

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 primitives on contracts, i.e. *compliance* and *culpability testing*, and we study some relevant properties.

In § 3 we present CO₂ (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 CO₂, which preserves the transitions of a participant $A[P]$, while abstracting those of the context wherein $A[P]$ 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 CO₂ 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 A, B, \dots) and a denumerable set of *atoms* (ranged over by a, b, \dots). We postulate an involution $\text{co}(a)$, also written as \bar{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:

$$c, d ::= \bigoplus_{i \in \mathcal{I}} a_i ; c_i \mid \sum_{i \in \mathcal{I}} a_i . c_i \mid \text{ready } a.c \mid \text{rec } X. c \mid X$$

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 $A \text{ says } c \mid B \text{ says } d$, where $A \neq B$ and at most one occurrence of “ready” is present. The order of unilateral contracts in γ is immaterial, i.e. $A \text{ says } c \mid B \text{ says } d \equiv B \text{ says } d \mid A \text{ says } c$.

An internal sum $\bigoplus_{i \in \mathcal{I}} a_i ; c_i$ allows to choose one of the branches $a_i ; c_i$, to perform the action a_i , and then to behave according to c_i . Dually, an external sum $\sum_{i \in \mathcal{I}} a_i . c_i$ allows to wait for the other participant to choose one of the branches $a_i . c_i$, then to perform the corresponding a_i and behave according to c_i . Separators $;$ and $.$ allow for distinguishing singleton internal sums $a ; c$ from singleton external sums $a . c$. 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`

$$\begin{array}{c}
A \text{ says } (a; c \oplus c') \mid B \text{ says } (\bar{a}.d + d') \xrightarrow{A \text{ says } a} A \text{ says } c \mid B \text{ says ready } \bar{a}.d \quad [\text{INTEXT}] \\
A \text{ says ready } a. c \mid B \text{ says } d \xrightarrow{A \text{ says } a} A \text{ says } c \mid B \text{ says } d \quad [\text{RDY}]
\end{array}$$

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:

$$\begin{array}{l}
c_{\text{pay}} = \text{pay} . (\overline{\text{ok}}; 0 \oplus \overline{\text{no}}; 0) \\
c_A = \text{addToCart} . (\text{rec } Z. \text{addToCart} . Z + c_{\text{pay}} + \text{cancel} . 0)
\end{array}$$

Instead, a buyer contract could be expressed as:

$$c_B = \text{rec } Z. (\overline{\text{addToCart}}; Z \oplus \overline{\text{pay}}; (\text{ok} . 0 + \text{no} . 0))$$

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` is 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 $\xrightarrow{\mu}$, 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 $\text{rec } X. c \equiv c\{\text{rec } X. c/X\}$ and $\bigoplus_{i \in \emptyset} a_i; c_i \equiv \sum_{i \in \emptyset} a_i . c_i$. The label $\mu = A \text{ says } a$ 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{a} in his external sum. This is done by marking that branch with

ready \bar{a} , while discarding all the other branches; B will then perform his action in the subsequent step, by rule $[\text{RDY}]$.

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:

```

cr1 [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 .

r1 [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 $\text{rdy}(\gamma)$ is undefined (i.e. equal to \perp). Otherwise, $\text{rdy}(\gamma)$ contains the atoms which could be fired in the first step.

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

$$\text{rdy}\left(A \text{ says } \bigoplus_{i \in \mathcal{I}} a_i ; c_i \mid B \text{ says } \sum_{j \in \mathcal{J}} b_j . c_j\right) = \{a_i\}_{i \in \mathcal{I}} \quad \text{if } \{a_i\}_{i \in \mathcal{I}} \subseteq \{\bar{b}_j\}_{j \in \mathcal{J}} \\ \text{and } (\mathcal{I} = \emptyset \implies \mathcal{J} = \emptyset)$$

$$\text{rdy}(A \text{ says ready } a.c \mid B \text{ says } d) = \{a\}$$

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

- (1) $\text{rdy}(A \text{ says } c \mid B \text{ says } d) \neq \perp$
- (2) $A \text{ says } c \mid B \text{ says } d \xrightarrow{\mu} A \text{ says } c' \mid B \text{ says } d' \implies c' \bowtie d'$

Example 2.4. Let $\gamma = A \text{ says } c \mid B \text{ says } d$, where $c = a ; c_1 \oplus b ; c_2$ and $d = \bar{a} . d_1 + \bar{c} . d_2$. If the participant A internally chooses to perform a , then γ will take a transition to $A \text{ says } c_1 \mid B \text{ says ready } \bar{a} . d_1$. Suppose instead that A chooses to perform b , which is not offered by B in his external choice. In this case, $\gamma \not\xrightarrow{A \text{ says } b}$. We have that $\text{rdy}(\gamma) = \perp$, 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 $\gamma = A \text{ says } c \mid B \text{ says } d$:*

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

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

```

eq <{1} 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 ($A \dot{\smile} \gamma$ in symbols) iff $\gamma \xrightarrow{A \text{ says } a}$ for some a . When A is not culpable in γ we write $A \dot{\smile} \neg \gamma$.

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

Theorem 2.9. Let $\gamma = A \text{ says } c \mid B \text{ says } d$, with $c \bowtie d$. If $\gamma \rightarrow^* \gamma'$, then either $\gamma' = A \text{ says } 0 \mid B \text{ says } 0$, 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 $\gamma = A \text{ says } c \mid B \text{ says } d$. For all γ' such that $\gamma \rightarrow^* \gamma'$, we have that:

- (1) $\gamma' \not\bowtie \Rightarrow A \dot{\smile} \neg \gamma'$ and $B \dot{\smile} \neg \gamma'$
- (2) $A \dot{\smile} \gamma' \Rightarrow \forall \gamma'' . \gamma' \rightarrow \gamma'' \Rightarrow \begin{cases} A \dot{\smile} \gamma'' , \text{ or} \\ \forall \gamma''' . \gamma'' \rightarrow \gamma''' \Rightarrow A \dot{\smile} \gamma''' \end{cases}$

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 $\{1\} g \mid = \text{--A-->>}$ 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 0 --A-->> (where 0 is the “next” operator of LTL). This is verified through the Maude model checker.

commutative monoidal laws for $|$ on processes and systems

$$\begin{aligned}
A[(v)P] &\equiv (v)A[P] & Z | (u)Z' &\equiv (u)(Z | Z') \text{ if } u \notin \text{fv}(Z) \cup \text{fn}(Z) \\
(v)(v)Z &\equiv (v)(u)Z & (u)Z &\equiv Z \text{ if } u \notin \text{fv}(Z) \cup \text{fn}(Z) & \{\downarrow_s c\}_A &\equiv \mathbf{0}
\end{aligned}$$

Figure 2: Structural equivalence for CO₂ (Z, Z' range over systems or processes).

```

op --_-->> : Participant -> Prop .
eq {A says a} g |= -- A -->> = true .
eq {1} g |= -- A -->> = false [otherwise] .
op _ :C _ : Participant BiContract -> Bool .
eq A :C g = modelCheck(g, 0 -- A -->>) == true .

```

3 Modelling contracting processes

We model agents and systems through the process calculus CO₂ [2], which we instantiate with the contracts introduced in § 2. The primitives of CO₂ 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, \dots) and *session names* (ranged over by s, t, \dots). Let u, v, \dots 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 CO₂ is given as follows:*

$$\begin{array}{lcl}
\text{Systems } S & ::= & \mathbf{0} \quad | \quad A[P] \quad | \quad s[\gamma] \quad | \quad S | S \quad | \quad (u)S \quad | \quad \{\downarrow_u c\}_A \\
\text{Processes } P & ::= & \sum_i \pi_i.P_i \quad | \quad P | P \quad | \quad (u)P \quad | \quad X(\vec{u}) \\
\text{Prefixes } \pi & ::= & \tau \quad | \quad \text{tell } \downarrow_u c \quad | \quad \text{do}_u a \quad | \quad \text{ask}_u \phi
\end{array}$$

