To do that, we safely over-approximate the honesty property by abstracting from the actual contexts a process may be engaged with. Then, we develop a model-checking technique for this abstraction, we describe an implementation in Maude, and we discuss some experiments with it.

1 Introduction

Contract-oriented computing is a software design paradigm where the interaction between clients and services is disciplined through contracts [5, 3]. Contract-oriented services start their life-cycle by advertising contracts which specify their required and offered behaviour. When compliant contracts are found, a session is created among the respective services, which may then start interacting to fulfil their contracts. Differently from other design paradigms (e.g. those based on the session types discipline [9]), services are not assumed to be *honest*, in that they might not respect the promises made [4]. This may happen either unintentionally (because of errors in the service specification), or because of malicious behaviour.

Dishonest behaviour is assumed to be automatically detected and sanctioned by the service infrastructure. This gives rise to a new kind of attacks, that exploit possible discrepancies between the promised and the actual behaviour. If a service does not behave as promised, an attacker can induce it to a situation where the service is sanctioned, while the attacker is reckoned honest. A crucial problem is then how to avoid that a service results definitively culpable of a contract violation, despite of the honest intentions of its developer.

In this paper we present an executable specification in Maude [8] of CO_2, a calculus for contract-oriented computing [3]. Furthermore, we devise and implement a sound verification technique for honesty. We start in § 2 by introducing a new model for contracts. Borrowing from other approaches to behavioural contracts [7, 4], ours are bilateral contracts featuring internal/external choices, and recursion. We define and implement in Maude two crucial primives on contracts, i.e. *compliance* and *culpability testing*, and we study some relevant properties.

In § 3 we present CO_2 (instantiated with the contracts above), and an executable specification of its semantics in Maude. In § 4 we formalise a weak notion of honesty, i.e. when a process P is honest *in a given context*, and we implement and experiment with it through the Maude model checker.

The main technical results follow in § 5, where we deal with the problem of checking honesty in *all* possible contexts. To do that, we start by defining an abstract semantics of CO_2, which preserves the transitions of a participant $\mathsf{A}[P]$, while abstracting those of the context wherein $\mathsf{A}[P]$ is run. Building upon the abstract semantics, we then devise an abstract notion of honesty (*α-honesty*, Def. 5.6), which neglects the execution context. Theorem 5.7 states that α-honesty correctly approximates honesty, and that — under certain hypotheses — it is also complete. We then propose a verification technique for α-honesty, and we provide an implementation in Maude. Some experiments have then been carried out; quite notably, our tool has allowed us to determine the dishonesty of a supposedly-honest CO_2 process appeared in [4] (see Ex. 5.12).

2 Modelling contracts

We model contracts as processes in a simple algebra, with internal/external choice and recursion. Compliance between contracts ensures progress, until a successful state is reached. We prove that our model enjoys some relevant properties. First, 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 always recovers from culpability in at most two steps (Theorem 2.10).

Syntax. We assume a finite set of *participant names* (ranged over by $\mathsf{A}, \mathsf{B}, \ldots$) and a denumerable set of *atoms* (ranged over by $\mathsf{a}, \mathsf{b}, \ldots$). We postulate an involution $\mathrm{co}(\mathsf{a})$, also written as $\bar{\mathsf{a}}$, extended to sets of atoms in the natural way. Def. 2.1 introduces the syntax of contracts. We distinguish between (*unilateral*) contracts c, which model the promised behaviour of a single participant, and *bilateral* contracts γ, which combine the contracts advertised by two participants.

Definition 2.1. Unilateral contracts *are defined by the following grammar:*

$$c, d \;::=\; \bigoplus_{i\in\mathcal{I}} \mathsf{a}_i \,;\, c_i \;\;\Big|\;\; \sum_{i\in\mathcal{I}} \mathsf{a}_i \,.\, c_i \;\;\Big|\;\; \textit{ready}\ \mathsf{a}.c \;\;\Big|\;\; \mathrm{rec}\ X.\, c \;\;\Big|\;\; X$$

where (i) the index set $\mathcal{I}$ is finite; (ii) the "ready" prefix may appear at the top-level, only; (iii) recursion is guarded.

Bilateral contracts γ *are terms of the form* A *says* $c \mid \mathsf{B}$ *says* d*, where* $\mathsf{A} \neq \mathsf{B}$ *and at most one occurrence of "ready" is present. The order of unilateral contracts in* γ *is immaterial, i.e.* A *says* $c \mid \mathsf{B}$ *says* $d \;\equiv\; \mathsf{B}$ *says* $d \mid \mathsf{A}$ *says* c.

An internal sum $\bigoplus_{i\in\mathcal{I}} \mathsf{a}_i \,;\, c_i$ allows to choose one of the branches $\mathsf{a}_i \,;\, c_i$, to perform the action a_i, and then to behave according to c_i. Dually, an external sum $\sum_{i\in\mathcal{I}} \mathsf{a}_i \,.\, c_i$ allows to wait for the other participant to choose one of the branches $\mathsf{a}_i \,.\, c_i$, then to perform the corresponding a_i and behave according to c_i. Separators ; and . allow for distinguishing singleton internal sums $\mathsf{a} \,;\, c$ from singleton external sums $\mathsf{a} \,.\, c$. Empty internal/external sums are denoted with 0. We will only consider contracts without free occurrences of recursion variables X.

Example 2.2. *An online store* A *has the following contract: buyers can iteratively add items to the shopping cart (*`addToCart`*); when at least one item has been added, the client can either* `cancel`

$$\mathsf{A}\ \mathit{says}\ (\mathsf{a}\,;\,c \oplus c') \mid \mathsf{B}\ \mathit{says}\ (\bar{\mathsf{a}}\,.\,d + d') \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}}\!\!\!\rightarrow \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ \mathit{ready}\ \bar{\mathsf{a}}.d \qquad \text{[IntExt]}$$

$$\mathsf{A}\ \mathit{says}\ \mathit{ready}\ \mathsf{a}.\ c \mid \mathsf{B}\ \mathit{says}\ d \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}}\!\!\!\rightarrow \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \qquad \text{[Rdy]}$$

Figure 1: Semantics of contracts (symmetric rules for B actions omitted)

the order or `pay`*; then, the store can accept (*`ok`*) or decline (*`no`*) the payment. Such a contract may be expressed as* c_A *below:*

$$\begin{aligned} c_{pay} &= \mathsf{pay}\,.\,\big(\overline{\mathsf{ok}}\,;\,0 \oplus \overline{\mathsf{no}}\,;\,0\big) \\ c_\mathsf{A} &= \mathsf{addToCart}\,.\,(\mathrm{rec}\ Z.\ \mathsf{addToCart}\,.\,Z\ +\ c_{pay}\ +\ \mathsf{cancel}\,.\,0) \end{aligned}$$

Instead, a buyer contract could be expressed as:

$$c_\mathsf{B} \quad = \quad \mathrm{rec}\ Z.\ \big(\overline{\mathsf{addToCart}}\,;\,Z\ \oplus\ \overline{\mathsf{pay}}\,;\,(\mathsf{ok}\,.\,0\ +\ \mathsf{no}\,.\,0)\big)$$

The Maude specification of the syntax of contracts is defined as follows:

```
sorts Atom UniContract Participant AdvContract BiContract
      IGuarded EGuarded IChoice EChoice Var Id RdyContract .
subsort Id < IGuarded < IChoice < UniContract < RdyContract .
subsort Id < EGuarded < EChoice < UniContract < RdyContract .
subsort Var < UniContract .
```

The sorts `IGuarded` and `EGuarded` represent singleton internal/external sums, respectively, while `IChoice` and `EChoice` are for arbitrary internal/external sums. `Id` represents empty sums, and it is a subsort of internal and external sums (either singleton or not). `RdyContract` if for contracts which may have a top-level *ready* , while `AdvContract` is a unilateral contract advertised by some participant.

```
op -_ : Atom -> Atom [ctor] .
eq - - a:Atom = a:Atom .
op 0 : -> Id [ctor] .
op _._ : Atom UniContract -> EGuarded [frozen ctor] .
op _;_ : Atom UniContract -> IGuarded [frozen ctor] .
op _+_ : EChoice EChoice -> EChoice [frozen comm assoc id: 0 ctor] .
op _(+)_ : IChoice IChoice -> IChoice [frozen comm assoc id: 0 ctor] .
op ready _._ : Atom UniContract -> RdyContract [frozen ctor] .
op rec _._ : Var IChoice -> UniContract [frozen ctor] .
op rec _._ : Var EChoice -> UniContract [frozen ctor] .
op _ says _  : Participant RdyContract -> AdvContract [ctor] .
op _ | _ : AdvContract AdvContract -> BiContract [comm ctor] .
```

The operator `-` models the involution on atoms, with `eq - - a:Atom = a:Atom`. The other operators are rather standard, and they guarantee that each `UniContract` respects the syntactic constraints imposed by Def. 2.1.

