Verifiable abstractions for contract-oriented systems

Massimo Bartoletti*, Maurizio Murgia, Alceste Scalas

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

Roberto Zunino

Dipartimento di Matematica, Università degli Studi 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 respect them. A key issue is that the *honesty* property, which characterises those agents which respect their contracts in *all* possible execution contexts, is undecidable in general. The main contribution of this paper is a sound verification technique for honesty, targeted at agents modelled in a value-passing version of the calculus CO_2. To do that, we safely over-approximate the honesty property by abstracting from the actual values and from the contexts a process may be engaged with. Then, we develop a model-checking technique for this abstraction, we describe its implementation in Maude, and we discuss some experiments with it.

Keywords: contract-oriented computing, verification, rewriting logic, session types

1. Introduction

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

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

When services behave in the "right way" for all their advertised contracts, they are called *honest*. Instead, when services are *not* honest, they do *not* always respect the contracts they advertise, at least in some execution context. This may happen either unintentionally (because of errors in the service specification or in its implementation), or even because of malicious behaviour. Since discrepancies between the advertised

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

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

and the actual behaviour can be sanctioned, a new kind of attacks becomes possible: if a service does not behave as promised, an attacker can induce it to a situation where the service is sanctioned, while the attacker is not. A crucial problem is then how to ensure that a service will *never* result responsible of a contract violation, before deploying it in an unknown (and possibly adversarial) environment.

Contract-oriented computing in CO_2. The distributed, contract-oriented systems outlined in the previous paragraphs can be formally modelled and studied in CO_2, a core process calculus for contract-oriented computing [1, 3]. CO_2 is not tied to a specific language or semantics for contracts. This flexibility allows to adopt one of the many different contract formalisms available in literature: these include behavioural types [4, 5, 6, 7], Petri nets [8, 9, 10], multi-player games [11, 12], logics [1, 3], *etc.* Among behavioural types, *session types* [13, 5] have been devoted a lot of attention in the last few years, both at the foundational level [6, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23] and at the application level [24, 25, 26, 27]. In their simplest incarnation, session types are terms of a process algebra featuring a *selection* construct (i.e., an internal choice among a set of branches), a *branching* construct (i.e., an external choice offered to the environment), and recursion.

In the present work, we adopt CO_2 with binary session types as our contract model of choice. Therefore, once formalised in CO_2, a service will be represented as an agent $\mathsf{A}[P]$ that can offer (or require) some behaviour by advertising it in the form of a session type c. In order to establish a session, another compliant session type needs to be advertised by another agent: intuitively, compliance is based on the standard notions of duality and subtyping [28, 29, 30], and ensures that, when run in parallel, the two session types enjoy progress. Thus, when an agent $\mathsf{B}[Q]$ advertises a session type d which is compliant with c, a new session s between $\mathsf{A}[P]$ and $\mathsf{B}[Q]$ is created. Then, $\mathsf{A}[P]$ and $\mathsf{B}[Q]$ can start interacting through s, by performing the actions prescribed by c and d, respectively — or even by choosing not to do so.

The problem of verifying honesty, even with this simplistic contract model, and in the most basic version of CO_2, is not trivial: the honesty of an agent turns out to be undecidable (the proof in [31] exploits the fact that the value-free fragment of CO_2 is Turing-powerful). Some preliminary research on the static verification of honesty uses a *type system*: in [32], it is shown that type safety guarantees honesty, but no type inference algorithm is provided; moreover, only a simple version of CO_2 (e.g., without value passing) is addressed. Effective techniques to safely *approximate* honesty of contract-oriented services are therefore in order.

Contributions. In this paper we devise and implement a sound verification technique for honesty in an extended version of CO_2, featuring expressions, value-passing, and conditionals. The main technical insight is an abstract semantics of CO_2 which preserves the transitions of an agent $\mathsf{A}[P]$, while abstracting from values and from the context wherein $\mathsf{A}[P]$ is run. Building upon this abstract semantics, we then devise an abstract notion of honesty (α*-honesty*, Definition 4.10), which approximates the execution context. The main technical result is Theorem 4.12, which states that our approximation is correct (i.e., α-honesty implies honesty), and that — under certain hypotheses on the syntax of processes — it is also *complete* (i.e., honesty implies α-honesty). We then propose a model-checking approach for verifying α-honesty, and we provide an implementation in Maude. A relevant fact about our theoretical work is that, although in this paper we have focussed on binary session types, our verification technique appears to be directly reusable to deal with different contract models, e.g. all models satisfying Theorem 4.5. We have validated our technique through a set of case studies; quite notably, our implementation has allowed us to determine the dishonesty of a supposedly-honest CO_2 process appeared in [31] (see Section 5.2). Throughout the paper we shall use a running example (a simple online store), to clarify the different notions as they are introduced.

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

In Section 6 we discuss related work in the area, and finally in Section 7 we draw some conclusions. Sections A and B contain the proofs of all our statements. The code of our verification tool and that of all the experiments is available online [33].

2. Session types as contracts

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

2.1. Syntax

We assume a set of *participants* (ranged over by $\mathsf{A}, \mathsf{B}, \ldots$), a set of *branch labels* (ranged over by $\mathtt{a}, \mathtt{b}, \ldots$), and a set of *sorts* ranged over by $\mathtt{T}, \mathtt{T}', \ldots$ (e.g. $\mathtt{int}$, $\mathtt{bool}$, $\mathtt{unit}$). Each sort $\mathtt{T}$ is populated by a set of values, ranged over by $\mathtt{v}, \mathtt{v}', \ldots$; as usual, we write $\mathtt{v} : \mathtt{T}$ to indicate that $\mathtt{v}$ has sort $\mathtt{T}$.

Definition 2.1 (Contracts). *Contracts are* binary session types*, i.e. terms defined by the grammar:*

$$c, d \quad ::= \quad \bigoplus_{i \in \mathcal{I}} \mathtt{a}_i!\mathtt{T}_i \,.\, c_i \;\;\Big|\;\; \sum_{i \in \mathcal{I}} \mathtt{a}_i?\mathtt{T}_i \,.\, c_i \;\;\Big|\;\; \mathtt{rec}\, X \,.\, c \;\Big|\;\; X$$

where (i) the index set $\mathcal{I}$ is finite, (ii) the labels $\mathtt{a}_i$ in the prefixes of each summation are pairwise distinct, and (iii) recursion variables X are prefix-guarded.

An internal sum $\bigoplus_i \mathtt{a}_i!\mathtt{T}_i \,.\, c_i$ allows a participant to choose one of the labels $\mathtt{a}_i$, to pass a value of sort $\mathtt{T}_i$, and then to behave according to the branch c_i. Dually, an external sum $\sum_i \mathtt{a}_i?\mathtt{T}_i \,.\, c_i$ allows to wait for the other participant to choose one of the labels $\mathtt{a}_i$, and then to receive a value of sort $\mathtt{T}_i$ and behave according to the branch c_i. Empty internal/external sums are identified, and they are denoted with $\mathtt{0}$, which represents a *success state* wherein the interaction has terminated correctly.

We use the (commutative and associative) binary operators to isolate a branch in a sum: e.g., $c = (\mathtt{a}!\mathtt{T}.c') \oplus c''$ means that c has the form $\bigoplus_{i \in \mathcal{I}} \mathtt{a}_i!\mathtt{T}_i.c_i$ and there exists some $i \in \mathcal{I}$ such that $\mathtt{a}!\mathtt{T}.c' = \mathtt{a}_i!\mathtt{T}_i.c_i$. Hereafter, we will omit the $\mathtt{unit}$ sort and the trailing occurrences of $\mathtt{0}$, and we will only consider contracts without free occurrences of recursion variables X.

2.2. Semantics

While a contract describes the intended behaviour of *one* of the two participants involved in a session, the behaviour of two interacting participants A and B is modelled by the composition of two contracts, denoted by $\mathsf{A} : c \mid \mathsf{B} : d$. We specify in Definition 2.2 an operational semantics of contracts, where the two participants alternate in firing actions. To do that, we extend the syntax of Definition 2.1 with the term $\mathtt{rdy}\ \mathtt{a}?\mathtt{v} \,.\, c$, which models a participant ready to input a value $\mathtt{v}$ in a branch with label $\mathtt{a}$, and then to continue as c. In other words, $\mathtt{rdy}\ \mathtt{a}?\mathtt{v}$ acts as a one-position buffer shared between the two participants.

Definition 2.2 (Semantics of contracts). *A* contract configuration *γ is a term of the form $\mathsf{A} : c \mid \mathsf{B} : d$, where $\mathsf{A} \neq \mathsf{B}$ and the syntax of contracts is extended with terms $\mathtt{rdy}\ \mathtt{a}?\mathtt{v} \,.\, c$. We postulate that at most one occurrence of $\mathtt{rdy}$ is present, and if so $\mathtt{rdy}$ is at the top-level. We define a congruence relation $\equiv$ between contracts as the least equivalence including α-conversion of recursion variables, and satisfying $\mathtt{rec}\, X \,.\, c \equiv c\{^{\mathtt{rec}\, X.c}/_X\}$. Also, we assume that $\mathsf{A} : c \mid \mathsf{B} : d$ is equivalent to $\mathsf{B} : d \mid \mathsf{A} : c$. The semantics of contracts is modelled by a the labelled transition relation $\twoheadrightarrow$, which is the smallest relation closed under the rules in Figure 1 and under $\equiv$. We denote with $\twoheadrightarrow^*$ the reflexive and transitive closure of $\twoheadrightarrow$. A* computation *(of an initial configuration γ_0) is a possibly infinite sequence of transitions $\gamma_0 \twoheadrightarrow \gamma_1 \twoheadrightarrow \cdots$.*

$$\mathsf{A} : (\mathtt{a!T} \,.\, c \oplus c') \mid \mathsf{B} : (\mathtt{a?T} \,.\, d + d') \xrightarrow{\mathsf{A}:\mathtt{a!v}} \mathsf{A} : c \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{a?v} \,.\, d \quad \text{if } \mathtt{v} : \mathtt{T} \qquad \text{[IntExt]}$$

$$\mathsf{A} : \mathtt{rdy}\ \mathtt{a?v} \,.\, c \mid \mathsf{B} : d \xrightarrow{\mathsf{A}:\mathtt{a?v}} \mathsf{A} : c \mid \mathsf{B} : d \qquad \text{[Rdy]}$$

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

In rule [IntExt], A can perform any of the actions in the intersection between its internal sum labels, and the external sum labels of B; then, B is forced to commit to the corresponding branch in its external sum. This is done by marking such a committed branch with $\mathtt{rdy}\ \mathtt{a?v}$, while discarding all the other branches; the transition label $\mathsf{A} : \mathtt{a!v}$ models A selecting the branch with label $\mathtt{a}$, and passing value $\mathtt{v}$. Participant B can then perform his action in the subsequent step, by rule [Rdy]. Note that this semantics causes an *alternation* between output and input actions, not present in other semantics of sessions types (for instance, the one in [30]). This alternation allows for a "contractual" interpretation of session types: when a transition with label $\mathsf{A} : \cdots$ is enabled, it means that A is in charge to perform the next contractual action. In particular, A is in charge either when she has an internal choice, or she is committed to a branch of an external choice (with $\mathtt{rdy}$). Observe that this interpretation would not fit with standard CCS-style synchronisation, since the latter does not allow to distinguish A's turn in sending from and B's turn in receiving.

2.3. Compliance

We now define a notion of compliance between contracts. The intuition is that if a contract c is compliant with a contract d, then in all the configurations of a computation of $\mathsf{A} : c \mid \mathsf{B} : d$, whenever a participant wants to choose a branch in an internal sum, the other participant always offers the opportunity to do it. Compliance guarantees that whenever a computation of $\mathsf{A} : c \mid \mathsf{B} : d$ becomes stuck, then both participants have reached the success state $\mathtt{0}$.

Definition 2.3 (Compliance). *We say that a configuration γ is* safe *iff either:*

$$\begin{array}{ll} \textit{(i)} & \gamma = \mathsf{A} : \bigoplus_{i\in I} \mathtt{a}_i!\mathtt{T}_i \,.\, c_i \mid \mathsf{B} : \sum_{j\in J} \mathtt{a}_j?\mathtt{T}_j \,.\, d_j \quad \textit{with} \quad \emptyset \neq I \subseteq J \\ \textit{or} \quad \textit{(ii)} & \gamma = \mathsf{A} : \mathtt{rdy}\ \mathtt{a?v}.c \mid \mathsf{B} : d \\ \textit{or} \quad \textit{(iii)} & \gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0} \end{array}$$

Then, we say that c and d are compliant (in symbols, $c \bowtie d$) whenever:

$$\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow^* \gamma \implies \gamma \textit{ safe}$$

We observe that the notion of compliance in Definition 2.3 is equivalent to that of *progress* in [30, 23]. This can be proved as in [12], by exploiting the fact that the alternating semantics of session types is *turn-bisimilar* to the standard LTS semantics (as shown in Lemma 5.10 in [12]).

Example 2.4. *Let $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, where $c = \mathtt{a!} \,.\, c_1 \oplus \mathtt{b!} \,.\, c_2$ and $d = \mathtt{a?} \,.\, d_1 + \mathtt{c?} \,.\, d_2$. If participant A internally chooses label $\mathtt{a}$, then γ will take a transition to $\mathsf{A} : c_1 \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{a?v}.\ d_1$, for some $\mathtt{v}$. Suppose instead that A chooses $\mathtt{b}$, which is not offered by B in his external choice. In this case, $\gamma \not\xrightarrow{\mathsf{A}:\mathtt{b!v}}$, and indeed γ is* not *safe according to Definition 2.3. Therefore, c and d are* not *compliant.*

The following lemma states that each contract has a compliant one.

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

$$\mathsf{A} : (a \,.\, c \oplus c') \mid \mathsf{B} : (\mathrm{co}(a) \,.\, d + d') \xrightarrow{\mathsf{A}:a}\!\!\twoheadrightarrow_\star \mathsf{A} : c \mid \mathsf{B} : \mathtt{rdy}\ \mathrm{co}(a) \,.\, d \quad \text{[AbsIntExt]}$$

$$\mathsf{A} : \mathtt{rdy}\ a \,.\, c \mid \mathsf{B} : d \xrightarrow{\mathsf{A}:a}\!\!\twoheadrightarrow_\star \mathsf{A} : c \mid \mathsf{B} : d \quad \text{[AbsRdy]}$$

Figure 2: Semantics of value-abstract contracts (symmetric rules for B actions omitted)

Definition 2.3 cannot be directly exploited as an algorithm for checking compliance, as the transition system of contracts is infinite state (and infinitely branching), because of values $\mathtt{v}$ in transition labels and in states. However, note that values do not play any role in the dynamics of contracts, except for their occurrence in transition labels (which will be exploited later on in Section 3). Therefore, for the sake of checking compliance we can consider an alternative semantics of contracts, where we abstract from values (Figure 2). The configurations in this semantics are terms of the form $\mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d)$, where the abstraction $\alpha^\star$ encodes sorts in branch labels, and removes values from $\mathtt{rdy}$. For instance, $\mathtt{a!T}.\, c$ is abstracted as $(\mathtt{a}, \mathtt{T})\mathtt{!}.\, \alpha^\star(c)$, while $\mathtt{rdy}\ \mathtt{a?v}.\, c$ is abstracted as $\mathtt{rdy}\ (\mathtt{a}, \mathtt{T})\mathtt{?}.\, \alpha^\star(c)$ whenever $\mathtt{v} : \mathtt{T}$. The branch labels of value-abstract contracts (ranged over by $a, b, \ldots$) are terms of the form $(\mathtt{a}, \mathtt{T})\circ$, where $\circ \in \{\mathtt{!}, \mathtt{?}\}$. We postulate an involution operator $\mathrm{co}(\cdot)$ of value-abstract branch labels, satisfying $\mathrm{co}((\mathtt{a}, \mathtt{T})\mathtt{?}) = (\mathtt{a}, \mathtt{T})\mathtt{!}$ and $\mathrm{co}((\mathtt{a}, \mathtt{T})\mathtt{!}) = (\mathtt{a}, \mathtt{T})\mathtt{?}$. Further details about the definition of value-abstract contracts and of the abstraction function $\alpha^\star$ are provided in Section B.

The semantics of value-abstract contracts leads to a finite state system, so it provides us with a model-checkable characterisation of compliance (see Section 2.5 for some implementation details).

Lemma 2.6. *For all contracts c, d:*

$$c \bowtie d \iff (\forall \gamma.\ \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \twoheadrightarrow_\star{}^* \gamma \implies \gamma \textit{ safe})$$

Example 2.7 (Online store). *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 proceed to* `checkout`*. Then, the client can either* `cancel` *the order, or* `pay`*. In the latter case, the store can accept the payment (*`ok`*), or decline it (*`no`*, in which case it lets the user try again), or it can* `abort` *the transaction. Such a contract may be expressed as the session type c_A below:*

$$\begin{aligned} c_\mathsf{A} &= \mathtt{addToCart?int} \,.\, \big(\mathtt{rec}\, Z \,.\, \mathtt{addToCart?int} \,.\, Z \;+\; \mathtt{checkout?} \,.\, c_{\mathtt{pay}}\big) \\ \textit{where}\quad c_{\mathtt{pay}} &= \mathtt{rec}\, Y \,.\, \big(\mathtt{pay?string} \,.\, (\mathtt{ok!} \;\oplus\; \mathtt{no!} \,.\, Y \;\oplus\; \mathtt{abort!}) \;+\; \mathtt{cancel?}\big) \end{aligned}$$

A possible contract of some buyer B *could be expressed as follows:*

$$c_\mathsf{B} = \mathtt{rec}\, Z \,.\, \big(\mathtt{addToCart!int} \,.\, Z \;\oplus\; \mathtt{checkout!} \,.\, \mathtt{pay!string} \,.\, (\mathtt{ok?} \;+\; \mathtt{no?} \,.\, \mathtt{cancel!} \;+\; \mathtt{abort?})\big)$$

The above contracts are not compliant: in fact, c_B can choose to perform the branch `checkout` *before doing an* `addToCart`*. Instead, the contract $\mathtt{addToCart!int} \,.\, c_\mathsf{B}$ is compliant with c_A. The Maude implementation of this and of the subsequent examples is available in [33].*

2.4. Culpability

We now tackle the problem of determining who is expected to make the next step in an interaction. We call a participant A *culpable* in γ if she is expected to perform some actions so to make γ progress. Note that culpability does not imply a permanent status of contract configurations; instead, it is a *transient* notion, because (as formally stated in Theorem 2.10), a participant can always move out from this state.

Definition 2.8 (Culpability). *A participant* A *is culpable in γ ($\mathsf{A} \frown \gamma$ in symbols) iff $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\twoheadrightarrow$ for some $\mathtt{a}, \mathtt{v}$ and $\circ \in \{\mathtt{!}, \mathtt{?}\}$. When* A *is* not *culpable in γ we write $\mathsf{A} \smile \gamma$.*

Theorem 2.9 below establishes that, when starting from a configuration of compliant contracts, exactly one participant is culpable in all subsequent configurations. The only exception is $\mathsf{A} : 0 \mid \mathsf{B} : 0$, which represents a successfully terminated interaction, where nobody is culpable.

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

The following theorem states that a participant is always able to recover from culpability by performing a bounded number of actions.

Theorem 2.10 (Contractual exculpation). *Let $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, and let $\gamma \twoheadrightarrow^* \gamma'$. Then:*

1. $\gamma' \not\twoheadrightarrow \implies \mathsf{A} \,\overset{\smile}{\cdot}\, \gamma'$ *and* $\mathsf{B} \,\overset{\smile}{\cdot}\, \gamma'$
2. $\mathsf{A} \,\overset{\frown}{\cdot\cdot}\, \gamma' \implies \forall \gamma'' : \gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathsf{A} \,\overset{\smile}{\cdot}\, \gamma'', \text{ or} \\ \forall \gamma''' : \gamma'' \twoheadrightarrow \gamma''' \implies \mathsf{A} \,\overset{\smile}{\cdot}\, \gamma''' \end{cases}$

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

2.5. Maude implementation

In this section we describe our executable specification of (value abstract) contracts in Maude. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [34]. Here we are interested in using Maude as a semantic framework for concurrent systems. For instance, we exploit Maude equational logic to express structural equivalence and basic term transformations (like, e.g., variable substitution), and rewriting rules to model labelled transition semantics. The Maude features used in the paper will be introduced as needed.

We use *sorts* to specify the syntactic categories of contracts, some of which are linked by the *subsort* relation. Sorts and subsorts for contracts are defined as follows:

```
sorts   UniContract Participant AdvContract BiContract IGuarded EGuarded IChoice EChoice
        Var Id RdyContract .
sorts   BType InAction OutAction IOAction ActName .
subsort Id < IGuarded < IChoice < UniContract < RdyContract .
subsort Id < EGuarded < EChoice < UniContract < RdyContract .
subsort Var < UniContract .
subsort InAction < IOAction .
subsort OutAction < IOAction .
```

The sort `UniContract` describes session types as in Definition 2.1; here we are specifying internal/external sums as commutative and associative binary operators between *singleton* internal/external sums; the neutral element is the only inhabitant of sort `Id`. 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, which is a subsort of both internal/external sums. `RdyContract` is for contracts which may have a top-level `rdy`, `AdvContract` is the sort of contracts advertised by some participant, and `BiContract` is for contract configurations.

Sorts are inhabited by operators, which are defined by the keyword `op`, with the general schema:

```
op <OpName> : <Sort-1> ... <Sort-k> -> <Sort> [<OperatorAttributes>] .
```

where `<OpName>` is the name of the operator, `<Sort-1> ... <Sort-k>` is the (possibly empty) list of sorts of the operator parameters, `<Sort>` is the sort of the result and `<OperatorAttributes>` is an optional parameter for specifying the operator attributes. Constants are represented as operators without parameters. The special symbol _ in `OpName` is used for expressing operators in *mixfix* form. The operator attribute `prec`

establishes the operator precedence. The attribute `ctor` is used to specify *constructors*, while operators declared without `ctor` are *defined functions*. The attribute `frozen` restricts the application of rewriting rules (to be specified later) at the top-level, only. The attributes `comm` and `assoc` are used to specify, respectively, commutative and associative operators, while the attribute `id:<t>` declares the neutral element.

The Maude specification of the contract syntax is the following:

```
op _!_ : ActName BType -> OutAction [prec 20 ctor] .
op _?_ : ActName BType -> InAction  [prec 20 ctor] .
op 0 : -> Id [ctor] .
op _._ : InAction  UniContract -> EGuarded [frozen ctor] .
op _._ : OutAction 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 _._ : InAction 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] .
```

Operations between terms can be expressed by means of *equations* (with keyword `eq`). Basically, they are simplification rules (applied left-to-right), and they are required to be Church-Rosser and terminating.