Systems are the parallel composition of *participants* $A[P]$, *delimited systems* $(u)S$, *sessions* $s[\gamma]$ and *latent contracts* $\{\downarrow_u c\}_A$. A latent contract $\{\downarrow_x c\}_A$ 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 $(\vec{u})(A[P] | B[Q] | \dots)$, $A \neq B$. We denote with K a special participant name (playing the role of contract broker) such that, in each system $(\vec{u})(A[P] | \dots)$, $A \neq K$. 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 $X(x_1, \dots, x_j) \stackrel{\text{def}}{=} P$ such that $\text{fv}(P) \subseteq \{x_1, \dots, x_j\} \subseteq \mathcal{V}$, 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 $\text{tell } \downarrow_u c$, action execution $\text{do}_u a$, and contract query $\text{ask}_u \phi$ (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 CO₂ 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 .

```

$$\begin{array}{c}
\frac{}{A[\tau.P + P' \mid Q] \xrightarrow{A: \tau} A[P \mid Q]} \quad [\text{TAU}] \\
\frac{}{A[\text{tell } \downarrow_u c.P + P' \mid Q] \xrightarrow{A: \text{tell } \downarrow_u c} A[P \mid Q] \mid \{\downarrow_u c\}_A} \quad [\text{TELL}] \\
\frac{c \bowtie d \quad \gamma = A \text{ says } c \mid B \text{ says } d \quad \sigma = \{s/x, y\} \quad s \text{ fresh}}{(x, y)(S \mid \{\downarrow_x c\}_A \mid \{\downarrow_y d\}_B) \xrightarrow{K: \text{fuse}} (s)(S\sigma \mid s[\gamma])} \quad [\text{FUSE}] \\
\frac{\gamma \xrightarrow{A \text{ says } a} \gamma'}{A[\text{do}_s a.P + P' \mid Q] \mid s[\gamma] \xrightarrow{A: \text{do}_s a} A[P \mid Q] \mid s[\gamma']} \quad [\text{DO}] \\
\frac{\gamma \vdash \phi}{A[\text{ask}_s \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{A: \text{ask}_s \phi} A[P \mid Q] \mid s[\gamma]} \quad [\text{ASK}] \\
\frac{X(\bar{u}) \stackrel{\text{def}}{=} P \quad A[P\{\bar{v}/\bar{u}\} \mid Q] \mid S \xrightarrow{\mu} S'}{A[X(\bar{v}) \mid Q] \mid S \xrightarrow{\mu} S'} \quad [\text{DEF}] \quad \frac{S \xrightarrow{\mu} S'}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \quad [\text{PAR}] \\
\frac{S \xrightarrow{A: \pi} S'}{(u)S \xrightarrow{A: \text{del}_u(\pi)} (u)S'} \quad [\text{DEL}] \quad \text{where } \text{del}_u(\pi) = \begin{cases} \tau & \text{if } u \in \text{fnv}(\pi) \\ \pi & \text{otherwise} \end{cases}
\end{array}$$

Figure 3: Reduction semantics of CO₂.

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 CO₂ semantics is formalised by the relation $\xrightarrow{\mu}$ in Fig. 3, where

$$\mu \in \{A: \pi \mid A \neq K\} \cup \{K: \text{fuse}\}$$

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 $\{\downarrow_x c\}_A$. Rule [FUSE] finds *agreements* among the latent contracts: it happens when there exist $\{\downarrow_x c\}_A$ and $\{\downarrow_y d\}_B$ such that $A \neq B$ and $c \bowtie d$. Once the agreement is reached, a fresh session containing $\gamma = A \text{ says } c \mid B \text{ says } d$ 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 $\text{del}_u(\pi)$, where the set $\text{fnv}(\pi)$ contains the free names/variables in π . For instance, $(x)A[\text{tell } \downarrow_x c.P] \xrightarrow{A: \tau} (x)(A[P \mid \{\downarrow_x c\}_A])$. Here, it would make little sense to have the label $A: \text{tell } \downarrow_x c$, as x (being delimited) may be α -converted.

Implementing in Maude the semantics of CO₂ is almost straightforward [18]; here we show only the main rules (see § C for the others). Rule [DO] uses the transition relation \Rightarrow on bilateral contracts. Rule [ASK] exploits the Maude model checker to verify if the bilateral contract \mathbf{g} satisfies the LTL formula ϕ . Rule [FUSE] uses the operator $|X|$ to check compliance between the contracts

c and d, then creates the session $s[A \text{ says } c \mid B \text{ says } d]$ (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 CO₂ 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 $O_s^A(S)$ 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 $O_s^A(S)$.

Definition 4.1. *We define the set of atoms $O_s^A(S)$ as:*

$$O_s^A(S) = \left\{ a \mid \exists \gamma, S' . S \equiv s[\gamma] \mid S' \text{ and } \gamma \xrightarrow{A \text{ says } a} \right\}$$

We say that A is culpable at s in S iff $O_s^A(S) \neq \emptyset$.

The set of atoms $RD_s^A(S)$ (“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 $RD_s^A(S) = \emptyset$). The set $WRD_s^A(S)$ (“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:*

$$RD_s^A(S) = \left\{ a \mid \exists S' . S \xrightarrow{A: do_s a} S' \right\}$$

$$WRD_s^A(S) = \left\{ a \mid \exists S' . S \xrightarrow{A: \neq do_s}^* S' \wedge a \in RD_s^A(S') \right\}$$

where we write $S \xrightarrow{A: \neq do_s} S'$ if $\exists \pi . S \xrightarrow{A: \pi} S' \wedge \forall a . \pi \neq do_s a$.

A participant is *ready* in a system if 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, $A[P]$ is honest *in a given system* S when A is ready in all evolutions of $A[P] \mid S$. Then, $A[P]$ 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 $S \equiv (\vec{u})S' \wedge O_s^A(S') \neq \emptyset \implies WRD_s^A(S') \cap O_s^A(S') \neq \emptyset$
3. P is honest in S iff $\forall A : (S \text{ is A-free} \wedge A[P] \mid S \rightarrow^* S') \implies A \text{ is ready in } S'$

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). CO₂ can simulate Turing machines [4], hence reachability in CO₂ 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 *do_s* 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 CO₂ (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 $O_s^A(S) \neq \emptyset$.

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 CO₂ 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:

$$\begin{array}{ll}
c = a ; 0 & d = \bar{a} . 0 \\
P = (x) \text{ tell } \downarrow_x c. \text{ do}_x a & Q = (y) \text{ tell } \downarrow_y d. X \quad \text{where } X \stackrel{\text{def}}{=} \tau.X
\end{array}$$

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

$$\begin{array}{l}
S \xrightarrow{A: \tau} (x)(A[\text{do}_x a] \mid B[Q] \mid \{\downarrow_x c\}_A) \\
\quad \xrightarrow{B: \tau} (x, y)(A[\text{do}_x a] \mid B[X] \mid \{\downarrow_x c\}_A \mid \{\downarrow_y d\}_B) \\
\quad \xrightarrow{K: \text{fuse}} (s) (A[\text{do}_s a] \mid B[X] \mid s[A \text{ says } c \mid B \text{ says } d]) = S' \xrightarrow{B: \tau} S' \xrightarrow{B: \tau} \dots
\end{array}$$

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 `dos 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 CO₂ which preserves the honesty property, while neglecting the actual context where the process A[P] is executed.

The definition of the abstract semantics of CO₂ 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 $\alpha_A(\gamma)$ of a bilateral contract $\gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d$ (Definition 5.1 below) is either c , or $ctx.c$ when d has a *ready*.

Definition 5.1. For all γ , we define the abstract contract $\alpha_A(\gamma)$ as:

$$\alpha_A(\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d) = \begin{cases} c & \text{if } d \text{ is ready-free} \\ ctx \ a.c & \text{if } d = \text{ready } a.d' \end{cases}$$

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 $\alpha_A(S)$ as:

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

Abstract semantics. We now introduce the semantics of abstract contracts and systems. For all participants A, the abstract LTSs $\xrightarrow{\ell}_A$ and $\xrightarrow{\mu}_A$ 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 $\mathbf{A} : \pi$, where A is the participant in \rightarrow_A , and π is a CO₂ 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 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 $[\alpha\text{-Do}]$ involves the abstract transitions of contracts. The behaviour of abstract systems also considers context actions, labelled with *ctx*. If $c \vdash \phi$, then the *ask* ϕ passes, independently from the context (rule $[\alpha\text{-Ask}]$). If $c \not\vdash \neg\phi$, then the *ask* ϕ may pass or not, depending and the context (rule $[\alpha\text{-AskCtx}]$). The correctness of these rules is guaranteed by Lemma 5.11. Rule $[\alpha\text{-Fuse}]$ 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 ($[\alpha\text{-DoCtx}]$). Unobservable context actions are modelled by rules $[\alpha\text{-Ctx}]$ and $[\alpha\text{-DelCtx}]$.

To check if $\mathbf{A}[P]$ 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_{ctx} .

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

$$\begin{array}{llll} S \equiv \mathbf{0} & S \equiv \mathbf{A}[P] & S \equiv s[\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d] & S \equiv \{\downarrow_x c\}_A \\ S \equiv S' \mid S'' & \text{where } S' \text{ and } S'' \text{ A-solo} & S \equiv (u)S' & \text{where } S' \text{ A-solo} \end{array}$$

$$\begin{array}{c}
\mathbf{a}; c \oplus c' \xrightarrow{\mathbf{a}}_{\mathbf{A}} \text{ctx } \bar{\mathbf{a}}.c \quad \mathbf{a}.c + c' \xrightarrow{\text{ctx}:\bar{\mathbf{a}}}_{\mathbf{A}} \text{ready } \mathbf{a}.c \quad \text{ready } \mathbf{a}.c \xrightarrow{\mathbf{a}}_{\mathbf{A}} c \quad \text{ctx } \mathbf{a}.c \xrightarrow{\text{ctx}:\mathbf{a}}_{\mathbf{A}} c \\
\\
\frac{c \xrightarrow{\mathbf{a}}_{\mathbf{A}} c'}{\mathbf{A}[\text{do}_s \mathbf{a}.P + P' \mid Q] \mid s[c] \xrightarrow{\mathbf{A}:\text{do}_s \mathbf{a}}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid s[c']} \quad [\alpha\text{-Do}] \\
\\
\frac{s \text{ fresh}}{(x)(\tilde{S} \mid \{\downarrow_x c\}_{\mathbf{A}}) \xrightarrow{\text{ctx}}_{\mathbf{A}} (s)(s[c] \mid \tilde{S}\{s/x\})} \quad [\alpha\text{-FUSE}] \\
\\
\frac{c \vdash \phi}{\mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[c] \xrightarrow{\mathbf{A}:\text{ask}_s \phi}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid s[c]} \quad [\alpha\text{-ASK}] \\
\\
\frac{c \not\vdash \neg\phi}{\mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[c] \xrightarrow{\text{ctx}}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid s[c]} \quad [\alpha\text{-ASKCTX}] \\
\\
\frac{c \xrightarrow{\text{ctx}}_{\mathbf{A}} c'}{s[c] \xrightarrow{\text{ctx}}_{\mathbf{A}} s[c']} \quad [\alpha\text{-DoCTX}] \quad S \xrightarrow{\text{ctx}}_{\mathbf{A}} S \quad [\alpha\text{-CTX}] \quad \frac{\tilde{S} \xrightarrow{\text{ctx}}_{\mathbf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{\text{ctx}}_{\mathbf{A}} (u)\tilde{S}'} \quad [\alpha\text{-DELCTX}]
\end{array}$$

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

We say that S is \mathbf{A} -safe iff $S \equiv (\bar{s})(S_{\mathbf{A}} \mid S_{\text{ctx}})$, with $S_{\mathbf{A}}$ \mathbf{A} -solo and S_{ctx} \mathbf{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 \mathbf{A} -safe systems S , and for all concrete traces η :*

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

Furthermore, if η is \mathbf{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:*

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

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 $\mathbf{A}[P] \rightarrow_{\mathbf{A}}^* \tilde{S}$, \mathbf{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, \dots 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 $\text{TS} = (\Sigma, \rightarrow, I, \mathbf{A}, L)$ as follows:

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

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

- 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 $\widetilde{\text{TS}} = (\Sigma, \rightarrow, I, \mathbf{A}, L)$ as follows:

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

$$(b, \tilde{c}) \rightarrow (a, \tilde{c}') \quad \text{if } c \xrightarrow{a} c' \vee \tilde{c} \xrightarrow{\text{ctx}:a} \tilde{c}'$$

- 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 $\text{Paths}(s_0)$ of maximal traces from a state s_0 as:

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

Then, we write:

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