Semantics. The evolution of bilateral contracts is modelled by $\xrightarrow{\mu}\!\!\!\rightarrow$, the smallest relation closed under the rules in Fig. 1 and under $\equiv$. The congruence $\equiv$ is the least relation including α-conversion of recursion variables, and satisfying $\mathrm{rec}\ X.\ c \equiv c\{^{\mathrm{rec}\ X.\ c}/_X\}$ and $\bigoplus_{i\in\emptyset} \mathsf{a}_i\,;\,c_i \equiv \sum_{i\in\emptyset} \mathsf{a}_i\,.\,c_i$. The label $\mu = \mathsf{A}\ \mathit{says}\ \mathsf{a}$ models A performing action a. Hereafter, we shall consider contracts up-to $\equiv$.

In rule [IntExt], participant A selects the branch a in an internal sum, and B is then forced to commit to the corresponding branch $\bar{\mathsf{a}}$ in his external sum. This is done by marking that branch with *ready* $\bar{\mathsf{a}}$, while discarding all the other branches; B will then perform his action in the subsequent step, by rule [Rdy].

In Maude, the semantics of contracts is an almost literal translation of that in Fig. 1 (except that labels are moved to configurations). The one-step transition relation is defined as follows:

```
crl [IntExt]: A says a ; c (+) c' | B says b . d + d'
 => {A says a} A says c | B says ready b . d              if a = - b .

rl [Rdy]: A says ready a.c | B says d => {A says a} A says c | B says d .
```

Compliance. Two contracts are *compliant* if, whenever a participant A wants to choose a branch in an internal sum, then participant B always offers A the opportunity to do it. To formalise compliance, we first define a partial function rdy from bilateral contracts to sets of atoms. Intuitively, if the unilateral contracts in γ do not agree on the first step, then $\mathrm{rdy}(\gamma)$ is undefined (i.e. equal to $\bot$). Otherwise, $\mathrm{rdy}(\gamma)$ contains the atoms which could be fired in the first step.

Definition 2.3 (Compliance)**.** *Let the partial function* rdy *be defined as:*

$$\mathrm{rdy}\Big(\mathsf{A}\ \mathit{says} \bigoplus_{i\in\mathcal{I}} \mathsf{a}_i\,;\,c_i \mid \mathsf{B}\ \mathit{says} \sum_{j\in\mathcal{J}} \mathsf{b}_j\,.\,c_j\Big) \;=\; \{\mathsf{a}_i\}_{i\in\mathcal{I}} \qquad \begin{array}{cl} \mathit{if} & \{\mathsf{a}_i\}_{i\in\mathcal{I}} \subseteq \{\bar{\mathsf{b}}_j\}_{j\in\mathcal{J}} \\ \mathit{and} & (\mathcal{I}=\emptyset \implies \mathcal{J}=\emptyset) \end{array}$$

$$\mathrm{rdy}(\mathsf{A}\ \mathit{says\ ready}\ \mathsf{a}.c \mid \mathsf{B}\ \mathit{says}\ d) \;=\; \{\mathsf{a}\}$$

Then, the compliance relation $\bowtie$ *between unilateral contracts is the largest relation such that, whenever* $c \bowtie d$*:*

(1) $\mathrm{rdy}(\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d) \neq \bot$

(2) $\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \xrightarrow{\mu}\!\!\!\rightarrow \mathsf{A}\ \mathit{says}\ c' \mid \mathsf{B}\ \mathit{says}\ d' \implies c' \bowtie d'$

Example 2.4. *Let* $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$*, where* $c = \mathsf{a}\,;c_1 \oplus \mathsf{b}\,;c_2$ *and* $d = \bar{\mathsf{a}}\,.\,d_1 + \bar{\mathsf{c}}\,.\,d_2$*. If the participant* A *internally chooses to perform* a*, then* γ *will take a transition to* $\mathsf{A}\ \mathit{says}\ c_1 \mid \mathsf{B}\ \mathit{says\ ready}\ \bar{\mathsf{a}}.d_1$*. Suppose instead that* A *chooses to perform* b*, which is not offered by* B *in his external choice. In this case,* $\gamma \overset{\mathsf{A}\ \mathit{says}\ \mathsf{b}}{\not\twoheadrightarrow}$*. We have that* $\mathrm{rdy}(\gamma) = \bot$*, which does not respect item* (1) *of Def. 2.3. Therefore, c and d are* not *compliant.*

We say that a contract is *proper* if the prefixes of each summation are pairwise distinct. The next lemma states that each proper contract has a compliant one.

Lemma 2.5. *For all proper contracts c, there exists d such that* $c \bowtie d$*.*

Def. 2.3 cannot be directly exploited as an algorithm for checking compliance. Lemma 2.6 gives an alternative, model-checkable characterisation of $\bowtie$.

Lemma 2.6. *For all bilateral contracts* $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$*:*

$$c \bowtie d \iff (\forall \gamma'.\ \gamma \twoheadrightarrow^{*} \gamma' \implies \mathrm{rdy}(\gamma') \neq \bot)$$

In Maude, the compliance relation is defined as suggested by Lemma 2.6. The predicate `isBottom` is true for a contract γ whenever $\mathrm{rdy}(\gamma) = \bot$. The operator `<>` used below allows for the transitive closure of the transition relation. The relation `c |X| d` is implemented by verifying that the contract $\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$ satisfies the LTL formula $\Box\neg$ `isBottom`. This is done through the Maude model checker.

```
eq <{l} g> |= isBottom = is rdy(g) eq bottom .
op _|X|_ : UniContract UniContract -> Bool .
eq c |X| d = modelCheck(<A says c | B says d>, [] ~isBottom) == true .
```

Example 2.7. *Recall the store contract c_{A} in Ex. 2.2. Its Maude version is:*

```
op Z : -> Var .
ops addToCart pay ok no cancel : -> Atom .
ops CA CPay CB : -> UniContract .
eq CPay = pay . (- ok ; 0 (+) - no ; 0) .
eq CA = addToCart . (rec Z . addToCart . Z + CPay + cancel . 0) .
```

Instead, the Maude implementation of the buyer contract c_{B} in Ex. 2.2 is:

```
eq CB = rec Z . ( - addToCart ; Z (+) - pay ; (ok . 0 + no . 0)) .
```

We can verify with Maude that `CA` *and* `CB` *are* not *compliant:*

```
red CA |X| CB .
result Bool: false
```

The problem is that `CB` *may choose to* `pay` *even when the cart is empty. We can easily fix the buyer contract as follows, and then obtain compliance:*

```
red CA |X| (- addToCart ; CB) .
result Bool: true
```

Culpability We now tackle the problem of determining who is expected to make the next step for the fulfilment of a bilateral contract. We call a participant A *culpable* in γ if she is expected to perform some actions so to make γ progress.

Definition 2.8. *A participant A is culpable in γ ($\mathsf{A}\,\ddot{\frown}\,\gamma$ in symbols) iff $\gamma \xrightarrow{\mathsf{A}\ says\ \mathsf{a}}\!\!\!\!\rightarrow$ for some a. When A is* not *culpable in γ we write $\mathsf{A}\,\ddot{\smile}\,\gamma$.*

Theorem 2.9 below establishes that, when starting with compliant contracts, exactly one participant is culpable in a bilateral contract. The only exception is A *says* $0 \mid \mathsf{B}$ *says* 0, which represents a successfully terminated interaction, where nobody is culpable.

Theorem 2.9. *Let $\gamma = \mathsf{A}$ says $c \mid \mathsf{B}$ says d, with $c \bowtie d$. If $\gamma \twoheadrightarrow^* \gamma'$, then either $\gamma' = \mathsf{A}$ says $0 \mid \mathsf{B}$ says 0, or there exists a unique culpable in γ'.*

The following theorem states that a participant is always able to recover from culpability by performing some of her duties. This requires at most two steps.

Theorem 2.10 (Contractual exculpation)**.** *Let $\gamma = \mathsf{A}$ says $c \mid \mathsf{B}$ says d. For all γ' such that $\gamma \twoheadrightarrow^* \gamma'$, we have that:*

(1) $\gamma' \not\twoheadrightarrow \implies \mathsf{A}\,\ddot{\smile}\,\gamma'$ and $\mathsf{B}\,\ddot{\smile}\,\gamma'$

(2) $$\mathsf{A}\,\ddot{\frown}\,\gamma' \implies \forall \gamma''.\gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathsf{A}\,\ddot{\smile}\,\gamma'', \text{ or} \\ \forall \gamma'''.\gamma'' \twoheadrightarrow \gamma''' \implies \mathsf{A}\,\ddot{\smile}\,\gamma''' \end{cases}$$

Item (1) of Theorem 2.10 says that, in a stuck contract, no participant is culpable. Item (2) says that if A is culpable, then she can always exculpate herself in *at most* two steps, i.e.: one step if A has an internal choice, or a *ready* followed by an external choice; two steps if A has a *ready* followed by an internal choice.