For instance, the involution $\mathrm{co}(\cdot)$ on branch labels is specified as follows:

```
op co : IOAction -> IOAction .
eq co(a ! T) = (a ? T) .
eq co(a ? T) = (a ! T) .
```

To model the labelled transition semantics of contract configurations, we will exploit Maude *rewriting rules*, which specify local concurrent transitions of terms. Unlike equations, rewriting rules are not required to be Church-Rosser nor terminating. Rewriting rules can be either *unconditional* (keyword `rl`), or *conditional* (keyword `crl`), with the following schema:

```
 rl [<Label>] : <Term-1> => <Term-2> [<StatementAttributes>] .

crl [<Label>] : <Term-1> => <Term-2>
              if <Condition-1> /\ ... /\ <Condition-k> [<StatementAttributes>] .
```

The evaluation strategy adopted by Maude is to first apply equations to reduce terms in normal form (which exists and is unique by the assumption of Church-Rosser and terminating equations), and then to apply rewriting rules. Systems specified with rewriting rules can be verified with the Maude LTL model-checker [35], and with the Maude search capabilities.

The semantics of contract configurations is an almost-literal translation of that in Figure 1 using rewriting rules. A minor difference is that, since transitions in rewritings are unlabelled, we decorate states with labels, as done in [36]. Concretely, this is done by specifying the labelled transition $\gamma \xrightarrow{\mu}\!\!\twoheadrightarrow_\star \gamma'$ as the unlabelled transition $\gamma \twoheadrightarrow_\star \{\mu\}\gamma'$. In Maude, this is specified as follows:

```
sort LBiContract .
subsort BiContract < LBiContract .
sort Label .

op _ says _ : Participant IOAction -> Label [ctor] .
op {_}_ : Label LBiContract -> LBiContract [frozen ctor] .

crl [AbsIntExt] : A says ol . c (+) ci | B says il . d + de
 => { A says ol } A says c | B says ready co(ol) . d
if co(il) = ol .

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

Note that, since the operator contructor for `LBiContract` has the `frozen` attribute, the above rewriting rules can be directly used for computing one step successors, only. Sequences of transitions can be obtained similarly to [36]: we define the constructor `<_>`, and we provide it with the rewriting rule `[Rifl]` below, which computes one step successors for decorated states too, and keeps track of the last label, only. This choice is motivated by the following reasons: first, keeping track of the whole trace might make the set of states in the transition system infinite; second, the last label is needed to correctly implement queries on contract configurations (see Definition 3.3). The Maude code for the transition relation is the following:

```
sort TBiContract .
op <_> : LBiContract -> TBiContract [frozen] .
op dummy : -> Label [ctor] .

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

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

The compliance relation is defined as suggested by Lemma 2.6, exploiting the Maude LTL model-checker. Basically, we model the predicate *safe* as an atomic proposition, defined by its satisfaction relation `|=`:

```
op safe : -> Prop .
op isSafe : BiContract -> Bool .

eq isSafe(A says 0 | B says 0) = true .
ceq isSafe(A says IS | B says ES) = IS subset ES if IS =/= 0 .
eq isSafe(A says ready il . c | B says d) = true .
eq isSafe(C:BiContract) = false [owise] .

eq < { l } g > |= safe = isSafe(g) .
```

where `Prop` is the built-in sort for propositions, the attribute `owise` (for otherwise) tells Maude to apply the equation only if the other ones do not apply, and `ceq` is the keyword for conditional equations.

The compliance relation `c |X| d` is implemented by verifying that the configuration $\mathsf{A} : c \mid \mathsf{B} : d$ satisfies the LTL formula `[] safe` (i.e., "globally" safe). This is done through the Maude LTL model checker.

```
op _|X|_ : UniContract UniContract -> Bool .
eq c |X| d = modelCheck(< Al says c | Bo says d >, ([] safe)) == true .
```

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

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

3. Contract-oriented computing & Honesty

We model agents and systems in the process calculus CO_2 [1, 31, 32], which we instantiate with the contracts introduced in Section 2. In Section 3.1 we provide the syntax of CO_2: its primitives allow agents to advertise contracts, to open sessions between agents with compliant contracts, to fulfil them by performing the required actions, and to query contracts. Then, in Section 3.2 we define the semantics of CO_2, and in Section 3.3 we formalise the concept of honesty.

$\mathsf{A}, \mathsf{B}, \ldots$	Participant names	$u, v, \ldots$	Union of:
$\mathtt{a}, \mathtt{b}, \ldots$	Branch labels	$s, t, \ldots \in \mathcal{N}$	Session names
$\mathtt{T}, \mathtt{T}', \ldots$	Sorts	$x, y, \ldots \in \mathcal{V}$	Variables
$\mathtt{v}, \mathtt{v}', \ldots$	Values	$e, e', \ldots$	Expressions
$c, d, \ldots$	Contracts	$P, Q, \ldots$	Processes
$\gamma, \gamma', \ldots$	Contract configurations	$S, S', \ldots$	Systems
$\gamma \twoheadrightarrow \gamma'$	Transition of contracts	$S \rightarrow S'$	Transition of systems

Table 1: Summary of notation.

commutative monoidal laws for $\mid$ on processes and systems

$$\mathsf{A}[(v)P] \equiv (v)\,\mathsf{A}[P] \qquad Z \mid (u)Z' \equiv (u)(Z \mid Z') \;\text{ if } u \notin \text{fv}(Z) \cup \text{fn}(Z)$$

$$(u)(v)Z \equiv (v)(u)Z \qquad (u)Z \equiv Z \;\text{ if } u \notin \text{fv}(Z) \cup \text{fn}(Z) \qquad \{\downarrow_s c\}_{\mathsf{A}} \equiv \mathbf{0}$$

Figure 3: Structural equivalence for CO_2 (Z, Z' range over systems or processes).

3.1. Syntax

Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of *variables* (ranged over by $x, y, \ldots$) and *names* (ranged over by $s, t, \ldots$). We assume a language of *expressions* (ranged over by $e, e', \ldots$), containing variables, values, and operators (e.g. the usual arithmetic/logic ones). The actual choice of operators is almost immaterial for the subsequent technical development; here we just postulate a function $[\![\cdot]\!]$ which maps (closed) expressions to values. We assume that the sort of an expression is uniquely determined by the sorts of its variables. We use $u, v, \ldots$ to range over $\mathcal{V} \cup \mathcal{N}$, we use $\boldsymbol{u}, \boldsymbol{v}, \ldots$ to range over sequences of variables/names, and $\boldsymbol{e}$ to range over sequences of expressions. To make symbols lookup easier, we have summarised the syntactic categories and some notation in Table 1.

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

$$\begin{array}{lllllll} S & ::= & \mathbf{0} & \mid\; \mathsf{A}[P] & \mid\; s[\gamma] & \mid\; (u)S & \mid\; S \mid S \;\;\mid\; \{\downarrow_u c\}_{\mathsf{A}} \\ P & ::= & \sum_i \pi_i.P_i & \mid\; \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P & \mid\; \mathrm{X}(\boldsymbol{u}, \boldsymbol{e}) & \mid\; (u)P & \mid\; P \mid P \\ \pi & ::= & \tau & \mid\; \mathtt{tell} \downarrow_u c & \mid\; \mathtt{do}_u\ \mathtt{a!}e & \mid\; \mathtt{do}_u\ \mathtt{a?}x : \mathtt{T} & \mid\; \mathtt{ask}_u \phi \end{array}$$

If $\boldsymbol{u} = u_0, \ldots, u_n$, we will use $(\boldsymbol{u})S$ and $(\boldsymbol{u})P$ as shorthands for $(u_0) \cdots (u_n)S$ and $(u_0) \cdots (u_n)P$, respectively. We also assume the following syntactic constraints on processes and systems:

1. *each occurrence of named processes is prefix-guarded;*
2. *in $(\boldsymbol{u})(\mathsf{A}[P] \mid \mathsf{B}[Q] \mid \cdots)$, it must be $\mathsf{A} \neq \mathsf{B}$;*
3. *in $(\boldsymbol{u})(s[\gamma] \mid t[\gamma'] \mid \cdots)$, it must be $s \neq t$.*

Systems $S, S', \ldots$ are the parallel composition of *participants* $\mathsf{A}[P]$, *sessions* $s[\gamma]$, *delimited systems* $(u)S$, and *latent contracts* $\{\downarrow_u c\}_{\mathsf{A}}$. A latent contract $\{\downarrow_x c\}_{\mathsf{A}}$ represents a contract c (advertised by A) which has not been stipulated yet; upon stipulation, the variable x will be instantiated to a fresh session name.

Processes $P, Q, \ldots$ are prefix-guarded (finite) sums of processes, conditionals $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$ (where e is a boolean valued expression), named processes $\mathrm{X}(\boldsymbol{u}, \boldsymbol{e})$ (used e.g. to specify recursive behaviours), delimited processes $(u)P$, and parallel compositions $P \mid P$.

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

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}:\,\tau} \mathsf{A}[P \mid Q] \qquad \text{[TAU]}$$

$$\mathsf{A}[\texttt{tell}\downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{tell}\downarrow_u c} \mathsf{A}[P \mid Q] \;\mid\; \{\downarrow_u c\}_{\mathsf{A}} \qquad \text{[TELL]}$$

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

$$\mathsf{A}[(\texttt{if}\; e\; \texttt{then}\; P_{\texttt{true}}\; \texttt{else}\; P_{\texttt{false}}) \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{if}} \mathsf{A}[P_{[\![e]\!]} \mid Q] \qquad \text{[IF]}$$

$$\frac{[\![e]\!] = \texttt{v} \qquad \gamma \xrightarrow{\mathsf{A}:\texttt{a!v}}\!\!\!\!\rightarrow \gamma'}{\mathsf{A}[\texttt{do}_s\,\texttt{a!}e.P + P' \mid Q] \;\mid\; s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{do}_s\,\texttt{a!v}} \mathsf{A}[P \mid Q] \;\mid\; s[\gamma']} \qquad \text{[DO!]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:\texttt{a?v}}\!\!\!\!\rightarrow \gamma' \qquad \texttt{v} : \texttt{T}}{\mathsf{A}[\texttt{do}_s\,\texttt{a?}x : \texttt{T}.P + P' \mid Q] \;\mid\; s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{do}_s\,\texttt{a?v}} \mathsf{A}[P\{{}^{\texttt{v}}/_{x}\} \mid Q] \;\mid\; s[\gamma']} \qquad \text{[DO?]}$$

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

$$\frac{\mathrm{X}(\boldsymbol{x}, \boldsymbol{y}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{{}^{\boldsymbol{u}}/_{\boldsymbol{x}}\}\{{}^{\boldsymbol{e}}/_{\boldsymbol{y}}\} \mid Q] \mid S \xrightarrow{\mu} S'}{\mathsf{A}[\mathrm{X}(\boldsymbol{u}, \boldsymbol{e}) \mid Q] \mid S \xrightarrow{\mu} S'} \;\text{[DEF]} \qquad \frac{S \xrightarrow{\mu} S'}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \;\text{[PAR]}$$

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

Figure 4: Reduction semantics of CO_2.

The only binder for names is the delimitation (u), both in systems and processes. Instead, variables have two binders: delimitations (x) (both in systems and processes), and input actions. Namely, in a process $\texttt{do}_u\,\texttt{a?}x : \texttt{T}.\,P$, the variable x in the prefix binds the occurrences of x within P. Note that "value-kinded" variables in input actions will be replaced by values, while "name-kinded" variables used in delimitations will be replaced by session names. Accordingly, we avoid confusion between these two kinds of variables. For instance, we forbid $\texttt{do}_u\,\texttt{a?}x.\texttt{do}_x\,\texttt{b!v}$ and $(x)\,\texttt{do}_u\,\texttt{a!}x$.

Free *session* names/variables in a prefix are defined as follows: $\mathrm{fnv}(\tau) = \emptyset$, and $\mathrm{fnv}(\texttt{tell}\downarrow_u c) = \{u\} = \mathrm{fnv}(\texttt{do}_u\,\texttt{a!}\,e) = \mathrm{fnv}(\texttt{do}_u\,\texttt{a?}x : \texttt{T})$. Free variables/names of systems/processes are defined accordingly, and they are denoted by fv and fn. A system or a process is *closed* when it has no free variables.

We write $\pi_1.P_1 + \pi_2.P_2$ for $\sum_{i\in\{1,2\}} \pi_i.P_i$, and $\mathbf{0}$ for $\sum_{\emptyset} P$. We stipulate that each process identifier X has a unique defining equation $\mathrm{X}(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$ such that $\mathrm{fv}(P) \subseteq \{x_1, \ldots, x_j\} \subseteq \mathcal{V}$. We will sometimes omit the arguments of $\mathrm{X}(\boldsymbol{u}, \boldsymbol{e})$ when they are clear from the context. As usual, we omit trailing occurrences of $\mathbf{0}$ in processes.

3.2. Semantics

The operational semantics of CO_2 systems is formalised by the labelled transition relation $\rightarrow$ in Figure 4, where we consider processes and systems up-to the congruence relation $\equiv$ in Figure 3. The axioms for $\equiv$ are fairly standard — except the bottom-rightmost one: it collects garbage terms possibly arising from variable substitutions. The labels μ of the transition relation can be of the following forms: $\mathsf{A} : \pi$ (where π is not $\texttt{do}$), $\mathsf{A} : \texttt{if}$, $\mathsf{A} : \texttt{do}_s\,\texttt{a!v}$, $\mathsf{A} : \texttt{do}_s\,\texttt{a?v}$, or $\mathsf{K} : \texttt{fuse}$.

We now briefly discuss the rules in Figure 4. Rule [TAU] just fires a τ prefix. Rule [TELL] advertises a latent contract $\{\downarrow_x c\}_{\mathsf{A}}$. Rule [FUSE] finds *agreements* among the latent contracts: this happens when there

exist $\{\downarrow_x c\}_{\mathsf{A}}$ and $\{\downarrow_y d\}_{\mathsf{B}}$ such that $\mathsf{A} \neq \mathsf{B}$ and $c \bowtie d$. Once an agreement is found, a fresh session containing $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$ is created. Rule [If] evaluates the guard of a conditional, and then proceeds with one of the branches. Rule [Do!] allows a participant A to choose a branch label in a contract configuration γ within session s, and to send the value resulting from the evaluation of e (which results in γ evolving to a suitable γ'). Rule [Do?] allows A to receive a value v, resulting from a [Rdy] transition of the contract configuration γ within session s, and to bind to v the free occurrences of x within the continuation P. Rule [Ask] allows A to proceed only if the contract γ at session s satisfies the formula ϕ (the predicate $\gamma \vdash \phi$ used in the rule premise is specified in Definition 3.3). The last three rules are mostly standard. In rule [Del] the label π fired in the premise becomes τ in the consequence, when π contains the delimited name/variable. This transformation is defined by the function $\mathrm{del}_u(\pi)$: for instance, $(x)\, \mathsf{A}[\mathtt{tell} \downarrow_x c.P] \xrightarrow{\mathsf{A}:\,\tau} (x)\,(\mathsf{A}[P] \mid \{\downarrow_x c\}_{\mathsf{A}})$. Here, it would make little sense to have the label $\mathsf{A} : \mathtt{tell} \downarrow_x c$, as x (being delimited) may be α-converted.

Hereafter, we shall assume that systems are always *well-typed*, i.e.: (i) the syntactic constraints required in Section 3.1 are respected; (ii) the guards in conditionals have sort `bool`; (iii) the sorts of the expressions passed to named processes are coherent with those in the corresponding defining equations. Ensuring such form of well-typedness can be easily done through standard type checking techniques.

Example 3.2 (Online store). *Recall c_{A} from Example 2.7. A possible specification of the store is:*

$$\begin{aligned}
P_{\mathsf{A}} &= (x)\ (\mathtt{tell} \downarrow_x c_{\mathsf{A}}.\mathtt{do}_x\ \mathtt{addToCart?}n.P_{\mathsf{add}}(x,n)) \\
P_{\mathsf{add}}(x,t) &\stackrel{\text{def}}{=} \mathtt{do}_x\ \mathtt{addToCart?}n.P_{\mathsf{add}}(x,n+t)\ +\ \mathtt{do}_x\ \mathtt{checkout?}.P_{\mathsf{pay}}(x,n+t) \\
P_{\mathsf{pay}}(x,t) &\stackrel{\text{def}}{=} \mathtt{do}_x\ \mathtt{pay?}.P_{\mathsf{ack}}(x,t)\ +\ \mathtt{do}_x\ \mathtt{cancel?} \\
P_{\mathsf{ack}}(x,t) &\stackrel{\text{def}}{=} \mathtt{if}\ t < 100\ \mathtt{then}\ \mathtt{do}_x\ \mathtt{ok!}\ \mathtt{else}\ \mathtt{do}_x\ \mathtt{no!}.P_{\mathsf{pay}}(x,t)
\end{aligned}$$

The process P_{A} first advertises the contract c_{A}, and then waits that the user has performed the first `addToCart` (note that it also requires that x is instantiated with a new session containing c_{A}). Then, the store enters a recursive process P_{add}, wherein it accepts two user choices: `addToCart`, which is followed by a recursive call to P_{add}, and `checkout`, which passes the control to process P_{pay}. In the meanwhile, the overall amount to pay is accumulated in variable t (we are assuming that the value n passed by `addToCart` is the price of an item; in more concrete implementations, such value could be obtained e.g. from product database). Within P_{pay}, after the payment is received, the store internally chooses (with a rather arbitrary policy) whether to accept it or not: in the first case, it terminates; in the second case, it proceeds with P_{pay}, thus allowing the user to retry the payment. In more concrete implementations (see e.g. Example 3.9), the process of determining whether to accept the payment may be delegated to a third party, e.g. a bank.

Now, let B be a buyer with contract $d_{\mathsf{B}} = \mathtt{addToCart?int}.\,c_{\mathsf{B}}$ as in Example 2.7, and let:

$$Q_{\mathsf{B}} = (y)\,\mathtt{tell} \downarrow_y d_{\mathsf{B}}.Y \qquad\qquad Y \stackrel{\text{def}}{=} \mathtt{do}_y\ \mathtt{addToCart!}51.\mathtt{do}_y\ \mathtt{checkout!}.\mathtt{do}_y\ \mathtt{pay!}cc.\mathtt{do}_y\ \mathtt{ok?}$$

A possible, successful computation of the system $S = \mathsf{A}[P_{\mathsf{A}}] \mid \mathsf{B}[Q_{\mathsf{B}}]$ is the following:

$$\begin{aligned}
S &\to^* (x,y)\ \big(\{\downarrow_x c_{\mathsf{A}}\}_{\mathsf{A}} \mid \{\downarrow_y d_{\mathsf{B}}\}_{\mathsf{B}} \mid \mathsf{A}[\mathrm{X}] \mid \mathsf{B}[Y]\big) \quad \textit{where } \mathrm{X} = \mathtt{do}_x\ \mathtt{addToCart?}n.\,P_{\mathsf{add}}(x,n) \\
&\to (s)\ \big(\mathsf{A}[\mathrm{X}\{s/x\}] \mid \mathsf{B}[Y\{s/y\}] \mid s[\mathsf{A} : c_{\mathsf{A}} \mid \mathsf{B} : d_{\mathsf{B}}]\big) \\
&\to^* (s)\ \big(\mathsf{A}[\mathrm{X}\{s/x\}] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{checkout!}.\cdots] \mid s[\mathsf{A} : c_{\mathsf{A}} \mid \mathsf{B} : c_{\mathsf{B}}]\big) \\
&\to^* (s)\ \big(\mathsf{A}[P_{\mathsf{add}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{checkout!}.\cdots] \mid s[\mathsf{A} : c'_{\mathsf{A}} \mid \mathsf{B} : c_{\mathsf{B}}]\big) \quad \textit{where } c'_{\mathsf{A}} = \mathtt{rec}\ Z.\cdots \\
&\to^* (s)\ \big(\mathsf{A}[P_{\mathsf{add}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{pay!}cc.\,\mathtt{do}_s\ \mathtt{ok?}] \mid s[\mathsf{A} : c'_{\mathsf{A}} \mid \mathsf{B} : c'_{\mathsf{B}}]\big) \quad \textit{where } c'_{\mathsf{B}} = \mathtt{pay!string}.\cdots \\
&\to^* (s)\ \big(\mathsf{A}[P_{\mathsf{pay}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{pay!}cc.\,\mathtt{do}_s\ \mathtt{ok?}] \mid s[\mathsf{A} : c_{\mathsf{pay}} \mid \mathsf{B} : c'_{\mathsf{B}}]\big) \\
&\to (s)\ \big(\mathsf{A}[P_{\mathsf{pay}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{ok?}] \mid s[\mathsf{A} : c'_{\mathsf{pay}} \mid \mathsf{B} : c''_{\mathsf{B}}]\big) \quad \textit{where } c''_{\mathsf{B}} = \mathtt{ok?} + \cdots \textit{ and } c'_{\mathsf{pay}} = \mathtt{rdy}\ \mathtt{pay?}cc\,.\cdots \\
&\to (s)\ \big(\mathsf{A}[P_{\mathsf{ack}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{ok?}] \mid s[\mathsf{A} : c_{\mathsf{ack}} \mid \mathsf{B} : c''_{\mathsf{B}}]\big) \quad \textit{where } c_{\mathsf{ack}} = \mathtt{ok!} + \cdots \\
&\to (s)\ \big(\mathsf{A}[\mathbf{0}] \mid \mathsf{B}[\mathtt{do}_s\ \mathtt{ok?}] \mid s[\mathsf{A} : 0 \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{ok?}\,.\,0]\big) \\
&\to (s)\ \big(\mathsf{A}[\mathbf{0}] \mid \mathsf{B}[\mathbf{0}] \mid s[\mathsf{A} : 0 \mid \mathsf{B} : 0]\big)
\end{aligned}$$

Notice that the buyer has been quite lucky, since the store has chosen `ok` *in the last step; otherwise, the buyer would have been culpable of a contract violation at session s.*

We now formalise the meaning of the entailment relation $\vdash$ used in rule [Ask]. We denote with $\mathbf{A}$ a set of atomic propositions, whose elements are terms of the form $(\mathtt{a}, \mathtt{T})\circ$, where $\mathtt{a}$ is a branch label, $\mathtt{T}$ is a sort, and $\circ \in \{\mathtt{!}, \mathtt{?}\}$. We let $\ell, \ell', \ldots$ range over $\mathbf{A} \cup \{\varepsilon\}$.