Lemma 5.11 below guarantees the correctness of the abstract semantics of `ask`. Item (1) shows the correctness of rule $[\alpha\text{-ASK}]$: if $\tilde{c} \vdash \phi$, then the rule allows for a transition of \mathbf{A} , and then by Lemma 5.11 we know that `ask` ϕ will pass in each possible concrete context. Rule $[\alpha\text{-ASKCTX}]$ 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) $\tilde{c} \vdash \phi \iff \forall \gamma . \alpha_{\mathbf{A}}(\gamma) = \tilde{c} \implies \gamma \vdash \phi$
- (2) $\exists \gamma . \alpha_{\mathbf{A}}(\gamma) = \tilde{c} \wedge \gamma \vdash \phi \implies \tilde{c} \not\vdash \neg\phi$

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 CO₂ [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 a ; c | B says \bar{a} ; d. In our model, if both participants have an internal (or external)

choice, then their contracts are *not* compliant, whereas e.g. $a.c$ and $\bar{a}.d$ (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 $_{\text{FUSE}}$). 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. Neuhäuser 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.

A Proofs for § 2

Lemma A.1. *For all γ , if $\gamma \rightarrow A \text{ says ready } a.c \mid B \text{ says } d$, then d is ready-free.*

Proof. By the semantics of contracts, a *ready* can only occur at top-level of a contract. By Definition 2.1, only one *ready* can occur in γ . The thesis then follows by straightforward analysis of the rules in Figure 1. \square

Below we establish that contracts are deterministic, under the condition that contracts are *proper*, i.e. the prefixes of each summation are pairwise distinct. Determinism ensures that the obligations of a participant at any given time are uniquely determined by their past actions.

Lemma A.2 (Determinism). *Let c, d be proper contracts, and let $\gamma = A \text{ says } c \mid B \text{ says } d$. Then, for all labels μ , and for all bilateral contracts γ' and γ'' :*

$$\gamma \xrightarrow{\mu} \gamma' \wedge \gamma \xrightarrow{\mu} \gamma'' \implies \gamma' = \gamma''$$

Proof. Let $\gamma = A \text{ says } c \mid B \text{ says } d$, and w.l.o.g. assume that $\mu = A \text{ says } a$. Each rule in Figure 1 may generate at most one μ transition for γ . It is therefore enough to consider the set of applicable rules. We have the following two cases:

[RDY] We have that $\gamma = A \text{ says ready } a.c' \mid B \text{ says } d$. The only possible transition is $\gamma \xrightarrow{\mu} A \text{ says } c' \mid B \text{ says } d$, so we are done.

[INTEXT] We have that $\gamma = A \text{ says } (a; c \oplus c') \mid B \text{ says } (\bar{a}.d + d')$. The rule gives the transition $\gamma \xrightarrow{\mu} A \text{ says } c \mid B \text{ says ready } \bar{a}.d$, which is unique because the contracts are proper.

Note that the cases where A cannot perform any action need not be considered. \square

Lemma A.3. *Let $\gamma = A \text{ says } c \mid B \text{ says } d$. Then:*

$$c \bowtie d \iff (\forall c', d'. \gamma \rightarrow^* A \text{ says } c' \mid B \text{ says } d' \implies c' \bowtie d')$$

Proof. The “if” direction is trivial. For the “only if” direction, we proceed by induction on the length n of the derivation $\gamma \rightarrow^n A \text{ says } c' \mid B \text{ says } d'$. Suppose that $c \bowtie d$. The base case ($n = 0$) holds trivially. Now, suppose that $\gamma \rightarrow^{n-1} A \text{ says } c'' \mid B \text{ says } d'' \rightarrow A \text{ says } c' \mid B \text{ says } d'$. By the induction hypothesis, $c'' \bowtie d''$. Then, by item (2) of Definition 2.3 we conclude that $c' \bowtie d'$. \square

The following lemma shows that $\text{rdy}(\gamma)$ produces the same output for equivalent contracts.

Lemma A.4. $\gamma \equiv \gamma' \implies \text{rdy}(\gamma) = \text{rdy}(\gamma')$.

Proof. Straightforward by Def. 2.3 and by the definition of the equivalence \equiv . \square

The following lemma guarantees, for all proper contracts c , the existence of a contract d compliant with c . Intuitively, we can construct d from c by turning internal choices into external ones (and *viceversa*), and by turning actions into co-actions.

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

Proof. For all proper contracts c , let the contract $\text{dual}(c)$ be inductively defined as follows:

$$\begin{aligned} \text{dual}\left(\bigoplus_{i \in I} a_i; c_i\right) &= \sum_{i \in I} \bar{a}_i. \text{dual}(c_i) & \text{dual}(\text{rec } X. c) &= \text{rec } X. \text{dual}(c) \\ \text{dual}\left(\sum_{i \in I} a_i. c_i\right) &= \bigoplus_{i \in I} \bar{a}_i; \text{dual}(c_i) & \text{dual}(X) &= X \end{aligned}$$

Now, let R be the smallest relation such that, for all ready-free c and for all atoms \mathbf{a} :

$$(c, \text{dual}(c)) \in R \quad (1)$$

$$(c, \text{ready } \mathbf{a}. \text{dual}(c)) \in R \quad (2)$$

$$(\text{ready } \mathbf{a}. \text{dual}(c), c) \in R \quad (3)$$

Also, for all pair X of proper contracts, let:

$$F(X) = \left\{ (c, d) \mid \begin{array}{l} \text{rdy}(\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d) \neq \perp, \text{ and} \\ \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d \twoheadrightarrow \mathbf{A} \text{ says } c' \mid \mathbf{B} \text{ says } d' \implies (c', d') \in X \end{array} \right\}$$

By the coinduction proof principle, we have to show that $R \subseteq F(R)$. Suppose that $(c, d) \in R$. The three equations defining R satisfy the first requirement of F , i.e. $\text{rdy}(\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d) \neq \perp$: it is trivial for 2 and 3, while for 1 it can be easily proved by cases on the structure of c . We now prove that (c, d) satisfies the second requirement of F . We have the following three cases, according to the equation used to prove $(c, d) \in R$.

- (1), $d = \text{dual}(c)$. We have two subcases, according to the form of c . Suppose first that c is an external sum, i.e. $c = (\mathbf{a}; c') + c''$. Then $d = \bar{\mathbf{a}}. \text{dual}(c') \oplus \text{dual}(c'')$. By rule [INT_{EXT}], $\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d \twoheadrightarrow \mathbf{A} \text{ says } c' \mid \mathbf{B} \text{ says } \text{ready } \bar{\mathbf{a}}. \text{dual}(c')$. We obtain the thesis by observing that $(c', \text{ready } \bar{\mathbf{a}}. \text{dual}(c')) \in R$ follows by (2). The case where c is an internal sum is similar.
- (2), $d = \text{ready } \mathbf{a}. \text{dual}(c)$. By rule [RDY], $\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d \twoheadrightarrow \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } \text{dual}(c)$. The thesis follows by observing that $(c', \text{dual}(c')) \in R$ follows by 1.
- (3), $c = \text{ready } \mathbf{a}. \text{dual}(d)$. Similar to the previous case.

Summing up, R is a compliance relation, hence by item (1) it follows that $c \bowtie \text{dual}(c)$. \square

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

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

Proof. The “only if” part follows directly by Lemma A.3 and by Definition 2.3.

For the “if” part, assume that $c \bowtie d$, and let:

$$R = \left\{ (\hat{c}, \hat{d}) \mid \gamma \twoheadrightarrow^* \mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d} \right\}$$

$$F(X) = \left\{ (\hat{c}, \hat{d}) \mid \begin{array}{l} \text{rdy}(\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d}) \neq \perp, \text{ and} \\ \mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d} \twoheadrightarrow \mathbf{A} \text{ says } \hat{c}' \mid \mathbf{B} \text{ says } \hat{d}' \implies (\hat{c}', \hat{d}') \in X \end{array} \right\}$$

By the coinduction proof principle, we have to show that $R \subseteq F(R)$. Let $(\hat{c}, \hat{d}) \in R$. By the “only if” direction of Lemma 2.6, it follows that $\text{rdy}(\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d}) \neq \perp$ (thus satisfying the first condition of F), and either:

- $\text{rdy}(\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d}) = \emptyset$. Then, $\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d} \not\twoheadrightarrow$, and thus (\hat{c}, \hat{d}) vacuously satisfies the second clause of F ;
- $\text{rdy}(\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d}) \neq \emptyset$. Then, $\mathbf{A} \text{ says } \hat{c} \mid \mathbf{B} \text{ says } \hat{d} \twoheadrightarrow \mathbf{A} \text{ says } \hat{c}' \mid \mathbf{B} \text{ says } \hat{d}'$, for some \hat{c}', \hat{d}' , and so also $(\hat{c}', \hat{d}') \in R$. Thus, (\hat{c}, \hat{d}) satisfies the second clause of F .

Therefore, in both cases, we have that $(\hat{c}, \hat{d}) \in R \implies (\hat{c}, \hat{d}) \in F(R)$. \square

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

Proof. By Lemma 2.6 it follows that $\text{rdy}(\gamma') \neq \perp$. So, by Definition 2.3 we have the following three cases (symmetric ones are omitted):

- $\gamma' = \mathbf{A}$ says $0 \mid \mathbf{B}$ says 0 , from which the thesis follows directly.
- $\gamma' = \mathbf{A}$ says *ready* $\mathbf{a}.c' \mid \mathbf{B}$ says d' , with d' ready-free (Lemma A.1). We have only one transition from γ' , given by rule $[\text{RDY}]$, which prescribes a move of \mathbf{A} . By Definition 2.8, we have that $\mathbf{A} \dot{\leadsto} \gamma'$ and $\mathbf{B} \dot{\leadsto} \gamma'$.
- $\gamma' = \mathbf{A}$ says $\bigoplus_{i \in \mathcal{I}} \mathbf{a}_i ; c_i \mid \mathbf{B}$ says $\sum_{i \in \mathcal{I}} \bar{\mathbf{a}}_i . d_i + \sum_{j \in \mathcal{J}} \mathbf{b}_j . d_j$. Only rule $[\text{INTEXT}]$ can be applied, and it prescribes a move of \mathbf{A} . By Definition 2.8, we conclude that $\mathbf{A} \dot{\leadsto} \gamma'$ and $\mathbf{B} \dot{\leadsto} \gamma'$. \square

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

- (1) $\gamma' \not\Rightarrow \implies \mathbf{A} \dot{\leadsto} \gamma'$ and $\mathbf{B} \dot{\leadsto} \gamma'$
- (2) $\mathbf{A} \dot{\leadsto} \gamma' \implies \forall \gamma''. \gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathbf{A} \dot{\leadsto} \gamma'', \text{ or} \\ \forall \gamma'''. \gamma'' \twoheadrightarrow \gamma''' \implies \mathbf{A} \dot{\leadsto} \gamma''' \end{cases}$

Proof. Item (1) follows directly from Definition 2.8.

For item (2), we proceed by cases on the rule used to deduce $\gamma' \twoheadrightarrow \gamma''$ (the symmetric cases are omitted).

$[\text{INTEXT}]$ We have that $\gamma' = \mathbf{A}$ says $\mathbf{a} ; c' \oplus c'' \mid \mathbf{B}$ says $\bar{\mathbf{a}} . d' + d''$, and $\gamma'' = \mathbf{A}$ says $c' \mid \mathbf{B}$ says *ready* $\bar{\mathbf{a}}.d'$. By rule $[\text{RDY}]$, γ'' can only take a move of \mathbf{B} . Therefore, $\mathbf{A} \dot{\leadsto} \gamma''$.

$[\text{RDY}]$ We have that $\gamma' = \mathbf{A}$ says *ready* $\mathbf{a}.c' \mid \mathbf{B}$ says d' , and $\gamma'' = \mathbf{A}$ says $c' \mid \mathbf{B}$ says d' . Now we have two further subcases. If $\mathbf{A} \dot{\leadsto} \gamma''$, then we have the thesis. Otherwise, if $\mathbf{A} \dot{\leadsto} \gamma''$, since c' and d' are *ready*-free, then γ'' must have the form \mathbf{A} says $\mathbf{a} ; c' \oplus c'' \mid \mathbf{B}$ says $\bar{\mathbf{a}} . d' + d''$. By rule $[\text{INTEXT}]$ we have that, if $\gamma'' \xrightarrow{\mathbf{A} \text{ says } \mathbf{b}} \gamma'''$ for some \mathbf{b} , then γ''' will have the form *ready* $\bar{\mathbf{b}}.d'''$, for some d''' . Therefore, $\mathbf{A} \dot{\leadsto} \gamma'''$. \square

B Proofs for § 5

Remark B.1 (On contextes and closed participants). *When defining honesty, we limit ourselves to closed participants (i.e., participants whose processes have no free variables). This is necessary in order to rule out tricky cases in which a context with properly crafted delimitations would make a participant trivially dishonest.*

Consider, for instance, the process $P = \text{tell} \downarrow_x c$, where $c = \mathbf{a} ; 0$. Let $S = (x)(\mathbf{A}[P] \mid \mathbf{B}[\text{tell} \downarrow_x d])$, where $d = \mathbf{a} . 0$. The system S is not \mathbf{A} -safe. Indeed, since the delimitation (x) appears in both \mathbf{A} 's and \mathbf{B} 's processes, then S cannot be rewritten (via congruence rules) in the form $(\bar{s})(S_{\mathbf{A}} \mid S_{\text{ctx}})$ with $S_{\mathbf{A}}$ \mathbf{A} -solo and S_{ctx} \mathbf{A} -free, as required by Def. 5.3.