We specify culpability in Maude as follows. The formula `{l} g |= --A-->>` is true whenever `g` has been reached by some transitions of `A`. The participant `A` is culpable in `g`, written `A :C g`, if `g` satisfies the LTL formula `O --A-->>` (where `O` is the "next" operator of LTL). This is verified through the Maude model checker.

commutative monoidal laws for | on processes and systems

$$\mathsf{A}[(v)P] \equiv (v)\,\mathsf{A}[P] \qquad Z \mid (u)Z' \equiv (u)(Z \mid Z') \;\text{ if } u \not\in \text{fv}(Z) \cup \text{fn}(Z)$$

$$(u)(v)Z \equiv (v)(u)Z \qquad (u)Z \equiv Z \;\text{ if } u \not\in \text{fv}(Z) \cup \text{fn}(Z) \qquad \{\downarrow_s c\}_{\mathsf{A}} \equiv \mathbf{0}$$

Figure 2: Structural equivalence for CO_2 (Z, Z' range over systems or processes).

```
op --_->> : Participant -> Prop .
eq {A says a} g |= -- A ->> = true .
eq {l} g |= -- A ->> = false [owise] .
op _ :C _ : Participant BiContract -> Bool .
eq A :C g = modelCheck(g, O -- A ->>) == true .
```

3 Modelling contracting processes

We model agents and systems through the process calculus CO_2 [2], which we instantiate with the contracts introduced in § 2. The primitives of CO_2 allow agents to advertise contracts, to open sessions between agents with compliant contracts, to execute them by performing some actions, and to query contracts.

Syntax. Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of *session variables* (ranged over by $x, y, \ldots$) and *session names* (ranged over by $s, t, \ldots$). Let $u, v, \ldots$ range over $\mathcal{V} \cup \mathcal{N}$, and $\vec{u}, \vec{v}$ range over $2^{\mathcal{V} \cup \mathcal{N}}$.

Definition 3.1. *The syntax of CO_2 is given as follows:*

$$\begin{array}{rccccccccccccc}
\textit{Systems} & S & ::= & \mathbf{0} & \mid & \mathsf{A}[P] & \mid & s[\gamma] & \mid & S \mid S & \mid & (u)S & \mid & \{\downarrow_u c\}_{\mathsf{A}} \\
\textit{Processes} & P & ::= & \sum_i \pi_i.P_i & \mid & P \mid P & \mid & (u)P & \mid & X(\vec{u}) & & & & \\
\textit{Prefixes} & \pi & ::= & \tau & \mid & \mathsf{tell} \downarrow_u c & \mid & \mathsf{do}_u\,\mathsf{a} & \mid & \mathsf{ask}_u \phi & & & &
\end{array}$$

Systems are the parallel composition of *participants* $\mathsf{A}[P]$, *delimited systems* $(u)S$, *sessions* $s[\gamma]$ and *latent contracts* $\{\downarrow_u c\}_{\mathsf{A}}$. A latent contract $\{\downarrow_x c\}_{\mathsf{A}}$ represents a contract c (advertised by A) which has not been stipulated yet; upon stipulation, the variable x will be instantiated to a fresh session name. We assume that, in a system of the form $(\vec{u})(\mathsf{A}[P] \mid \mathsf{B}[Q]) \mid \cdots)$, $\mathsf{A} \neq \mathsf{B}$. We denote with K a special participant name (playing the role of contract broker) such that, in each system $(\vec{u})(\mathsf{A}[P] \mid \cdots)$, $\mathsf{A} \neq \mathsf{K}$. We allow for prefix-guarded finite sums of processes, and write $\pi_1.P_1 + \pi_2.P_2$ for $\sum_{i \in \{1,2\}} \pi_i.P_i$, and $\mathbf{0}$ for $\sum_{\emptyset} P$. Recursion is allowed only for processes; we stipulate that each process identifier X has a unique defining equation $X(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$ such that $\text{fv}(P) \subseteq \{x_1, \ldots, x_j\} \subseteq \mathcal{V}$, and each occurrence of process identifiers in P is prefix-guarded. We will sometimes omit the arguments of $X(\vec{u})$ when they are clear from the context.

Prefixes include silent action τ, contract advertisement $\mathsf{tell} \downarrow_u c$, action execution $\mathsf{do}_u\,\mathsf{a}$, and contract query $\mathsf{ask}_u\,\phi$ (where ϕ is an LTL formula on γ). In each prefix $\pi \neq \tau$, u refers to the target session involved in the execution of π.

In Maude, we translate the syntax of CO_2 almost literally. Here we just show the sorts used; see § C for the full details.

```
sorts System Process Prefix SessionName SessionVariable SessionIde
      GuardProc Sum IdeVec ProcIde ParamList .
subsort SessionName < SessionIde < IdeVec .
subsort Qid < SessionVariable < SessionIde < IdeVec .
subsort GuardProc < Sum < Process .
subsort SessionIde < ParamList .
```

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}\colon \tau} \mathsf{A}[P \mid Q] \qquad \text{[Tau]}$$

$$\mathsf{A}[\mathsf{tell} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}\colon \mathsf{tell} \downarrow_u c} \mathsf{A}[P \mid Q] \mid \{\downarrow_u c\}_{\mathsf{A}} \qquad \text{[Tell]}$$

$$\frac{c \bowtie d \qquad \gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \qquad \sigma = \{{}^{s}/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \mid \{\downarrow_x c\}_{\mathsf{A}} \mid \{\downarrow_y d\}_{\mathsf{B}}) \xrightarrow{\mathsf{K}\colon \mathsf{fuse}} (s)(S\sigma \mid s[\gamma])} \qquad \text{[Fuse]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}}\!\!\!\!\rightarrow \gamma'}{\mathsf{A}[\mathsf{do}_s\, \mathsf{a}.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}\colon \mathsf{do}_s\, \mathsf{a}} \mathsf{A}[P \mid Q] \mid s[\gamma']} \qquad \text{[Do]}$$

$$\frac{\gamma \vdash \phi}{\mathsf{A}[\mathsf{ask}_s\, \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}\colon \mathsf{ask}_s\, \phi} \mathsf{A}[P \mid Q] \mid s[\gamma]} \qquad \text{[Ask]}$$

$$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{\vec{v}/\vec{u}\} \mid Q] \mid S \xrightarrow{\mu} S'}{\mathsf{A}[X(\vec{v}) \mid Q] \mid S \xrightarrow{\mu} S'} \text{[Def]} \qquad \frac{S \xrightarrow{\mu} S'}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \text{[Par]}$$

$$\frac{S \xrightarrow{\mathsf{A}\colon \pi} S'}{(u)S \xrightarrow{\mathsf{A}\colon \mathrm{del}_u(\pi)} (u)S'} \text{[Del]} \qquad \text{where } \mathrm{del}_u(\pi) = \begin{cases} \tau & \text{if } u \in \mathrm{fnv}(\pi) \\ \pi & \text{otherwise} \end{cases}$$

Figure 3: Reduction semantics of CO$_2$.

The sort `SessionIde` is a super sort of both `SessionVariable` and `SessionName`. Session variables can be of sort `Qid`; session names can not. Sort `IdeVec` models sets of `SessionIde` (used as syntactic sugar for delimitations), while `ParamList` models vectors of `SessionIde` (used for parameters of defining equations).

Semantics. The CO$_2$ semantics is formalised by the relation $\xrightarrow{\mu}$ in Fig. 3, where

$$\mu \in \{\mathsf{A}\colon \pi \mid \mathsf{A} \neq \mathsf{K}\} \cup \{\mathsf{K}\colon \mathsf{fuse}\}$$

We will consider processes and systems up-to the congruence relation $\equiv$ in Fig. 2. The axioms for $\equiv$ are fairly standard — except the last one: it collects garbage terms possibly arising from variable substitutions.

Rule [Tau] just fires a τ prefix. Rule [Tell] advertises a latent contract $\{\downarrow_x c\}_{\mathsf{A}}$. Rule [Fuse] finds *agreements* among the latent contracts: it happens when there exist $\{\downarrow_x c\}_{\mathsf{A}}$ and $\{\downarrow_y d\}_{\mathsf{B}}$ such that $\mathsf{A} \neq \mathsf{B}$ and $c \bowtie d$. Once the agreement is reached, a fresh session containing $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$ is created. Rule [Do] allows a participant A to perform an action in the session s containing γ (which, accordingly, evolves to γ'). Rule [Ask] allows A to proceed only if the contract γ at session s satisfies the property ϕ. The last three rules are mostly standard. In rule [Del] the label π fired in the premise becomes τ in the consequence, when π contains the delimited name/variable. This transformation is defined by the function $\mathrm{del}_u(\pi)$, where the set $\mathrm{fnv}(\pi)$ contains the free names/variables in π. For instance, $(x)\, \mathsf{A}[\mathsf{tell} \downarrow_x c.P] \xrightarrow{\mathsf{A}\colon \tau} (x)\, (\mathsf{A}[P] \mid \{\downarrow_x c\}_{\mathsf{A}})$. Here, it would make little sense to have the label $\mathsf{A}\colon \mathsf{tell} \downarrow_x c$, as x (being delimited) may be α-converted.