Definition 3.3. *We define the (unlabelled) transition system* $\mathrm{TS} = (\Sigma, \rightarrow, \mathbf{A}, L)$ *as follows:*

- $\Sigma = \{(\ell, \gamma) \mid \ell \in \mathbf{A} \cup \{\varepsilon\},$ *and* γ *is a configuration of compliant contracts*$\}$,
- *the transition relation* $\rightarrow \subseteq \Sigma \times \Sigma$ *is defined by the following rule:*

$$(\ell, \gamma) \rightarrow ((\mathtt{a}, \mathtt{T})\circ, \gamma') \qquad \text{if } \gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\!\rightarrow \gamma' \text{ and } \mathtt{v} : \mathtt{T}$$

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

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

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

Then, we write $\gamma \vdash \phi$ *whenever* $\forall \lambda \in \mathrm{Paths}((\varepsilon, \gamma))$, $\lambda \models \phi$ *holds in LTL.*

The last line of Definition 3.3 explains the use of the symbol ε: it is a dummy label, needed to decorate the initial state of a trace. Note that the standard LTL semantics is defined for infinite traces only. However, finite traces can be transformed into infinite ones by adding to the TS a special final state with a self loop, and a transition from every terminal state (of the original TS) to that final state. We have chosen to do not explicitly perform the above transformation, in order to keep the theory faithful to the Maude implementation. Indeed, the transformation is internally accomplished by the Maude engine.

3.3. Honesty

CO_2 allows for writing *dishonest* agents which do not fulfil their contractual obligations, in some contexts. To formalise the notion of honesty, we start by defining the set $\mathrm{O}^{\mathsf{A}}_s(S)$ of *obligations* of a participant A at a session s in S. The intuition is that, whenever A is culpable at some session s, she has to fire one of the actions in $\mathrm{O}^{\mathsf{A}}_s(S)$ to exculpate herself (possibly in two steps, according to Theorem 2.10).

Definition 3.4 (Obligations). *We define the set of branch labels* $\mathrm{O}^{\mathsf{A}}_s(S)$ *as:*

$$\mathrm{O}^{\mathsf{A}}_s(S) \;=\; \left\{ \mathtt{a} \;\middle|\; \exists \gamma, S', \mathtt{v}, \circ \;:\; S \equiv s[\gamma] \mid S' \text{ and } \gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\!\rightarrow \right\}$$

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

A participant A is *ready* in a system S if, whenever A obligations in S, she can fulfil some of them (so, if A does not occur in S or has no obligations there, then she is trivially ready). 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 $\mathtt{do}_s$ of A. The set $\mathrm{Rdy}^{\mathsf{A}}_s$ collects all the systems where A may perform some action at s after a finite sequence of transitions of A not involving any $\mathtt{do}$ at s. Note that A is vacuously ready in all systems in which she does not have any obligations.

Definition 3.5 (Readiness). *We define the set of systems* $\mathrm{Rdy}^{\mathsf{A}}_s$ *as the smallest set such that:*

1. $S \xrightarrow{\mathsf{A}:\,\mathtt{do}_s} \implies S \in \mathrm{Rdy}^{\mathsf{A}}_s$
2. $\left(S \xrightarrow{\mathsf{A}:\,\neq\mathtt{do}_s} S' \wedge S' \in \mathrm{Rdy}^{\mathsf{A}}_s\right) \implies S \in \mathrm{Rdy}^{\mathsf{A}}_s$

Then, we say that:

1. A *is* ready at s in S *whenever* $S \in \mathrm{Rdy}_s^{\mathsf{A}}$.

2. A *is* ready in S *iff* $S \equiv (\boldsymbol{u})\, S' \;\wedge\; \forall s \;:\; \mathrm{O}_s^{\mathsf{A}}(S') \neq \emptyset \implies S' \in \mathrm{Rdy}_s^{\mathsf{A}}$

We can now formalise when a participant is *honest*. Roughly, $\mathsf{A}[P]$ is honest in a *fixed* system S when A is ready in all evolutions of $\mathsf{A}[P] \mid S$. Then, we say that $\mathsf{A}[P]$ is honest when she is honest in *all* systems S.

Definition 3.6 (Honesty). *We say that:*

1. S is A-free *iff it has no latent/stipulated contracts of* A*, nor processes of* A

2. P *is* honest in S *iff* $\forall \mathsf{A} : \big(S$ *is* A*-free* $\wedge\; \mathsf{A}[P] \mid S \to^* S'\big) \implies \mathsf{A}$ *is ready in* S'

3. P *is* honest *iff* $\forall S : P$ *is honest in* S.

Note that in item 2 we are quantifying over all A: this is just needed to associate P to a participant name, with the only constraint that such name must not be present in the environment S used to test P. In the absence of the A-freeness constraint, the notion of honesty would be impractically strict: indeed, were S already carrying stipulated or latent contracts of A, e.g. with $S = s[\mathsf{A} : \mathtt{pay100Keu!} \mid \mathsf{B} : \mathtt{pay100Keu?}]$, it would be unreasonable to ask participant A to fulfil them. Note however that S can contain latent contracts and sessions involving *any* other participant different from A: in a sense, the honesty of $\mathsf{A}[P]$ ensures a good behaviour even in the (quite realistic) case where $\mathsf{A}[P]$ is inserted in a system which has already started.

Example 3.7 (Basic examples of honesty). *Consider the following processes:*

1. $P_1 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a!} \oplus \mathtt{b!}.\mathtt{do}_x\,\mathtt{a!}$

2. $P_2 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a!}.(\tau.\mathtt{do}_x\,\mathtt{a!} + \tau.\mathtt{do}_x\,\mathtt{b!})$

3. $P_3 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a?} + \mathtt{b?}.\mathtt{do}_x\,\mathtt{a?}$

4. $P_4 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a?} + \mathtt{b?}.\big(\mathtt{do}_x\,\mathtt{a?} + \mathtt{do}_x\,\mathtt{b?} + \mathtt{do}_x\,\mathtt{c?}\big)$

5. $P_5 \;=\; (x, y)\;\mathtt{tell}\downarrow_x \mathtt{a?}.\mathtt{tell}\downarrow_y \mathtt{b!}.\mathtt{do}_x\,\mathtt{a?}.\mathtt{do}_y\,\mathtt{b!}$

6. $P_6 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a!}.\mathrm{X}(x) \qquad \mathrm{X}(x) \stackrel{\text{def}}{=} \tau.\tau.\mathrm{X}(x) + \tau.\mathtt{do}_x\,\mathtt{a!}$

7. $P_7 \;=\; (x)\;\mathtt{tell}\downarrow_x \mathtt{a!}.\mathrm{X}(x) \qquad \mathrm{X}(x) \stackrel{\text{def}}{=} \mathtt{if\ true\ then}\ \tau.\mathrm{X}(x)\ \mathtt{else}\ \mathtt{do}_x\,\mathtt{a!}$

We now discuss the honesty (or dishonesty) of these processes.

- P_1 *is honest: it is advertising an internal choice between* $\mathtt{a!}$ *and* $\mathtt{b!}$*, and then it is doing* $\mathtt{a!}$.

- P_2 *is dishonest: if the rightmost* τ *is fired, then the process cannot do the promised* $\mathtt{a!}$*. Note that* P_2 *is dishonest in* all *contexts where the session* x *is fused.*

- P_3 *is dishonest: indeed, if the other participant involved at session* x *chooses* $\mathtt{b!}$*, then* P_3 *cannot do the corresponding input. Note instead that* P_3 *is honest in all contexts where either the session* x *is not fused, or the other participant at* x *does not fire* $\mathtt{b!}$.

- P_4 *is honest: note that the branch* $\mathtt{c?}$ *can never be taken. Indeed, an action can be fired by a process at session* s *only if it is enabled by the contract configuration in* s *(see the premise of rule* [Do?]*).*

- P_5 *is dishonest, for two different reasons. First, in contexts where session* y *is fused and* x *is not, the* $\mathtt{do}_y\,\mathtt{b!}$ *cannot be reached (and so the contract at session* y *is not fulfilled). Second, also in those contexts where both sessions are fused, if the other participant at session* x *never does* $\mathtt{a!}$*, then* $\mathtt{do}_y\,\mathtt{b!}$ *cannot be reached.*

- P_6 *is honest. The process is advertising a singleton internal choice, and then non-deterministically choosing to either perform an internal action followed by the* do *action, or to perform an internal action and then loop. Although there exists a computation where* a! *is never performed (the infinite sequence of internal actions), under a fair scheduler the rightmost* τ*s, which is enabled infinitely often, will be performed. We now show that* P_6 *is honest in* $S = (y)(\{\downarrow_y \texttt{a?}\}_{\mathsf{B}})$*, with* $\mathsf{A} \neq \mathsf{B}$ *(the generalisation to arbitrary contexts is straightforward). The reachable states from* $\mathsf{A}[P_6] \mid S$ *(up-to structural congruence) are the following:*

  1. $\mathsf{A}[P_6] \mid S$*. Here* A *is vacuously ready, because no session has been established yet.*
  2. $(x, y)(\mathsf{A}[\mathrm{X}(x)] \mid \{\downarrow_x \texttt{a!}\}_{\mathsf{A}} \mid \{\downarrow_y \texttt{a?}\}_{\mathsf{B}})$*. Here* A *is vacuously ready, as in the previous item.*
  3. $(s)(\mathsf{A}[\texttt{do}_s\, \texttt{a!}] \mid s[\mathsf{A} : \texttt{a!} \mid \mathsf{B} : \texttt{a?}])$*. Here* A *is ready, because* $\mathsf{A}[\texttt{do}_s\, \texttt{a!}] \mid s[\mathsf{A} : \texttt{a!} \mid \mathsf{B} : \texttt{a?}] \in \mathrm{Rdy}_s^{\mathsf{A}}$*, by item 1 of Definition 3.5.*
  4. $(s)(\mathsf{A}[\mathrm{X}(s)] \mid s[\mathsf{A} : \texttt{a!} \mid \mathsf{B} : \texttt{a?}])$*. Since* $\mathsf{A}[\mathrm{X}(s)] \mid s[\mathsf{A} : \texttt{a!} \mid \mathsf{B} : \texttt{a?}] \xrightarrow{\mathsf{A}:\,\tau} \mathsf{A}[\texttt{do}_s\, \texttt{a!}] \mid s[\mathsf{A} : \texttt{a!} \mid \mathsf{B} : \texttt{a?}]$*, which is ready for the previous item, then* A *is ready.*
  5. $(s)(\mathsf{A}[\mathbf{0}] \mid s[\mathsf{A} : 0 \mid \mathsf{B} : \texttt{rdy a?}])$*. Here* A *is vacuously ready, because she has not obligations in* s*.*

- P_7 *is dishonest: it is advertising a singleton internal choice, and then, since the condition in the* if *is* true*, it can never take the branch which would fulfil the obligation to do* a!*.*

Observe that items 6 and 7 in Example 3.7 show that it would be *incorrect* to verify the honesty of a process if e then P else Q as we would do for process $\tau.P + \tau.Q$.

Example 3.8 (Online store). *The specification of the online store* P_{A} *in Example 3.2 is honest. Instead, the buyer process* Q_{B} *in the same example in* not *honest. For instance, assume* Q_{B} *is run in the context* P_{A}*. If the store chooses to refuse the payment (e.g., it executes* no!*), then the system* $P_{\mathsf{A}} \mid Q_{\mathsf{B}}$ *reaches the state:*

$$S' = (s)\ S'' \qquad \textit{where} \quad S'' \;=\; (s)\ \big(\mathsf{A}[\mathbf{0}] \mid \mathsf{B}[\texttt{do}_s\, \texttt{ok?}] \mid s[\mathsf{A} : 0 \mid \mathsf{B} : \texttt{rdy no?}.\,0]\big)$$

By Definition 3.4, we have that $\mathrm{O}_s^{\mathsf{B}}(S'') = \{\texttt{no}\}$*, and by Definition 3.5 it follows that* $S'' \notin \mathrm{Rdy}_s^{\mathsf{B}}$*. Therefore,* B *is* not *ready in* S'*, and so by Definition 3.6 we conclude that* Q_{B} *is not honest.* □

Example 3.9 (Online store with bank). *We now refine the specifications of the online store provided in Example 3.2, by delegating a bank* B *to determine whether to accept or not buyer payments. The store advertises the following contract between itself and a bank:*

$$d_{\mathsf{B}} \;=\; \texttt{rec}\,Y.\ \texttt{ccnum!string}.\big(\texttt{amount!int}.\ (\texttt{accept?} + \texttt{deny?}.\,Y)\ \oplus\ \texttt{abort!}\big)\ \oplus\ \texttt{abort!}$$

*The store first sends a credit card number to the bank (*ccnum*), and then the* amount *to pay. After this, it waits the response from the bank, which can be either* accept *or* deny*; in the latter case, the whole procedure can be repeated. Note that each internal choice is associated to an* abort *branch, which allows the store to terminate the interaction with the bank.*

A possible implementation of the online store is the following:

$$\begin{array}{rcl}
P_{\mathsf{A}} & = & (x)\ (\texttt{tell}\downarrow_x c_{\mathsf{A}}.\,\texttt{do}_x\,\texttt{addToCart?}n.\,P_{\texttt{add}}(x,n)) \\
P_{\texttt{add}}(x,t) & \stackrel{def}{=} & \texttt{do}_x\,\texttt{addToCart?}n.\,P_{\texttt{add}}(x,n+t)\ +\ \texttt{do}_x\,\texttt{checkout?}.\,(y)\,\texttt{tell}\downarrow_y d_{\mathsf{B}}.\,P_{\texttt{pay}}(x,y,t) \\
P_{\texttt{pay}}(x,y,t) & \stackrel{def}{=} & \texttt{do}_x\,\texttt{pay?}w.\,P_{\texttt{bank}}(x,y,w,t)\ +\ \texttt{do}_x\,\texttt{cancel?}.\,\texttt{do}_y\,\texttt{abort!} \\
P_{\texttt{bank}}(x,y,w,t) & \stackrel{def}{=} & \texttt{do}_y\,\texttt{ccnum!}w.\,\texttt{do}_y\,\texttt{amount!}t.\,P_{\texttt{ack}}(x,y,t) \\
P_{\texttt{ack}}(x,y,t) & \stackrel{def}{=} & \texttt{do}_y\,\texttt{accept?}.\,\texttt{do}_x\,\texttt{ok!}\ +\ \texttt{do}_y\,\texttt{deny?}.\,\texttt{do}_x\,\texttt{no!}.\,P_{\texttt{pay}}(x,y,t)
\end{array}$$

We now comment the differences with respect to Example 3.2. After checkout*, the process* $P_{\texttt{add}}$ *advertises the contract* d_{B} *for the bank. In process* $P_{\texttt{bank}}$*, the store sends to the bank the credit card number* w *provided by the buyer, and the amount* t *to pay. Then, in process* $P_{\texttt{ack}}$*, the store waits for the bank response, and it acknowledges the buyer accordingly.*

The process P_{A} is not *honest. One reason is that if either the buyer or the bank is dishonest, the store is not capable of keeping a correct interaction with the other party: for instance, if the bank does not respond in process $P_{\mathtt{ack}}$, then the buyer will never receive the expected $\mathtt{ok}/\mathtt{no}$ message. Another reason is that if contract d_{B} is not fused in a session (i.e., an agreement is not found with a bank), then $P_{\mathtt{bank}}$ will be stuck, since session y will not be initialised. Therefore, in both these scenarios the store will remain culpable of a contract violation towards the buyer.*

In Section 5.1 we will show how to use our Maude honesty checker to verify the dishonesty of the store. Further, we will show how to use the output of the tool in order to revise the specification of the store, so to finally make it honest (and verified as such by the tool). □

Example 3.10 (Unfair scheduler). *Consider the following processes:*

$$P = (x)\ \mathtt{tell} \downarrow_x \mathtt{a!}.\ \mathtt{do}_x\, \mathtt{a!} \qquad\qquad Q = (y)\ \mathtt{tell} \downarrow_y \mathtt{a?}.\ \mathrm{X} \quad \text{where } \mathrm{X} \stackrel{\text{def}}{=} \tau.\ \mathrm{X}$$

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

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

In the above computation, an unfair *scheduler prevents A from making her moves, and so A remains persistently culpable in such computation. However, A is ready in S' (because the $\mathtt{do}_s\, \mathtt{a!}$ is enabled), and therefore P is honest according to Definition 3.6. 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.* □

4. Model checking honesty

We now address the problem of automatically verifying honesty. As mentioned in Section 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 [31] (also when restricting to CO_2 without value-passing), our goal is a verification technique which safely over-approximates it: i.e., only honest processes must be classified as such.

A first issue is that Definition 3.6 requires readiness to be preserved in *all* possible contexts, and there is an *infinite* number of such contexts. Clearly, this prevents us from using standard techniques for model checking finite-state systems. Another issue is that, even considering a fixed context and the usual syntactic restrictions required to make processes finite-state (e.g. no delimitation/parallel under process definitions), value-passing makes the semantics of CO_2 infinite-state.

To overcome these problems, we present below an *abstract* semantics of CO_2 which safely approximates the honesty property of a process P, while neglecting values and the actual context wherein it is executed. The definition of the abstract semantics of CO_2 is obtained in two steps.

1. First, in Section 4.1 we devise a *value abstraction* $\alpha^\star$ of systems (Definition 4.1), which replaces each expression e with a special value $\star$. In Theorem 4.3 we show that value abstraction is *sound* with respect to honesty: i.e., if $\alpha^\star(P)$ is honest, then also the concrete process P is honest. Furthermore, value abstraction is *complete* whenever P contains no conditional expressions, i.e. if P is honest and it is $\mathtt{if}$-free, then $\alpha^\star(P)$ is honest, too.

2. Second, in Sections 4.2 and 4.3 we provide a *context abstraction* α_{A} of contracts and systems (Definitions 4.4 and 4.9, respectively). The abstraction α_{A} is parameterised by the participant A the honesty of which is under consideration: basically, $\alpha_{\mathsf{A}}(S)$ discards the part of the system S not governed by A, by over-approximating its moves. Theorem 4.11 states that this abstraction is sound, too, and it is also complete for $\mathtt{ask}$-free processes.

$$\mathsf{A}[(\texttt{if} \star \texttt{ then } P_0 \texttt{ else } P_1) \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{if}}_\star \mathsf{A}[P_i \mid Q] \qquad (i \in \{0,1\}) \qquad [\alpha^\star\text{IF}]$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:a}\!\!\!\twoheadrightarrow_\star \gamma'}{\mathsf{A}[\texttt{do}_s\, a.P + P' \mid Q] \;\mid\; s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{do}_s\, a}_\star \mathsf{A}[P \mid Q] \;\mid\; s[\gamma']} \qquad [\alpha^\star\text{DO}]$$

$$\frac{\gamma \vdash_\star \phi}{\mathsf{A}[\texttt{ask}_s\, \phi.P + P' \mid Q] \;\mid\; s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{ask}_s\, \phi}_\star \mathsf{A}[P \mid Q] \;\mid\; s[\gamma]} \qquad [\alpha^\star\text{ASK}]$$

Figure 5: Reduction semantics of value-abstract systems (full set of rules in Section B.1).

Summing up, by composing the two abstractions we obtain a sound over-approximation of honesty: namely, if $\alpha_{\mathsf{A}}(\alpha^\star(P))$ is honest, then the concrete process P is honest (Theorem 4.12). Conversely, if P is honest, `if`-free and `ask`-free, then $\alpha_{\mathsf{A}}(\alpha^\star(P))$ is honest, too. When P is a finite state process (i.e., without delimitation/parallel under process definitions), then the honesty of $\alpha_{\mathsf{A}}(\alpha^\star(P))$ can be verified by model checking its state space. In Section 4.5 we outline an implementation in Maude of our verification technique.

4.1. Value abstraction

Our first step towards the verification of honesty is defining a transformation of CO_2 processes and systems which abstracts from values (Definition 4.1). We provide *value-abstract systems* with a semantics (Figure 5), and in Definition 4.2 we accordingly refine the notion of honesty. Theorem 4.3 states the soundness (and, for `if`-free processes, also the completeness) of value abstraction.

Definition 4.1 (Value abstraction of processes). *For all processes P, and for all functions Γ from (value-kinded) variables to sorts, we define the value-abstract process $\alpha^\star_\Gamma(P)$ inductively as follows:*

$$\alpha^\star_\Gamma(\sum_i \pi_i.P_i) = \sum_i \alpha^\star_\Gamma(\pi_i.P_i) \qquad \alpha^\star_\Gamma(\pi.P) = \begin{cases} \texttt{do}_u\,(\texttt{a},\texttt{T})!.\alpha^\star_\Gamma(P) & \text{if } \pi = \texttt{do}_u\,\texttt{a}!e \text{ and } e : \texttt{T} \text{ in } \Gamma \\ \texttt{do}_u\,(\texttt{a},\texttt{T})?.\alpha^\star_{\Gamma,x:\texttt{T}}(P) & \text{if } \pi = \texttt{do}_u\,\texttt{a}?x : \texttt{T} \\ \pi.\alpha^\star_\Gamma(P) & \text{if } \pi \neq \texttt{do} \end{cases}$$

$$\alpha^\star(\mathrm{X}(\boldsymbol{u}, \boldsymbol{e})) = \mathrm{X}(\boldsymbol{u}, \star) \qquad \alpha^\star_\Gamma(\texttt{if } e \texttt{ then } P \texttt{ else } Q) = \texttt{if} \star \texttt{ then } \alpha^\star_\Gamma(P) \texttt{ else } \alpha^\star_\Gamma(Q)$$

$$\alpha^\star_\Gamma((u)\,P) = (u)\,\alpha^\star_\Gamma(P) \qquad \alpha^\star_\Gamma(P \mid Q) = \alpha^\star_\Gamma(P) \mid \alpha^\star_\Gamma(Q)$$

and we simply write $\alpha^\star(P)$ when the mapping Γ is empty. The value abstraction of systems just applies $\alpha^\star(\cdot)$ to each process and contract configuration within a system (see Definition B.1 for details).