However, the above process P is trivially honest, since it advertises the contract c using a free variable x , and c will never be stipulated if $\mathbf{A}[P]$ is composed in parallel with some \mathbf{A} -free system (as per Def. 4.3). But if $\mathbf{A}[P]$ is put in a context which delimits x , such as in S above, then c may indeed be stipulated: $S \twoheadrightarrow^ (s)(\mathbf{A}[0] \mid \mathbf{B}[0] \mid s[\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d]) = S'$, and \mathbf{A} is not ready in S' .*

$$\begin{array}{c}
\mathbf{a}; c \oplus c' \xrightarrow{\mathbf{a}}_{\mathbf{A}} \text{ctx } \bar{\mathbf{a}}.c \quad \mathbf{a}.c + c' \xrightarrow{\text{ctx}:\bar{\mathbf{a}}}_{\mathbf{A}} \text{ready } \mathbf{a}.c \quad \text{ready } \mathbf{a}.c \xrightarrow{\mathbf{a}}_{\mathbf{A}} c \quad \text{ctx } \mathbf{a}.c \xrightarrow{\text{ctx}:\mathbf{a}}_{\mathbf{A}} c \\
\\
\frac{}{\mathbf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathbf{A}:\tau}_{\mathbf{A}} \mathbf{A}[P \mid Q]} \quad [\alpha\text{-TAU}] \\
\\
\frac{}{\mathbf{A}[\text{tell } \downarrow_x c.P + P' \mid Q] \xrightarrow{\mathbf{A}:\text{tell } \downarrow_x c}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid \{\downarrow_x c\}_{\mathbf{A}}} \quad [\alpha\text{-TELL}] \\
\\
\frac{c \xrightarrow{\mathbf{a}}_{\mathbf{A}} c'}{s[c] \mid \mathbf{A}[\text{do}_s \mathbf{a}.P + P' \mid Q] \xrightarrow{\mathbf{A}:\text{do}_s \mathbf{a}}_{\mathbf{A}} s[c'] \mid \mathbf{A}[P \mid Q]} \quad [\alpha\text{-DO}] \\
\\
\frac{s \text{ fresh}}{(x)(\tilde{S} \mid \{\downarrow_x c\}_{\mathbf{A}}) \xrightarrow{\text{ctx}}_{\mathbf{A}} (s)(s[c] \mid \tilde{S}\{s/x\})} \quad [\alpha\text{-FUSE}] \\
\\
\frac{c \vdash \phi}{\mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[c] \xrightarrow{\mathbf{A}:\text{ask}_s \phi}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid s[c]} \quad [\alpha\text{-ASK}] \\
\\
\frac{c \not\vdash \neg\phi}{\mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[c] \xrightarrow{\text{ctx}}_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid s[c]} \quad [\alpha\text{-ASKCTX}] \\
\\
\frac{X(\bar{u}) \stackrel{\text{def}}{=} P \quad \mathbf{A}[P\{\bar{v}/\bar{u}\} \mid Q] \mid \tilde{S} \xrightarrow{\mu}_{\mathbf{A}} \tilde{S}'}{\mathbf{A}[X(\bar{v}) \mid Q] \mid \tilde{S} \xrightarrow{\mu}_{\mathbf{A}} \tilde{S}'} \quad [\alpha\text{-DEF}] \quad \frac{\tilde{S} \xrightarrow{\mu} \tilde{S}'}{\tilde{S} \mid \tilde{S}'' \xrightarrow{\mu}_{\mathbf{A}} \tilde{S}' \mid \tilde{S}''} \quad [\alpha\text{-PAR}] \\
\\
\frac{\tilde{S} \xrightarrow{\mathbf{A}:\pi}_{\mathbf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{\mathbf{A}:\text{del}_u(\pi)}_{\mathbf{A}} (u)\tilde{S}'} \quad [\alpha\text{-DEL}] \quad \frac{\tilde{S} \xrightarrow{\text{ctx}}_{\mathbf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{\text{ctx}}_{\mathbf{A}} (u)\tilde{S}'} \quad [\alpha\text{-DELCCTX}] \\
\\
\frac{c \xrightarrow{\text{ctx}}_{\mathbf{A}} c'}{s[c] \xrightarrow{\text{ctx}}_{\mathbf{A}} s[c']} \quad [\alpha\text{-DOCTX}] \quad S \xrightarrow{\text{ctx}}_{\mathbf{A}} S \quad [\alpha\text{-CTX}]
\end{array}$$

Figure 5: Abstract LTSs for contracts and systems (full set of rules).

In Lemma B.2 below we show that \mathbf{A} -safety is preserved by transitions. Consequently, we can safely consider \mathbf{A} -safe systems only.

Lemma B.2. *For all systems S, S' such that $S \rightarrow S'$:*

- (1) *If S is \mathbf{A} -solo, then S' is \mathbf{A} -solo.*
- (2) *If S is \mathbf{A} -free, then S' is \mathbf{A} -free.*
- (3) *If S is \mathbf{A} -safe, then S' is \mathbf{A} -safe.*

Proof. Items (1) and (2) are straightforward by Def. 5.3 and by the semantics of CO_2 . For item (3), we proceed by induction on the depth of the derivation of $S \rightarrow S'$. Let S be \mathbf{A} -safe. Then, by Def. 5.3, $S \equiv (\vec{s})(S_{\mathbf{A}} \mid S_{\text{ctx}})$, for some \mathbf{A} -solo $S_{\mathbf{A}}$ and \mathbf{A} -free S_{ctx} . By repeatedly applying rule $[\text{DEL}]$, we have that:

$$\frac{\vdots}{S_{\mathbf{A}} \mid S_{\text{ctx}} \rightarrow S'_0}_{[\text{DEL}]}$$

$$\frac{\vdots}{S \equiv (\vec{s})(S_{\mathbf{A}} \mid S_{\text{ctx}}) \rightarrow (\vec{s})S'_0 = S'}_{[\text{DEL}]}$$

We have then the following cases, according to the rule used to deduce $S_{\mathbf{A}} \mid S_{\text{ctx}} \rightarrow S'_0$.

[Do] There are the following three subcases.

- The **do** is fired by A. Then, $S'_0 = S'_A \mid S_{ctx}$. By item (1), S'_A is A-solo, hence S' is A-safe.
- The **do** is fired by some $B \neq A$, in a session not involving A. Then, $S'_0 = S_A \mid S'_{ctx}$. By item (2), S'_{ctx} is A-free, hence S' is A-safe.
- The $\text{do}_s \mathbf{a}$ is fired by some $B \neq A$, in a session s involving A. We have that $S_A \equiv S'_A \mid s[\gamma]$, with $\gamma = A \text{ says } c \mid B \text{ says } d$, and $S_{ctx} = B[\text{do}_s \mathbf{a}.P + Q \mid R] \mid S'_{ctx}$. Then, by rule [Do]:

$$\frac{\gamma \xrightarrow{B \text{ says } \mathbf{a}} \gamma'}{S_A \mid S_{ctx} \rightarrow S'_A \mid s[\gamma'] \mid B[P \mid R] \mid S'_{ctx}}$$

We have that $S'_A \mid s[\gamma']$ is A-solo, and $B[P \mid R] \mid S'_{ctx}$ is A-free. Therefore, S' is A-safe.

[ASK] Similar to the case [Do].

[FUSE] We have $S_A \equiv (x)(S'_A \mid \{\downarrow_x c\}_A)$, $S_{ctx} \equiv (y)(S'_{ctx} \mid \{\downarrow_y d\}_B)$, and:

$$\frac{c \bowtie d \quad \gamma = A \text{ says } c \mid B \text{ says } d \quad \sigma = \{s/x, y\} \quad s \text{ fresh}}{S_A \mid S_{ctx} \rightarrow (s)(S'_A \sigma \mid s[\gamma] \mid S'_{ctx} \sigma) = S'_0}$$

We have that $S'_A \sigma \mid s[\gamma]$ is A-solo, and $S'_{ctx} \sigma$ is A-free, then by Def. 5.3, S'_0 is A-safe. Therefore, $S = (\vec{s})S'_0$ is A-safe as well.

[PAR] There are the following two subcases.

- $S_A \rightarrow S'_A$, and $S'_0 = S'_A \mid S_{ctx}$. By item (1), S'_A is A-solo, hence S' is A-safe.
- $S_{ctx} \rightarrow S'_{ctx}$. By item (2), S'_{ctx} is A-free, hence S' is A-safe.

[DEF] There are two subcases, according to which participant has moved.

- move of A. We have that $S_A = A[X(\vec{v}) \mid Q] \mid S'_A$, for some A-solo S'_A , and:

$$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \quad A[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'_A \mid S_{ctx} \xrightarrow{\mu} S'_0}{S_A \mid S_{ctx} \xrightarrow{\mu} S'_0} \text{ [DEF]}$$

Since $A[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'_A$ is A-solo, then by the induction hypothesis it follows that S'_0 is A-safe. Therefore, S' is A-safe as well.

- move of some $B \neq A$. Similar to the previous case. □

Definition B.3. We define the following sets:

$$\begin{aligned} \text{O}_s^A(\tilde{S}) &= \left\{ \mathbf{a} \mid \tilde{S} \equiv s[c] \mid \tilde{S}' \wedge c \xrightarrow{\mathbf{a}}_A \right\} \\ \text{RD}_s^A(\tilde{S}) &= \left\{ \mathbf{a} \mid \exists \tilde{S}' . \tilde{S} \xrightarrow{A: \text{do}_s \mathbf{a}}_A \tilde{S}' \right\} \\ \text{WRD}_s^A(\tilde{S}) &= \left\{ \mathbf{a} \mid \exists S' . S \xrightarrow{A: \neq \text{do}_s}_A^* \tilde{S}' \wedge \mathbf{a} \in \text{RD}_s^A(\tilde{S}') \right\} \end{aligned}$$

and we say that A is ready in \tilde{S} iff:

$$\tilde{S} \equiv (\vec{u})\tilde{S}' \wedge \text{O}_s^A(\tilde{S}') \neq \emptyset \implies \text{WRD}_s^A(\tilde{S}') \cap \text{O}_s^A(\tilde{S}') \neq \emptyset$$

The following lemmata state the correspondence between the concrete and the abstract semantics of contracts.

Lemma B.4. For all bilateral contracts γ, γ' :

$$(1) \gamma \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \gamma' \implies \alpha_{\mathbf{A}}(\gamma) \xrightarrow{\mathbf{a}}_{\mathbf{A}} \alpha_{\mathbf{A}}(\gamma')$$

$$(2) \gamma \xrightarrow{\mathbf{B} \text{ says } \mathbf{a}} \gamma' \implies \alpha_{\mathbf{A}}(\gamma) \xrightarrow{\text{ctx} : \mathbf{a}}_{\mathbf{A}} \alpha_{\mathbf{A}}(\gamma')$$

Proof. For (1) we proceed by cases on the rule being used:

[INTEXT] By the semantics of concrete and abstract contracts and by Def. 5.1, we have:

$$\begin{array}{l} \gamma = \mathbf{A} \text{ says } (\mathbf{a}; c \oplus c') \mid \mathbf{B} \text{ says } (\bar{\mathbf{a}}.d + d') \rightarrow \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \bar{\mathbf{a}}.d = \gamma' \\ \alpha_{\mathbf{A}}(\gamma) = \mathbf{a}; c \oplus c' \xrightarrow{\mathbf{a}}_{\mathbf{A}} \text{ctx } \mathbf{a}.c = \alpha_{\mathbf{A}}(\gamma') \end{array}$$

[RDY] Since d is ready-free, we have:

$$\begin{array}{l} \gamma = \mathbf{A} \text{ says ready } \mathbf{a}.c \mid \mathbf{B} \text{ says } d \rightarrow \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d = \gamma' \\ \alpha_{\mathbf{A}}(\gamma) = \text{ready } \mathbf{a}.c \xrightarrow{\mathbf{a}}_{\mathbf{A}} c = \alpha_{\mathbf{A}}(\gamma') \end{array}$$

For (2) we proceed by cases on the rule being used:

[INTEXT] We have:

$$\begin{array}{l} \gamma = \mathbf{A} \text{ says } (\bar{\mathbf{a}}.c + c') \mid \mathbf{B} \text{ says } (\mathbf{a}; d \oplus d') \rightarrow \mathbf{A} \text{ says ready } \bar{\mathbf{a}}.c \mid \mathbf{B} \text{ says } d = \gamma' \\ \alpha_{\mathbf{A}}(\gamma) = \bar{\mathbf{a}}.c + c' \xrightarrow{\text{ctx} : \mathbf{a}}_{\mathbf{A}} \text{ready } \bar{\mathbf{a}}.c = \alpha_{\mathbf{A}}(\gamma') \end{array}$$

[RDY] We have:

$$\begin{array}{l} \gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \mathbf{a}.d \rightarrow \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d = \gamma' \\ \alpha_{\mathbf{A}}(\gamma) = \text{ctx } \mathbf{a}.c \xrightarrow{\text{ctx} : \mathbf{a}}_{\mathbf{A}} c = \alpha_{\mathbf{A}}(\gamma') \end{array}$$

□

Definition B.5. Let $\gamma_{\mathbf{B}}$ be a function defined as follows:

$$\begin{array}{l} \gamma_{\mathbf{B}}(\text{ctx } \mathbf{a}.c) = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \mathbf{a}. \text{dual}(c) \\ \gamma_{\mathbf{B}}(\text{ready } \mathbf{a}.c) = \mathbf{A} \text{ says ready } \mathbf{a}.c \mid \mathbf{B} \text{ says } \text{dual}(c) \\ \gamma_{\mathbf{B}}(c) = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } \text{dual}(c) \quad \textit{otherwise} \end{array}$$

Lemma B.6. For all proper abstract contracts \tilde{c} :

$$(1) \tilde{c} \xrightarrow{\mathbf{a}}_{\mathbf{A}} \tilde{c}' \implies \gamma_{\mathbf{B}}(\tilde{c}) \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \gamma_{\mathbf{B}}(\tilde{c}')$$

$$(2) \tilde{c} \xrightarrow{\text{ctx} : \mathbf{a}}_{\mathbf{A}} \tilde{c}' \implies \gamma_{\mathbf{B}}(\tilde{c}) \xrightarrow{\mathbf{B} \text{ says } \mathbf{a}} \gamma_{\mathbf{B}}(\tilde{c}')$$

$$(3) \alpha_{\mathbf{A}}(\gamma_{\mathbf{B}}(\tilde{c})) = \tilde{c}$$

Proof. (1) There are two cases:

- $\tilde{c} = \mathbf{a}; c \oplus c' \xrightarrow{\mathbf{a}}_{\mathbf{A}} \text{ctx } \bar{\mathbf{a}}.c = \tilde{c}'$
 $\gamma_{\mathbf{B}}(\tilde{c}) = \mathbf{A} \text{ says } \mathbf{a}; c \oplus c' \mid \mathbf{B} \text{ says } \bar{\mathbf{a}}. \text{dual}(c) + \text{dual}(c')$
 $\gamma_{\mathbf{B}}(\tilde{c}') = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \bar{\mathbf{a}}. \text{dual}(c)$
Clearly: $\gamma_{\mathbf{B}}(\tilde{c}) \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \gamma_{\mathbf{B}}(\tilde{c}')$

- $\tilde{c} = \text{ready } \mathbf{a} . c' \xrightarrow{\mathbf{a}}_{\mathbf{A}} c' = \tilde{c}'$
 $\gamma_{\mathbf{B}}(\tilde{c}) = \mathbf{A} \text{ says ready } \mathbf{a} . c' \mid \mathbf{B} \text{ says dual}(c') \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \mathbf{A} \text{ says } c' \mid \mathbf{B} \text{ says dual}(c') = \gamma_{\mathbf{B}}(\tilde{c}')$

(2) There are two cases:

- $\tilde{c} = \bar{\mathbf{a}} . c + c' \xrightarrow{\text{ctx}:\mathbf{a}}_{\mathbf{A}} \text{ready } \bar{\mathbf{a}} . c = \tilde{c}'$
 $\gamma_{\mathbf{B}}(\tilde{c}) = \mathbf{A} \text{ says } \bar{\mathbf{a}} . c + c' \mid \mathbf{B} \text{ says } \mathbf{a} ; \text{dual}(c) \oplus \text{dual}(c')$