Implementing in Maude the semantics of CO$_2$ is almost straightforward [18]; here we show only the main rules (see § C for the others). Rule `[Do]` uses the transition relation `=>` on bilateral contracts. Rule `[Ask]` exploits the Maude model checker to verify if the bilateral contract `g` satisfies the LTL formula `phi`. Rule `[Fuse]` uses the operator `|X|` to check compliance between the contracts c and d, then creates the session s[A says c | B says d] (with s fresh), and finally applies the substitution {s / x}{s / y} (delimitations are dealt with as in Fig. 3).

```
crl [Do] : A[do s a . P + P' | Q] | s[g] => {A : do s a} (A[P | Q] | s[g'])
           if g => {A says a} g' .

crl [Ask] : A[ask s phi . P + P' | Q] | s[g] => {A : ask s phi} A[P | Q]
            if g |- phi .

crl [Fuse] : (uVec , vVec) ({x c}A | {y d}B | S) => {K : fuse}
             (s , vVec) (s[A says c | B says d] | S{s / x}{s / y})
             if uVec == (x , y) / c |X| d / s := fresh(0 , S) .
```

4 Honesty

A remarkable feature of CO$_2$ is that it allows for writing *dishonest* agents which do not keep their promises. Intuitively, a participant is honest if she always fulfils her contractual obligations, in all possible contexts. Below we formalise the notion of honesty, by slightly adapting the one appeared in [2]. Then, we show how we verify in Maude a weaker notion, i.e. honesty *in a given context.*

We start by defining the set $O^{\mathsf{A}}_s(S)$ of *obligations* of A at s in S. Whenever A is culpable at some session s, she has to fire one of the actions in $O^{\mathsf{A}}_s(S)$.

Definition 4.1. *We define the set of atoms* $O^{\mathsf{A}}_s(S)$ *as:*

$$O^{\mathsf{A}}_s(S) \;=\; \Big\{ \mathsf{a} \mid \exists \gamma, S' \,.\, S \equiv s[\gamma] \mid S' \text{ and } \gamma \xrightarrow{\mathsf{A}\ says\ \mathsf{a}}\!\!\!\!\rightarrow \Big\}$$

We say that A *is* culpable at s in S *iff* $O^{\mathsf{A}}_s(S) \neq \emptyset$.

The set of atoms $\mathrm{RD}^{\mathsf{A}}_s(S)$ ("Ready Do") defined below comprises all the actions that A can perform at s in one computation step within S (note that, by rule [Del], if s is a bound name then $\mathrm{RD}^{\mathsf{A}}_s(S) = \emptyset$). The set $\mathrm{WRD}^{A}_s(S)$ ("Weak Ready Do") contains all the actions that A may possibly perform at s after a finite sequence of transitions of A not involving any do at s.

Definition 4.2. *For all* S, A *and* s, *we define the sets of atoms:*

$$\mathrm{RD}^{\mathsf{A}}_s(S) = \Big\{ \mathsf{a} \mid \exists S' \,.\, S \xrightarrow{\mathsf{A}:\ \mathsf{do}_s\, \mathsf{a}} S' \Big\}$$

$$\mathrm{WRD}^{\mathsf{A}}_s(S) = \Big\{ \mathsf{a} \mid \exists S' \,.\, S \xrightarrow{\mathsf{A}:\ \neq \mathsf{do}_s}{}^{*} S' \;\wedge\; \mathsf{a} \in \mathrm{RD}^{\mathsf{A}}_s(S') \Big\}$$

where we write $S \xrightarrow{\mathsf{A}:\ \neq \mathsf{do}_s} S'$ *if* $\exists \pi.\ S \xrightarrow{\mathsf{A}:\ \pi} S' \;\wedge\; \forall \mathsf{a}.\ \pi \neq \mathsf{do}_s\, \mathsf{a}$.

A participant is *ready* in a systemif she can fulfil some of her obligations. To check if A is ready in S, we consider all the sessions s in S involving A. For each of them, we check that some obligations of A at s are exposed after some steps of A *not* preceded by other do_s of A. We can now formalise when a participant is *honest.* Roughly, $\mathsf{A}[P]$ is honest *in a given system* S when A is ready in all evolutions of $\mathsf{A}[P] \mid S$. Then, $\mathsf{A}[P]$ is honest when she is honest in *all* systems$S$.

Definition 4.3 (Honesty)**.** *We say that:*

1. S is A-free *iff it has no latent/stipulated contracts of* A, *nor processes of* A
2. A *is* ready in S *iff* $S \equiv (\vec{u})S' \;\wedge\; O^{\mathsf{A}}_s(S') \neq \emptyset \implies \mathrm{WRD}^{\mathsf{A}}_s(S') \cap O^{\mathsf{A}}_s(S') \neq \emptyset$
3. P *is* honest in S *iff* $\forall \mathsf{A} : (S \textit{ is } \mathsf{A}\textit{-free} \wedge \mathsf{A}[P] \mid S \rightarrow^{*} S') \implies \mathsf{A} \textit{ is ready in } S'$

4. P is honest *iff, for all S, P is honest in S*

We have implemented items 2 and 3 of the above definition in Maude (item 4 is dealt with in the next section). CO_2 can simulate Turing machines [4], hence reachability in CO_2 is undecidable, and consequently WRD, readiness and honesty are undecidable as well. To recover decidability, we then restrict to finite state processes: roughly, these are the processes with neither delimitations nor parallel compositions under process definitions.

In Maude we verify readiness in a session s by searching if A can reach (with her moves only), a state which allows for a $\mathsf{do}_s\ \mathsf{a}$ move, for some a.

```
op ready? : Participant SessionName System Module -> Bool .
eq ready?(A,s,S,M:Module) =  metaSearch(M:Module, upTerm(< S > A s),
   '<_>__['S1:System , upTerm(A) , upTerm(s)],
   'S1:System => '`_`_['l:SLabel,'S2:System] /\
   '_:_[upTerm(A),'do__[upTerm(s),'a:Atom]] := 'l:SLabel,
   '*, unbounded, 0 ) =/= failure .
```

We start the search from the term `< S > A s`, whose meta-representation is obtained through the `upTerm` function. The search is performed according to the `A`-solo semantics of CO_2 (see Definition 5.3), which blocks all `do` at `s`. This is done by the operator `<_>__`. Then, we look for reachable systems `S1` where `A` can fire a `do` at `s`. If the search succeeds, `ready?` returns true. Note that if `A` has no obligations at `s` in `S`, `ready?` returns false — uncoherently with Def. 4.3. To correctly check readiness, we define the function `ready` (see § C), which invokes `ready?` only when $O^{\mathtt{A}}_{\mathtt{s}}(\mathtt{S}) \neq \emptyset$.

Verifying honesty in a context is done similarly. We use `metaSearch` to check that `A` is ready in all reachable states. The operator `<_>` gives the CO_2 semantics.

```
op search-honest-ctx : Participant System Module -> ResultTriple? .
eq search-honest-ctx(A,S,M:Module) = metaSearch(M:Module, upTerm(< S >),
  '<_>['S:System], 'ready[upTerm(A), 'S:System,'S:System, upTerm(M:Module)]
  = 'false.Bool, '*, unbounded, 0) .
op honest-ctx : Participant System Module -> Result .
ceq honest-ctx (A , S , M:Module) = true
  if search-honest-ctx (A , S , M:Module) == failure .
ceq honest-ctx (A , S , M:Module) = downTerm (T:Term , < (0).System > )
  if {T:Term,Ty:Type,S:Substitution} := search-honest-ctx (A,S,M:Module) .
```

Example 4.4. *A travel agency* `A` *queries in parallel an airline ticket broker* `F` *and a hotel reservation service* `H` *in order to organise a trip for some user* `U`*. The agency first requires* `U` *to pay, and then chooses either to commit the reservation or to issue a refund (contract* `CU`*). When querying the ticket broker (contract* `CF`*), the agency first receives a quotation, and then chooses either to commit and pay the ticket, or to abort the transaction. The contract* `CH` *between* `A` *and* `H` *is similar.*

```
eq CU = pay . (commit ; 0 (+) refund ; 0) .
eq CF = ticket . ( commitF ; payF ; 0 (+) abortF ; 0) .
eq CH = hotel . ( commitH ; payH ; 0 (+) abortH ; 0) .
```

In addition to the contracts above, the agency should respect the following constraints: (a) the agency refunds `U` *only if both the transactions with* `F` *and* `H` *are aborted; (b)* `A` *pays the ticket and the hotel reservation only after it has committed the transaction with* `U`*; (c) either both the transactions with* `F` *or* `H` *are committed, or they are both aborted. A possible specification in Maude respecting the above constraints is given by the following process* `P`*:*

```
eq P = ( xu , xf , xh ) ( tell xu CU . do xu pay .
       ( (tell xf CF . PF) | (tell xh CH . PH) | PU ) ) .

eq PF = do xf ticket . (do xh commitH . 0 + do xf abortF . 0) .
eq PH = do xh hotel  . (do xf commitF . 0 + do xh abortH . 0) .

eq PU = ask xh ([] ~ payH) . do xu refund . 0 +
        t . do xu commit . (do xf payF . 0 | do xh payH . 0) .
```