The semantics of value-abstract systems is given by a set of rules, most of which are similar to those in Figure 4. Hence, in Figure 5 we only show those which differ substantially from the rules for concrete systems (the full set of rules is provided in Section B.1). When clear from the context, we shall overload the meaning of metavariables $S, S', \ldots$, and we use them also to range over value-abstract systems. Rule [$\alpha^\star$IF] takes into account the fact that the guards in conditionals are abstracted as $\star$: hence, a correct over-approximation of the concrete semantics requires to take both branches of a conditional. Rule [$\alpha^\star$DO] abstracts the concrete [DO!] and [DO?] rules. To do that, it just exploits the semantics $\twoheadrightarrow_\star$ of value-abstract contracts (Section 2.3), whose transitions do not carry values. Rule [$\alpha^\star$ASK] uses in its premise a value-abstract entailment relation $\vdash_\star$ (this is a minor modification of the concrete relation, see Definition B.3).

The notion of honesty for value-abstract systems (Definition 4.2 below) requires a slight modification of that for concrete systems. When the move of A is an `if`, value-abstract readiness requires that *both* branches of the conditional are ready at s (item 3). Note that this introduces an over-approximation of readiness:

$$a\,.\,c \oplus c' \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} ctx\ \mathrm{co}(a).\,c \qquad a\,.\,c + c' \xrightarrow{ctx:\mathrm{co}(a)}\!\!\!\!\to_{\mathsf{A}} \mathtt{rdy}\ a.\,c \qquad \mathtt{rdy}\ a.\,c \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} c \qquad ctx\ a.\,c \xrightarrow{ctx:a}\!\!\!\!\to_{\mathsf{A}} c$$

Figure 6: Semantics of context-abstract contracts.

for instance, if $S = \mathsf{A}[\mathtt{if\ true\ then\ do}_s\ \mathtt{a!v\ else\ 0}]$, then A is ready at s in S according to Definition 3.5, while A is *not* $\alpha^\star$-ready at s in $\alpha^\star(S)$ according to Definition 4.2, because item 3 is not satisfied.

Definition 4.2 (Value-abstract readiness). *Given a session name s and participant A, we define the set of value-abstract systems $\alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$ as the smallest set such that:*

1. $S \xrightarrow{\mathsf{A}:\,\mathtt{do}_s}_\star \implies S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$
2. $\big(S \xrightarrow{\mathsf{A}:\,\neq\{\mathtt{do}_s,\mathtt{if}\}}_\star S' \ \wedge\ S' \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}\,\big) \implies S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$
3. $S \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star \ \wedge (\forall S' : S \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star S' \implies S' \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}) \implies S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$

We say that A is $\alpha^\star$-ready at s in S when $S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$.

We define value-abstract honesty ($\alpha^\star$-honesty) as in Definition 3.6, except for using the value-abstract notion of readiness instead of the concrete one.

Theorem 4.3 below states that $\alpha^\star$-honesty is a sound over-approximation of the concrete notion. This approximation is also complete in the absence of conditional statements.

Theorem 4.3. *Let P be a concrete process. If $\alpha^\star(P)$ is $\alpha^\star$-honest, then P is honest. Conversely, if P is honest and* `if`*-free, then $\alpha^\star(P)$ is $\alpha^\star$-honest.*

4.2. Context abstraction of contracts

After having defined a value abstraction for CO_2 systems (Section 4.1), we now introduce our second step towards the verification of honesty: we fix a participant A, and we abstract her contracts from the context, by removing all the information related to other participants (whose names are abstracted as ctx). After introducing *context abstraction for contracts* (Definition 4.4 and Figure 6), we will prove its main properties (Theorem 4.5), and finally ensure that such an abstraction is sound with respect to the semantics of the CO_2 `ask` primitive (Definition 4.6, Definition 4.7 and Lemma 4.8).

Let γ be a value-abstract contract configuration (Section 2.3): the *context-abstraction* $\alpha_{\mathsf{A}}(\gamma)$ (Definition 4.4) maintains only the contract of A, while recording whether the opponent contract has a `rdy`.

Definition 4.4 (Context abstraction of contracts). *For all value-abstract contract configurations γ, we define the context-abstract contract $\alpha_{\mathsf{A}}(\gamma)$ as follows:*

$$\alpha_{\mathsf{A}}(\mathsf{A} : c \mid \mathsf{B} : d) \quad = \quad \begin{cases} c & \text{if } d \text{ is } \mathtt{rdy}\text{-free} \\ ctx\ a.\,c & \text{if } d = \mathtt{rdy}\ a.\,d' \end{cases}$$

For all participants A, the LTS $\twoheadrightarrow_{\mathsf{A}}$ on context-abstract contracts is defined by the rules in Figure 6. Labels of $\twoheadrightarrow_{\mathsf{A}}$ are atoms, possibly prefixed with ctx — which indicates a contractual action performed by the context. In an internal sum, A chooses a branch; in an external sum, the choice is made by the context; in a `rdy` $a.c$ the atom a is fired. The rightmost rule handles a `rdy` in the context contract.

The following theorem highlights some relevant properties of context abstraction of contracts, which will be exploited to prove the correctness of our verification technique for honesty. Items 1 and 2 state that each

transition of a value-abstract contract configuration γ can be simulated by its context abstraction $\alpha_{\mathsf{A}}(\gamma)$. Conversely, item 3 states that each (non-ctx) transition of a context-abstract contract c can be simulated by all its concretisations γ_c (the ctx case is only needed to prove the completeness of our verification technique, and it is dealt with by Lemma B.22).

Theorem 4.5. *For all value-abstract contract configurations γ, γ', for all context-abstract contracts c, c':*

1. $\gamma \xrightarrow{\mathsf{A}:a}\!\!\!\!\rightarrow_\star \gamma' \implies \alpha_{\mathsf{A}}(\gamma) \xrightarrow{a}\!\!\!\!\rightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\gamma')$
2. $\gamma \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_\star \gamma' \implies \alpha_{\mathsf{A}}(\gamma) \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\gamma') \qquad (\mathsf{B} \neq \mathsf{A})$
3. $c \xrightarrow{a}\!\!\!\!\rightarrow_{\mathsf{A}} c' \implies \forall$ *compliant* $\gamma_c : \big(\alpha_{\mathsf{A}}(\gamma_c) = c \implies \exists \gamma_{c'} : \gamma_c \xrightarrow{\mathsf{A}:a}\!\!\!\!\rightarrow_\star \gamma_{c'} \wedge \alpha_{\mathsf{A}}(\gamma_{c'}) = c'\big)$

Another desirable property of context abstraction of contracts is formalised in Definition 4.6 below. There, we consider the soundness of context abstraction with respect to the entailment relation $\vdash_\star$ used in rule [$\alpha^\star$Ask] of the value-abstract semantics of CO_2 (Figure 5). Assume we are given two relations $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ between context-abstract contracts and formulae (of some temporal logics):

- item 1 requires that, whenever $c \vdash_{\mathsf{A}} \phi$, then also *all* (value-abstract) concretisations γ of c entail ϕ. When this holds, each concrete transition of A enabled by rule [Ask] (with premise $\gamma \vdash_\star \phi$) can be simulated by an abstract transition (with premise $\alpha_{\mathsf{A}}(\gamma) \vdash_{\mathsf{A}} \phi$). This allows for abstracting from the context of A — in particular, from the actual contracts advertised by the other participants.
- item 2 plays the same role with the moves of the context. Here, soundness requires that, whenever there exists some concretisation γ of c which entails ϕ, then $c \vdash_{ctx} \phi$ must also hold.

Definition 4.6 (Sound context-abstract entailment). *Let $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ be relations between context-abstract contracts and formulae. We say that $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ are* sound *whenever they satisfy, respectively:*

1. $c \vdash_{\mathsf{A}} \phi \implies (\forall$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \implies \gamma \vdash_\star \phi)$
2. $c \vdash_{ctx} \phi \impliedby (\exists$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \wedge \gamma \vdash_\star \phi)$

In Section 4.3 we will use sound context-abstract entailment relations to define the semantics of context-abstract CO_2 systems (see the premises of rules [α-AskCtx] and [α-AskCtx] in Figure 7). In Definition 4.7 below we show a possible instantiation of the relations $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$, which is appropriate for the contract model introduced in Section 2. Lemma 4.8 will then show that these context-abstract relations are sound according to Definition 4.6. Recall from Definition 3.3 that $\mathbf{A}$ is the set of terms $(\mathtt{a}, \mathtt{T})\circ$, where $\mathtt{a}$ is a branch label.

Definition 4.7 (Context abstraction of entailment). *For all participants A, we define the (unlabelled) transition system $\mathrm{TS}_{\mathsf{A}} = (\Sigma, \to, \mathbf{A}, L)$ as follows:*

- $\Sigma = \{(\ell, c) \mid c$ *is a context-abstract contract and* $\ell \in \mathbf{A} \cup \{\varepsilon\}\}$,
- *the transition relation $\to \subseteq \Sigma \times \Sigma$ is defined by the following rule:*

$$(\ell, c) \to (a, c') \qquad\qquad \text{if } c \xrightarrow{a}\!\!\!\!\rightarrow_{\mathsf{A}} c' \vee c \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} c'$$

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

For all context-abstract contracts c, we define the predicate $c \vdash \phi$ as true whenever $\forall \lambda \in \mathrm{Paths}((\epsilon, c)).\ \lambda \models \phi$ holds in LTL. Then, we define the relations:

$$\vdash_{\mathsf{A}} = \{(c, \phi) \mid c \vdash \phi\} \qquad\qquad \vdash_{ctx} = \{(c, \phi) \mid c \nvdash \neg\phi\}$$

Lemma 4.8 below guarantees the soundness of our instantiation of $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ with respect to Definition 4.6. Note that item 1 of Lemma 4.8 ensures a stronger condition than item 1 of Definition 4.6: this improves the accuracy of the analysis.

Lemma 4.8. *For all context-abstract contracts c and for all LTL formulae ϕ:*

1. $c \vdash \phi \quad \Longleftrightarrow \quad (\forall$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \implies \gamma \vdash_{\star} \phi)$
2. $c \nvdash \neg\phi \quad \Longleftarrow \quad (\exists$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \ \wedge \ \gamma \vdash_{\star} \phi)$

4.3. Context abstraction of systems

After having defined a value abstraction for CO_2 systems (Section 4.1), and a context abstraction for contracts (Section 4.2), we now extend the latter to CO_2 systems. For all participants A, we define the *context abstraction* α_{A} of value-abstract systems, which just discards all the components not involving A, and "projects" the contracts involving A. This final abstraction step requires a corresponding notion of context-abstract honesty (Definition 4.10), whose properties (Theorem 4.11) will allow us to verify CO_2 systems via model checking (Section 4.5).

Definition 4.9 (Context abstraction of systems). *For all value-abstract P, we define $\alpha_{\mathsf{A}}(P) = P$. For all value-abstract S, we define the context-abstract system $\alpha_{\mathsf{A}}(S)$ inductively as follows:*

$$\alpha_{\mathsf{A}}(\mathsf{A}[P]) = \mathsf{A}[P] \qquad \alpha_{\mathsf{A}}(s[\gamma]) = s[\alpha_{\mathsf{A}}(\gamma)] \ \textit{if} \ \gamma = \mathsf{A} : c \mid \mathsf{B} : d$$

$$\alpha_{\mathsf{A}}(\{\downarrow_x c\}_{\mathsf{A}}) = \{\downarrow_x c\}_{\mathsf{A}} \qquad \alpha_{\mathsf{A}}(S \mid S') = \alpha_{\mathsf{A}}(S) \mid \alpha_{\mathsf{A}}(S')$$

$$\alpha_{\mathsf{A}}((u)S) = (u)(\alpha_{\mathsf{A}}(S)) \qquad \alpha_{\mathsf{A}}(S) = \mathbf{0}, \ \textit{otherwise}$$

We now introduce the semantics of context-abstract systems. For all participants A, the LTS $\rightarrow_{\mathsf{A}}$ on context-abstract systems is defined by the rules in Figure 7. Labels of $\rightarrow_{\mathsf{A}}$ are either ctx or they have the form $\mathsf{A}\colon \pi$, where A is the participant in $\rightarrow_{\mathsf{A}}$, and π is a CO_2 prefix.

The rules in Figure 7 can be arranged in two groups:

Rules for A. Rule [α-Do] requires a context-abstract transitions of contracts. Rule [α-Ask] allows for a transition of A, whenever c entails ϕ according to a sound context-abstract entailment relation $\vdash_{\mathsf{A}}$. Item 1 of Definition 4.6 guarantees that such an `ask` ϕ will pass in each possible value-abstract context with session $s[\mathsf{A} : c \mid \mathsf{B} : \cdots]$.

Rules for ctx. Rule [α-Fuse] says that a latent contract of A may always be fused (the context may choose whether this is the case or not). Rule [α-AskCtx] allows an `ask` ϕ to fire a ctx transition, whenever c entails ϕ according to a sound context-abstract entailment relation $\vdash_{ctx}$. Item 2 of Definition 4.6 ensures that, if `ask` ϕ would be fired in some concrete system, then it can be fired also in the context-abstract one. The context may also decide whether to perform actions within sessions ([α-DoCtx]). Non-observable context actions are modelled by rules [α-Ctx] and [α-DelCtx].

The remaining context-abstract rules are similar to the value-abstract ones.

The notion of honesty for context-abstract systems (Definition 4.10), named *α-honesty*, follows the lines of that of honesty in Definition 3.6. As expected, we use a notion of readiness for context-abstract systems (named *α-readiness*): this is just a minor revisitation of Definition 4.2, where we use the context-abstract semantics instead of the value-abstract one (see Section B.3 for details).

Definition 4.10 (Context-abstract honesty). *Let P be a context-abstract process. We say that P is* α-honest *iff, for all context-abstract systems S such that $\mathsf{A}[P] \rightarrow_{\mathsf{A}}^{*} S$, A is α-ready in S.*

$$\mathsf{A}[(\texttt{if} \star \texttt{then } P_0 \texttt{ else } P_1) \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{if}}_{\mathsf{A}} \mathsf{A}[P_i \mid Q] \qquad (i \in \{0,1\}) \quad [\alpha\text{-IF}]$$

$$\frac{c \xrightarrow{a}\!\!\!\!\rightarrow_{\mathsf{A}} c'}{\mathsf{A}[\texttt{do}_s\, a.P + P' \mid Q] \;\mid\; s[c] \xrightarrow{\mathsf{A}:\,\texttt{do}_s\, a}_{\mathsf{A}} \mathsf{A}[P \mid Q] \;\mid\; s[c']} \quad [\alpha\text{-DO}]$$

$$\frac{c \vdash_{\mathsf{A}} \phi}{\mathsf{A}[\texttt{ask}_s\, \phi.P + P' \mid Q] \;\mid\; s[c] \xrightarrow{\mathsf{A}:\,\texttt{ask}_s\, \phi}_{\mathsf{A}} \mathsf{A}[P \mid Q] \;\mid\; s[c]} \quad [\alpha\text{-ASK}]$$

$$\frac{s \textit{ fresh}}{(x)(S \;\mid\; \{\downarrow_x c\}_{\mathsf{A}}) \xrightarrow{ctx}_{\mathsf{A}} (s)(s[c] \;\mid\; S\{s/x\})} \quad [\alpha\text{-FUSE}]$$

$$\frac{c \vdash_{ctx} \phi}{\mathsf{A}[\texttt{ask}_s\, \phi.P + P' \mid Q] \;\mid\; s[c] \xrightarrow{ctx}_{\mathsf{A}} \mathsf{A}[P \mid Q] \;\mid\; s[c]} \quad [\alpha\text{-ASKCTX}]$$

$$\frac{c \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} c'}{s[c] \xrightarrow{ctx}_{\mathsf{A}} s[c']} \; [\alpha\text{-DOCTX}] \qquad S \xrightarrow{ctx}_{\mathsf{A}} S \; [\alpha\text{-CTX}] \qquad \frac{S \xrightarrow{ctx}_{\mathsf{A}} S'}{(u)S \xrightarrow{ctx}_{\mathsf{A}} (u)S'} \; [\alpha\text{-DELCTX}]$$

Figure 7: Reduction semantics of context-abstract contracts and systems (full set of rules in Section B.3).

Theorem 4.11 below establishes a link between context-abstract honesty and value-abstract honesty: the former implies the latter, and the *vice versa* holds when the ask primitive is not used. Since we have already established that value-abstract honesty implies concrete honesty (Theorem 4.3), Theorem 4.11 is the final step for guaranteeing the soundness (and completeness) of our verification technique (see Section 4.5).

Theorem 4.11. *Let P be a context-abstract process. If P is α-honest, then P is $\alpha^\star$-honest. Conversely, if P is $\alpha^\star$-honest and* ask*-free, then P is α-honest.*

We now sketch the proof of Theorem 4.11 (full details are available in Section B.3). Correctness of α-honesty (first part of Theorem 4.11) follows because value-abstract transitions of systems can be mimicked by context-abstract ones, and (for the moves of A) also the *vice versa* holds. To prove these properties we exploit the corresponding properties of the context abstraction of contracts (Theorem 4.5). More precisely, to show that P is $\alpha^\star$-honest we have to prove that, for all A-free value-abstract S:

$$\mathsf{A}[P] \mid S \rightarrow_\star^* S' \quad\Longrightarrow\quad \mathsf{A} \text{ is } \alpha^\star\text{-ready in } S'$$

By Theorem B.28 in Section B.3 we have that each value-abstract transition of $\mathsf{A}[P] \mid S$ can be mimicked by a context-abstract transition of $\mathsf{A}[P]$, i.e. $\mathsf{A}[P] \rightarrow_{\mathsf{A}}^* \tilde{S}'$, where $\tilde{S}' = \alpha_{\mathsf{A}}(S')$. By hypothesis we have that P is α-honest, and so A must be α-ready in $\tilde{S}'$. Lemma B.29 in Section B.3 ensures that each context-abstract transition of A in $\tilde{S}'$ can be mimicked by a value-abstract transition of A in S'. Therefore, A must be $\alpha^\star$-ready in S', and so P is $\alpha^\star$-honest.

Completeness of α-honesty (second part of Theorem 4.11) follows by a similar argument. Theorem B.32 in Section B.3 ensures that, in the ask-free fragment, each context-abstract transition is mimicked by a value-abstract one. Lemma B.27 is then used to prove that $\alpha^\star$-readiness implies α-readiness.

4.4. Main result

We now put together the results of Sections 4.1 to 4.3, to devise a model checking technique for honesty. The main result of this paper follows: it states that α-honesty safely approximates honesty, and it is complete for processes without if and ask. This paves us the way for a verification procedure for honesty.

Theorem 4.12. *Let P be a concrete CO_2 process, and let $\tilde{P} = \alpha_{\mathsf{A}}(\alpha^\star(P))$. Then:*

Soundness *If $\tilde{P}$ is α-honest, then P is honest.*

Completeness *If P is honest,* `ask`*-free, and* `if`*-free, then $\tilde{P}$ is α-honest.*

Decidability *α-honesty of $\tilde{P}$ is decidable if P has no delimitation/parallel under process definitions.*

Proof. Soundness and completeness follow directly from Theorems 4.3 and 4.11. Decidability involves two steps: first, we compute the value abstraction of P; second, we model check the state space of P (under the contex-abstract semantics), searching for states where A is *not* α-ready. If the search fails, then A is honest. This is decidable for finite state processes, such as those without delimitation/parallel under process definitions. □

In Example 4.13 below we show two counterexamples to completeness, in case the process under observation is not `ask`-free and `if`-free.

4.5. Maude implementation

In order to implement α-honesty in Maude, we proceed similarly to Section 2.5: we provide abstract systems and contracts with one-step semantics; then, we define the operator `<_>` to specify sequences of transitions. We then specify the relation $\tilde{S} \xrightarrow{\mathsf{A}:\neq\{\mathtt{do}_s,\mathtt{if}\}}_{\mathsf{A}} \tilde{S}'$, which is needed to implement α-readiness (see Definition B.34 in the appendix). This is done as follows:

```
op prefCheck : Prefix SessionName -> Bool .
eq prefCheck (do s a , s) = false .
eq prefCheck (pi , s) = true [owise] .        // 'if' is not a Prefix

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

The predicate "A is α-ready at s in S" is implemented as the operator `ready-at`, which is defined recursively. To guarantee its termination, we collect the visited states through the recursive calls: in this way, no state is visited twice. To allow for a more fine-grained control over `ready-at`, we also add the parameter `b`, of the built-in sort `Bound`. This parameter specifies the maximum depth of the search, when model checking α-readiness. By default, `b` is the constant `unbounded`, but it can also be a number. When analysing processes with no parallel/delimitation under recursion, the `unbounded` default leads to a complete analysis w.r.t. α-readiness. In the general case (arbitrary processes), fixing a value for `b` guarantees termination of `ready-at`: this makes us lose completeness, while soundness is preserved. Increasing the value `b` improves the accuracy of the analysis.

Function `ready-at` is defined by conditional equations, corresponding to the three items of Definition B.34 in the appendix. The first one checks for the base case, where the required `do` action is immediately available.

```
ceq ready-at(s,S,M:Module,b, set) = true if
b =/= 0 /\ not S in set /\ not RDempty(s,S,M:Module) .
```

With `b =/= 0` we check that the bound is not zero, with `not S in set` we check that `S` has not already been visited, while the function `RDempty(s,S,M:Module)` checks that $S \not\xrightarrow{\mathsf{A}:\mathtt{do}_s}_{\mathsf{A}}$.

The second conditional equation instead checks if there is a next state, reached without using $\mathtt{do}_s$ or `if`, from which (recursively) the process is found ready. This exploits the Maude search capabilities; technically, the abstract semantics of CO_2 is reflected at the meta-level, where the search is performed using the `metaSearch` function. Intuitively, `metaSearch` takes a starting state, and searches for all the reachable states matching a given pattern and condition. Below, `metaSearch` is fed with (the meta-representation of): the starting state `< S > s`; the searched pattern `< S':System > s`; the condition, involving the recursive call to `ready-at` (with `S'` as starting state, `b` decremented and `S` added to the set of visited states). Further, the parameter `'+` stands for the transitive (but not reflexive) closure of $\rightarrow_{\mathsf{A}}$; the parameter 1 specifies the depth of the search (we are looking for one-step successors only); the last parameter is 0 asking for the the first element of the solution set. If the `metaSearch` succeeds, then `ready-at` returns `true`.

```
ceq ready-at(s,S,M:Module,b, set) = true if
b =/= 0 /\
not S in set /\
metaSearch(M:Module,
upTerm(< S > s),
'<_>_['S':System , upTerm(s)],
'ready-at[upTerm(s),'S':System,upTerm(M:Module),upTerm(pred(b)),
      '_`,_[upTerm(S),upTerm(set)]] = 'true.Bool,
'+,
1,
0
) =/= failure .
```

The third conditional equation checks that an `if` transition is enabled, and that all such transitions lead to ready states. With **`not succIfempty(S,M:Module)`** we verify that $\tilde{S} \xrightarrow{\mathsf{A}:\,\mathtt{if}}_{\mathsf{A}}$. To check that *all* the `if` transitions lead to a ready state, we search for a counterexample, i.e., a transition leading to a *non-ready* one: when the search *fails*, we return `true`. We call `metaSearch` with the following parameters: the abstract system `S`; the pattern `{l:ASLabel}S':System`; the condition requiring an `if` transition (`l:ASLabel = A :if`) and the non-readiness of the residual (a recursive call to `ready-at` must return `false`).