 $\gamma_{\mathbf{B}}(\tilde{c}') = \mathbf{A} \text{ says ready } \bar{\mathbf{a}} . c \mid \mathbf{B} \text{ says dual}(c)$
Clearly: $\gamma_{\mathbf{B}}(\tilde{c}) \xrightarrow{\mathbf{B} \text{ says } \mathbf{a}} \gamma_{\mathbf{B}}(\tilde{c}')$
- $\tilde{c} = \text{ctx } \mathbf{a} . c \xrightarrow{\text{ctx}:\mathbf{a}}_{\mathbf{A}} c = \tilde{c}'$
 $\gamma_{\mathbf{B}}(\tilde{c}) = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \mathbf{a} . \text{dual}(c) \xrightarrow{\mathbf{B} \text{ says } \mathbf{a}} \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says dual}(c)$

(3) By cases on the form of \tilde{c} :

- $\alpha_{\mathbf{A}}(\gamma_{\mathbf{B}}(\text{ctx } \mathbf{a}.c)) = \alpha_{\mathbf{A}}(\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says ready } \mathbf{a}.\text{dual}(c)) = \text{ctx } \mathbf{a}.c = \tilde{c}$
- $\alpha_{\mathbf{A}}(\gamma_{\mathbf{B}}(\text{ready } \mathbf{a}.c)) = \alpha_{\mathbf{A}}(\mathbf{A} \text{ says ready } \mathbf{a}.c \mid \mathbf{B} \text{ says dual}(c)) = \text{ready } \mathbf{a}.c = \tilde{c}$
- $\alpha_{\mathbf{A}}(\gamma_{\mathbf{B}}(c)) = \alpha_{\mathbf{A}}(\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says dual}(c)) = c = \tilde{c}$, with c ready-free.

□

Lemma B.7. For all proper abstract contracts \tilde{c} , we have:

- (1) $\tilde{c} \xrightarrow{\mathbf{a}}_{\mathbf{A}} \tilde{c}' \implies \forall \gamma . \alpha_{\mathbf{A}}(\gamma) = \tilde{c} \implies \exists \gamma' . \gamma \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \gamma' \wedge \alpha_{\mathbf{A}}(\gamma') = \tilde{c}'$
- (2) $\tilde{c} \xrightarrow{\text{ctx}:\mathbf{a}}_{\mathbf{A}} \tilde{c}' \implies \exists \gamma, \gamma' . \gamma \xrightarrow{\mathbf{B} \text{ says } \mathbf{a}} \gamma' \wedge \alpha_{\mathbf{A}}(\gamma) = \tilde{c} \wedge \alpha_{\mathbf{A}}(\gamma') = \tilde{c}'$

Proof. For (1), we have two cases:

- i. For $\tilde{c} = \mathbf{a} ; c' \oplus c''$ we have that $\tilde{c} \xrightarrow{\mathbf{a}}_{\mathbf{A}} \text{ctx } \mathbf{a}.c' = \tilde{c}'$. Let $\gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d$, with $c = \tilde{c}$, $d = \bar{\mathbf{a}} . \text{dual}(c') + \text{dual}(c'') + d'$, and d' ranges over all external sums such that d does not contain duplicate atoms at top level. Clearly $\gamma \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \mathbf{A} \text{ says } c' \mid \mathbf{B} \text{ says ready } \bar{\mathbf{a}}.d' = \gamma'$. Of course $\alpha_{\mathbf{A}}(\gamma) = \tilde{c}$ and $\alpha_{\mathbf{A}}(\gamma') = \tilde{c}'$.
- ii. For $\tilde{c} = \text{ready } \mathbf{a}.c'$ we have that $\tilde{c} \xrightarrow{\mathbf{a}}_{\mathbf{A}} c' = \tilde{c}'$. Let $\gamma = \mathbf{A} \text{ says } \tilde{c} \mid \mathbf{B} \text{ says } d$, with d ready-free and $d \boxtimes c'$. $\gamma \xrightarrow{\mathbf{A} \text{ says } \mathbf{a}} \mathbf{A} \text{ says } c' \mid \mathbf{B} \text{ says } d = \gamma'$. Clearly $\alpha_{\mathbf{A}}(\gamma) = \tilde{c}$ and $\alpha_{\mathbf{A}}(\gamma') = \tilde{c}'$.

For (2) we already know, by items (2) and (3) of Lemma: B.6, that thesis holds, and in particular $\gamma = \gamma_{\mathbf{B}}(\tilde{c})$. □

Lemma B.8. For all substitutions σ , $(\alpha_{\mathbf{A}}(S))\sigma = \alpha_{\mathbf{A}}(S\sigma)$.

Proof. Straightforward structural induction. □

Lemma B.9. For all systems S : $\text{fv}(\alpha_{\mathbf{A}}(S)) \cup \text{fn}(\alpha_{\mathbf{A}}(S)) \subseteq \text{fv}(S) \cup \text{fn}(S)$

Proof. We proceed by induction on the structure of S . There are the following cases:

- $\alpha_{\mathbf{A}}(S) = \mathbf{0}$. The thesis holds trivially.
- $\mathbf{A}[P]$. Obvious, by the fact that $\alpha_{\mathbf{A}}(\mathbf{A}[P]) = \mathbf{A}[P]$.
- $s[\mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d]$. We have $\alpha_{\mathbf{A}}(S) = s[c]$. Clearly u not free in S only if $u \neq s$. But this holds also for $\alpha_{\mathbf{A}}(S)$.

- $\{\downarrow_x c\}_A$. We have $\alpha_A(S) = \{\downarrow_x c\}$. Clearly u is not free in S only if $x \neq s$. But this holds also for $\alpha_A(S)$.
- $S' \mid S''$. We have $\alpha_A(S) = \alpha_A(S') \mid \alpha_A(S'')$. By induction hypothesis we have that u not free in $S' \implies u$ not free in $\alpha_A(S') \wedge u$ not free in $S'' \implies u$ not free in $\alpha_A(S'')$. Assume that u is not free in S , so u must be not free in S' and in S'' , but then, by induction hypothesis, we have that u is not free in $\alpha_A(S')$ and in $\alpha_A(S'')$, which implies u not free in $\alpha_A(S)$.
- $(u')S'$. We have $\alpha_A(S) = (u')(\alpha_A(S'))$. Under the assumption that u is not free in S , we have that $u = u'$ or u is not free in S' . If the first holds we are done. If u is not free in S' the thesis follows by induction hypothesis. \square

Lemma B.10. *Let $\Gamma_{\tilde{c}} = \{\gamma \mid \alpha_A(\gamma) = \tilde{c}\}$. Then, for all $a \in \mathbf{A} \cup \{\varepsilon\}$, and for all \tilde{c} :*

$$\text{Paths}((a, \tilde{c})) = \bigcup_{\gamma \in \Gamma_{\tilde{c}}} \text{Paths}((a, \gamma))$$

Proof. \supseteq : Suppose that:

$$a_0 a_1 \dots \in \bigcup_{\gamma \in \Gamma_{\tilde{c}}} \text{Paths}((a, \gamma))$$

i.e.:

$$\exists \gamma \in \Gamma_{\tilde{c}}. a_0 a_1 \dots \in \text{Paths}((a, \gamma))$$

Let $s_0 = (a_0, \gamma_0)$, with $a_0 = a$ and $\gamma_0 = \gamma$. By Definition 5.10, $L(s_i) = a_i$ and $s_{i-1} \rightarrow s_i$, for all i . For all i , $\tilde{s}_i = (a_i, \alpha_A(\gamma_i))$. $L(s_i) = L(\tilde{s}_i)$. We have to show that $\tilde{s}_i \rightarrow \tilde{s}_{i+1}$, i.e., by Definition 5.9, $\alpha_A(\gamma_i) \xrightarrow{\mu} \alpha_A(\gamma_{i+1})$ with $\tilde{\mu} = a_{i+1}$ or $\tilde{\mu} = \text{ctx} : a_{i+1}$. Since $s_i \rightarrow s_{i+1}$ holds by hypothesis, by Definition 5.8 we have that $\gamma_i \xrightarrow{\mu} \gamma_{i+1}$, with $\mu = \mathbf{A}$ says a_{i+1} or $\mu = \mathbf{B}$ says a_{i+1} . But then, by Lemma B.4, the thesis follows. In case of finite paths, the proof above is still valid, except that $s_n \not\rightarrow$ for some n , and we have to show that $\tilde{s}_n \not\rightarrow$. By Definition 5.8, we have that $\gamma_n \not\rightarrow$, which is only possible when $\gamma_n = \mathbf{A}$ says $0 \mid \mathbf{B}$ says 0 . But then $\alpha_A(\gamma_n) = 0$ and, clearly, $(a, 0) \not\rightarrow$.

\subseteq : Suppose that $a_0 a_1 \dots \in \text{Paths}((a, \tilde{c}))$. Let $\tilde{s}_0 = (a, \tilde{c})$. By Definition 5.10, $L(\tilde{s}_i) = a_i$ and $\tilde{s}_{i-1} \rightarrow \tilde{s}_i$ for all i . Let $s_i = (a_i, \gamma_B(\tilde{c}_i))$, for all i . Clearly, $L(s_i) = L(\tilde{s}_i)$. We have to show that, for all $i > 0$, $\gamma_B(\tilde{c}_{i-1}) \xrightarrow{\mu} \gamma_B(\tilde{c}_i)$, with $\mu = \mathbf{A}$ says a_i or \mathbf{B} says a_i . This follows by Definition 5.9 and items (1) and (2) of Lemma B.6. In case of finite paths, we have to show that $(a_n, \tilde{c}_n) \not\rightarrow \implies (a_n, \gamma_B(\tilde{c}_n)) \not\rightarrow$. By Definitions 5.9 and 5.8 it holds when $\tilde{c}_n \not\rightarrow_{\mathbf{A}} \implies \gamma_B(\tilde{c}_n) \not\rightarrow$. But this follows by item (3) of Lemma B.6 and Lemma B.4. \square

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

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

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

Proof. (1) \implies : Suppose $\tilde{c} \vdash \phi$. By the definition of \vdash follows that $\forall \pi \in \text{Paths}(\varepsilon, \tilde{c}). \pi \vdash \phi$. Let γ be a contract such that $\alpha_A(\gamma) = \tilde{c}$. Suppose $\pi \in \text{Paths}(\varepsilon, \gamma)$. Thanks to Lemma B.10 we can conclude that $\pi \in \text{Paths}(\varepsilon, \tilde{c})$ and then $\pi \vdash \phi$.

\impliedby : Suppose $\tilde{c} \not\vdash \phi$. By the definition of \vdash follows that $\exists \pi \in \text{Paths}(\varepsilon, \tilde{c}). \pi \not\vdash \phi$. By Lemma B.10 we know that $\exists \gamma. \pi \in \text{Paths}(\varepsilon, \gamma)$. But then $\gamma \not\vdash \phi$.

(2) Assume that $\exists \gamma. \alpha_A(\gamma) = \tilde{c} \wedge \gamma \vdash \phi$. Then also holds that $\exists \gamma. \alpha_A(\gamma) = \tilde{c} \wedge \gamma \not\vdash \neg \phi$. By item (1) we obtain the thesis. \square

Lemma B.11. For all A-safe systems S , whenever $S \equiv (x_1, \dots, x_n, \vec{s})(S_A \mid S_{ctx})$:

$$x_i \text{ is free in } S_A \implies x_i \text{ is not free in } S_{ctx}$$

Proof. Suppose, by contradiction, that there exists i such that x_i is free in both S_A and S_{ctx} . Then no rule of Table: 2 allows to put S in the form $(\vec{s})(S'_A \mid S'_{ctx})$, and so S is not A-safe. \square

Lemma B.12. For all systems S, S' , and for all labels μ :

$$(1) S \xrightarrow{\mu} S' \implies \exists \tilde{\mu}. \alpha_A(S) \xrightarrow{\tilde{\mu}}_A \alpha_A(S')$$

(2) furthermore, if $\mu = A : \text{do}_s \mathbf{a}$ or $(\mu = A : \pi \wedge S \text{ ask-free})$ then $\mu = \tilde{\mu}$.

Proof. By induction on the depth of the derivation of $S \xrightarrow{\mu} S'$. According to the last rule used in such derivation, we have the following exhaustive cases:

[TAU] We have two subcases, according to which participant has moved:

- $S \equiv A[\tau.P + P' \mid Q] \xrightarrow{A: \tau} A[P \mid Q] = S'$. Both the theses (1) and (2) follow because $\alpha_A(S) = S$, $\alpha_A(S') = S'$, and $S \xrightarrow{A: \tau}_A S'$ by rule $[\alpha\text{-TAU}]$.
- $S \equiv B[\tau.P + P' \mid Q] \xrightarrow{B: \tau} B[P \mid Q] = S'$. The thesis (1) follows because, by rule $[\alpha\text{-CTX}]$, $\alpha_A(S) = \mathbf{0} \xrightarrow{ctx}_A \mathbf{0} = \alpha_A(S')$. The thesis (2) follows trivially.

[TELL] We have two subcases, according to which participant has moved:

- $S \equiv A[\text{tell } \downarrow_x c.P + P' \mid Q] \xrightarrow{A: \text{tell } \downarrow_x c} A[P \mid Q] \mid \{\downarrow_x c\}_A = S'$. Both the theses (1) and (2) follow because $\alpha_A(S) = S$, $\alpha_A(S') = S'$, and $S \xrightarrow{A: \text{tell } \downarrow_x c}_A S'$ by rule $[\alpha\text{-TELL}]$.
- $S \equiv B[\text{tell } \downarrow_x c.P + P' \mid Q] \xrightarrow{B: \text{tell } \downarrow_x c} B[P \mid Q] \mid \{\downarrow_x c\}_B = S'$. The thesis (1) follows because, by rule $[\alpha\text{-CTX}]$, $\alpha_A(S) = \mathbf{0} \xrightarrow{ctx}_A \mathbf{0} = \alpha_A(S')$. The thesis (2) follows trivially.

[DO] We have three subcases, the first for A moves, the others for context moves:

- $S \equiv s[\gamma] \mid A[\text{do}_s \mathbf{a}.P + P' \mid Q] \xrightarrow{A: \text{do}_s \mathbf{a}} s[\gamma'] \mid A[P \mid Q] = S'$, where $\gamma \xrightarrow{A \text{ says } \mathbf{a}} \gamma'$. By item (1) of Lemma B.4 we have $\alpha_A(\gamma) \xrightarrow{\mathbf{a}}_A \alpha_A(\gamma')$. Therefore both the theses (1) and (2) follow because $\alpha_A(S) = s[\alpha_A(\gamma)] \mid A[\text{do}_s \mathbf{a}.P + P' \mid Q] \xrightarrow{A: \text{do}_s \mathbf{a}}_A s[\alpha_A(\gamma')] \mid A[P \mid Q] = \tilde{S}' = \alpha_A(S')$, by rule $[\alpha\text{-DO}]$.
- $S \equiv s[\gamma] \mid B[\text{do}_s \mathbf{a}.P + P' \mid Q] \xrightarrow{B: \text{do}_s \mathbf{a}} s[\gamma'] \mid B[P \mid Q] = S'$, where $\gamma \xrightarrow{B \text{ says } \mathbf{a}} \gamma'$ and $\gamma = B \text{ says } c \mid B' \text{ says } d$. The thesis (1) follows because $\alpha_A(S) = \mathbf{0} \xrightarrow{ctx}_A \mathbf{0} = \alpha_A(S')$, by rule $[\alpha\text{-CTX}]$. The thesis (2) holds trivially.
- $S \equiv s[\gamma] \mid B[\text{do}_s \mathbf{a}.P + P' \mid Q] \xrightarrow{B: \text{do}_s \mathbf{a}} s[\gamma'] \mid B[P \mid Q] = S'$, where $\gamma \xrightarrow{B \text{ says } \mathbf{a}} \gamma'$ and $\gamma = A \text{ says } c \mid B \text{ says } d$. By item (2) of Lemma B.4, $\alpha_A(\gamma) \xrightarrow{ctx: \mathbf{a}}_A \alpha_A(\gamma')$. The thesis (1) follows because $\alpha_A(S) = s[\alpha_A(\gamma)] \xrightarrow{ctx}_A s[\alpha_A(\gamma')] = \tilde{S}' = \alpha_A(S')$, by rule $[\alpha\text{-DOCTX}]$. The thesis (2) holds trivially.