The process `P` *first opens a session with* `U`*, and then advertises the contracts* `CF` *and* `CH`*, and in parallel executes* `PU`*. The process* `PF` *gets the ticket quotation, then either commits the hotel reservation, or aborts the flight reservation. Dually,* `PH` *gets the hotel quotation, then either commits the flight reservation, or aborts the hotel reservation. Note that the two choices in* `PF` *and* `PH` *ensure that constraint* (c) *above is satisfied: e.g., if* `PF` *fires the* `commitH` *(resp.* `abortF`*) prefix, the* `abortH` *(resp.* `commitF`*) branch in* `PH` *is disabled, and only* `commitF` *(resp.* `abortH`*) can be selected. The process* `PU` *checks if a refund is due to* `U`*. When the atom* `payH` *is no longer reachable in session* `xh`*, the* `ask` *passes, and the refund is issued. This guarantees constraint* (a)*. In the* τ*-branch,* `PU` *commits the transaction with* `U`*, and then proceeds to pay both* `F` *and* `H`*. This satisfies constraint* (b)*. Note that it may happen that* `PU` *chooses to* `commit` *even when* `CF` *or* `CH` *are not stipulated. Although this behaviour is conceptually wrong, it does not affect honesty. Indeed, honesty does not consider the domain-specific constraints among actions (e.g.* (a)*,* (b)*,* (c) *above), but only that the advertised contracts are respected.*

We have experimented the function `honest-ctx` *by inserting* `P` *in some contexts* `S` *where all the other participants* `U`*,* `F` *and* `H` *are honest (see §B for details). The Maude model checker has correctly determined that* `P` *is honest in* `S`*.*

```
red honest-ctx(A , S , ['TRAVEL-AGENCY-CTX]) .
rewrites: 53950741 in 38062ms cpu (38058ms real) (1417429 rewrites/second)
result Bool: true
```

Even though we conjecture that `P` *is honest (in all contexts), we anticipate here that the verification technique proposed in § 5 does not classify* `P` *as honest. This is because the analysis is (correct but) not complete in the presence of* `ask`*: indeed, the precise behaviour of an* `ask` *is lost by the analysis, because it abstracts from the contracts of the context.*

Example 4.5. *Consider the following contracts and processes:*

$$c = \mathsf{a}\,;\,0 \qquad\qquad d = \bar{\mathsf{a}}\,.\,0$$

$$P = (x)\ \mathsf{tell}\downarrow_x c.\ \mathsf{do}_x\,\mathsf{a} \qquad\qquad Q = (y)\ \mathsf{tell}\downarrow_y d.\,X \quad \textit{where } X \overset{\textit{def}}{=} \tau.X$$

We have the following (concrete) computation in the system $S = \mathsf{A}[P] \mid \mathsf{B}[Q]$*:*

$$\begin{aligned} S &\xrightarrow{\mathsf{A}:\ \tau} (x)(\mathsf{A}[\mathsf{do}_x\,\mathsf{a}] \mid \mathsf{B}[Q] \mid \{\downarrow_x c\}_{\mathsf{A}}) \\ &\xrightarrow{\mathsf{B}:\ \tau} (x,y)(\mathsf{A}[\mathsf{do}_x\,\mathsf{a}] \mid \mathsf{B}[X] \mid \{\downarrow_x c\}_{\mathsf{A}} \mid \{\downarrow_y d\}_{\mathsf{B}}) \\ &\xrightarrow{\mathsf{K}:\ \mathsf{fuse}} (s)\ (\mathsf{A}[\mathsf{do}_s\,\mathsf{a}] \mid \mathsf{B}[X] \mid s[\mathsf{A}\ \textit{says}\ c \mid \mathsf{B}\ \textit{says}\ d]) = S' \xrightarrow{\mathsf{B}:\ \tau} S' \xrightarrow{\mathsf{B}:\ \tau} \cdots \end{aligned}$$

In the above computation, an unfair *scheduler prevents* A *from making her moves, and so* A *remains persistently culpable in such computation. However,* A *is ready in* S' *(because the* $\mathsf{do}_s\,\mathsf{a}$ *is enabled), and therefore* P *is honest according to Def. 4.3. This is coherent with our intuition about honesty: an honest participant will always exculpate herself in all fair computations, but she might stay culpable in the unfair ones, because an unfair scheduler might always give precedence to the actions of the context.*

5 Model checking honesty

We now address the problem of automatically verifying honesty. As mentioned in § 1, this is a desirable goal, because it alerts system designers before they deploy services which could violate contracts at run-time (so possibly incurring in sanctions). Since honesty is undecidable in general [4], our goal is a verification technique which safely over-approximates honesty, i.e. it never classifies a process as honest when it is not. The first issue is that Def. 4.3 requires readiness to be preserved in all possible contexts, and there is an *infinite* number of such contexts. To overcome this problem, we present below an *abstract* semantics of CO_2 which preserves the honesty property, while neglecting the actual context where the process $\mathsf{A}[P]$ is executed.

The definition of the abstract semantics of CO_2 is obtained in two steps. First, we provide the projections from concrete contracts/systems to the abstract ones. Then, we define the semantics of abstract contracts and systems, and we relate the abstract semantics with the concrete one. The abstraction is always parameterised in the participant A the honesty of which is under consideration.

The abstraction $\alpha_{\mathsf{A}}(\gamma)$ of a bilateral contract $\gamma = \mathsf{A}\ \textit{says}\ c \mid \mathsf{B}\ \textit{says}\ d$ (Definition 5.1 below) is either c, or $\textit{ctx}.c$ when d has a *ready*.

Definition 5.1. *For all γ, we define the abstract contract $\alpha_{\mathsf{A}}(\gamma)$ as:*

$$\alpha_{\mathsf{A}}(\mathsf{A}\ \textit{says}\ c \mid \mathsf{B}\ \textit{says}\ d) \quad = \quad \begin{cases} c & \textit{if } d \textit{ is ready-free} \\ \textit{ctx}\ \mathsf{a}.c & \textit{if } d = \textit{ready}\ \mathsf{a}.d' \end{cases}$$

We now define the abstraction α_{A} of concrete systems, which just discards all the components not involving A, and projects the contracts involving A.

Definition 5.2. *For all A, S we define the* abstract system *$\alpha_{\mathsf{A}}(S)$ as:*

$$\begin{array}{ll} \alpha_{\mathsf{A}}(\mathsf{A}[P]) = \mathsf{A}[P] & \alpha_{\mathsf{A}}(s[\gamma]) = s[\alpha_{\mathsf{A}}(\gamma)] \ \textit{if}\ \gamma = \mathsf{A}\ \textit{says}\ c \mid \mathsf{B}\ \textit{says}\ d \\ \alpha_{\mathsf{A}}(\{\downarrow_x c\}_{\mathsf{A}}) = \{\downarrow_x c\}_{\mathsf{A}} & \alpha_{\mathsf{A}}(S \mid S') = \alpha_{\mathsf{A}}(S) \mid \alpha_{\mathsf{A}}(S') \\ \alpha_{\mathsf{A}}((u)S) = (u)(\alpha_{\mathsf{A}}(S)) & \alpha_{\mathsf{A}}(S) = \mathbf{0},\ \textit{otherwise} \end{array}$$

Abstract semantics. We now introduce the semantics of abstract contracts and systems. For all participants A, the abstract LTSs $\xrightarrow{\ell}\!\!\!\!\rightarrow_{\mathsf{A}}$ and $\xrightarrow{\mu}_{\mathsf{A}}$ on abstract contracts and systems, respectively, are defined by the rules in Fig. 4. Labels ℓ are atoms, with or without the special prefix *ctx* — which indicates a contractual action performed by the context. Labels μ are either *ctx* or they have the form $\mathsf{A}\colon \pi$, where A is the participant in $\rightarrow_{\mathsf{A}}$, and π is a CO_2 prefix.

Rules for abstract contracts (first row in Fig. 4) are simple: in an internal sum, A chooses a branch; in an external sum, the choice is made by the context; in a *ready* $\mathsf{a}.c$ the atom a is fired. The rightmost rule handles a *ready* in the context contract. For abstract systems, some rules are similar to the concrete ones, hence we discuss only the most relevant ones. Rule [α-Do] involves the abstract transitions of contracts. The behaviour of abstract systems also considers context actions, labelled with *ctx*. If $c \vdash \phi$, then the $\mathsf{ask}\ \phi$ passes, indepedently from the context (rule [α-Ask]). If $c \nvdash \neg\phi$, then the $\mathsf{ask}\ \phi$ may pass or not, depending and the context (rule [α-AskCtx]). The correctness of these rules is guaranteed by Lemma 5.11. Rule [α-Fuse] says that a latent contract of A may always be fused (the context may choose whether this is the case or not). The context may also decide whether to perform actions within sessions ([α-DoCtx]). Unobservable context actions are modelled by rules [α-Ctx] and [α-DelCtx].