```
ceq ready-at(s,S,M:Module,b,set) = true if
b =/= 0 /\
not S in set /\
not succIfempty(S,M:Module) /\
metaSearch(M:Module,
upTerm(S),
'`{_`}_['l:ASLabel,'S':System],
'_:if[upTerm(A)] = 'l:ASLabel /\
  'ready-at[upTerm(s),'S':System,upTerm(M:Module),upTerm(pred(b)),
      '_`,_[upTerm(S),upTerm(set)]] = 'false.Bool,
'+,
1,
0
) == failure .
```

Finally, `ready-at` returns `false` in the other cases.

```
eq ready-at(s,S,M:Module,b,set) = false [owise] .
```

The function `search-dishonest` searches for reachable non-ready states. It returns a term of the built-in sort `ResultTriple?`: this is either `failure`, representing an unsuccessful search, or a counterexample whenever the process `P` is not α-honest. To implement `search-dishonest` we exploit the auxiliary function `ready`, which just applies `ready-at` (discussed above) to check that A is ready at all sessions s where A has some obligations.

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

Finally, the function `honest` verifies α-honesty, by exploiting `search-dishonest`. When `honest` does not return `true`, it returns a state where the participant is potentially not α-ready (see e.g. Section 5.2).

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

Example 4.13. *Consider the following processes:*

$$P_{\texttt{if}} = (x)\ \texttt{tell} \downarrow_x \texttt{a!}.\ \texttt{if true then do}_x\ \texttt{a! else}\ \mathbf{0}$$

$$P_{\texttt{ask}} = (x)\ \texttt{tell} \downarrow_x \texttt{a?} + \texttt{b?}.\ \big(\texttt{ask}_x \,\Box\neg\texttt{b?}.\texttt{do}_x\ \texttt{a?}\ +\ \texttt{ask}_x \,\Box\neg\texttt{a?}.\texttt{do}_x\ \texttt{b?}\big)$$

Both $P_{\texttt{if}}$ and $P_{\texttt{ask}}$ are honest, but they are not α-honest. The honesty of $P_{\texttt{if}}$ is straightforward; however, $P_{\texttt{if}}$ is not α-honest, because $\mathsf{A}[P_{\texttt{if}}] \rightarrow_{\mathsf{A}}{}^{*}\ (s)\,(\mathsf{A}[\mathbf{0}] \mid s[\texttt{a!}])$, wherein A is not α-ready.

Checking the honesty of $P_{\texttt{ask}}$ is a bit more complex. The key observation is that, if the session x is fused, then either the participant B at the other endpoint of x stays culpable, or one of the two `ask`*s will eventually be fired. If the leftmost* `ask` *is fired, then either the contract of B was just* `a!`*, or it was* `a!` $\oplus$ `b!`*, but B has already committed to branch* `a!`*. In both cases, A is ready to fire the required input* `a?`*. The other* `ask` *branch is symmetrical. To check that $P_{\texttt{ask}}$ is not α-honest, it is enough to use the Maude tool, since α-honesty is decidable on such process. The output produced by the tool is the following:*

```
result TSystem: < ($ 0)(A[do $ 0 b ? unit . (0).Sum] | $ 0[ready a ? unit . 0]) >
```

This means that $\mathsf{A}[P_{\texttt{ask}}] \rightarrow_{\mathsf{A}}{}^{*}\ (s)\,(\mathsf{A}[\texttt{do}_s\ \texttt{b?}] \mid s[\texttt{rdy a?}])$, wherein A is not α-ready. To statically verify this process as honest we could refine the context-abstract semantics of contracts, by keeping information about which `ask` *prefixes have passed. More precisely, after an* `ask` *passes, we gain information on the contract of the counterparty, and this additional information could be used to prove honesty. Our analysis instead completely abstracts from the context, discarding such information.* □

5. Experiments

In this section we validate our verification technique through some experiments. We consider five case studies: the online store with bank (Section 5.1), a voucher distribution system (Section 5.2), a car purchase financed with a loan (Section 5.3), an online casino featuring blackjack (Section 5.4), and a travel agency (Section 5.5). We specify each of these case studies in CO_2, and we (successfully) verify their honesty using our tool. Finally, in Section 5.6 we evaluate the performance of our model checking tool.

The full Maude implementation of all our experiments is available in [33].

5.1. Online store with bank

Our first experiment concerns the online store with bank introduced in Example 3.9. When using the Maude honesty checker to verify it, we obtain:

```
reduce in ONLINE-STORE : honest(PA, ['ONLINE-STORE], unbounded) .

result TSystem: < ($ 0,$ 1)
(  A[do $ 0 pay ? string . Pbank(($ 0) ; ($ 1) ; expr)
   + do $ 0 cancel ? unit . do $ 1 abort ! unit . (0).Sum]
 | $ 0[pay ? string . (ok ! unit . 0(+)no ! unit .
     (rec Y . pay ? string . (ok ! unit . 0(+)no ! unit . Y(+)abort ! unit . 0)
            + cancel ? unit . 0)(+)abort ! unit . 0)
     + cancel ? unit . (0).Id]
 | $ 1[ccnum ! string .  ··· omitted ···  (+) abort ! unit . 0] >
```

The output produced by Maude is a counterexample to honesty. By analysing it, the source of the dishonesty of the store becomes apparent. The store is enabling `pay?` and `cancel?` on session x (in Maude, denoted `$ 0`). On session y (in Maude, `$ 1`), the store has the obligation to do either action `ccnum!` or `abort!`: however, if the other endpoint at session x does not perform `pay!` or `cancel!`, then the store is not ready to fulfil its obligaton at session y.

We now revise the specification of the store. In order to make it honest, we have to deal with all the cases — as the one shown above — where the other endpoint involved in a session does not fulfil its obligations.

This is done by adding branches (prefixed by τ's, modelling timeouts) whenever the store is waiting some input on a session. In all these branches, the revised store does all the actions needed to exculpate itself in the other sessions.

For instance, process $P_{\texttt{pay}}$ in Example 3.9 is modified as follows:

$$P_{\texttt{pay}}(x,y,t) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{pay?}w.\,P_{\texttt{bank}}(x,y,w,t) \;+\; \texttt{do}_x\,\texttt{cancel?}.\texttt{do}_y\,\texttt{abort!} \;+\; \tau.(P_{\texttt{abortC}}(x) \mid P_{\texttt{abortB}}(y))$$
$$P_{\texttt{abortB}}(y) \stackrel{\text{def}}{=} \texttt{do}_y\,\texttt{abort!} \mid (\texttt{do}_y\,\texttt{accept?} + \texttt{do}_y\,\texttt{deny?})$$
$$P_{\texttt{abortC}}(x) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{abort!} \mid (\texttt{do}_x\,\texttt{pay?}w + \texttt{do}_x\,\texttt{cancel?})$$

Note that in the processes $P_{\texttt{abortB}}$ and $P_{\texttt{abortC}}$, only one output is performed (`abort!`); the other `do` prefixes are only needed to receive residual pending inputs, if any.

The honesty of the revised store is correctly verified by the Maude model checker.

5.2. Voucher distribution system

A store A offers buyers two payment options: `clickPay` or `clickVoucher`. If a buyer B chooses `clickPay`, A requires B to `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 the contracts c_{B} (between A and B) and c_{V} (between A and V) as follows:

$$c_{\mathsf{B}} = \texttt{clickPay?}\,.\,\texttt{pay?string} \;+\; \texttt{clickVoucher?}\,.\,(\texttt{reject!}\,.\,\texttt{pay?string} \oplus \texttt{accept!}\,.\,\texttt{voucher?string})$$
$$c_{\mathsf{V}} = \texttt{ok?} \;+\; \texttt{no?}$$

In [31] a CO_2 process for A was specified as follows:

$$P = (x)\;\big(\texttt{tell}\downarrow_x c_{\mathsf{B}}.(\texttt{do}_x\,\texttt{clickPay?}.\texttt{do}_x\,\texttt{pay?} \;+\; \texttt{do}_x\,\texttt{clickVoucher?}.(y)\;\texttt{tell}\downarrow_y c_{\mathsf{V}}.Q)\big)$$
$$Q = \texttt{do}_y\,\texttt{ok?}.\texttt{do}_x\,\texttt{accept!}.\texttt{do}_x\,\texttt{voucher?} \;+\; \texttt{do}_y\,\texttt{no?}.\texttt{do}_x\,\texttt{reject!}.\texttt{do}_x\,\texttt{pay?} \;+\; \tau.R$$
$$R = \texttt{do}_x\,\texttt{reject!}.\texttt{do}_x\,\texttt{pay?}$$

Variables x and y in P correspond to two separate sessions, where A interacts with B and V, respectively. The advertisement of c_{V} causally depends on the stipulation of the contract c_{B}, because A must fire `clickVoucher` before the rightmost `tell`. In process Q the store waits for an answer from V: if V validates the voucher (first branch), then A accepts it from B; if V rejects the voucher (second branch), then A requires B to pay. The third branch $\tau.R$ allows A to fire a τ action, and then reject the voucher. Here τ models a timeout, to deal with the fact that c_{V} might either not be stipulated, or V might take too long to answer.

The process P above was erroneously classified as honest in [31]. The Maude model checker has determined the dishonesty of that process, and by exploiting the Maude tracing facilities we managed to fix it. Actually, when we check the honesty of P, Maude gives the following output:

```
red honest(P , ['VOUCHER], unbounded) .
rewrites: 36668 in 76ms cpu (76ms real) (482473 rewrites/second)
result TSystem: < ($ 0,$ 1)(A[do $ 0 reject ! unit . do $ 0 pay ? string . (0).Sum] |
$ 0[accept ! unit . voucher ? string . 0(+)reject ! unit . pay ? string . 0] |
$ 1[ready ok ? unit . 0]) >
```

The last three lines of the output above show a state where A is not ready: there, A must do `ok` in session `$1` (which corresponds to variable y in the CO_2 specification), while A is only ready to do a `reject` at session `$0` (which corresponds to x). This problem occurs when branch $\tau.R$ is chosen (actually, the code within `A[...]` is that of R). Since P is `ask`-free and `if`-free, by completeness of abstract honesty (Theorem 4.12) it follows that P is dishonest. To recover honesty, it suffices to replace R with the following process R', where A is ready to handle V's answer when y is istantiated:

$$R' \;=\; (\texttt{do}_x\,\texttt{reject!}.\texttt{do}_x\,\texttt{pay?}) \mid (\texttt{do}_y\,\texttt{no?} + \texttt{do}_y\,\texttt{ok?})$$

Let P' be the store process with the modifications above. Using the Maude model checker, now we obtain:

```
red honest(P' , ['VOUCHER], 3) .
rewrites: 51201 in 44ms cpu (42ms real) (1163659 rewrites/second)
result Bool: true
```

Therefore, by Theorem 4.12 we deduce that the P' is honest.

5.3. *Car loan*

In this example, we show how the $\mathtt{ask}$ prefix allows to query an already established session. Alice wants to buy a car, but she is undecided between an used one, or a new Ferrari: it depends on whether she will manage to obtain a loan, and on its amount. Therefore, she plans to ask for *either* a €10,000 or a €200,000 loan by advertising the following contracts:

$$c_{10K} \ = \ \mathtt{loan10K!} \oplus \mathtt{abort10K!} \qquad\qquad c_{200K} \ = \ \mathtt{loan200K!} \oplus \mathtt{abort200K!}$$

Advertising these contracts, Alice is looking for a bank which is willing to commit to an offer for a loan. Alice instead is not committing herself to the loan, right now: after one of the two contracts is fused, Alice will then choose whether to actually accept the offer (e.g., $\mathtt{loan10k!}$), or not (e.g., $\mathtt{abort10k!}$). Then, depending on which loan can be granted, she plans to advertise *one* of the following contracts for actually buying the car:

$$c_U \ = \ \mathtt{used!}\,.\,(\mathtt{ok?} + \mathtt{no?}) \oplus \mathtt{abortU!} \qquad\qquad c_F \ = \ \mathtt{ferrari!}\,.\,(\mathtt{ok?} + \mathtt{no?}) \oplus \mathtt{abortF!}$$

where she can either select the car (and then accept $\mathtt{ok?}$ or $\mathtt{no?}$ as an answer from the car dealer), or abort the transaction. Her CO_2 process is the following:

$$\begin{aligned} P = (x)\ \mathtt{tell}\downarrow_x c_{10K}\,.\,\mathtt{tell}\downarrow_x c_{200K}\,.\,\big(&\\ &\mathtt{ask}_x\, O(\mathtt{loan10k!} \vee \mathtt{abort10k!})\ \ .\,(y)\ \mathtt{tell}\downarrow_y c_U\,.\,P_{\mathtt{used}}(y)\\ +\ &\mathtt{ask}_x\, O(\mathtt{loan200k!} \vee \mathtt{abort200k!})\,.\,(y)\ \mathtt{tell}\downarrow_y c_F\,.\,P_{\mathtt{ferrari}}(y)\big) \end{aligned}$$

$$\begin{aligned} P_{\mathtt{used}}(y) \stackrel{\text{def}}{=}\ & \mathtt{do}_y\,\mathtt{used!}\,.\,(\mathtt{do}_y\,\mathtt{ok?}\,.\,\mathtt{do}_x\,\mathtt{loan10K!}\ +\ \mathtt{do}_y\,\mathtt{no?}\,.\,\mathtt{do}_x\,\mathtt{abort10K!}\ +\ \tau\,.\,P_{\mathtt{abortU1}})\\ & +\ \tau\,.\,P_{\mathtt{abortU}} \end{aligned}$$

$$\begin{aligned} P_{\mathtt{ferrari}}(y) \stackrel{\text{def}}{=}\ & \mathtt{do}_y\,\mathtt{ferrari!}\,.\,(\mathtt{do}_y\,\mathtt{ok?}\,.\,\mathtt{do}_x\,\mathtt{loan200K!}\ +\ \mathtt{do}_y\,\mathtt{no?}\,.\,\mathtt{do}_x\,\mathtt{abort200K!}\ +\ \tau\,.\,P_{\mathtt{abortF1}})\\ & +\ \tau\,.\,P_{\mathtt{abortF}} \end{aligned}$$

In P, Alice $\mathtt{tell}$s the two loan contracts c_{10K} and c_{200K} on the *same* session variable x: this guarantees that only *one* of them can be fused into a new session (more on this later). Then, P performs a choice whose two branches are guarded by $\mathtt{ask}_x$: by rule [Ask] on Figure 4, they will both be blocked until x is replaced by some session name s (i.e., after rule [Fuse] is applied and a new session is established). Then, we have that:

- the *first* branch will only pass if the *next step* of the contracts fused on x is either $\mathtt{loan10K!}$ or $\mathtt{abort10K!}$ (the symbol O denotes the standard *"next"* operator in LTL);
- similarly, the *second* branch will only pass if the next step of the contracts fused on x is either $\mathtt{loan200K!}$ or $\mathtt{abort200K!}$.

The *first* $\mathtt{ask}_x$ prefix passes only if c_{10K} is fused, and thus Alice can obtain a €10,000 loan: in this case, she advertises the contract c_U above on session y, and then executes $P_{\mathtt{used}}(y)$: there, she tries to buy an used car on y, and if the car dealer's answer is $\mathtt{ok?}$, she selects $\mathtt{loan10K!}$ on x.

The *second* $\mathtt{ask}_x$-guarded branch is similar — except that its $\mathtt{ask}_x$ passes only if c_{200K} is fused, and thus Alice can obtain a €200,000 loan: in this case, she advertises c_F on y, and executes $P_{\mathtt{ferrari}}(y)$; there, she tries to buy a Ferrari on y, and if the car dealer's answer is $\mathtt{ok?}$, she selects $\mathtt{loan200K!}$ on x.

The processes $P_{\texttt{abortU1}}$, $P_{\texttt{abortU}}$, $P_{\texttt{abortF1}}$ and $P_{\texttt{abortF1}}$ are used to ensure that all the sessions are aborted correctly, whenever the participants at the other endpoints are not cooperating.

$$P_{\texttt{abortU1}}(x,y) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{abort10K!} \mid (\texttt{do}_y\,\texttt{ok?} + \texttt{do}_y\,\texttt{no?}) \qquad P_{\texttt{abortU}}(x,y) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{abort10K!} \mid \texttt{do}_y\,\texttt{abortU!}$$
$$P_{\texttt{abortF1}}(x,y) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{abort200K!} \mid (\texttt{do}_y\,\texttt{ok?} + \texttt{do}_y\,\texttt{no?}) \qquad P_{\texttt{abortF}}(x,y) \stackrel{\text{def}}{=} \texttt{do}_x\,\texttt{abort200K!} \mid \texttt{do}_y\,\texttt{abortF!}$$

The Maude model checker correctly verifies that P is honest. In particular, it correctly determines that only *one* contract between c_{10K} and c_{200K} can be fused on x. In fact, after the first two $\texttt{tell}$ prefixes of $\mathsf{A}[P]$ are fired, we have a system of the form:

$$(x,\dots)\big(\,\mathsf{A}[\texttt{ask}_x \cdots + \texttt{ask}_x \cdots] \;\mid\; \{\downarrow_x c_{10K}\}_{\mathsf{A}} \;\mid\; \{\downarrow_x c_{200K}\}_{\mathsf{A}} \;\mid\; S\,\big) \;\mid\; \cdots$$

Depending on the context S, rule [Fuse] might be fired by involving c_{10K} or c_{200K}. In the first case, we obtain:

$$(s,\dots)\big(\,\mathsf{A}[\texttt{ask}_s \cdots + \texttt{ask}_s \cdots] \;\mid\; s[\gamma] \;\mid\; \{\downarrow_s c_{200K}\}_{\mathsf{A}} \;\mid\; \cdots\,\big) \;\mid\; \cdots$$

i.e., c_{10K} becomes part of γ, while c_{200K} remains latent. However, x has now been replaced by s: this prevents rule [Fuse] to be fired on $\{\downarrow_s c_{200K}\}_{\mathsf{A}}$, and allows such a term to be garbage-collected by the last rule in Figure 3. Instead, if c_{200K} is fused in a session, then c_{10K} remains latent, and $\{\downarrow_s c_{10K}\}_{\mathsf{A}}$ can be garbage-collected. The outcome of this session establishment will later influence the behaviour of the $\texttt{ask}_x$-guarded choices in P. This chain of events is precisely reflected by the abstract semantics of CO_2 — in particular, by rules [α-Fuse] and [α-Ask] in Figure 7: this allows the model checker to establish that P is α-honest, and hence honest.

5.4. Blackjack

We model an online blackjack server, using simplified casino rules. The game involves two players: P (for player) and A (for *dealer*). The goal of P is to beat the dealer, by accumulating a hand of cards whose value is greater than that of the dealer; furthermore, the value of the hand must not exceed 21. The game has two turns: first the player turn, and then the dealer turn. In the player turn, A deals cards to the player; after a card is received, the player can decide whether to get another one ($\texttt{hit}$) or to terminate his turn ($\texttt{stand}$). In the dealer turn, A deals cards for himself, with the goal of obtaining a hand with value greater than the player's hand. The player (possibly, A) which exceeds 21 loses the game.

The contract advertised by the dealer to players is the following:

$$c_{\mathsf{P}} = \texttt{rec}\,Z\,.\;\texttt{hit?}\,.\,c_{\texttt{hit}} + \texttt{stand?}\,.\,c_{\texttt{end}}$$
$$c_{\texttt{hit}} = \texttt{card!int}\,.\,Z \oplus \texttt{lose!} \oplus \texttt{abort!}$$
$$c_{\texttt{end}} = \texttt{win!} \oplus \texttt{lose!} \oplus \texttt{abort!}$$

Players can choose between taking a card ($\texttt{hit}$) or passing the turn ($\texttt{stand}$). In the first case, the dealer either gives a $\texttt{card}$ to the player (and returns to the beginning of the contract), or it notifies that the player $\texttt{lose}$s (or it may $\texttt{abort}$ the game). In the second case ($\texttt{stand}$), the dealer notifies to the player if he has won or lost (or if the game has been aborted).

To implement the game, the dealer resorts to an external service which provides the features of a deck of cards. The contract between the dealer and the deck of cards is formalised by c_{D} as follows:

$$c_{\mathsf{D}} \;=\; \texttt{rec}\,Z\,.\;\texttt{next!}\,.\,\texttt{card?int}\,.\,Z \oplus \texttt{abort!}$$

The dealer can recursively ask for a new card ($\texttt{next}$) and receive it ($\texttt{card}$) as an integer value, or it may $\texttt{abort}$ the interaction with the deck of cards service.

We specify the dealer as the following process P:

$$P \;=\; (x_d)\,(x_p)\,\texttt{tell}\downarrow_{x_d} c_{\mathsf{D}}\,.\,\texttt{ask}_{x_d}\,\texttt{true}\,.\,\texttt{tell}\downarrow_{x_p} c_{\mathsf{P}}\,.\,P_{\texttt{play}}(x_p, x_d, 0)$$

The first $\texttt{tell}$ in P advertises the contract for the deck of cards. The dealer waits (via the $\texttt{ask}$ prefix) that such contract is fused, and then it advertises the contract for the player (with the second $\texttt{tell}$). At this point the control is passed to the process $P_{\texttt{play}}$, which is specified as follows:

$$\begin{aligned} P_{\texttt{play}}(x_p, x_d, n_p) \stackrel{\text{def}}{=}\ & \texttt{do}_{x_p}\ \texttt{hit?}\,.\,\texttt{do}_{x_d}\ \texttt{next!}\,.\,P_{\texttt{deck}}(x_p, x_d, n_p) \\ & + \texttt{do}_{x_p}\ \texttt{stand?}\,.\,Q_{\texttt{stand}}(x_p, xd, n_p, 0) \\ & + \tau\,.\,\texttt{do}_{x_d}\ \texttt{abort!}\,.\,P_{\texttt{abortP}}(x_p) \end{aligned}$$

Process $P_{\texttt{play}}$ waits for a player decision. If the player chooses $\texttt{hit}$ then the dealer asks the deck for the next card, and the control passes to $P_{\texttt{deck}}$. Instead, if the player chooses $\texttt{stand}$, the control passes to $Q_{\texttt{stand}}$. The third branch models a timeout, where the dealer stops waiting for the player decision, and it just aborts all the sessions. The parameter n_p is used to accumulate the value of the player hand (i.e., the summation of the value of the cards he has received).

$$P_{\texttt{deck}}(x_p, x_d, n_p) \stackrel{\text{def}}{=} \texttt{do}_{x_d}\ \texttt{card?}n\,.\,P_{\texttt{card}}(x_p, x_d, n_p + n, n) + \tau\,.\,\texttt{do}_{x_p}\ \texttt{abort!}\,.\,P_{\texttt{abortD}}(x_d)$$

Process $P_{\texttt{deck}}$ waits for the value n of the card provided by the deck, and then passes the control to $P_{\texttt{card}}$. Also in this case, a timeout branch ensures that sessions are aborted in case the deck does not reply timely.

$$P_{\texttt{card}}(x_p, x_d, n_p, n) \stackrel{\text{def}}{=} \texttt{if}\ n_p \leq 21\ \texttt{then}\ \texttt{do}_{x_p}\ \texttt{card!}n\,.\,P_{\texttt{play}}(x_p, x_d, n_p)\ \texttt{else}\ \texttt{do}_{x_p}\ \texttt{lose!}\,.\,P_{\texttt{abortD}}(x_d)$$

Process $P_{\texttt{card}}$ checks whether the player hand exceeds 21: if so, it tells the player that he has lost; otherwise, the player is allowed to take another turn.

$$Q_{\texttt{stand}}(x_p, x_d, n_p, n_d) \stackrel{\text{def}}{=} \texttt{if}\ n_d \leq 21\ \texttt{then}\ \texttt{do}_{x_d}\ \texttt{next!}\,.\,Q_{\texttt{deck}}(x_p, x_d, n_p, n_d)\ \texttt{else}\ \texttt{do}_{x_p}\ \texttt{win!}\,.\,\texttt{do}_{x_d}\ \texttt{abort!}$$

Process $Q_{\texttt{stand}}$ is invoked upon the player has decided to stand. The dealer checks that the value n_d of its hand (initially set to 0) is less then 21. If so, the dealer asks the deck for the next card, and the control passes to $Q_{\texttt{deck}}$; otherwise, it tells the player that he has won.

$$Q_{\texttt{deck}}(x_p, x_d, n_p, n_d) \stackrel{\text{def}}{=} \texttt{do}_{x_d}\ \texttt{card?}n\,.\,Q_{\texttt{card}}(x_p, x_d, n_p, n_d + n)\ +\ \tau\,.\,\texttt{do}_{x_p}\ \texttt{abort!}\,.\,P_{\texttt{abortD}}(x_d)$$

Process $Q_{\texttt{deck}}$ waits for the card, and then proceeds to $Q_{\texttt{card}}$ (as above, also in this case we use a timeout branch to avoid deadlock).

$$Q_{\texttt{card}}(x_p, x_d, n_p, n_d) \stackrel{\text{def}}{=} \texttt{if}\ n_d < n_p\ \texttt{then}\ Q_{\texttt{stand}}(x_p, x_d, n_p, n_d)\ \texttt{else}\ \texttt{do}_{x_p}\ \texttt{lose!}\,.\,P_{\texttt{abortD}}(x_d)$$

Process $Q_{\texttt{card}}$ compares the hand n_p of the player with that n_d of the dealer. If the dealer hand has not reached n_p, the dealer takes another card; otherwise, the player has lost.

$$\begin{aligned} P_{\texttt{abortP}}(x_p) &\stackrel{\text{def}}{=} \texttt{do}_{x_p}\ \texttt{hit?}\,.\,\texttt{do}_{x_p}\ \texttt{abort!}\ +\ \texttt{do}_{x_p}\ \texttt{stand?}\,.\,\texttt{do}_{x_p}\ \texttt{abort!} \\ P_{\texttt{abortD}}(x_d) &\stackrel{\text{def}}{=} \texttt{do}_{x_d}\ \texttt{abort!}\ \mid\ \texttt{do}_{x_d}\ \texttt{card?}\,.\,\texttt{do}_{x_d}\ \texttt{abort!} \end{aligned}$$

Finally, processes $P_{\texttt{abortP}}$ and $P_{\texttt{abortD}}$ ensure that the sessions with the player and with the deck of cards, respectively, are aborted correctly.

The Maude honesty checker correctly determines that P is honest.

5.5. Travel agency

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 contract c_{U} between the travel agency and the user first requires U to provide the trip details and the available budget; then, it chooses either to send a quote to U, or to abort the transaction. In the first case, the continuation c' requires first U to pay (the details of the payment are abstracted away; see Example 3.9 for

a more concrete treatment of payments). Then, the agency may decide whether to commit the transaction or to abort it:

$$c_{\mathsf{U}} = \mathtt{tripDets?string} \,.\, \mathtt{budget?int} \,.\, (\mathtt{quote!int} \,.\, c' \oplus \mathtt{abort!})$$
$$c' = \mathtt{pay?} \,.\, (\mathtt{commit!} \oplus \mathtt{abort!})$$

The contract c_{F} between the travel agency and the ticket broker first requires A to send the flight details to F. Then, F replies with a quote for the ticket, after which A can choose whether to pay or abort the transaction. If A opts to pay, then it will receive a confirmation, after which it may eventually choose to commit or to abort the transaction:

$$c_{\mathsf{F}} = \mathtt{flightDets!string} \,.\, d$$
$$d = \mathtt{quote?int} \,.\, \big(\mathtt{pay!} \,.\, d' \oplus \mathtt{abort!}\big)$$
$$d' = \mathtt{confirm?} \,.\, (\mathtt{commit!} \oplus \mathtt{abort!})$$

The contract c_{H} between the agency and the hotel reservation service is similar (except for the first action):

$$c_{\mathsf{H}} = \mathtt{hotelDets!string} \,.\, d$$

In addition to the contracts above, the agency should respect the following constraints: (i) the agency commits the transaction with U iff both the transactions with F and H are committed; (ii) A pays the ticket and the hotel reservation only after it has received the payment from U; (iii) either both the transactions with F or H are committed, or they are both aborted.

A specification of the travel agency respecting the above constraints is given by the following process P:

$$P = (x_u)\,\mathtt{tell}\downarrow_{x_u} c_{\mathsf{U}} \,.\, \mathtt{do}_{x_u}\,\mathtt{tripDets?}y_t \,.\, \mathtt{do}_{x_u}\,\mathtt{budget?}y_b \,.\, P'$$
$$P' = (x_f\ x_h)\,\big((\mathtt{tell}\downarrow_{x_f} c_{\mathsf{F}} \,.\, \mathtt{do}_{x_f}\,\mathtt{flightDets!}y_t) \mid (\mathtt{tell}\downarrow_{x_h} c_{\mathsf{H}} \,.\, \mathtt{do}_{x_h}\,\mathtt{hotelDets!}y_t) \mid P_{\mathtt{quote}}(x_u, x_f, x_h, y_b)\big)$$

Process P first advertises the contract c_{U}, then receives from U the trip details and the budget. Then, process P' advertises the contracts c_{F} and c_{H}, and requests in parallel the quotes to F and H.

$$P_{\mathtt{quote}}(x, x_1, x_2, y) \stackrel{\text{def}}{=} P_{\mathtt{quote1}}(x, x_1, x_2, y) + P_{\mathtt{quote1}}(x, x_2, x_1, y) + \tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2)$$
$$P_{\mathtt{quote1}}(x, x_1, x_2, y) \stackrel{\text{def}}{=} \mathtt{do}_{x_1}\,\mathtt{quote?}y_1 \,.\, \mathtt{if}\ y_1 < y\ \mathtt{then}\ P_{\mathtt{quote2}}(x, x_1, x_2, y_1, y)\ \mathtt{else}\ P_{\mathtt{abort}}(x, x_1, x_2)$$
$$P_{\mathtt{quote2}}(x, x_1, x_2, y_1, y) \stackrel{\text{def}}{=} \big(\mathtt{do}_{x_2}\,\mathtt{quote?}y_2 \,.\, \mathtt{if}\ y_1 + y_2 < y\ \mathtt{then}\ P_{\mathtt{pay}}(x, x_1, x_2, y_1 + y_2)\ \mathtt{else}\ P_{\mathtt{abort}}(x, x_1, x_2)\big) + \tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2)$$

In the continuation $P_{\mathtt{quote}}$, the agency receives the quotes from F and H. In the leftmost invocation $P_{\mathtt{quote1}}(x_u, x_1, x_2, y)$ the quote from F is received first, while in the rightmost invocation $P_{\mathtt{quote1}}(x_u, x_2, x_1, y)$, the priority is given to the quote from H. The branch $\tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2)$ models a timeout, where the agency stops waiting for the quotes, and it just aborts all the sessions. These are aborted also in case one of the quotes (or their sum) is greater than the user budget.

$$P_{\mathtt{abort}}(x, x_1, x_2) \stackrel{\text{def}}{=} \mathtt{do}_x\,\mathtt{abort!} \mid \mathtt{do}_{x_1}\,\mathtt{abort!} \mid \mathtt{do}_{x_2}\,\mathtt{abort!}$$
$$\mid \mathtt{do}_x\,\mathtt{pay?} \mid \mathtt{do}_{x_1}\,\mathtt{quote?} \mid \mathtt{do}_{x_2}\,\mathtt{quote?} \mid \mathtt{do}_{x_1}\,\mathtt{confirm?} \mid \mathtt{do}_{x_2}\,\mathtt{confirm?}$$

The second line of process $P_{\mathtt{abort}}$ ensures that pending input messages that might remain are eventually consumed. After both quotes are received, the control passes to $P_{\mathtt{pay}}$.

$$P_{\mathtt{pay}}(x, x_1, x_2, y) \stackrel{\text{def}}{=} \mathtt{do}_x\,\mathtt{quote!}y \,.\, (\mathtt{do}_x\,\mathtt{pay?} \,.\, P_{\mathtt{confirm1}}(x, x_1, x_2) + \tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2))$$
$$P_{\mathtt{confirm1}}(x, x_1, x_2) \stackrel{\text{def}}{=} \mathtt{do}_{x_1}\,\mathtt{pay!} \,.\, \mathtt{do}_{x_2}\,\mathtt{pay!} \,.\, \big((\mathtt{do}_{x_1}\,\mathtt{confirm?} \,.\, P_{\mathtt{confirm2}}(x, x_1, x_2)) + \tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2)\big)$$
$$P_{\mathtt{confirm2}}(x, x_1, x_2) \stackrel{\text{def}}{=} (\mathtt{do}_{x_2}\,\mathtt{confirm?} \,.\, P_{\mathtt{commit}}) + \tau \,.\, P_{\mathtt{abort}}(x, x_1, x_2)$$

Example	Ref.	Rewritings	Avg. time (ms)	Std. dev.
Online store with bank (dishonest)	Example 3.9	18478	36.8	1.03
Online store with bank (honest)	Section 5.1	223379	147.6	2.27
Voucher distribution system (dishonest)	Section 5.2	36668	49.8	1.03
Voucher distribution system (honest)	Section 5.2	51201	54.8	2.69
Car loan	Section 5.3	110804	114.8	5.00
Blackjack	Section 5.4	125720	140.9	3.14
Travel Agency	Section 5.5	15143028	6118	80.95

Table 2: Benchmarks for the honesty checker.

In process $P_{\texttt{pay}}$, the agency sends the overall quote to U. Then, it waits for the user payment, or it aborts all the sessions if a timeout has occurred. If the payment from U is received, the agency proceeds to pay the ticket and the hotel reservation. Then, in $P_{\texttt{confirm1}}$ it waits the confirmation from F, and after that, in $P_{\texttt{confirm2}}$ waits the confirmation from H. As above, waiting can always be terminated by a timeout, which is followed by $P_{\texttt{abort}}$. Finally, in process $P_{\texttt{commit}}$ the agency commits all the transactions.

The Maude honesty checker correctly determines that P is honest.

5.6. Benchmarks

To empirically valuate the effectiness of our verification technique, we have applied the Maude honesty checker on all the case studies in Section 5. The experiments have measured, for each case study, the total number of rewritings, and the average completion time. The testing environment is a PC with an Intel Core i7-4790K CPU @ 4.00GHz and 32G of RAM, running Ubuntu 14.04. The results are reported in Table 2.

6. Related work

The development of the tool presented in this paper witnesses the usefulness and flexibility of rewriting logic (in general) and of Maude (in particular) as a support for the specification and verification of concurrent programming languages. Indeed, rewriting logic [37] 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 [36], the π-calculus [38], Petri nets [39], Erlang [40], Klaim [41], adaptive systems [42], *etc.* A more comprehensive list of calculi, programming languages, tools and applications implemented in Maude is collected in [43].

6.1. CO_2 and contract-oriented computing

To the best of our knowledge, the concept of *contract-oriented computing* (in the meaning used in this paper) has been introduced in [1]. CO_2, a contract-agnostic calculus for contract-oriented computing has been instantiated with several contract models — both bilateral [31, 32] and multiparty [3, 44, 45]. Here we have instantiated it with binary session types (Section 2). A minor difference w.r.t. [31, 32, 44] is that here we no longer have `fuse` as a language primitive, but rather the creation of fresh sessions is performed non-deterministically by the context (rule [Fuse]). This is equivalent to assume a contract broker which collects all contracts, and may establish sessions when compliant ones are found. Another difference w.r.t. [31] is that there a participant A is considered honest when, in each possible context, she can always exculpate herself by a sequence of A-solo moves. Here, instead, we require that A is *ready* (Definition 3.5) in all possible contexts, similarly to [32, 44]. We conjecture that these two notions are equivalent in the contract model based on binary session types considered in this paper.

In [32] a type system has been proposed to safely over-approximate honesty. The type of a process P is a function which maps each session 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". While the type system of [32] allows to type check finite-state processes (as in the current paper) and also some kinds of infinite-state processes (which are not dealt with in the current paper) no actual algorithm is given for such verification, hence type inference remains an open issue. In contrast, here we have implemented a verification algorithm for honesty, by model checking the abstract semantics in Section 4. Furthermore, here we can verify honesty of value-passing processes, which are not dealt with in [32].

The programming model envisioned by CO_2 has been implementated as a *contract-oriented middleware* [46], which uses *timed* session types [47] as contracts to regulate the interaction of distributed services. Such a middleware collects the contracts advertised by services, and upon finding a pair of compliant contracts, it creates a session between the respective services. The infrastructure then behaves as a *message-oriented middleware (MOM)*, which additionally monitors all the messages exchanged in sessions (also checking that the time constraints are respected). When a participant is culpable of a contract violation, its reputation is decreased (similarly to [2]): as a consequence, its chances of being involved in further sessions are reduced.

6.2. Comparison with other calculi