[DEL] We have two subcases, according to which participant has moved:

- $S \equiv (u)S_0 \xrightarrow{A: \pi} (u)S'_0 = S'$, with $S_0 \xrightarrow{A: \pi'} S'_0$ and $\pi = \text{del}_u(\pi')$. By the induction hypothesis, there exists $\tilde{\mu}'$ such that $\alpha_A(S_0) \xrightarrow{\tilde{\mu}'}_A \alpha_A(S'_0)$. There are two further subcases, according to the form of the label π :

- $\pi = \text{do}_s \mathbf{a}$. Then it must be $\pi = \pi'$. By the induction hypothesis of item (2), it follows that $\tilde{\mu}' = \mathbf{A} : \pi'$. Then, by rule $[\alpha\text{-DEL}]$:

$$\frac{\alpha_{\mathbf{A}}(S_0) \xrightarrow{\mathbf{A} : \pi'}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'_0)}{(u)\alpha_{\mathbf{A}}(S_0) \xrightarrow{\tilde{\mu}}_{\mathbf{A}} (u)\alpha_{\mathbf{A}}(S'_0)} \text{ where } \tilde{\mu} = \mathbf{A} : \text{del}_u(\pi')$$

The thesis (1) follows because $\alpha_{\mathbf{A}}(S) = (\vec{u})\alpha_{\mathbf{A}}(S_0)$ and $\alpha_{\mathbf{A}}(S') = (\vec{u})\alpha_{\mathbf{A}}(S'_0)$. The thesis (2) follows because $\tilde{\mu} = \mathbf{A} : \text{del}_u(\pi') = \mathbf{A} : \text{del}_u(\pi) = \mu$.

- $\mu = \mathbf{A} : \pi$ and S ask-free. Similar to the previous case.
- Otherwise, by rule $[\alpha\text{-DELCtx}]$:

$$\frac{\alpha_{\mathbf{A}}(S_0) \xrightarrow{ctx}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'_0)}{(u)\alpha_{\mathbf{A}}(S_0) \xrightarrow{ctx}_{\mathbf{A}} (u)\alpha_{\mathbf{A}}(S'_0)}$$

The thesis (1) follows because $\alpha_{\mathbf{A}}(S) = (\vec{u})\alpha_{\mathbf{A}}(S_0)$ and $\alpha_{\mathbf{A}}(S') = (\vec{u})\alpha_{\mathbf{A}}(S'_0)$. The thesis (2) follows trivially.

- $S \equiv (u)S_0 \xrightarrow{\mathbf{B} : \pi}_{\mathbf{B}} (u)S'_0 = S'$. Then $S_0 \xrightarrow{\mathbf{B} : \pi'}_{\mathbf{B}} S'_0$, with $\pi = \text{del}_u(\pi')$. By the induction hypothesis $\alpha_{\mathbf{A}}(S_0) \xrightarrow{ctx}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'_0)$, so $\alpha_{\mathbf{A}}(S) \xrightarrow{ctx}_{\mathbf{A}} \alpha_{\mathbf{A}}(S')$.

$[\text{PAR}]$ Let $S \equiv S'' \mid S'''$. Assuming that $S'' \xrightarrow{\mu}_{\mathbf{A}} S''''$ we have $S \xrightarrow{\mu}_{\mathbf{A}} S'''' \mid S''' = S'$, and by induction hypothesis for thesis (1), we have:

$$\frac{\alpha_{\mathbf{A}}(S'') \xrightarrow{\tilde{\mu}}_{\mathbf{A}} \alpha_{\mathbf{A}}(S''')}{\alpha_{\mathbf{A}}(S'') \mid \alpha_{\mathbf{A}}(S''') \xrightarrow{\tilde{\mu}}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'') \mid \alpha_{\mathbf{A}}(S''')} \quad [\alpha\text{-PAR}]$$

Using the induction hypothesis for thesis (2):

$$\frac{\alpha_{\mathbf{A}}(S'') \xrightarrow{\mathbf{A} : \pi}_{\mathbf{A}} \alpha_{\mathbf{A}}(S''')}{\alpha_{\mathbf{A}}(S'') \mid \alpha_{\mathbf{A}}(S''') \xrightarrow{\mathbf{A} : \pi}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'') \mid \alpha_{\mathbf{A}}(S''')} \quad [\alpha\text{-PAR}]$$

$[\text{DEF}]$ We can have two subcases:

- Let $X(\vec{u}) \stackrel{\text{def}}{=} P$, $S \equiv \mathbf{A}[X(\vec{v}) \mid Q] \mid S''$, and assume that $\mathbf{A}[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'' \xrightarrow{\mathbf{A} : \pi}_{\mathbf{A}} S'$. Both the theses (1) and (2) hold, assuming they hold in the premise of the rule:
$$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \quad \mathbf{A}[P\{\vec{v}/\vec{u}\} \mid Q] \mid \tilde{S} \xrightarrow{\mathbf{A} : \pi}_{\mathbf{A}} \tilde{S}'}{\mathbf{A}[X(\vec{v}) \mid Q] \mid \tilde{S} \xrightarrow{\mathbf{A} : \pi}_{\mathbf{A}} \tilde{S}'} \quad [\alpha\text{-DEF}].$$
- Let $X(\vec{u}) \stackrel{\text{def}}{=} P$, $S \equiv \mathbf{B}[X(\vec{v}) \mid Q] \mid S''$, and assume that $\mathbf{B}[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'' \xrightarrow{\mu}_{\mathbf{B}} S'$. By the induction hypothesis $\alpha_{\mathbf{A}}(\mathbf{B}[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'') = \alpha_{\mathbf{A}}(S'') \xrightarrow{\tilde{\mu}}_{\mathbf{A}} \alpha_{\mathbf{A}}(S')$. But then both the theses (1) and (2) hold: $\alpha_{\mathbf{A}}(S) = \alpha_{\mathbf{A}}(S'') \xrightarrow{\tilde{\mu}}_{\mathbf{A}} \alpha_{\mathbf{A}}(S'') = \alpha_{\mathbf{A}}(S')$.

$[\text{FUSE}]$ There are two cases:

- Let $S \equiv (x, y)(S_{\mathbf{A}} \mid S_{ctx})$, with $S_{ctx} \equiv S'_{ctx} \mid \{\downarrow_x c\}_{\mathbf{B}} \mid \{\downarrow_y d\}_{\mathbf{B}'}$. Suppose:

$$\frac{c \bowtie d \quad \gamma = \mathbf{B} \text{ says } c \mid \mathbf{B}' \text{ says } d \quad \sigma = \{s/x, y\} \quad s \text{ fresh}}{S \xrightarrow{\mathbf{K} : \text{fuse}} (s)((S_{\mathbf{A}} \mid S'_{ctx})\sigma \mid s[\gamma]) = S'} \quad [\text{FUSE}]$$

By Lemma: B.11 both x and y are not free in $S_{\mathbf{A}}$, and then $S' \equiv (s)(S_{\mathbf{A}} \mid S_{ctx}\sigma \mid s[\gamma])$.
By rule $[\alpha\text{-Fuse}]$: $\alpha_{\mathbf{A}}(S) = \alpha_{\mathbf{A}}(S_{\mathbf{A}}) \xrightarrow{ctx}_{\mathbf{A}} \alpha_{\mathbf{A}}(S_{\mathbf{A}}) = \tilde{S}'$.
 $\alpha_{\mathbf{A}}(S') = \alpha_{\mathbf{A}}((s)(S_{\mathbf{A}})) \equiv \alpha_{\mathbf{A}}(S_{\mathbf{A}}) = \tilde{S}'$.

ii. Let $S \equiv (x, y)(S_A \mid S_{ctx})$, with $S_A \equiv S'_A \mid \{\downarrow_x c\}_A$ and $S_{ctx} \equiv S'_{ctx} \mid \{\downarrow_y d\}_B$. Suppose:

$$\frac{c \bowtie d \quad \gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d \quad \sigma = \{s/x, y\} \quad s \text{ fresh}}{S \xrightarrow{\text{Fuse}} (s)((S'_A \mid S'_{ctx})\sigma \mid s[\gamma]) = S'} \text{[FUSE]}$$

By Lemma: B.11 y is not free in S_A and then $\alpha_A(S) = (x, y)(\alpha_A(S'_A) \mid \{\downarrow_x c\}_A) \equiv (x)(\alpha_A(S'_A) \mid \{\downarrow_x c\}_A)$. By rule $[\alpha\text{-Fuse}]$: $\alpha_A(S) \xrightarrow{ctx}_A (s)(\alpha_A(S'_A)\{s/x\} \mid s[c]) = \tilde{S}'$.

$$\begin{aligned} \alpha_A(S') &= \\ (s)(\alpha_A(S'_A)\sigma \mid s[c]) &\equiv \\ (s)(\alpha_A(S'_A)\{s/x\} \mid s[c]) &\equiv \quad \text{because } y \text{ is not free in } \alpha_A(S'_A) \\ (s)(\alpha_A(S'_A)\{s/x\} \mid s[c]) &= \quad \text{by Lemma: B.8} \\ \tilde{S}' & \end{aligned}$$

$[\text{ASK}]$ There are two cases, according to the participant which has moved:

- $S \equiv \mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathbf{A}: \text{ask}_s \phi} \mathbf{A}[P \mid Q] \mid s[\gamma]$. Let $\gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d$. By item (2) of Lemma 5.11, $c \not\vdash \neg\phi$. The thesis (1) holds because, by rule $[\alpha\text{-ASKCTX}]$:

$$\frac{c \not\vdash \neg\phi}{\alpha_A(S) = \mathbf{A}[\text{ask}_s \phi.P + P' \mid Q] \mid s[\alpha_A(\gamma)] \xrightarrow{ctx}_A \mathbf{A}[P \mid Q] \mid s[\alpha_A(\gamma)]}$$

The thesis (2) follows trivially.

- $S \equiv \mathbf{B}[\text{ask}_s \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathbf{B}: \text{ask}_s \phi} \mathbf{B}[P \mid Q] \mid s[\gamma]$. The thesis (1) follows by rule $[\alpha\text{-CTX}]$:

$$\alpha_A(S) \xrightarrow{ctx}_A \alpha_A(S) = \alpha_A(S')$$

The thesis (2) follows trivially. □

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

$$S \xrightarrow{\eta}^* S' \implies \exists \tilde{\eta} : \alpha_A(S) \xrightarrow{\tilde{\eta}}_A^* \alpha_A(S')$$

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

Proof. We proceed by induction on the length of the derivation. The base case holds trivially. For the inductive case, suppose that $S \xrightarrow{\eta'}^n S''$, for $n > 0$. By the induction hypothesis, we have:

$$\exists \tilde{\eta}' : \alpha_A(S) \xrightarrow{\tilde{\eta}'}_A^n \alpha_A(S'') \quad (4)$$

$$\eta' \text{ A-solo and } S \text{ ask-free} \implies \eta' = \tilde{\eta}' \quad (5)$$

Now, assume that $S'' \xrightarrow{\mu} S'$, and let $\eta = \eta' \mu$. By Lemma B.12, it follows that:

$$\exists \tilde{\mu} : \alpha_A(S'') \xrightarrow{\tilde{\mu}}_A \alpha_A(S') \quad (6)$$

$$\mu = \mathbf{A} : \pi \wedge S \text{ ask-free} \implies \mu = \tilde{\mu} \quad (7)$$

By (4) and (6), it follows that:

$$\alpha_A(S) \xrightarrow{\tilde{\eta}}_A^n \alpha_A(S'') \xrightarrow{\tilde{\mu}}_A \alpha_A(S')$$

For the “furthermore” part, let $\tilde{\eta} = \tilde{\eta}' \tilde{\mu}$, and assume that η is A-solo and S is ask-free. Therefore, by (5) and (7) we conclude that: $\eta = \eta' \mu = \tilde{\eta}' \tilde{\mu} = \tilde{\eta}$. □

Lemma B.13. $\alpha_A(S) \xrightarrow{A: \pi}_A \tilde{S}' \implies \exists S' \text{ A-safe} . S \xrightarrow{A: \pi} S' \wedge \alpha_A(S') = \tilde{S}'$

Proof. Let S_{ctx} be an A-free system. By rule induction:

[α -TAU] Let $S \equiv A[\tau.P + P' \mid Q] \mid S_{ctx}$, and let $\alpha_A(S) \equiv A[\tau.P + P' \mid Q] \xrightarrow{A: \tau}_A A[P \mid Q] = \tilde{S}'$. We have $S \xrightarrow{A: \tau} A[P \mid Q] \mid S_{ctx} = S' = \alpha_A(S') = \tilde{S}'$.

[α -TELL] Let $S \equiv A[\text{tell } \downarrow_x c.P + P' \mid Q] \mid S_{ctx}$, and also let $\alpha_A(S) \equiv A[\text{tell } \downarrow_x c.P + P' \mid Q] \xrightarrow{A: \text{tell } \downarrow_x c}_A A[P \mid Q] \mid \{\downarrow_x c\}_A = \tilde{S}'$. We have $S \xrightarrow{A: \text{tell } \downarrow_x c} A[P \mid Q] \mid \{\downarrow_x c\}_A \mid S_{ctx} = S' = \alpha_A(S') = \tilde{S}'$.

[α -DO] Let $\alpha_A(\gamma) \xrightarrow{a}_A \tilde{c}'$ — which, by item (1) of Lemma B.7, implies $\gamma \xrightarrow{A \text{ says } a} \gamma'$, with $\alpha_A(\gamma') = \tilde{c}'$. Furthermore, let $S \equiv s[\gamma] \mid A[\text{do}_s a.P + P' \mid Q] \mid S_{ctx}$, and $S \xrightarrow{A: \text{do}_s a} s[\gamma'] \mid A[P \mid Q] \mid S_{ctx} = S'$. We have $\alpha_A(S) = s[c] \mid A[\text{do}_s a.P + P' \mid Q] \xrightarrow{A: \text{do}_s a}_A s[c'] \mid A[P \mid Q] = \alpha_A(S') = \tilde{S}'$.

[α -ASK] We have $S \equiv s[\gamma] \mid A[\text{ask}_s \phi.P + P' \mid Q] \mid S_{ctx}$, $c \vdash \phi$ and:

$$\alpha_A(S) = s[c] \mid A[\text{ask}_s \phi.P + P' \mid Q] \xrightarrow{A: \text{ask}_s \phi}_A s[c'] \mid A[P \mid Q] = \alpha_A(S') = \tilde{S}'$$

By item (1) of Lemma 5.11, since $c \vdash \phi$ and $\alpha_A(\gamma) = c$, then $\gamma \vdash c$. Then, by rule [ASK]:

$$S \xrightarrow{A: \text{ask}_s \phi} s[\gamma] \mid A[P \mid Q] \mid S_{ctx} = S'$$

[α -DEL] Let $S \equiv (u)S''$ and $S \xrightarrow{A: \pi}_A (u)S''' = S'$. Also, let $\alpha_A(S) \equiv (u)\alpha_A(S'')$. Assuming $\alpha_A(S'') \xrightarrow{A: \pi'}_A \tilde{S}'''$, we have, by induction hypothesis, $S'' \xrightarrow{A: \pi'}_A S'''$ and $\alpha_A(S''') = \tilde{S}'''$ (note that π and π' may differ, depending on whether $u \in \text{fv}(\pi')$). But then $\alpha_A(S) \xrightarrow{A: \pi}_A (u)\alpha_A(S''') = \tilde{S}'$. Clearly $\alpha_A(S') = \tilde{S}'$.

[α -PAR] Let $S \equiv S'' \mid S'''$ and $S'' \xrightarrow{A: \pi}_A S''''$ — and therefore, $S \xrightarrow{A: \pi}_A S'''' \mid S''' = S'$. We have $\alpha_A(S) \equiv \alpha_A(S'') \mid \alpha_A(S''')$. Assuming that $\alpha_A(S'') \xrightarrow{A: \pi}_A \tilde{S}''''$ we have, by induction hypothesis, $\alpha_A(S''') = \tilde{S}''''$. But then $\alpha_A(S) \xrightarrow{A: \pi}_A \alpha_A(S'''' \mid S''') = \tilde{S}'$. Clearly $\alpha_A(S') = \tilde{S}'$.

[α -DEF] Let $X(\vec{u}) \stackrel{\text{def}}{=} P$. Also, let $S \equiv A[X(\vec{v}) \mid Q] \mid S''$ and $A[P\{\vec{v}/\vec{u}\} \mid Q] \mid S'' \xrightarrow{A: \pi}_A S'$. We have $\alpha_A(S) \equiv A[X(\vec{v}) \mid Q] \mid \alpha_A(S'')$. Suppose $A[P\{\vec{v}/\vec{u}\} \mid Q] \mid \alpha_A(S'') \xrightarrow{A: \pi}_A \tilde{S}'$. By applying the induction hypothesis, we have $\alpha_A(S') = \tilde{S}'$. Then the thesis follows trivially. \square

Lemma B.14. For all A-safe systems S , and for all A-solo traces η :

$$\alpha_A(S) \xrightarrow{\eta}_A \tilde{S}' \implies \exists S' \text{ A-safe} . S \xrightarrow{\eta} S' \wedge \alpha_A(S') = \tilde{S}'$$

Proof. By induction on the length of the derivation. Base case holds trivially. Suppose that $\alpha_A(S) \xrightarrow{\eta'}_A \tilde{S}''$, with η' A-solo. By inductive hypothesis $S \xrightarrow{\eta'} S'' \wedge \alpha_A(S'') = \tilde{S}''$. Let $\alpha_A(S'') \xrightarrow{A: \pi}_A \tilde{S}'$. By Lemma B.13 it trivially follows that $\exists S' \text{ A-safe} . S'' \xrightarrow{A: \pi} S' \wedge \alpha_A(S') = \tilde{S}'$. \square