To check if $\mathsf{A}[P]$ is honest, we must only consider those A-free contexts not already containing advertised/stipulated contracts of A. Such systems will always evolve to a system which can be split in two parts: an A-*solo* system S_{A} containing the process of A, the contracts advertised by A and all the sessions containing contracts of A, and an A-*free* system $S_{\textit{ctx}}$.

Definition 5.3. *We say that a system S is* A-solo *iff one of the following holds:*

$$\begin{array}{llll} S \equiv \mathbf{0} & S \equiv \mathsf{A}[P] & S \equiv s[\mathsf{A}\ \textit{says}\ c \mid \mathsf{B}\ \textit{says}\ d] & S \equiv \{\downarrow_x\ c\}_{\mathsf{A}} \\ \multicolumn{2}{l}{S \equiv S' \mid S'' \quad \textit{where } S' \textit{ and } S''\ \mathsf{A}\textit{-solo}} & \multicolumn{2}{l}{S \equiv (u)S' \quad \textit{where } S'\ \mathsf{A}\textit{-solo}} \end{array}$$

$$\mathsf{a}\,;\,c \oplus c' \overset{\mathsf{a}}{\twoheadrightarrow}_{\mathsf{A}} \mathit{ctx}\ \bar{\mathsf{a}}.c \qquad \mathsf{a}\,.\,c + c' \overset{\mathit{ctx}\,:\,\bar{\mathsf{a}}}{\twoheadrightarrow}_{\mathsf{A}} \mathit{ready}\ \mathsf{a}.\ c \qquad \mathit{ready}\ \mathsf{a}.\ c \overset{\mathsf{a}}{\twoheadrightarrow}_{\mathsf{A}} c \qquad \mathit{ctx}\ \mathsf{a}.c \overset{\mathit{ctx}\,:\,\mathsf{a}}{\twoheadrightarrow}_{\mathsf{A}} c$$

$$\frac{c \overset{\mathsf{a}}{\twoheadrightarrow}_{\mathsf{A}} c'}{\mathsf{A}[\mathsf{do}_s\,\mathsf{a}.P + P' \mid Q] \ \mid\ s[c] \xrightarrow{\mathsf{A}:\ \mathsf{do}_s\ \mathsf{a}}_{\mathsf{A}} \mathsf{A}[P \mid Q] \ \mid\ s[c']} \ [\alpha\text{-Do}]$$

$$\frac{s\ \mathit{fresh}}{(x)(\tilde{S} \ \mid\ \{\downarrow_x c\}_{\mathsf{A}}) \xrightarrow{\mathit{ctx}}_{\mathsf{A}} (s)(s[c] \ \mid\ \tilde{S}\{s/x\})} \ [\alpha\text{-Fuse}]$$

$$\frac{c \vdash \phi}{\mathsf{A}[\mathsf{ask}_s\,\phi.P + P' \mid Q] \ \mid\ s[c] \xrightarrow{\mathsf{A}:\ \mathsf{ask}_s\ \phi}_{\mathsf{A}} \mathsf{A}[P \mid Q] \ \mid\ s[c]} \ [\alpha\text{-Ask}]$$

$$\frac{c \nvdash \neg\phi}{\mathsf{A}[\mathsf{ask}_s\,\phi.P + P' \mid Q] \ \mid\ s[c] \xrightarrow{\mathit{ctx}}_{\mathsf{A}} \mathsf{A}[P \mid Q] \ \mid\ s[c]} \ [\alpha\text{-AskCtx}]$$

$$\frac{c \overset{\mathit{ctx}}{\twoheadrightarrow}_{\mathsf{A}} c'}{s[c] \xrightarrow{\mathit{ctx}}_{\mathsf{A}} s[c']} \ [\alpha\text{-DoCtx}] \qquad S \xrightarrow{\mathit{ctx}}_{\mathsf{A}} S \ [\alpha\text{-Ctx}] \qquad \frac{\tilde{S} \xrightarrow{\mathit{ctx}}_{\mathsf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{\mathit{ctx}}_{\mathsf{A}} (u)\tilde{S}'} \ [\alpha\text{-DelCtx}]$$

Figure 4: Abstract LTSs for contracts and systems (full set of rules in § B).

We say that S is A-safe *iff $S \equiv (\vec{s})(S_{\mathsf{A}} \mid S_{ctx})$, with S_{A}* A*-solo and S_{ctx}* A*-free.*

The following theorems establish the relations between the concrete and the abstract semantics of CO$_2$. Theorem 5.4 states that the abstraction is *correct*, i.e. for each concrete computation there exists a corresponding abstract computation. Theorem 5.5 states that the abstraction is also *complete*, provided that a process has neither ask nor non-proper contracts.

Theorem 5.4. *For all* A*-safe systems S, and for all concrete traces η:*

$$S \xrightarrow{\eta}{}^{*} S' \implies \exists\tilde{\eta}\ :\ \alpha_{\mathsf{A}}(S) \xrightarrow{\tilde{\eta}}{}_{\mathsf{A}}^{*} \alpha_{\mathsf{A}}(S')$$

Furthermore, if η is A*-solo and S is* ask*-free, then $\eta = \tilde{\eta}$.*

Theorem 5.5. *For all* ask*-free abstract system $\tilde{S}$ with proper contracts only:*

$$\tilde{S} \rightarrow_{\mathsf{A}}^{*} \tilde{S}' \implies \exists S, S'\ \mathsf{A}\text{-}\mathit{safe}.\ \alpha_{\mathsf{A}}(S) = \tilde{S}\ \wedge\ S \rightarrow^{*} S'\ \wedge\ \alpha_{\mathsf{A}}(S') = \tilde{S}'$$

The abstract counterparts of Ready Do, Weak Ready Do, and readiness are defined as expected, by using the abstract semantics instead of the concrete one (see §B for details). The notion of honesty for abstract systems, namely *α-honesty*, follows the lines of that of honesty in Def. 4.3.

Definition 5.6 (α-honesty)**.** *We say that P is* α-honest *iff for all $\tilde{S}$ such that $\mathsf{A}[P] \rightarrow_{\mathsf{A}}^{*} \tilde{S}$,* A *is ready in $\tilde{S}$.*

The main result of this paper follows. It states that α-honesty is a sound approximation of honesty, and — under certain conditions — it is also complete.

Theorem 5.7. *If P is α-honest, then P is honest. Conversely, if P is honest,* ask*-free, and has proper contracts only, then P is α-honest.*

In Maude, we implement abstract semantics for system and contracts for one-step transitions. We obtain their transitive closure, discarding labels, with the operator `<_>`. The function `ready` in `search-honest` computes abstract readiness.

```
op search-honest : Process Module -> ResultTriple? .
eq search-honest(P , M:Module) = metaSearch(M:Module, upTerm(< A[P] >),
   '<_>['S:System], 'ready['S:System,'S:System, upTerm(M:Module)]
   = 'false.Bool, '*, unbounded, 0) .

op honest : Process Module -> Result .
ceq honest (P, M:Module) = true if search-honest (P,M:Module) == failure .
ceq honest (P, M:Module) = downTerm (T:Term , < (0).System > )
    if {T:Term, Ty:Type, S:Substitution} := search-honest (P , M:Module) .
```

Honesty is checked by searching for states such that A is *not* ready. If the search fails, then A is honest. As in § 4, this function is decidable for finite state processes, i.e. those without delimitation/parallel under process definitions.

On the verification of ask We now formalise the meaning of the relation $\vdash$ used in the concrete and abstract semantics of ask . Let $\mathbf{A}$ be the set of atoms, and let $\varepsilon \notin \mathbf{A}$. Let $a, b, \cdots$ range over $\mathbf{A} \cup \{\varepsilon\}$. In the following all γ are assumed to be compositions of compliant unilateral contracts.

Definition 5.8. *We define the (unlabelled) transition system* $\mathrm{TS} = (\Sigma, \rightarrow, I, \mathbf{A}, L)$ *as follows:*

- $S = \{(a, \gamma) \mid \gamma$ *is a bilateral contract and* $a \in \mathbf{A} \cup \{\varepsilon\}\}$,
- $I = \{(a, \gamma) \in S \mid a = \varepsilon\}$
- *the transition relation* $\rightarrow \subseteq S \times S$ *is defined by the following rule:*

$$(b, \gamma) \rightarrow (\mathsf{a}, \gamma') \qquad\qquad \text{if } \gamma \xrightarrow{\mathsf{A}\ says\ \mathsf{a}} \gamma'$$

- *the labelling function* $L : S \rightarrow \mathbf{A} \cup \{\varepsilon\}$ *is defined as* $L((a, \gamma)) = a$.