Lemma B.15. For all A-safe systems S' , and for all ask-free abstract systems \tilde{S} :

$$\tilde{S} \rightarrow_A \tilde{S}' \wedge \alpha_A(S') = \tilde{S}' \implies \exists S \text{ A-safe} . S \rightarrow S' \wedge \alpha_A(S) = \tilde{S}$$

Proof. In the following S_{ctx} is a generic \mathbf{A} -free system. We show that lemma holds by rule induction:

[α -TAU] Let $\tilde{S} = \mathbf{A}[\tau.P + P' \mid Q] \rightarrow_{\mathbf{A}} \mathbf{A}[P \mid Q] = \tilde{S}'$. S' must be in the form $\mathbf{A}[P \mid Q] \mid S_{ctx}$.
Then $S = \mathbf{A}[\tau.P + P' \mid Q] \mid S_{ctx}$.

[α -TELL] Let $\tilde{S} = \mathbf{A}[\text{tell}_x c.P + P' \mid Q] \rightarrow_{\mathbf{A}} \mathbf{A}[P \mid Q] \mid \{\downarrow_x c\}_{\mathbf{A}} = \tilde{S}'$. S' must be in the form $\mathbf{A}[P \mid Q] \mid \{\downarrow_x c\}_{\mathbf{A}} \mid S_{ctx}$. Then $S = \mathbf{A}[\text{tell}_x c.P + P' \mid Q] \mid S_{ctx}$.

[α -DO] Let \tilde{c}, \tilde{c}' be abstract contracts such that $\tilde{c} \stackrel{\mathbf{a}}{\rightarrow}_{\mathbf{A}} \tilde{c}'$, and let $\tilde{S} = s[\tilde{c}]$ and $\tilde{S}' = s[\tilde{c}']$. We have that $\tilde{S} \rightarrow_{\mathbf{A}} s[\tilde{c}'] = \tilde{S}'$. Then $S' = s[\gamma'] \mid S_{ctx}$, with $\alpha_{\mathbf{A}}(\gamma') = \tilde{c}'$. Let γ be a bilateral contract such that $\alpha_{\mathbf{A}}(\gamma) = \tilde{c}$. By Lemma B.7 $\gamma \rightarrow \gamma'$. So $S = s[\gamma] \mid S_{ctx} \rightarrow s[\gamma'] \mid S_{ctx}$.

[α -FUSE] Suppose: $\frac{s \text{ fresh}}{\tilde{S} = (x)(\tilde{S}_0 \mid \{\downarrow_x c\}_{\mathbf{A}}) \xrightarrow{ctx}_{\mathbf{A}} (s)(s[c] \mid \tilde{S}_0\{s/x\}) = \tilde{S}'}$ [α -FUSE].

S' must be in the form $(s')(s'[c] \mid S'_A \mid S'_{ctx})$, with s' not free in \tilde{S}_0 , $\gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d$ and $\alpha_{\mathbf{A}}(S'_A) = \tilde{S}_0\{s'/x\}$.

$$\begin{aligned} \alpha_{\mathbf{A}}(S') &= \\ (s')(s'[c] \mid \tilde{S}_0\{s'/x\}) &= \\ (s)(s[c] \mid (\tilde{S}_0\{s'/x\})\{s'/s'\}) &= \quad s \text{ is fresh then trivially not free in } \tilde{S}_0 \\ (s)(s[c] \mid (\tilde{S}_0\{s'/s'\})\{s/x\}) &= \\ (s)(s[c] \mid \tilde{S}_0\{s/x\}) &= \quad s' \text{ is not free in } \tilde{S}_0 \\ &= \tilde{S}'. \end{aligned}$$

S can be as follows: $S = (x, y)(S_A \mid \{\downarrow_x c\}_{\mathbf{A}} \mid S_{ctx} \mid \{\downarrow_y d\}_{\mathbf{B}})$, with $\alpha_{\mathbf{A}}(S_A) = \tilde{S}_0$, y not free in both S_A and S'_{ctx} , $S_{ctx} = S'_{ctx}\{y/s'\}$, x not free in S_{ctx} and s' fresh.

$$\frac{c \bowtie d \quad \gamma = \mathbf{A} \text{ says } c \mid \mathbf{B} \text{ says } d \quad \sigma = \{s'/x, y\} \quad s \text{ fresh}}{(x, y)(S_A \mid \{\downarrow_x c\}_{\mathbf{A}} \mid S_{ctx} \mid \{\downarrow_y d\}_{\mathbf{B}}) \xrightarrow{\text{K: fuse}} (s')((S_A \mid S_{ctx})\sigma \mid s'[\gamma])}$$
 [α -FUSE]

$$\begin{aligned} \alpha_{\mathbf{A}}(S) &= \\ (x, y)(\tilde{S}_0 \mid \{\downarrow_x c\}_{\mathbf{A}}) &= \\ (x)(\tilde{S}_0 \mid \{\downarrow_x c\}_{\mathbf{A}}) &= \quad \text{since } y \text{ not free in } S_A, \text{ by Lemma:B.9, } y \text{ is not free in } S_0 \\ &= \tilde{S}'. \end{aligned}$$

[α -DEF] Assume $\alpha_{\mathbf{A}}(S') = \tilde{S}'$ and suppose:

$$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \quad \mathbf{A}[P\{\vec{v}/\vec{u}\} \mid Q] \mid \tilde{S}_0 \rightarrow_{\mathbf{A}} \tilde{S}'}{\tilde{S} = \mathbf{A}[X(\vec{v}) \mid Q] \mid \tilde{S}_0 \rightarrow_{\mathbf{A}} \tilde{S}'}$$

Let S_0 be a system such that $\alpha_{\mathbf{A}}(S_0) = \tilde{S}_0$. Since the thesis holds, by induction hypothesis, for the premise of the rule above, we have:

$$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \quad \mathbf{A}[P\{\vec{v}/\vec{u}\} \mid Q] \mid S_0 \rightarrow S'}{S = \mathbf{A}[X(\vec{v}) \mid Q] \mid S_0 \rightarrow S'} \quad [\text{DEF}]$$

[α -PAR] Assume $\alpha_A(S_1) = \tilde{S}_1$, $\alpha_A(S_2) = \tilde{S}_2$ and suppose:

$$\frac{\tilde{S}_0 \rightarrow \tilde{S}_1}{\tilde{S} = \tilde{S}_0 \mid \tilde{S}_2 \rightarrow_A \tilde{S}_1 \mid \tilde{S}_2}$$

By induction hypothesis $\alpha_A(S_0) = \tilde{S}_0$. Then we have:

$$\frac{S_0 \rightarrow S_1}{S = S_0 \mid S_2 \rightarrow S_1 \mid S_2} \text{ [PAR]}$$

[α -DEL][α -DELCtx] These cases can be treated together, since we are ignoring labels. Assume $\alpha_A(S_1) = \tilde{S}_1$ and suppose:

$$\frac{\tilde{S}_0 \rightarrow_A \tilde{S}_1}{(u)\tilde{S}_0 \rightarrow_A (u)\tilde{S}_1}$$

By induction hypothesis $\alpha_A(S_0) = \tilde{S}_0$ and:

$$\frac{S_0 \rightarrow S_1}{(u)S_0 \rightarrow (u)S_1}$$

[α -DoCtx] Let \tilde{c}, \tilde{c}' be abstract contracts such that $\tilde{c} \xrightarrow{ctx:a} \tilde{c}'$, and let $\tilde{S} = s[\tilde{c}]$ and $\tilde{S}' = s[\tilde{c}']$. We have that $\tilde{S} \rightarrow_A s[\tilde{c}'] = \tilde{S}'$. Then $S' = s[\gamma'] \mid S_{ctx}$, with $\alpha_A(\gamma') = \tilde{c}'$. By Lemma B.7 we have that there exists a γ such that $\alpha_A(\gamma) = \tilde{c}$ and $\gamma \rightarrow \gamma'$. So $S = s[\gamma] \mid S_{ctx} \rightarrow s[\gamma'] \mid S_{ctx}$.

[α -Ctx] Suppose $\tilde{S} \xrightarrow{\mu} \tilde{S}'$. Must be: $S' = S_A \mid S'_{ctx}$. Clearly $\alpha_A(S') = \tilde{S}'$. Then $S = S_A \mid S_{ctx}$, with $S_{ctx} = S'_{ctx} \mid B[\tau]$ such that B do not appear in S'_{ctx} . Clearly $S \xrightarrow{ctx} S'$.

□

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

$$\tilde{S} \rightarrow_A^* \tilde{S}' \implies \exists S, S' \text{ A-safe. } \alpha_A(S) = \tilde{S} \wedge S \rightarrow^* S' \wedge \alpha_A(S') = \tilde{S}'$$

Proof. By induction on the length n of the derivation. For $n = 0$ (base case), the thesis holds trivially. For the inductive step, suppose that $\tilde{S} \rightarrow_A \tilde{S}'' \rightarrow_A^n \tilde{S}'$. By the induction hypothesis, there exist S'', S' A-safe such that $\alpha_A(S'') = \tilde{S}''$, $\alpha_A(S') = \tilde{S}'$, and $S'' \rightarrow^* S'$. Since $\tilde{S} \rightarrow_A \tilde{S}''$, Lemma B.15 guarantees that there exists S A-safe such that $\alpha_A(S) = \tilde{S}$ and $S \rightarrow^* S''$. Summing up, we conclude that $S \rightarrow^* S'$. □

Lemma B.16. *For all A-safe S , and for all s :*

- (1) $O_s^A(S) = O_s^A(\alpha_A(S))$,
- (2) $RD_s^A(S) = RD_s^A(\alpha_A(S))$,
- (3) $WRD_s^A(\alpha_A(S)) \subseteq WRD_s^A(S)$.
- (4) $WRD_s^A(\alpha_A(S)) \supseteq WRD_s^A(S)$ if S is ask-free.

Proof. For item (1), we have that $O_s^A(S) \subseteq O_s^A(\alpha_A(S))$ follows by item (1) of Lemma B.4. The reverse inclusion holds by item (1) of Lemma B.7.

For item (2), we first show the inclusion \supseteq . Assume that $\mathbf{a} \in \text{RD}_s^A(\alpha_A(S))$. By Def. B.3, there exists \tilde{S}' such that $\alpha_A(S) \xrightarrow{\mathbf{A}: \text{do}_s \mathbf{a}} \tilde{S}'$. By Lemma B.13, there exists S' such that $S \xrightarrow{\mathbf{A}: \text{do}_s \mathbf{a}} S'$ and $\alpha_A(S') = \tilde{S}'$. Then, by Def. 4.2, it follows that $\mathbf{a} \in \text{RD}_s^A(S)$. For the inclusion \subseteq , assume that $\mathbf{a} \in \text{RD}_s^A(S)$. By Def. 4.2, there exists S' such that $S \xrightarrow{\mathbf{A}: \text{do}_s \mathbf{a}} S'$. By Lemma B.12 it follows that $\alpha_A(S) \xrightarrow{\mathbf{A}: \text{do}_s \mathbf{a}} \alpha_A(S')$. Therefore, by Def. B.3 we conclude that $\mathbf{a} \in \text{RD}_s^A(\alpha_A(S))$.

For item (3), assume that $\mathbf{a} \in \text{WRD}_s^A(\alpha_A(S))$. By Def. B.3, there exists \tilde{S}' such that $\alpha_A(S) \xrightarrow{\eta} \tilde{S}'$, where $\mathbf{a} \in \text{RD}_s^A(\tilde{S}')$, and η is an \mathbf{A} -solo trace without do_s . By Lemma B.14, there exists S' such that $S \xrightarrow{\eta} S'$ and $\alpha_A(S') = \tilde{S}'$. By Lemma B.16(2), $\text{RD}_s^A(\tilde{S}') = \text{RD}_s^A(S')$. Therefore, by Def. 4.2 we conclude that $\mathbf{a} \in \text{WRD}_s^A(S)$.

For item (4), assume that $\mathbf{a} \in \text{WRD}_s^A(S)$. By Def. 4.2, there exists S' such that $S \xrightarrow{\eta} S'$, where $\mathbf{a} \in \text{RD}_s^A(S')$, and η is an \mathbf{A} -solo trace without do_s . By Theorem 5.4, there exists a trace $\tilde{\eta}$ such that $\alpha_A(S) \xrightarrow{\tilde{\eta}} \alpha_A(S')$. Furthermore, since S is ask-free, then $\tilde{\eta} = \eta$. By Lemma B.16(2), $\text{RD}_s^A(\alpha_A(S')) = \text{RD}_s^A(S')$. Therefore, by Def. B.3 we conclude that $\mathbf{a} \in \text{WRD}_s^A(S)$. \square

Lemma B.17. *For all \mathbf{A} -safe S :*

- (1) *if \mathbf{A} is α -ready in $\alpha_A(S)$, then \mathbf{A} is ready in S .*
- (2) *if \mathbf{A} is ready in S and S is ask-free, then \mathbf{A} is α -ready in $\alpha_A(S)$.*

Proof. For item (1), suppose that \mathbf{A} is α -ready in $\tilde{S} = \alpha_A(S)$, and assume that $S \equiv (\bar{u})S_0$. By Def. 4.3, we have to show that $\text{WRD}_s^A(S_0) \cap O_s^A(S_0) \neq \emptyset$ whenever $O_s^A(S_0) \neq \emptyset$. Suppose that $O_s^A(S_0) \neq \emptyset$. Then, by Lemma B.16(1), $O_s^A(\alpha_A(S_0)) \neq \emptyset$. We have that:

$$\begin{aligned} \text{WRD}_s^A(S_0) \cap O_s^A(S_0) &= \text{WRD}_s^A(S_0) \cap O_s^A(\alpha_A(S_0)) && \text{by Lemma B.16(1)} \\ &\supseteq \text{WRD}_s^A(\alpha_A(S_0)) \cap O_s^A(\alpha_A(S_0)) && \text{by Lemma B.16(3)} \\ &\neq \emptyset && \text{since } \mathbf{A} \text{ is } \alpha\text{-ready in } \alpha_A(S) \end{aligned}$$

For item (2), suppose that \mathbf{A} is ready in S , and assume that $\alpha_A(S) \equiv (\bar{u})\tilde{S}_0$. Then, by Def. 5.2, there exists S_0 such that $\tilde{S}_0 = \alpha_A(S_0)$ and $S = (\bar{u})S_0$. By Def. B.3, we have to show that $\text{WRD}_s^A(\tilde{S}_0) \cap O_s^A(\tilde{S}_0) \neq \emptyset$ whenever $O_s^A(\tilde{S}_0) \neq \emptyset$. Suppose that $O_s^A(\tilde{S}_0) \neq \emptyset$. Then, by Lemma B.16(1), $O_s^A(S_0) = O_s^A(\tilde{S}_0) \neq \emptyset$. We have that:

$$\begin{aligned} \text{WRD}_s^A(\tilde{S}_0) \cap O_s^A(\tilde{S}_0) &= \text{WRD}_s^A(\tilde{S}_0) \cap O_s^A(S_0) && \text{by Lemma B.16(1)} \\ &\supseteq \text{WRD}_s^A(S_0) \cap O_s^A(S_0) && \text{by Lemma B.16(4)} \\ &\neq \emptyset && \text{since } \mathbf{A} \text{ is ready in } S \quad \square \end{aligned}$$

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.*

Proof. For the first part, suppose that P is α -honest. Let S_{ctx} be an \mathbf{A} -free system, and assume that $\mathbf{A}[P] \mid S_{ctx} \rightarrow^* S$ for some S . Since $\mathbf{A}[P] \mid S_{ctx}$ is \mathbf{A} -safe, then by item (3) of Lemma B.2 it follows that S is \mathbf{A} -safe. Let $\tilde{S} = \alpha_A(S)$. By Theorem 5.4 it follows that $\mathbf{A}[P] \rightarrow_{\mathbf{A}}^* \tilde{S}$. Since P is α -honest, then \mathbf{A} is α -ready in \tilde{S} . Therefore, item (1) of Lemma B.17 guarantees that \mathbf{A} is ready in S . Hence, by Def. 4.3 it follows that P is honest.

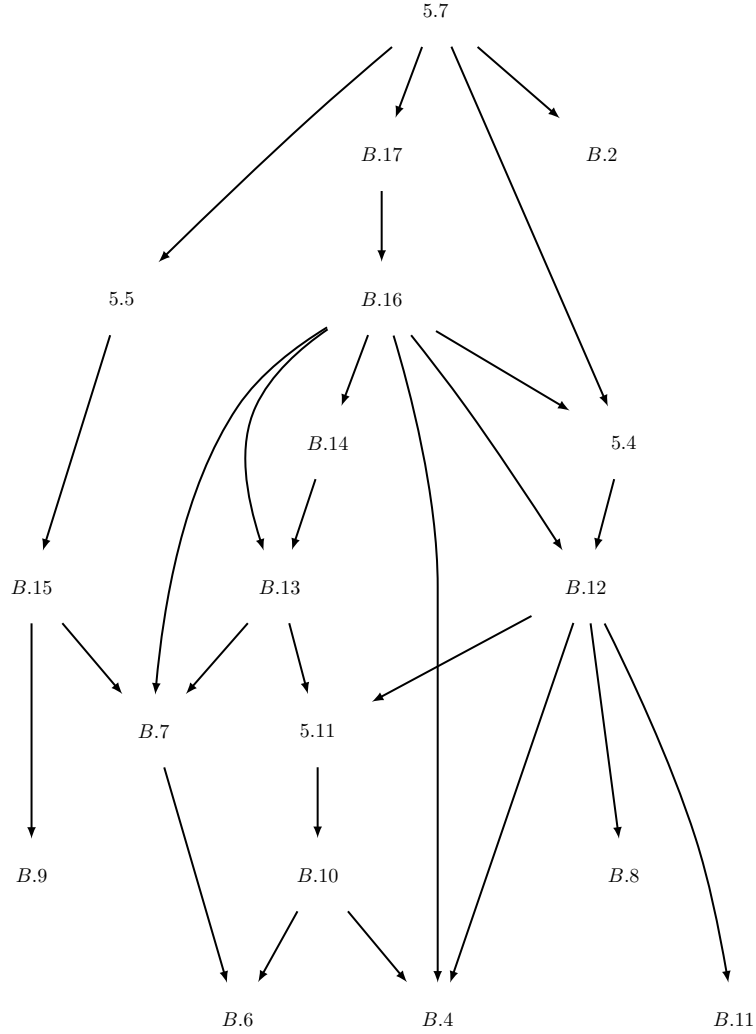


Figure 6: Dependencies among the proofs .

For the second part, suppose instead that P is honest, **ask**-free, and with proper contracts only. Let \tilde{S} be an arbitrary system such that $A[P] \rightarrow_A^* \tilde{S}$. By Theorem 5.5 it follows that there exist S, S' A -safe such that $\alpha_A(S) = A[P]$, $\alpha_A(S') = \tilde{S}$, and $S \rightarrow^* S'$. Since P is honest, then A is ready in S' , and so by item (2) of Lemma B.17 it follows that A is α -ready in \tilde{S} . Hence, by Def. 5.6 it follows that P is α -honest. \square

The diagram in Fig. 6 illustrates the dependencies among the proofs.

C Maude implementation

The diagram in Fig. 7 illustrates the graph of module importation used in our Maude implementation. The complete code is available at <http://tcs.unica.it/software/co2-maude>. Here we

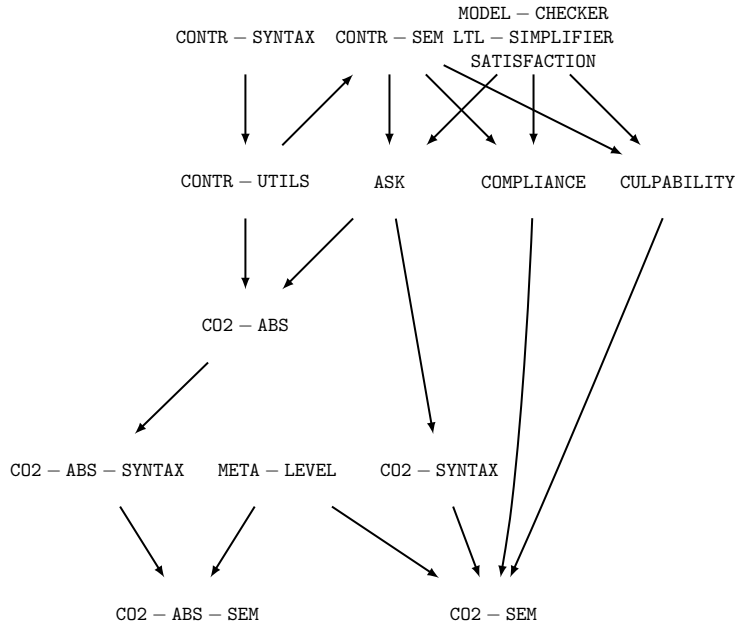


Figure 7: Graph of module importation.

only show the most relevant parts of the implementation.

Contracts: syntax.

```

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 .

op _ : Atom -> Atom [prec 20 ctor] .

eq - - a:Atom = a:Atom .

op 0 : -> Id [ctor] .
op _.. : Atom UniContract -> EGuarded [frozen prec 25 ctor] .
op _;_ : Atom UniContract -> IGuarded [frozen prec 25 ctor] .
op _+_ : EChoice EChoice -> EChoice [frozen comm assoc id: 0 prec 30 ctor] .
op _(+)_ : IChoice IChoice -> IChoice [frozen comm assoc id: 0 prec 30 ctor] .
op ready _.. : Atom UniContract -> RdyContract [frozen prec 25 ctor] .
op rec _.. : Var IChoice -> UniContract [frozen prec 35 ctor] .
op rec _.. : Var EChoice -> UniContract [frozen prec 35 ctor] .

op _ says _ : Participant RdyContract -> AdvContract [ctor] .

op _ | _ : AdvContract AdvContract -> BiContract [comm ctor] .

```

Contracts: semantics.

```

sort LBiContract .
subsort BiContract < LBiContract .

```

```

sort Label .

op _ says _ : Participant Atom -> Label [ctor] .

***labelled contracts
op {_}_ : Label LBiContract -> LBiContract [frozen ctor] .

vars A B : Participant .
vars a b : Atom .
vars c d : UniContract .
var ci : IChoice .
var de : EChoice .
var X : Var .
vars l l' : Label .
vars g g' : BiContract .

***structural congruence
eq A says rec X . ci = A says ci {rec X . ci / X} .
eq A says rec X . de = A says de {rec X . de / X} .

***one step semantics
crl [IntExt] : A says a ; c (+) ci | B says b . d + de =>
  {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 .

***semantics
sort TBiContract .
op <_> : LBiContract -> TBiContract [frozen] .

op dummy : -> Label [ctor] .

eq < g > = < {dummy} g > .

crl [Rifl] : < {l'} g > => < {l} g' > if g => {l} g' .

```

CO₂: syntax.

```

op empty : -> IdeVec [ctor] .
op void : -> ParamList [ctor] .
op 0 : -> System [ctor] .
op 0 : -> Process [ctor] .
op t : -> Prefix [ctor] .
op _;_ : ParamList ParamList -> ParamList [ctor assoc id: void] .
op _,_ : IdeVec IdeVec -> IdeVec [comm assoc ctor id: empty] .
op _[] : Participant Process -> System [frozen ctor] .
op _[] : SessionName BiContract -> System [frozen ctor] .
op _|_ : System System -> System [comm assoc frozen ctor id: 0] .
op (_)_ : IdeVec System -> System [frozen ctor] .
op {__}_ : SessionIde UniContract Participant -> System [frozen ctor] .
op __ : Prefix Process -> GuardProc [frozen ctor] .
op _+_ : Sum Sum -> Sum [comm assoc frozen ctor id: 0] .
op _|_ : Process Process -> Process [comm assoc frozen ctor id: 0] .
op (_)_ : IdeVec Process -> Process [frozen ctor] .
op tell __ : SessionIde UniContract -> Prefix [frozen ctor] .
op do __ : SessionIde Atom -> Prefix [frozen ctor] .
op ask __ : SessionIde Formula -> Prefix [frozen ctor] .
op _(_) : ProcIde ParamList -> Process [frozen ctor] .
op $_ : Nat -> SessionName [ctor] .
op __ : Qid Nat -> SessionVariable [ctor] .

```

CO₂: semantics.

```

sorts LSystem SLabel LPrefix .

```

```

subsort System < LSystem .
subsort Prefix < LPrefix .

op fuse : -> LPrefix .

op _:_ : Participant LPrefix -> SLabel [ctor] .

op {_}_ : SLabel System -> LSystem [frozen ctor] .

op Z : -> Participant .

vars A B : Participant .
vars P Q P' R : Process .
var u : SessionIde .
vars c d : UniContract .
vars s s' : SessionName .
vars g g' : BiContract .
var a : Atom .
var pi : LPrefix .
vars uVec vVec : IdeVec .
vars S S' S'' : System .
vars x y : SessionVariable .
var n : Nat .
var phi : Formula .
var X : ProcIde .
var uList : ParamList .
vars l l' : SLabel .

op del : IdeVec Prefix -> Prefix .
eq del((u , uVec) , do u a) = t .
eq del((u , uVec) , tell u c) = t .
eq del((u , uVec) , ask u phi) = t .
eq del(uVec , pi) = pi [owise] .

op fresh : Nat System -> SessionName .
eq fresh(n , S) = if not is $ n free in S then $ n else fresh(n + 1 , S) fi .

rl [Tau] : A[t . P + P' | Q] => {A : t} A[P | Q] .

rl [Tell] : A[tell u c . P + P' | Q] => {A : tell u c} A[P | Q] | {u c} A .

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 [Del] : (uVec) S => {A : del(uVec , pi)}(uVec) S' if S => {A : pi} S' .

crl [Par] : S | S'' => {A : pi} S' | S'' if S /= 0 /\ S'' /= 0 /\ S => {A : pi} S' .

crl [Fuse] : (uVec , vVec) ({x c}A | {y d}B | S) =>
{Z : 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) .

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

sort TSystem .

op <_> : System -> TSystem [frozen] .

crl [Rif1] : < S > => < S' > if S => {l'} S' .

sort ASystem .

op prefCheck : Prefix SessionName -> Bool .
eq prefCheck (do s a , s) = false .
eq prefCheck (pi , s) = true [owise] .

```

```

op <_>__ : System Participant SessionName -> ASystem .

*** non A-solo semantics: /= A: do s
crl < S > A s => < S' > A s if S => {B : pi} S' /\ (B /= A or prefCheck(pi , s)) .

*** A-solo semantics: A: /= do s
*** crl < S > A s => < S' > A s if S => {A : pi} S' /\ prefCheck(pi , s) .

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

op ready : Participant System System Module -> Bool .
ceq ready(A , S , s[g] , M:Module) = ready(A , s , S' , M:Module)
if A :C g /\ (uVec) S' := S .
ceq ready(A , S , s[g] , M:Module) = true if not A :C g .
eq ready(A , S , s[g] , M:Module) = ready(A , s , S , M:Module) [owise] .
ceq ready (A , S , S' | S'' , M:Module) = ready (A , S , S' , M:Module)
and ready(A , S , S'' , M:Module) if S' /= 0 /\ S'' /= 0 .
ceq ready (A , S , (uVec) S' , M:Module) = ready (A , S , S' , M:Module)
if (uVec) /= empty .
eq ready (A , S , 0 , M:Module) = true .
eq ready (A , S , B:Participant[P] , M:Module) = true .
eq ready (A , S , {u c}B:Participant , M:Module) = true .

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) .

sort Result .
subsort Result < TSystem .
subsort Result < Bool .

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) .

```

CO₂: abstract semantics.

```

op A : -> Participant [ctor] .

sorts LSystem ASLabel .

subsort System < LSystem .

op _:_ : Participant Prefix -> ASLabel [ctor] .
op ctx : -> ASLabel [ctor] .

op {_}_ : ASLabel System -> LSystem [frozen ctor] .

vars P Q : Process .

```

```

vars S S' S'' : System .
var P' : Sum .
vars c c' : UniContract .
vars g g' : RdyContract .
vars u v : SessionIde .
vars x y : SessionVariable .
var s : SessionName .
var a : Atom .
var pi : Prefix .
var uVec : IdeVec .
var X : ProcIde .
var mu : ASLabel .
vars n m : Nat .
var uList : ParamList .
var fi : Formula .

op del : IdeVec Prefix -> Prefix .
eq del((u , uVec) , do u a) = t .
eq del((u , uVec) , tell u c) = t .
eq del((u , uVec) , ask u fi) = t .
eq del(uVec , pi) = pi [owise] .

op fresh : Nat System -> SessionName .
eq fresh(n , S) = if not is $ n free in S then $ n else fresh(n + 1 , S) fi .

rl [Tau] : A[t . P + P' | Q] => {A : t} A[P | Q] .

rl [Tell] : A[tell u c . P + P' | Q] => {A : tell u c} A[P | Q] | {u c} A .

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

crl [Del] : (uVec) S => {A : del(uVec,pi)}(uVec) S' if S => {A : pi} S' .

crl [Del] : (uVec) S => {ctx}(uVec) S' if S => {ctx} S' .

crl [Par] : S | S'' => {mu} (S' | S'') if S'' /= 0 /\ S => {mu} S' .

crl [DoCTX] : s[g] => {ctx} s[g'] if g => {ctx} g' .

rl [Ask] : s[g] | A[ask s fi . P + P' | Q] => {ctx} s[g] | A[P | Q] .

crl [Fuse] : (x,uVec) (S | {x c}A) => {ctx} (s , uVec)(s[c] | S{s / x})
  if s := fresh(0 , S) .

*** unlabelled semantics
sort TSystem .
op <_> : System -> TSystem [ctor frozen] .

crl < S > => < S' > if S => {mu} S' .

*** A-solo semantics without do s
op prefCheck : Prefix SessionName -> Bool .
eq prefCheck (do s a , s) = false .
eq prefCheck (pi , s) = true [owise] .

sort ASystem .
op <_>_ : LSystem SessionName -> ASystem [ctor frozen] .
crl < S > s => < S' > s if S => {A : pi} S' /\ prefCheck (pi , s) .

```

Model checking honesty.

```

op ready : SessionName System Module -> Bool .
eq ready(s,S,M:Module) = metaSearch(M:Module,
upTerm(< S > s),
'<_>_'S:System , upTerm(s)),

```

```

'S:System => '{_}'_['l:ASLabel,'S':System] /\
'_:_[upTerm(A),'do_][upTerm(s),'a:Atom] := 'l:ASLabel,
'*,
unbounded,
0
) /= failure .

op culp : RdyContract Module -> Bool .
eq culp (g , M:Module) = metaSearch(M:Module,
upTerm(g),
'{'_}'_['l:ACLabel,'d:RdyContract],
'a:Atom := 'l:ACLabel,
'+,
1,
0
) /= failure .

op ready : System System Module -> Bool .
ceq ready(S , s[g] , M:Module) = ready(s , S' , M:Module)
if culp(g , M:Module) /\ (uVec) S' := S .
ceq ready(S , s[g] , M:Module) = true if not culp(g , M:Module) .
eq ready(S , s[g] , M:Module) = ready(s , S , M:Module) [owise] .
ceq ready (S , S' | S'' , M:Module) = ready (S , S' , M:Module)
and ready(S , S'' , M:Module) if S' /= 0 /\ S'' /= 0 .
ceq ready (S , (uVec) S' , M:Module) = ready (S , S' , M:Module) if (uVec) /= empty .
eq ready (S , 0 , M:Module) = true .
eq ready (S , B:Participant[P] , M:Module) = true .
eq ready (S , {u c}B:Participant , M:Module) = true .

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) .

sort Result .
subsort Result < TSystem .
subsort Result < Bool .

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) .

```

Example 4.4 (travel agency).

```

ops pay commit refund hotel ticket
commitH payH abortH commitF payF abortF : -> Atom [ctor] .
ops CU CF CH : -> UniContract .
ops P PU PU' PF PH : -> Process .
ops xu xf xh : -> SessionVariable .

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

ops refund? : -> Formula .
eq refund? = [] ~ payH .

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 refund? . do xu refund . 0 +
      t . (do xu commit . (do xf payF . 0 | do xh payH . 0)) .

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

*** context for honest-ctx checking

ops A U H F : -> Participant .
ops DU DH DF : -> UniContract .
ops Q Q' QU QH QF : -> Process .
ops yu yf yh : -> SessionVariable .
op S : -> System .

eq DU = - pay ; (- commit . 0 + - refund . 0) .
eq DF = - ticket ; (- commitF . - payF . 0 + - abortF . 0) .
eq DH = - hotel ; (- commitH . - payH . 0 + - abortH . 0) .

eq Q = ( yu , yf , yh )(tell yu DU . tell yf DF . tell yh DH .
      (do yu - pay . 0 | do yf - ticket . 0 | do yh - hotel . 0 | Q')) .
eq Q' = (do yu - commit . 0 + do yu - refund . 0) |
      (do yf - commitF . do yf - payF . 0 + do yf - abortF . 0) |
      (do yh - commitH . do yh - payH . 0 + do yh - abortH . 0) .

eq S = A[P | U[Q] .

*** A should be honest in the given context
red honest-ctx(A , S , ['TRAVEL-AGENCY-CTX]) .

```

Example 5.12 (voucher distribution system).

```

ops clickPay clickVoucher pay voucher accept reject ok no : -> Atom [ctor] .
ops x y : -> SessionVariable [ctor] .
ops CB CV : -> UniContract .
ops P Q R : -> Process .

eq CB = clickPay . pay . 0 + clickVoucher . (- reject ; pay . 0 (+
      - accept ; voucher . 0) .

eq CV = ok . 0 + no . 0 .

*** should be dishonest
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) .

ops P' Q' R' : -> Process .

*** should be honest
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 | (do y no . 0 + do y ok . 0)) .

reduce in STORE-VOUCHER : honest(P, ['STORE-VOUCHER]) .
rewrites: 31649 in 40ms cpu (39ms real) (791185 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]) >

reduce in STORE-VOUCHER : honest(P', ['STORE-VOUCHER]) .
rewrites: 44010 in 32ms cpu (31ms real) (1375226 rewrites/second)
result Bool: true

```