Definition 5.9. *We define the (unlabelled) transition system* $\widetilde{\mathrm{TS}} = (\Sigma, \rightarrow, I, \mathbf{A}, L)$ *as follows:*

- $S = \{(a, \tilde{c}) \mid \tilde{c}$ *is an abstract contract and* $a \in \mathbf{A} \cup \{\varepsilon\}\}$,
- $I = \{(a, \tilde{c}) \in S \mid a = \varepsilon\}$
- *the transition relation* $\rightarrow \subseteq S \times S$ *is defined by the following rule:*

$$(b, \tilde{c}) \rightarrow (\mathsf{a}, \tilde{c}') \qquad\qquad \text{if } c \xrightarrow{\mathsf{a}}_{\mathsf{A}} c' \vee \tilde{c} \xrightarrow{ctx:\mathsf{a}}_{\mathsf{A}} \tilde{c}'$$

- *the labelling function* $L : S \rightarrow \mathbf{A} \cup \{\varepsilon\}$ *is defined as* $L((a, \tilde{c})) = a$.

Definition 5.10. *We define the set* $\mathrm{Paths}(s_0)$ *of maximal traces from a state* s_0 *as:*

$$\{L(s_0)L(s_1)\cdots \mid \forall i > 0.\ s_{i-1} \rightarrow s_i\} \cup \{L(s_0)\cdots L(s_n) \mid (\forall i \in 1..n.\ s_{i-1} \rightarrow s_i) \wedge s_n \not\rightarrow\}$$

Then, we write:

- $\gamma \vdash \phi$ *whenever* $\forall \pi \in \mathrm{Paths}((\epsilon, \gamma)).\ \pi \models \phi$ *in LTL.*
- $\tilde{c} \vdash \phi$ *whenever* $\forall \pi \in \mathrm{Paths}((\epsilon, \tilde{c})).\ \pi \models \phi$ *in LTL.*

Lemma 5.11 below guarantees the correctness of the abstract semantics of ask. Item (1) shows the correctness of rule [α-Ask]: if $\tilde{c} \vdash \phi$, then the rule allows for a transition of A, and then by Lemma 5.11 we know that ask ϕ will pass in each possible concrete context. Rule [α-AskCtx] allows the context to fire the ask ϕ only if $\tilde{c} \not\vdash \neg\phi$. Item (2) of Lemma 5.11 shows that this condition is correct, because whenever the ask ϕ may pass in some concrete context, then $\tilde{c} \not\vdash \neg\phi$ holds.

Lemma 5.11. *For all abstract contracts* $\tilde{c}$ *and for all LTL formulae* ϕ*:*

(1) $\tilde{c} \vdash \phi \iff \forall \gamma\ .\ \alpha_{\mathsf{A}}(\gamma) = \tilde{c} \implies \gamma \vdash \phi$

(2) $\exists \gamma\ .\ \alpha_{\mathsf{A}}(\gamma) = \tilde{c} \wedge \gamma \vdash \phi \implies \tilde{c} \not\vdash \neg\phi$

Experiments

The following example shows a process which was erroneously classified as honest in [4]. The Maude model checker has determined the dishonesty of that process, and by exploiting the Maude tracing facilities we managed to fix it.

Example 5.12. *A store* `A` *offers buyers two options:* `clickPay` *or* `clickVoucher`*. If a buyer* `B` *chooses* `clickPay`*,* `A` *requires a payment (*`pay`*) otherwise* `A` *checks the validity of the voucher with* `V`*, an online voucher distribution system. If* `V` *validates the voucher (*`ok`*),* `B` *can use it (*`voucher`*), otherwise (*`no`*)* `B` *must* `pay`*. We specify in Maude the contracts* `CB` *(between* `A` *and* `B`*) and* `CV` *(between* `A` *and* `V`*) as:*

```
eq CB  = clickPay . pay . 0 +
         clickVoucher . (- reject ; pay . 0 (+) - accept ; voucher . 0) .
eq CV = ok . 0 + no . 0 .
```

We can specify in Maude a CO_2 process for `A` *as follows:*

```
eq P = (x)(tell x CB . (do x clickPay . do x pay . 0 +
                        do x clickVoucher . ((y) tell y CV . Q))) .
eq Q = do y ok . do x - accept . do x voucher . 0 +
       do y no . do x - reject . do x pay . 0     + R .
eq R = t . (do x - reject . do x pay . 0) .
```

Variables `x` *and* `y` *in* `P` *correspond to two separate sessions, where* `A` *respectively interacts with* `B` *and* `V`*. The advertisement of* `CV` *causally depends on the stipulation of the contract* `CB`*, because* `A` *must fire* `clickVoucher` *before* `tell y CV`*. In process* `Q` *the store waits for the answer of* `V`*: if* `V` *validates the voucher (first branch), then* `A` *accepts it from* `B`*; otherwise (second branch),* `A` *requires* `B` *to pay. The third branch* `R` *allows* `A` *to fire a τ action, and then reject the voucher. The intuition is that τ models a timeout, to deal with the fact that* `CV` *might not be stipulated. When we check the honesty of* `P` *with Maude, we obtain:*

```
red honest(P , ['STORE-VOUCHER]) .
rewrites: 31649 in 72ms cpu (77ms real) (439545 rewrites/second)
result TSystem: < ($ 0,$ 1)(A[do $ 0 - reject . do $ 0 pay . (0).Sum] |
  $ 0[- accept ; voucher . 0(+)- reject ; pay . 0] | $ 1[ready ok . 0]) >
```

This means that the process `P` *is dishonest: actually, the output provides a state where* `A` *is not ready. There,* `A` *must do* `ok` *in session y (*`$1`*), while* `A` *is only ready to do a* `-reject` *at session x (*`$0`*). This problem occurs when the branch* `R` *is chosen. To recover honesty, it suffices to replace* `R` *with the following process* `R'`*:*

```
eq R' = t . (do x - reject . do x pay . 0 | (do y no . 0 + do y ok . 0)) .
red honest(P' , ['STORE-VOUCHER]) .
rewrites: 44009 in 32ms cpu (30ms real) (1375195 rewrites/second)
result Bool: true
```

Example 5.13. *Recall the contract of the store in Example 2.7. A possible specification of the store is the following one:*

```
eq PA = (x) (tell x CA . do x addToCart . X(x)) .
eq PPay = do x pay . (t . do x - ok . 0 + t . do x - no . 0) .
eq env = (X(x) =def do x addToCart . X(x) + PPay + do x cancel . 0) .
```

The process `PA` *first advertises the contract* `CA`*, and when the session* `x` *is created, it waits that the user performs the first* `addToCart`*. Then, the store enters a recursive process* `X`*. In the body of* `X`*, the store accepts three actions from the user: an* `addToCart`*, which is followed by a recursive call to* `X`*, a* `cancel`*, which terminates the store process, or a* `pay` *from the user, which is handled by the process* `PPay`*. Within* `PPay`*, after the payment is received the store internally chooses whether to accept it or not, by firing the actions* `- ok` *or* `- no`*, respectively.*

The Maude model checker correctly classifies the process `PA` *as honest:*

```
red honest(PA , ['STORE-CART]) .
rewrites: 2895 in 40ms cpu (39ms real) (72371 rewrites/second)
result Bool: true
```

Example 5.14. *Consider an on-line food store* `A`*, which sells apples (*`a`*) and bottles of an expensive italian Brunello wine (*`b`*). Selling apples is quite easy: once an order is placed,* `A` *accepts it (with the feedback* `- ok`*) and waits for a payment (*`pay`*) before shipping the goods (*`- ship-a`*). However, if expensive bottles of Brunello are ordered, the store is entitled to either decline the order (by answering* `- no`*), or accept it (and, as above, ship the item after the payment).*

The store contract can be modelled in Maude as follows:

```
eq CA = a . - ok ; pay . - ship-a ; 0 +
        b . (- no ; 0 (+) - ok ; pay . - ship-b ; 0) .
```

and a possible specification of the store `A` *is:*

```
eq ship-a? = <> (- ship-a) .
eq PA = (x) (tell x CA . do x a . X(x) + do x b . X(x)) .
eq env = (X(x) =def do x - ok . do x pay . ask x ship-a? . do x - ship-a . 0) .
```

Here, `A` *creates a private channel* `x`*, and advertises the contract* `CA`*. Once the session at* `x` *is initiated,* `PA` *can accept an order for* `a` *or* `b` *on* `x`*. In both cases, the process* `X(x)` *is invoked. There,* `A` *accepts the transaction with* `ok`*, and waits for payment. Then* `A` *checks whether the contract requires to ship apples: if the query* `ask x ship-a?` *passes, the goods are shipped. Otherwise, when the customer* `B` *orders Brunello,* `A` *maliciously gets stuck, and so* `B` *has paid for nothing. Clearly, this store is dishonest, as it does not respect its own contract* `CA`*. The Maude model checker gives:*

```
red honest(PA , ['FOOD-STORE-MALICIOUS]) .
rewrites: 1433 in 20ms cpu (20ms real) (71646 rewrites/second)
result TSystem: < ($ 0)(A[do $ 0 a . X($ 0)] | $ 0[ready b . (- ok ; pay . -
    ship-b ; 0(+)- no ; 0)]) >
```

At session `$ 0` *(i.e.,* `x`*), the store* `A` *must perform a* `b` *(because of the* `ready b`*), while the process of* `A` *is only ready to perform an* `a`*.*

Consider now a non-malicious implementation of the store. Here, before accepting orders the store requires an insurance to cover shipment damages. — which may be particularly useful for the expensive (and fragile) Brunello bottles. The contract between `A` *and the insurance company is:*

```
eq CI = - payI . (- cover ; 0 (+) - cancel ; 0) .
```

There, `A` *promises to pay (*`payI`*) and then choose between getting the coverage, or cancelling the request. The new specification of the store process is:*

```
eq PA = (x , y) (tell y CI . do y - payI . tell x CA .
                (do x a . do x - ok . X(x) + do x b . Y(x ; y))) .

eq env = (
    X(x)    =def do x pay . (do x - ship-a . 0 + do x - ship-b . 0)
    &
    Y(x ; y) =def do y - cover . (do x - ok . X(x) + t . do x - no . 0)
) .
```

The store A first requests an insurance by advertising `CI`*. Once an insurance company* `C` *agrees,* `A` *pays the premium (on channel* `y`*), and then advertises* `CA`*; once an agreement with a customer* `B` *is reached,* `A` *waits for* `a` *or* `b` *orders. If apples are requested,* `A` *acknowledges (*`- ok`*) and invokes* `X(x)`*; there,* `A` *waits for payment, checks which good has to be shipped, and actually ships it. Otherwise, if Brunello is requested,* `Y(x,y)` *is invoked: there,* `A` *requests the insurance coverage paid in advance; then, either the order is accepted and* `X(x)` *is invoked for payment and shipment (as above), or the transaction is declined after an internal τ-action (e.g. a wake up after a timeout).*

When we check `PA` *for honesty, we obtain:*

```
red honest(PA , ['FOOD-STORE-NAIVE]) .
rewrites: 9377 in 20ms cpu (23ms real) (468803 rewrites/second)
result TSystem: < (x,$ 0)(A[tell x (a . - ok ; pay . - ship-a ; 0 + b . (- ok ;
    pay . - ship-b ; 0(+)- no ; 0)) . (do x a . do x - ok . X(x) + do x b . Y(x
    ; $ 0))] | $ 0[- cover ; 0(+)- cancel ; 0]) >
```

The specification of `PA` *does not seem malicious: however, it is keen to ship the goods, but it is not honest either due to the interaction between* `A` *and the insurance company. Actually, if* `C` *does not execute* `cover`*, then* `A` *gets stuck on* `do y - cover`*, unable to honour* `CA` *by providing the expected* `ok` *or* `no`*. Furthermore,* `A` *is dishonest w.r.t.* `CI`*: the premium is paid in advance, but* `A` *may never perform* `do y - cover` *nor* `do y - cancel` *— e.g. if no agreement on* `CA` *is found, or if the customer* `B` *is stuck, or if* `B` *simply chooses to buy apples. Thus, due to implementation naïveties,* `A` *may be blamed due to the unexpected (or malicious) behaviour of other participants.*

Finally, a honest implementation of the food store is the following:

```
eq PA = (x) (tell x CA . (do x a . X(x) + do x b . Y(x))) .

eq env = (
    X(x)   =def do x - ok . do x pay . (ask x (0 - ship-a) . do x - ship-a . 0
                                     +  ask x (0 - ship-b) . do x - ship-b . 0)
    &
    Y(x) =def (y) (tell y CI . do y - payI . 0 |
                  (do y - cover . X(x) + t . (do x - no . 0 | do y - cancel . 0)))
) .
```

Indeed, the Maude model checker gives:

```
red honest(PA , ['FOOD-STORE-HONEST]) .
rewrites: 35872 in 48ms cpu (48ms real) (747286 rewrites/second)
result Bool: true
```

6 Conclusions

We have described an executable specification in Maude of a calculus for contract-oriented systems. This has been done in two steps. First, we have specified a model for contracts, and we have formalised in Maude their semantics, and the crucial notions of compliance and culpability (§ 2). This specification has been exploited in § 3 to implement in Maude the calculus CO_2 [3]. Then, we have considered the problem of honesty [4], i.e. that of deciding when a participant always respects the contracts she advertises, in all possible contexts (§ 4). Writing honest processes is not a trivial task, especially when multiple sessions are needed for realising a contract (see e.g. Ex. 4.4 and Ex. 5.12). We have then devised a sound verification technique for deciding when a participant is honest, and we have provided an implementation of this technique in Maude (§ 5).

Related work. Rewriting logic [11] has been successfully used for more than two decades as a semantic framework wherein many different programming models and logics are naturally formalised, executed and analysed. Just by restricting to models for concurrency, there exist Maude specifications and tools for CCS [16], the π-calculus [15], Petri nets [14], Erlang [13], Klaim [17], adaptive systems [6], *etc.* A more comprehensive list of calculi, programming languages, tools and applications implemented in Maude is collected in [12].

The contract model presented in § 2 is a refined version of the one in [4], which in turn is an alternative formalisation of the one in [7]. Our version is simpler and closer to the notion of *session behaviour* [1], and enjoys several desirable properties. Theorem 2.9 establishes that only one participant may be culpable in a bilateral contract, whereas in [4] both participants may be culpable, e.g. in A *says* $\mathsf{a}\,;\,c \mid \mathsf{B}$ *says* $\bar{\mathsf{a}}\,;\,d$. In our model, if both participants have an internal (or external) choice, then their contracts are *not* compliant, whereas e.g. $\mathsf{a}.c$ and $\bar{\mathsf{a}}.d$ (both external choices) are compliant in [4, 7] whenever c and d are compliant. The exculpation property established by Theorem 2.10 is stronger than the corresponding one in [4]. There, a participant A is guaranteed to exculpate herself by performing (at most) two consecutive actions *of* A, while in our model two any actions (of *whatever* participant) suffice.

As far as we know, the concept of *contract-oriented computing* (in the meaning used in this paper) has been introduced in [5]. CO_2, a contract-agnostic calculus for contract-oriented computing, has been instantiated with several contract models — both bilateral [4, 2] and multiparty [10, 3]. Here we have instantiated it with the contracts in § 2. A minor difference w.r.t. [4, 2, 10] is that here we no longer have fuse as a language primitive, but rather the creation of fresh sessions is performed non-deterministically by the context (rule [Fuse]). This is equivalent to assume a contract broker which collects all contracts, and may establish sessions when compliant contracts are found. In [4], a participant A is considered honest when, in each possible context, she can always exculpate herself by a sequence of A-solo moves. Here we require that A is ready (i.e. some of her obligations are in the Weak Ready Do set) in all possible contexts, as in [2]. We conjecture that these two notions are equivalent. In [2] a type system has been proposed to safely over-approximate honesty. The type of a process P is a function which maps each variable to a *channel type*. These are behavioural types (in the form of Basic Parallel Processes) which essentially preserve the structure of P, by abstracting the actual prefixes as "non-blocking" and "possibly blocking". The type system relies upon checking honesty for channel types, but no actual algorithm is given for such verification, hence type inference remains an open issue. In contrast, here we have directly implemented in Maude a verification algorithm for honesty, by model checking the abstract semantics in § 5.

Acknowledgments. This work has been partially supported by Aut. Region of Sardinia under grants L.R.7/2007 CRP-17285 (TRICS) and P.I.A. 2010 project "Social Glue", and by MIUR PRIN 2010-11 project "Security Horizons", and by EU COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems" (BETTY).

References

[1] F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, 2010.

[2] M. Bartoletti, A. Scalas, E. Tuosto, and R. Zunino. Honesty by typing. In *FMOODS/FORTE*, volume 7892 of *LNCS*, 2013.

[3] M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1), 2012.

[4] M. Bartoletti, E. Tuosto, and R. Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION*, volume 7274 of *LNCS*, 2012.

[5] M. Bartoletti and R. Zunino. A calculus of contracting processes. In *LICS*, 2010.

[6] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In *WRLA*, volume 7571 of *LNCS*, 2012.

[7] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

[8] 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.

[9] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, 1998.

[10] J. Lange and A. Scalas. Choreography synthesis as contract agreement. In *ICE*, 2013.

[11] J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR*, volume 458 of *LNCS*, 1990.

[12] J. Meseguer. Twenty years of rewriting logic. *JLAP*, 81(7-8), 2012.

[13] M. Neuhäußer and T. Noll. Abstraction and model checking of core Erlang programs in Maude. *ENTCS*, 176(4), 2007.

[14] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, 2001.

[15] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. *ENTCS*, 71, 2002.

[16] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *ENTCS*, 71, 2002.

[17] M. Wirsing, J. Eckhardt, T. Mühlbauer, and J. Meseguer. Design and analysis of cloud-based architectures with KLAIM and Maude. In *WRLA*, volume 7571 of *LNCS*, 2012.

[18] T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305 – 340, 2009.