CO_2 takes inspiration from Concurrent Constraint Programming (CCP [48]): processes advertise (with the `tell` prefix) contracts representing promises on the future interactions, and once a session is established, the involved contracts can be queried (with the `ask` prefix). In CCP, the notion of consistency of the constraint store plays a central role, e.g. the primitive `check` φ allows to proceed only if the constraint store is consistent with φ. In session types, consistency is immaterial, thus `check` is not present in the process calculus. For other analogies and differences between CO_2 and CCP, and a discussion of alternative primitive sets for CO_2, see [3].

In cc-pi [49], CCP is mixed with communication via name fusion so that parties establish Service Level Agreements by merging the constraints (values in a nominal c-semiring) representing their requirements. cc-pi borrows the standard `tell` primitive from CCP, which is used to put constraints on names. Additionally, it features a communication primitive à la π-calculus: two processes can interact via prefixes $\bar{x}\langle z\rangle$ and $y\langle w\rangle$, *provided that* the constraint store entails the equality $x = y$; when this happens, the equality $z = w$ is added to the store, unless doing so leads to an inconsistency. A main consequence of this mechanism is that performing a `tell` *restricts* the future possible interactions with the other processes, since adding $z = w$ will lead to more inconsistencies. In a sense, initially a name can interact with every other name, and then its interaction capabilities can be constrained through `tell`. By contrast, in CO_2 performing a `tell` *enables* interaction with other participants. A session variable can *not* interact unless one or more `tell`s are performed to advertise contracts. The effect of a `tell` is therefore dual to that of cc-pi: in a sense, CO_2 advertises promises, while cc-pi advertises requirements.

One peculiar feature of CO_2 is that sessions are monitored with the contracts used to establish them, thus ensuring that "wrong" messages cannot be sent nor received. A similar notion can be found in [50, 51]; the main difference w.r.t. CO_2 is that, in those works, monitors discard unexpected messages without blocking the sender, whereas in CO_2 an agent process committing to the "wrong" output will get stuck; in other words, in the terminology of [52], the monitors of [50, 51] implement *suppression*, whereas CO_2's monitoring is closer to *truncation*.

6.3. Honesty vs. safety and progress

Honesty ensures that a CO_2 process will interact in a session according to some specific contract. This implies that P will communicate *safely* (i.e., will not diverge from its contracts), but, in general, does *not* imply that an honest process P also enjoys progress: this depends on its execution context. In fact, if $\mathsf{A}[P]$ is honest, but establishes a session with a *dishonest* agent $\mathsf{B}[Q]$, then $\mathsf{A}[P]$ may get stuck — albeit A will not be culpable (in *any* session).

The problem of ensuring safe communication is tackled in the session types literature, whose origins date back to [13, 53, 5] (and from which the contract model adopted in this paper is borrowed). Session typing systems do not include a notion of "culpability", and (besides some exceptions, discussed below) are

generally unable to guarantee progress for processes interacting on multiple interleaved sessions — unless some strong assumptions are made about their execution context. Intuitively, progress holds when assuming that a participant is always available to join a session, and all participants will respect their types, without deadlocking or performing unexpected communications. As a consequence, a well-session-typed process that interacts with other well-session-typed processes through multiple interleaved sessions may still get stuck — possibly in a state that, in our setting, would be deemed culpable. Consider, for instance, the following processes:

$$P \;=\; (x,y)\,\texttt{tell}\downarrow_x \texttt{a?.tell}\downarrow_y \texttt{b!.do}_x\,\texttt{a?.do}_y\,\texttt{b!} \qquad Q \;=\; (y,x)\,\texttt{tell}\downarrow_y \texttt{b?.tell}\downarrow_x \texttt{a!.do}_x\,\texttt{b?.do}_y\,\texttt{a!} \qquad (1)$$

Under the "classic" session typing approach, , the two (dishonest) processes P and Q above would be well-typed, because their do prefixes match the contracts at x and y. However, the composition $\mathsf{A}[P] \mid \mathsf{B}[Q]$ would deadlock after fusing the two sessions: in fact, A would remain waiting on x (while being persistently culpable at y), and B would remain waiting on y (while being persistently culpable at x).

The problem of guaranteeing *both* safety *and* progress of a given composition of session-typed processes is tackled in [54, 55, 56, 57], using type systems which track the dependencies among the active sessions: the result is that well-typed compositions of processes can interact on multiple interleaved sessions without incurring in deadlocks — and thus, in our setting, without remaining persistently culpable. For instance, the parallel composition of processes P and Q in (1) would not be typeable. Indeed, to guarantee deadlock freedom in case of interleaved sessions, a strong typing discipline is imposed on communications: for instance, in [56] all the interactions on a session must end before another session can be used. This restriction prevents the typing of e.g. P, which interleaves sessions x and y. In our approach, we do not necessarily classify as dishonest processes containing interleaved uses of sessions. For instance,

$$(x,y)\,\texttt{tell}\downarrow_x \texttt{a?.tell}\downarrow_y \texttt{b!.}\big(\texttt{do}_x\,\texttt{a?.do}_y\,\texttt{b!} + \texttt{do}_y\,\texttt{b!.do}_x\,\texttt{a?}\big)$$

is honest, despite it would not be typeable according to [56].

In [58], deadlock-freedom is obtained by using global types [6] (i.e., multiparty protocol specifications) to typecheck a *choreography.* A choreography is a global program that describes the active threads of each participant, when they are spawned, the sessions they are involved in, their communications, who performs conditional choices, and where. A choreography C is used to synthesise a parallel composition of *endpoint processes*, reflecting the structure of the threads of C. The result is that if C is well-typed and *linear* (i.e., there are no race conditions among its threads), then the synthesised composition will be race-free and error-free, and its behaviour will match that of C. Thus, the communication will be "safe" and deadlock-free, conforming to the global type.

Under this kind of approach, the main results depend on the *whole* execution context being known upfront. In our technique, instead, the process of a given participant is model-checked "in isolation" and without any assumptions about the context where it will run. In order to guarantee that A will never remain persistently culpable, the only requirement on the context is that it must be initially A-free. Despite progress is not implied by honesty in our framework, if *all* participants in a system are assumed honest, then no session will remain stuck before its conclusion, and so we obtain global progress. In a sense, assuming the honesty of the context is similar to assume its typeability.

6.4. Improvements w.r.t. [59]

This article is an extended and improved version of [59]. With respect to [59], here we have considered a more expressive (value-passing) calculus, while managing to preserve the main result (soundness, completeness and decidability of honesty verification, Theorem 4.12) in the new setting. A crucial issue we had to deal with is the verification of honesty in the presence of conditional expressions; this has required to refine the notion of abstract honesty, and to revise the Maude tool accordingly. Furthermore, we have streamlined some aspects of the theory (e.g. the definition of honesty), we have introduced additional results (e.g. Theorem 4.3), we have extended explanations, and provided full proofs of all our statements (Sections A and B). In addition, we have enriched the set of experiments used to validate our techniques (Section 5), and we have benchmarked our tool (Section 5.6).

7. Conclusions

We have devised a verification technique for honesty in contract-oriented systems. This is the problem of deciding whether a participant always respects the contracts she advertises, in all possible contexts [31]. Several case studies (e.g. Section 5.5 and example 3.9) show that writing honest processes is not a trivial task, especially when multiple sessions are needed for realising a contract.

We have dealt with the problem of verifying honesty in three steps. First, we have specified a model of contracts and of contract-oriented systems: we have formalised their syntax and semantics, and the crucial notions of compliance and honesty (Sections 2 and 3). We have then devised a verification technique for deciding when a participant is honest, and we have proved its soundness and (under some conditions) also its completeness (Theorem 4.12). Finally, we have provided an implementation of this technique in Maude, and we have validated our technique against a set of case studies (Sections 4.5 and 5).

The feedback obtained from the validation phase allows us to draw some preliminary conclusions about the proposed technique, and in general about the contract-oriented approach. A first observation is that the current version of the CO_2 calculus seems expressive enough to model sophisticated applications, like e.g. the online store with bank, the blackjack, and the travel agency case studies (Example 3.9 and sections 5.4 and 5.5). In all these case studies, we managed to write a specification whose honesty is automatically verified by our Maude tool. When more than two participants are involved in an application, writing honest processes has required us to deal with all those cases where an expected input is not received. This has been done by extending the code which performs the input with a timeout branch, which (basically) aborts all the sessions. In some cases, the assumption that *all* the participants may act dishonestly may be seen too strong, especially when some of them belong to the same organization (as it could be the case, e.g., for the dealer A and the deck of cards D in the blackjack case study). A possible improvement would be to verify the honesty of a participant (e.g., the dealer) under the assumption that some other participant (e.g., the deck of card) is honest. This would allow us to simplify the code of the participant under observation, and, possibly, also to increase the precision of the analysis.

A remarkable fact about our verification technique is that its correctness only depends on two properties of contracts, namely those stated in Theorem 4.5 (for the `ask`-free fragment), and in Definition 4.6 (for the `ask` statement). While in this paper we have shown that these properties hold for binary session types, we expect that similar results can be found for other contract models, e.g. the multiparty session types of [44]. For instance, the conditions in Definition 4.6 can be achieved trivially, for any contract model and logic, when $\vdash_{\mathsf{A}}$ is the empty relation and $\vdash_{ctx}$ is the cartesian product between the set of abstract contracts and the set of formulae (actually, in this paper we have used a more precise abstraction; see Definition 4.7 and Lemma 4.8). Therefore, our verification technique for honesty can be reused in any instantiation of CO_2 where binary session types are replaced by a contract model which admits a context-abstraction function α_{A} of contracts and a transition relation $\twoheadrightarrow_{\mathsf{A}}$ satisfying Theorem 4.5 and the conditions in Definition 4.6.

References

[1] M. Bartoletti, R. Zunino, A calculus of contracting processes, in: LICS, 2010, pp. 332–341. doi:10.1109/LICS.2010.25.

[2] A. Mukhija, A. Dingwall-Smith, D. Rosenblum, Qos-aware service composition in Dino, in: ECOWS, 2007, pp. 3–12. doi:10.1109/ECOWS.2007.24.

[3] M. Bartoletti, E. Tuosto, R. Zunino, Contract-oriented computing in CO_2, Sci. Ann. Comp. Sci. 22 (1) (2012) 5–60. doi:10.7561/SACS.2012.1.5.

[4] M. Bravetti, G. Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Software Composition, 2007, pp. 34–50. doi:10.1007/978-3-540-77351-1_4.

[5] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP, Vol. 1381 of LNCS, 1998, pp. 22–138. doi:10.1007/BFb0053567.

[6] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM, 2008, pp. 273–284. doi:10.1145/1328438.1328472.

[7] G. Castagna, N. Gesbert, L. Padovani, A theory of contracts for web services, ACM Transactions on Programming Languages and Systems 31 (5) (2009) 19:1–19:61. doi:10.1145/1538917.1538920.

[8] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, K. Wolf, Multiparty contracts: Agreeing and implementing interorganizational processes, Comput. J. 53 (1) (2010) 90–106. doi:10.1093/comjnl/bxn064.

[9] M. Bartoletti, T. Cimoli, G. M. Pinna, Lending Petri nets and contracts, in: Proc. FSEN, Vol. 8161 of LNCS, Springer, 2013, pp. 66–82. doi:10.1007/978-3-642-40213-5_5.
[10] M. Bartoletti, T. Cimoli, G. M. Pinna, Lending Petri nets, Science of Computer Programming (to appear).
[11] M. Bartoletti, T. Cimoli, R. Zunino, A theory of agreements and protection, in: Proc. POST, Vol. 7796 of LNCS, Springer, 2013, pp. 186–205. doi:10.1007/978-3-642-36830-1_10.
[12] M. Bartoletti, T. Cimoli, G. M. Pinna, R. Zunino, Contracts as games on event structures, JLAMP (to appear). doi:10.1016/j.jlamp.2015.05.001.
[13] K. Honda, Types for dyadic interaction, in: CONCUR, 1993, pp. 509–523. doi:10.1007/3-540-57208-2_35.
[14] L. Bocchi, K. Honda, E. Tuosto, N. Yoshida, A theory of design-by-contract for distributed multiparty interactions, in: CONCUR, 2010, pp. 162–176. doi:10.1007/978-3-642-15375-4_12.
[15] G. Castagna, M. Dezani-Ciancaglini, L. Padovani, On global types and multi-party session, Logical Methods in Comp. Sci. 8 (1). doi:10.2168/LMCS-8(1:24)2012.
[16] J. Lange, E. Tuosto, Synthesising choreographies from local session types, in: Proc. CONCUR, 2012, pp. 225–239. doi:10.1007/978-3-642-32940-1_17.
[17] P.-M. Deniélou, N. Yoshida, Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types, in: Proc. ICALP, 2013, pp. 174–186. doi:10.1007/978-3-642-39212-2_18.
[18] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: CONCUR, 2010, pp. 222–236. doi:10.1007/978-3-642-15375-4_16.
[19] P. Wadler, Propositions as sessions, J. Funct. Program. 24 (2-3) (2014) 384–418. doi:10.1017/S095679681400001X.
[20] S. J. Gay, V. T. Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (1) (2010) 19–50. doi:10.1017/S0956796809990268.
[21] M. Carbone, F. Montesi, C. Schürmann, Choreographies, logically, in: CONCUR, 2014, pp. 47–62. doi:10.1007/978-3-662-44584-6_5.
[22] G. Bernardi, M. Hennessy, Using higher-order contracts to model session types, in: CONCUR, 2014, pp. 387–401. doi:10.1007/978-3-662-44584-6_27.
[23] M. Bartoletti, A. Scalas, R. Zunino, A semantic deconstruction of session types, in: CONCUR, 2014, pp. 402–418. doi:10.1007/978-3-662-44584-6_28.
[24] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, J. J. Leifer, A secure compiler for session abstractions, Journal of Computer Security 16 (5).
[25] N. Yoshida, R. Hu, R. Neykova, N. Ng, The scribble protocol language, in: TGC, 2013, pp. 22–41.
[26] R. Neykova, N. Yoshida, Multiparty session actors, in: COORDINATION, 2014, pp. 131–146.
[27] J. Franco, V. T. Vasconcelos, A concurrent programming language with refined session types, in: SEFM Workshops: BEAT2, 2013, pp. 15–28. doi:10.1007/978-3-319-05032-4_2.
[28] S. J. Gay, M. Hole, Types and subtypes for client-server interactions, in: Proc. ESOP, 1999, pp. 74–90. doi:10.1007/3-540-49099-X_6.
[29] S. Gay, M. Hole, Subtyping for session types in the Pi calculus, Acta Inf. 42 (2). doi:10.1007/s00236-005-0177-z.
[30] F. Barbanera, U. de'Liguoro, Two notions of sub-behaviour for session-based client/server systems, in: PPDP, 2010, pp. 155–164. doi:10.1145/1836089.1836109.
[31] M. Bartoletti, E. Tuosto, R. Zunino, On the realizability of contracts in dishonest systems, in: COORDINATION, Vol. 7274 of LNCS, 2012, pp. 245–260. doi:10.1007/978-3-642-30829-1_17.
[32] M. Bartoletti, A. Scalas, E. Tuosto, R. Zunino, Honesty by typing, in: FMOODS/FORTE, Vol. 7892 of LNCS, 2013, pp. 305–320.
[33] M. Bartoletti, M. Murgia, A. Scalas, R. Zunino, The CO_2 honesty checker (2015).
URL http://tcs.unica.it/software/co2-maude
[34] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J. F. Quesada, Maude: Specification and programming in rewriting logic, TCS.
[35] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, Electr. Notes Theor. Comput. Sci. 71 (2002) 162–187. doi:10.1016/S1571-0661(05)82534-4.
[36] A. Verdejo, N. Martí-Oliet, Implementing CCS in Maude 2, Electr. Notes Theor. Comput. Sci. 71 (2002) 282–300, WRLA 2002, Rewriting Logic and Its Applications. doi:10.1016/S1571-0661(05)82540-X.
[37] J. Meseguer, Rewriting as a unified model of concurrency, in: CONCUR, Vol. 458 of LNCS, 1990, pp. 384–400.
[38] P. Thati, K. Sen, N. Martí-Oliet, An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0, ENTCS 71.
[39] M.-O. Stehr, J. Meseguer, P. C. Ölveczky, Rewriting logic as a unifying framework for Petri nets, in: Unifying Petri Nets, 2001, pp. 250–303.
[40] M. Neuhäußer, T. Noll, Abstraction and model checking of core Erlang programs in Maude, ENTCS 176 (4).
[41] M. Wirsing, J. Eckhardt, T. Mühlbauer, J. Meseguer, Design and analysis of cloud-based architectures with KLAIM and Maude, in: WRLA, Vol. 7571 of LNCS, 2012, pp. 54–82.
[42] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, A. Vandin, Modelling and analyzing adaptive self-assembly strategies with Maude, in: WRLA, Vol. 7571 of LNCS, 2012, pp. 118–138.
[43] J. Meseguer, Twenty years of rewriting logic, JLAP 81 (7-8).
[44] J. Lange, A. Scalas, Choreography synthesis as contract agreement, in: ICE, 2013, pp. 52–67.
[45] M. Bartoletti, J. Lange, A. Scalas, R. Zunino, Choreographies in the wild, Science of Computer Programming (2014) –To appear. doi:http://dx.doi.org/10.1016/j.scico.2014.11.015.
[46] M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, L. Pompianu, A contract-oriented middleware (2015).

URL http://co2.unica.it
[47] M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, L. Pompianu, Compliance and subtyping in timed session types, in: Proc. FORTE, 2015, pp. 161–177. doi:10.1007/978-3-319-19195-9_11.
[48] V. A. Saraswat, M. C. Rinard, Concurrent constraint programming, in: POPL, 1990, pp. 232–245. doi:10.1145/96709.96733.
[49] M. G. Buscemi, U. Montanari, CC-Pi: A constraint-based language for specifying service level agreements, in: ESOP, 2007, pp. 18–32. doi:10.1007/978-3-540-71316-6_3.
[50] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, N. Yoshida, Asynchronous distributed monitoring for multiparty session enforcement, in: R. Bruni, V. Sassone (Eds.), Trustworthy Global Computing, Vol. 7173 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 25–45. doi:10.1007/978-3-642-30065-3_2.
[51] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, N. Yoshida, Monitoring networks through multiparty session types, in: FORTE, Vol. 7892 of LNCS, Springer, 2013, pp. 50–65. doi:10.1007/978-3-642-38592-6_5.
[52] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Inf. Syst. Secur. 12 (3). doi:10.1145/1455526.1455532.
[53] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: PARLE, 1994, pp. 398–413. doi:10.1007/3-540-58184-7_118.
[54] M. Dezani-Ciancaglini, U. de'Liguoro, N. Yoshida, On progress for structured communications, in: Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers, 2007, pp. 257–275. doi:10.1007/978-3-540-78663-4_18.
[55] L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: CONCUR, 2008, pp. 418–433. doi:10.1007/978-3-540-85361-9_33.
URL http://dx.doi.org/10.1007/978-3-540-85361-9_33
[56] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, L. Padovani, Foundations of session types, in: PPDP, 2009, pp. 219–230. doi:10.1145/1599410.1599437.
[57] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, N. Yoshida, Inference of global progress properties for dynamically interleaved multiparty sessions, in: COORDINATION, 2013, pp. 45–59. doi:10.1007/978-3-642-38493-6_4.
[58] M. Carbone, F. Montesi, Deadlock-freedom-by-design: Multiparty asynchronous global programming, in: POPL, 2013, pp. 263–274. doi:10.1145/2429069.2429101.
[59] M. Bartoletti, M. Murgia, A. Scalas, R. Zunino, Modelling and verifying contract-oriented systems in Maude, in: WRLA, Vol. 8663 of LNCS, 2014, pp. 130–146. doi:10.1007/978-3-319-12904-4_7.

A. Proofs for Section 2

Lemma A.1. *If* $\gamma \twoheadrightarrow \mathsf{A} : \texttt{rdy a?v}.\, c \mid \mathsf{B} : d$*, then* d *is* $\texttt{rdy}$*-free.*

Proof. By Definition 2.1, only one $\texttt{rdy}$ can occur in γ, and if so it must occur at top-level of a contract. This properties are preserved by transitions (Definition 2.2). The thesis then follows by straightforward analysis of the rules in Figure 1. □

Below we establish that contracts are deterministic. Determinism ensures that the obligations of participants at any given time are uniquely determined by their past actions.

Lemma A.2 (Determinism). *For all transition labels* μ *of* $\twoheadrightarrow$*:*

$$\gamma \xrightarrow{\mu}\!\!\!\!\twoheadrightarrow \gamma' \;\wedge\; \gamma \xrightarrow{\mu}\!\!\!\!\twoheadrightarrow \gamma'' \quad\Longrightarrow\quad \gamma' \equiv \gamma''$$

Proof. Let $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$. We have the following two cases, according to the rule used to deduce $\gamma \xrightarrow{\mu}\!\!\!\!\twoheadrightarrow \gamma'$ (w.l.o.g. we assume that μ is a move of A):

[Rdy] Straightforward consequence of Lemma A.1.

[IntExt] The thesis follows by the assumption that the branch labels in c and d are pairwise distinct (item (ii) of Definition 2.1). □

Lemma A.3. *If* $\gamma \equiv \gamma'$*, then* γ *is safe iff* γ' *is safe.*

Proof. Straightforward by Definition 2.3 and by the definition of the equivalence $\equiv$ (Definition 2.2). □

The following lemma guarantees, for all 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 contracts* c*, there exists some* d *such that* $c \bowtie d$*.*

Proof. Let the contract $\mathrm{dual}(c)$ be inductively defined as follows:

$$\mathrm{dual}(\textstyle\bigoplus_i \texttt{a}_i\texttt{!T}_i \,.\, c_i) = \sum_i \texttt{a}_i\texttt{?T}_i \,.\, \mathrm{dual}(c_i) \qquad \mathrm{dual}(\texttt{rec}\, X \,.\, c) = \texttt{rec}\, X \,.\, \mathrm{dual}(c)$$
$$\mathrm{dual}(\textstyle\sum_i \texttt{a}_i\texttt{?T}_i \,.\, c_i) = \bigoplus_i \texttt{a}_i\texttt{!T}_i \,.\, \mathrm{dual}(c_i) \qquad \mathrm{dual}(X) = X$$

Now, let $\mathcal{R}$ be the smallest relation such that, for all $\texttt{rdy}$-free c:

$$(c, \mathrm{dual}(c)) \in \mathcal{R} \tag{2}$$
$$(c, \texttt{rdy a?v}.\, \mathrm{dual}(c)) \in \mathcal{R} \tag{3}$$
$$(\texttt{rdy a?v}.\, \mathrm{dual}(c)\,, c) \in \mathcal{R} \tag{4}$$

Also, for all pairs $\mathcal{X}$ of contracts, let:

$$F(\mathcal{X}) \;=\; \left\{ (c,d) \;\middle|\; \begin{array}{l} \mathsf{A} : c \mid \mathsf{B} : d \quad \text{is safe, and} \\ \mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow \mathsf{A} : c' \mid \mathsf{B} : d' \implies (c', d') \in \mathcal{X} \end{array} \right\}$$

By the coinduction proof principle, we have to show that $\mathcal{R} \subseteq F(\mathcal{R})$. Suppose that $(c,d) \in \mathcal{R}$. The three equations defining $\mathcal{R}$ satisfy the first requirement of F (safety). This is trivial for equations 3 and 4, while for 2 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 $(c,d) \in \mathcal{R}$.

(2) $d = \mathrm{dual}(c)$. We have two subcases, according to the form of c. Assume first that c is an external choice, i.e. $c = (\texttt{a?T} \,.\, c') + c''$. Then $d = \texttt{a!T} \,.\, \mathrm{dual}(c') \oplus \mathrm{dual}(c'')$. By rule [IntExt], $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow \mathsf{A} : c' \mid \mathsf{B} : \texttt{rdy a?v}.\, \mathrm{dual}(c')$, We obtain the thesis by observing that $(c', \texttt{rdy a?v}.\, \mathrm{dual}(c')) \in \mathcal{R}$ follows by (3). The case where c is an internal choice is similar.

(3) $d = \texttt{rdy a?v}.\,\mathrm{dual}(c)$. By rule [Rdy], $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : \mathrm{dual}(c)$. The thesis follows by observing that $(c, \mathrm{dual}(c)) \in \mathcal{R}$ follows by (2).

(4) $c = \texttt{rdy a?v}.\,\mathrm{dual}(d)$. Similar to the previous case.

Summing up, $\mathcal{R}$ is a compliance relation, hence by item (2) it follows that $c \bowtie \mathrm{dual}(c)$. □

Definition A.4 (Value abstraction of contracts). *Value-abstract contracts are terms of the grammar:*

$$\hat{c} \quad ::= \quad \bigoplus_{i \in \mathcal{I}} (\mathtt{a}_i, \mathtt{T}_i)\texttt{!} \,.\, \hat{c}_i \;\;\Big|\;\; \sum_{i \in \mathcal{I}} (\mathtt{a}_i, \mathtt{T}_i)\texttt{?} \,.\, \hat{c}_i \;\;\Big|\;\; \texttt{rdy}\ (\mathtt{a}_i, \mathtt{T}_i)\texttt{?}.\hat{c} \;\;\Big|\;\; \texttt{rec}\, X \,.\, \hat{c} \;\Big|\;\; X$$

where (i) the index set $\mathcal{I}$ is finite, (ii) the labels $\mathtt{a}_i$ in the prefixes of each summation are pairwise distinct, (iii) recursion variables X are guarded, and (iv) $\texttt{rdy}$ occurs at the top-level, only.

For all (concrete) contracts c, we define the value-abstract contract $\alpha^\star(c)$ as follows

$$\alpha^\star(c) = \begin{cases} \bigoplus_{i \in \mathcal{I}} (\mathtt{a}_i, \mathtt{T}_i)\texttt{!} \,.\, \alpha^\star(c_i) & \text{if } c = \bigoplus_{i \in \mathcal{I}} \mathtt{a}_i\texttt{!}\mathtt{T}_i \,.\, c_i \\ \sum_{i \in \mathcal{I}} (\mathtt{a}_i, \mathtt{T}_i)\texttt{?} \,.\, \alpha^\star(c_i) & \text{if } c = \sum_{i \in \mathcal{I}} \mathtt{a}_i\texttt{?}\mathtt{T}_i \,.\, c_i \\ \texttt{rdy}\ (\mathtt{a}, \mathtt{T})\texttt{?}.\,\alpha^\star(c) & \text{if } c = \texttt{rdy a?v}.\,c \text{ and } \mathtt{v} : \mathtt{T} \\ \texttt{rec}\, X \,.\, \alpha^\star(c) & \text{if } c = \texttt{rec}\, X \,.\, c \\ X & \text{if } c = X \end{cases}$$

For contract configurations $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, we define $\alpha^\star(\gamma) = \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d)$.

Lemma A.5. *For all contracts configurations γ, γ', values $\mathtt{v}$, sorts $\mathtt{T}$, and $\circ \in \{\texttt{!}, \texttt{?}\}$:*

1. $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\twoheadrightarrow \gamma' \;\wedge\; \mathtt{v} : \mathtt{T} \implies \alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\circ}\!\!\!\twoheadrightarrow_\star \alpha^\star(\gamma')$
2. $\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\circ}\!\!\!\twoheadrightarrow_\star \hat{\gamma}' \;\wedge\; \mathtt{v} : \mathtt{T} \implies \exists \gamma' \,.\, \gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\twoheadrightarrow \gamma' \;\wedge\; \hat{\gamma}' = \alpha^\star(\gamma')$

Proof. For item 1 we proceed by cases on the rule used to deduce $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\twoheadrightarrow \gamma'$. By the semantics of concrete and value-abstract contracts and by Definition A.4, we have:

- [IntExt]

$$\begin{aligned} \gamma &= \mathsf{A} : \mathtt{a}\texttt{!}\mathtt{T} \,.\, c \oplus c' \mid \mathsf{B} : \mathtt{a}\texttt{?}\mathtt{T} \,.\, d + d' \\ &\xrightarrow{\mathsf{A}:\mathtt{a}\texttt{!}\mathtt{v}}\!\!\!\twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : \texttt{rdy a?v}.\,d \\ &= \gamma' \\ \alpha^\star(\gamma) &= \mathsf{A} : (\mathtt{a}, \mathtt{T})\texttt{!} \,.\, \alpha^\star(c) \oplus \alpha^\star(c') \mid \mathsf{B} : (\mathtt{a}, \mathtt{T})\texttt{?} \,.\, \alpha^\star(d) + \alpha^\star(d') \\ &\xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\texttt{!}}\!\!\!\twoheadrightarrow_\star \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \texttt{rdy}\ (\mathtt{a}, \mathtt{T})\texttt{?}.\,\alpha^\star(d) \\ &= \alpha^\star(\gamma') \end{aligned}$$

- [Rdy]

$$\gamma = \mathsf{A} : \texttt{rdy a?v}.\,c \mid \mathsf{B} : d \xrightarrow{\mathsf{A}:\mathtt{a}\texttt{?}\mathtt{v}}\!\!\!\twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : d \;=\; \gamma'$$

$$\alpha^\star(\gamma) = \mathsf{A} : \texttt{rdy}\ (\mathtt{a}, \mathtt{T})\texttt{?}.\,\alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\texttt{?}}\!\!\!\twoheadrightarrow_\star \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \;=\; \alpha^\star(\gamma')$$

For item 2 we proceed by cases on the rule used to deduce $\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\circ}\!\!\!\twoheadrightarrow_\star \hat{\gamma}'$. By the semantics of concrete and value-abstract contracts and by Definition A.4, we have:

- [AbsIntExt]

$$\begin{aligned}\alpha^\star(\gamma) &= \mathsf{A} : (\mathtt{a},\mathtt{T})!\,.\,\alpha^\star(c) \oplus \alpha^\star(c') \mid \mathsf{B} : (\mathtt{a},\mathtt{T})?\,.\,\alpha^\star(d) + \alpha^\star(d') \\ &\xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})!}\!\!\twoheadrightarrow_\star \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \mathtt{rdy}\ (\mathtt{a},\mathtt{T})?.\,\alpha^\star(d) \\ &= \hat{\gamma}' \\ \gamma &= \mathsf{A} : \mathtt{a!T}\,.\,c \oplus c' \mid \mathsf{B} : \mathtt{a?T}\,.\,d + d' \\ &\xrightarrow{\mathsf{A}:\mathtt{a!v}}\!\!\twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{a?v}.\,d \\ &= \gamma'\end{aligned}$$

  Hence by Definition A.4 we conclude that $\alpha^\star(\gamma') = \hat{\gamma}'$.

- [AbsRdy]

$$\alpha^\star(\gamma) = \mathsf{A} : \mathtt{rdy}\ (\mathtt{a},\mathtt{T})?.\,\alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \xrightarrow{(\mathtt{a},\mathtt{T})?}\!\!\twoheadrightarrow_\star \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \;=\; \hat{\gamma}'$$

$$\gamma = \mathsf{A} : \mathtt{rdy}\ \mathtt{a?v}.\,c \mid \mathsf{B} : d \xrightarrow{\mathsf{A}:\mathtt{a?v}}\!\!\twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : d \;=\; \gamma'$$

  Hence by Definition A.4 we conclude that $\alpha^\star(\gamma') = \hat{\gamma}'$. □

Lemma 2.6. *For all contracts c, d:*

$$c \bowtie d \iff (\forall \gamma.\ \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) \twoheadrightarrow_\star^* \gamma \implies \gamma \textit{ safe})$$

Proof. Straightforward consequence of Lemma A.5 and of the fact that γ is safe iff $\alpha^\star(\gamma)$ is such. □

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

Proof. Let $\gamma = \mathsf{A} : c' \mid \mathsf{B} : d'$. By Definition 2.3 it follows that γ is safe. Hence, by Definition 2.3 we have the following three cases (symmetric ones are omitted):

- $\gamma = \mathsf{A} : 0 \mid \mathsf{B} : 0$, from which the thesis follows directly.
- $\gamma = \mathsf{A} : \mathtt{rdy}\ \mathtt{a?v}.\,c' \mid \mathsf{B} : d'$, with d' $\mathtt{rdy}$-free (Lemma A.1). We have only one transition from γ, given by rule [Rdy], which prescribes a move of A. By Definition 2.8, we have that $\mathsf{A} \frown \gamma$ and $\mathsf{B} \smile \gamma$.
- $\gamma = \mathsf{A} : \bigoplus_{i\in\mathcal{I}} \mathtt{a}_i!\mathtt{T}_i\,.\,c_i \mid \mathsf{B} : \sum_i \mathtt{a}_{i\in\mathcal{I}}?\mathtt{T}_i\,.\,d_i + \sum_j \mathtt{b}_j?\mathtt{T}_j\,.\,d_j$, with $\mathcal{I} \neq \emptyset$. Only rule [IntExt] can be applied, and it prescribes a move of A. By Definition 2.8, we conclude that $\mathsf{A} \frown \gamma$ and $\mathsf{B} \smile \gamma$. □

Theorem 2.10. *Let $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, and let $\gamma \twoheadrightarrow^* \gamma'$. Then:*

1. $\gamma' \not\twoheadrightarrow \implies \mathsf{A} \smile \gamma'$ *and* $\mathsf{B} \smile \gamma'$
2. $\mathsf{A} \frown \gamma' \implies \forall \gamma'' : \gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathsf{A} \smile \gamma'', \textit{ or} \\ \forall \gamma''' : \gamma'' \twoheadrightarrow \gamma''' \implies \mathsf{A} \smile \gamma''' \end{cases}$

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).

[IntExt] We have that $\gamma' = \mathsf{A} : \mathtt{a!T}\,.\,c' \oplus c'' \mid \mathsf{B} : \mathtt{a?T}\,.\,d' + d''$, and $\gamma'' = \mathsf{A} : c' \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{a?v}.\,d'$. Now, γ'' can only take a move of B (by rule [Rdy]). Therefore, $\mathsf{A} \smile \gamma''$.

[Rdy] We have that $\gamma' = \mathsf{A} : \mathtt{rdy}\ \mathtt{a?v}.\,c' \mid \mathsf{B} : d'$, and $\gamma'' = \mathsf{A} : c' \mid \mathsf{B} : d'$. Now we have two further subcases. If $\mathsf{A} \smile \gamma''$, then we have the thesis. Otherwise, if $\mathsf{A} \frown \gamma''$, since c' and d' are $\mathtt{rdy}$-free, then γ'' must have the form $\mathsf{A} : \mathtt{a!T}\,.\,c' \oplus c'' \mid \mathsf{B} : \mathtt{a?T}\,.\,d' + d''$. By rule [IntExt] we have that, if $\gamma'' \xrightarrow{\mathsf{A}:\mathtt{b!v}}\!\!\twoheadrightarrow \gamma'''$ for some $\mathtt{b}$ and $\mathtt{v}$, then γ''' will have the form $\mathsf{A} : c''' \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{b?v}.\,d'''$, for some c''', d'''. Therefore, $\mathsf{A} \smile \gamma'''$. □

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}:\,\tau}_{\star} \mathsf{A}[P \mid Q] \qquad [\alpha^\star\text{TAU}]$$

$$\mathsf{A}[(\texttt{if} \star \texttt{then } P_0 \texttt{ else } P_1) \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{if}}_{\star} \mathsf{A}[P_i \mid Q] \quad (i \in \{0,1\}) \qquad [\alpha^\star\text{IF}]$$

$$\mathsf{A}[\texttt{tell} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}:\,\texttt{tell} \downarrow_u c}_{\star} \mathsf{A}[P \mid Q] \mid \{\downarrow_u c\}_{\mathsf{A}} \qquad [\alpha^\star\text{TELL}]$$

$$\frac{c \bowtie d \qquad \gamma = \mathsf{A} : c \mid \mathsf{B} : d \qquad \sigma = \{^s/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \mid \{\downarrow_x c\}_{\mathsf{A}} \mid \{\downarrow_y d\}_{\mathsf{B}}) \xrightarrow{\mathsf{K}:\,\texttt{fuse}}_{\star} (s)(S\sigma \mid s[\gamma])} \qquad [\alpha^\star\text{FUSE}]$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:a} \gamma'}{\mathsf{A}[\texttt{do}_s\, a.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{do}_s\, a}_{\star} \mathsf{A}[P \mid Q] \mid s[\gamma']} \qquad [\alpha^\star\text{DO}]$$

$$\frac{\gamma \vdash_\star \phi}{\mathsf{A}[\texttt{ask}_s\, \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\,\texttt{ask}_s\, \phi}_{\star} \mathsf{A}[P \mid Q] \mid s[\gamma]} \qquad [\alpha^\star\text{ASK}]$$

$$\frac{\mathrm{X}(\boldsymbol{u},\star) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{^{\boldsymbol{v}}/_{\boldsymbol{u}}\} \mid Q] \mid S \xrightarrow{\mu}_{\star} S'}{\mathsf{A}[\mathrm{X}(\boldsymbol{v},\star) \mid Q] \mid S \xrightarrow{\mu}_{\star} S'} \; [\alpha^\star\text{DEF}] \qquad \frac{S \xrightarrow{\mu}_{\star} S'}{S \mid S'' \xrightarrow{\mu}_{\star} S' \mid S''} \; [\alpha^\star\text{PAR}]$$

$$\frac{S \xrightarrow{\mathsf{A}:\,\pi}_{\star} S'}{(u)S \xrightarrow{\mathsf{A}:\,\mathrm{del}_u(\pi)}_{\star} (u)S'} \; [\alpha^\star\text{DEL}] \qquad \text{where } \mathrm{del}_u(\pi) = \begin{cases} \tau & \text{if } u \in \mathrm{fnv}(\pi) \\ \pi & \text{otherwise} \end{cases}$$

Figure 8: Reduction semantics of value-abstract systems.

B. Proofs for Section 4

B.1. Proofs for Section 4.1

We shall use metavariables $\hat{S}, \hat{S}', \ldots$ to range over value-abstract systems. However, when there is no ambiguity about whether we are referring to a concrete or to a value-abstract system, we shall use the standard metavariables S, S'. We adopt similar conventions for value-abstract contract configurations.

Definition B.1 (Value abstraction of systems). *For all systems S, we define the value abstract system $\alpha^\star(S)$ as follows (see Definition A.4 for the specification of $\alpha^\star(\gamma)$):*

$$\alpha^\star(\mathsf{A}[P]) = \mathsf{A}[\alpha^\star(P)] \qquad \alpha^\star(s[\gamma]) = s[\alpha^\star(\gamma)]$$
$$\alpha^\star(\{\downarrow_x c\}_{\mathsf{A}}) = \{\downarrow_x c\}_{\mathsf{A}} \qquad \alpha^\star(S \mid S') = \alpha^\star(S) \mid \alpha^\star(S')$$
$$\alpha^\star((u)\, S) = (u)\, (\alpha^\star(S)) \qquad \alpha^\star(\mathbf{0}) = \mathbf{0}$$

Example B.2 (Online store). *The value-abstract counterpart of the process P_{A} in Example 3.2 is:*

$$\alpha^\star(P_{\mathsf{A}}) = (x)\ (\texttt{tell} \downarrow_x c_{\mathsf{A}}.\, \texttt{do}_x\, (\texttt{addToCart}, \texttt{int})?.\, Q_{\texttt{add}}(x,\star))$$
$$Q_{\texttt{add}}(x,t) \stackrel{def}{=} \texttt{do}_x\, (\texttt{addToCart}, \texttt{int})?.\, Q_{\texttt{add}}(x,\star) \;+\; \texttt{do}_x\, (\texttt{checkout}, \texttt{unit})?.\, Q_{\texttt{pay}}(x,\star)$$
$$Q_{\texttt{pay}}(x,t) \stackrel{def}{=} \texttt{do}_x\, (\texttt{pay}, \texttt{string})?.\, Q_{\texttt{ack}}(x,\star) \;+\; \texttt{do}_x\, (\texttt{cancel}, \texttt{unit})?$$
$$Q_{\texttt{ack}}(x,t) \stackrel{def}{=} \texttt{if} \star \texttt{then do}_x\, (\texttt{ok}, \texttt{unit})! \texttt{ else do}_x\, (\texttt{no}, \texttt{unit})!.\, Q_{\texttt{pay}}(x,\star)$$

We now formalise the meaning of the relation $\vdash$ used in rule [ASK]. We denote with $\mathbf{A}$ a set of atomic propositions, whose elements are terms of the form $(\texttt{a}, \texttt{T})\circ$, where $\texttt{a}$ is a branch label, $\texttt{T}$ is a sort, and $\circ \in \{!, ?\}$. Let $\ell, \ell', \cdots$ range over $\mathbf{A} \cup \{\varepsilon\}$.

Definition B.3. *We define the (unlabelled) transition system* $\mathrm{TS}_\star = (\Sigma, \to, \mathbf{A}, L)$ *as follows:*

- $\Sigma = \{(\ell, \hat{\gamma}) \mid \ell \in \mathbf{A} \cup \{\varepsilon\},$ *and* $\hat{\gamma}$ *is a configuration of compliant value-abstract contracts*$\}$,
- *the transition relation* $\to \subseteq \Sigma \times \Sigma$ *is defined by the following rule:*

$$(\ell, \hat{\gamma}) \to (a, \hat{\gamma}') \qquad\qquad \text{if } \hat{\gamma} \xrightarrow{\mathsf{A}:a}\!\!\!\!\twoheadrightarrow_\star \hat{\gamma}'$$

- *the labelling function* $L : \Sigma \to \mathbf{A} \cup \{\varepsilon\}$ *is defined as* $L((\ell, \hat{\gamma})) \; = \; \ell$.

We write $\hat{\gamma} \vdash_\star \phi$ *whenever* $\forall \lambda \in \mathrm{Paths}((\epsilon, \hat{\gamma}))$, $\lambda \models \phi$ *holds in LTL.*

Lemma B.4. *For all compliant* γ*:*

$$\mathrm{Paths}((\epsilon, \gamma)) = \mathrm{Paths}((\epsilon, \alpha^\star(\gamma)))$$

Proof. Straightforward consequence of Lemma A.5

Corollary B.5. *For all compliant* γ*:*

$$\gamma \vdash \phi \iff \alpha^\star(\gamma) \vdash \phi$$

Definition B.6 (Value-abstract labels). *For all transition labels* μ *of* $\to$ *(*CO_2 *semantics, Figure 4), we define the* value-abstract label $\alpha^\star(\mu)$ *as:*

$$\alpha^\star(\mu) = \begin{cases} \mathsf{A} : \mathtt{do}_s\,(\mathtt{a}, \mathtt{T})\circ & \text{if } \mu = \mathsf{A} : \mathtt{do}_s\, \mathtt{a} \circ \mathtt{v} \text{ and } \mathtt{v} : \mathtt{T} \\ \mathsf{A} : \mathtt{tell} \downarrow_u \alpha^\star(c) & \text{if } \mu = \mathsf{A} : \mathtt{tell} \downarrow_u c \\ \mu & \text{otherwise} \end{cases}$$

Lemma B.7. *For all substitutions* σ *and systems* S*:*

$$\alpha^\star(S)\sigma = \alpha^\star(S\sigma)$$

Furthermore, if σ *is a value-substitution:*

$$\alpha^\star(S)\sigma = \alpha^\star(S)$$

Proof. By structural induction on S.

Lemma B.8. *Let* $\mu = \mathsf{A} : \pi$ *and* $\alpha^\star(\mu) = \mathsf{A} : \hat{\pi}$*. Then:*

$$\alpha^\star(\mathsf{A} : \mathrm{del}_u(\pi)) = \mathsf{A} : \mathrm{del}_u(\hat{\pi})$$

where $\mathrm{del}_u(\cdot)$ *is defined in Figure 4.*

Proof. Trivial.

Lemma B.9. $S \xrightarrow{\mu} S' \implies \alpha^\star(S) \xrightarrow{\alpha^\star(\mu)}_\star \alpha^\star(S')$

Proof. Suppose $S \xrightarrow{\mu} S'$. We proceed by rule induction, rewriting the transitions according to Definition B.6 and Figure 8:

[Tau] Assume:

$$S = \mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}:\tau} \mathsf{A}[P \mid Q] = S'$$

Then:

$$\alpha^\star(S) = \mathsf{A}[\tau.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \xrightarrow{\mathsf{A}:\tau}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] = \alpha^\star(S')$$

[Tell] Assume:

$$S = \mathsf{A}[\mathtt{tell} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}:\,\mathtt{tell} \downarrow_u c} \mathsf{A}[P \mid Q] \mid \{\downarrow_u c\}_{\mathsf{A}} = S'$$

Then:

$$\alpha^\star(S) = \mathsf{A}[\mathtt{tell} \downarrow_u \alpha^\star(c).\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \xrightarrow{\mathsf{A}:\,\mathtt{tell} \downarrow_u \alpha^\star(c)}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \mid \{\downarrow_u \alpha^\star(c)\}_{\mathsf{A}} = \alpha^\star(S')$$

[Fuse] Assume:

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

By Lemma 2.6 (RHS of the double implication) and Definition 2.3, with an obvious overloading of $\bowtie$ we have $\alpha^\star(c) \bowtie \alpha^\star(d)$. Then, by Lemma B.7:

$$\frac{\alpha^\star(c) \bowtie \alpha^\star(d) \qquad \hat{\gamma} = \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) = \alpha^\star(\gamma) \qquad \sigma = \{{}^{s}/_{x,y}\} \qquad s \text{ fresh}}{\alpha^\star(S) = (x,y)(\alpha^\star(S'') \mid \{\downarrow_x \alpha^\star(c)\}_{\mathsf{A}} \mid \{\downarrow_y \alpha^\star(d)\}_{\mathsf{B}}) \xrightarrow{\mathsf{K}:\,\mathtt{fuse}}_\star (s)(\alpha^\star(S''\sigma) \mid s[\alpha^\star(\gamma)]) = \alpha^\star(S')}$$

[If] Suppose $[\![e]\!] = \mathtt{true}$ (the other case is similar) and:

$$S = \mathsf{A}[(\mathtt{if}\ e\ \mathtt{then}\ P_{\mathtt{true}}\ \mathtt{else}\ P_{\mathtt{false}}) \mid Q] \xrightarrow{\mathsf{A}:\,\mathtt{if}} \mathsf{A}[P_{\mathtt{true}} \mid Q] = S'$$

Then:

$$\alpha^\star(S) = \mathsf{A}[(\mathtt{if}\ \star\ \mathtt{then}\ \alpha^\star(P_{\mathtt{true}})\ \mathtt{else}\ \alpha^\star(P_{\mathtt{false}})) \mid \alpha^\star(Q)] \xrightarrow{\mathsf{A}:\,\mathtt{if}} \mathsf{A}[\alpha^\star(P_{\mathtt{true}}) \mid \alpha^\star(Q)] = \alpha^\star(S')$$

[Do!] Suppose:

$$\frac{[\![e]\!] = \mathtt{v} \qquad \gamma \xrightarrow{\mathsf{A}:\mathtt{a!v}}\!\!\!\!\rightarrow \gamma'}{S = \mathsf{A}[\mathtt{do}_s\ \mathtt{a!}e.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\mathtt{do}_s\ \mathtt{a!v}} \mathsf{A}[P \mid Q] \mid s[\gamma'] = S'}$$

Then, assuming $\mathtt{v} : \mathtt{T}$, by item 1 of Lemma A.5:

$$\frac{\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})!}\!\!\!\!\rightarrow \alpha^\star(\gamma')}{\alpha^\star(S) = \mathsf{A}[\mathtt{do}_s\ (\mathtt{a},\mathtt{T})!.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma)] \xrightarrow{\mathsf{A}:\mathtt{do}_s\ (\mathtt{a},\mathtt{T})!} \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma')] = \alpha^\star(S')}$$

[Do?] Suppose:

$$\frac{\gamma \xrightarrow{\mathsf{A}:\mathtt{a?v}}\!\!\!\!\rightarrow \gamma' \qquad \mathtt{v} : \mathtt{T}}{S = \mathsf{A}[\mathtt{do}_s\ \mathtt{a?}x : \mathtt{T}.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\mathtt{do}_s\ \mathtt{a?v}} \mathsf{A}[P\{\mathtt{v}/x\} \mid Q] \mid s[\gamma'] = S'}$$

Then, assuming $\mathtt{v} : \mathtt{T}$, by item 1 of Lemma A.5 and the "furthermore…" part of lemma B.7:

$$\frac{\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})?}\!\!\!\!\rightarrow \alpha^\star(\gamma')}{\alpha^\star(S) = \mathsf{A}[\mathtt{do}_s\ (\mathtt{a},\mathtt{T})?.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma)] \xrightarrow{\mathsf{A}:\mathtt{do}_s\ (\mathtt{a},\mathtt{T})!} \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma')] = \alpha^\star(S')}$$

[Ask] Suppose:

$$\frac{\gamma \vdash \phi}{S = \mathsf{A}[\mathtt{ask}_s\ \phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\,\mathtt{ask}_s\ \phi} \mathsf{A}[P \mid Q] \mid s[\gamma] = S'}$$

We can now notice that, by Definition 4.7, the premise $\gamma \vdash \phi$ only depends on the *labels* (and ignores the values) along the transitions of γ; therefore, since $\alpha^\star(\gamma)$ preserves such labels (see Definition A.4), with an obvious overloading of $\vdash$ we obtain $\gamma \vdash \phi \iff \alpha^\star(\gamma) \vdash \phi$. Hence, we have:

$$\frac{\alpha^\star(\gamma) \vdash \phi}{\alpha^\star(S) = \mathsf{A}[\mathtt{ask}_s\ \phi.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma)] \xrightarrow{\mathsf{A}:\,\mathtt{ask}_s\ \phi}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \mid s[\alpha^\star(\gamma)] = \alpha^\star(S')}$$

[DEF] Suppose:

$$\frac{\mathrm{X}(\boldsymbol{x},\boldsymbol{y}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{\boldsymbol{u}/\boldsymbol{x}\}\{\boldsymbol{e}/\boldsymbol{y}\} \mid Q] \mid S'' \xrightarrow{\mu} S'}{S = \mathsf{A}[\mathrm{X}(\boldsymbol{u},\boldsymbol{e}) \mid Q] \mid S'' \xrightarrow{\mu} S'}$$

Then, by applying the induction hypothesis, and by Lemma B.7:

$$\frac{\mathrm{X}(\boldsymbol{u},\star) \stackrel{\text{def}}{=} \alpha^\star(P) \qquad \mathsf{A}[\alpha^\star(P)\{\boldsymbol{v}/\boldsymbol{u}\} \mid \alpha^\star(Q)] \mid \alpha^\star(S'') \xrightarrow{\alpha^\star(\mu)}_\star \alpha^\star(S')}{\alpha^\star(S) = \mathsf{A}[\mathrm{X}(\boldsymbol{v},\star) \mid \alpha^\star(Q)] \mid \alpha^\star(S'') \xrightarrow{\alpha^\star(\mu)}_\star \alpha^\star(S')}$$

[PAR] Suppose:

$$\frac{S_0 \xrightarrow{\mu} S'_0}{S = S_0 \mid S'' \xrightarrow{\mu} S'_0 \mid S'' = S'}$$

Then, by applying the induction hypothesis:

$$\frac{\alpha^\star(S_0) \xrightarrow{\alpha^\star(\mu)} \alpha^\star(S'_0)}{\alpha^\star(S) = \alpha^\star(S_0) \mid \alpha^\star(S'') \xrightarrow{\alpha^\star(\mu)} \alpha^\star(S'_0) \mid \alpha^\star(S'') = \alpha^\star(S')}$$

[DEL] Suppose:

$$\frac{S_0 \xrightarrow{\mathsf{A}:\,\pi} S'_0}{S = (u)S_0 \xrightarrow{\mathsf{A}:\,\mathrm{del}_u(\pi)} (u)S'_0 = S'}$$

Then, by applying the induction hypothesis, and by Lemma B.8:

$$\frac{\alpha^\star(S_0) \xrightarrow{\alpha^\star(\mathsf{A}:\,\pi)}_\star \alpha^\star(S'_0)}{\alpha^\star(S) = (u)\alpha^\star(S_0) \xrightarrow{\alpha^\star(\mathsf{A}:\,\mathrm{del}_u(\pi))}_\star (u)\alpha^\star(S'_0) = \alpha^\star(S')}$$

Lemma B.10. *For all systems S, value-abstract systems $\hat{S}'$ and labels $\hat{\mu}$:*

$$\alpha^\star(S) \xrightarrow{\hat{\mu} \neq \mathsf{A}:\mathtt{if}}_\star \hat{S}' \implies \exists \mu, S' : S \xrightarrow{\mu} S' \,\wedge\, \alpha^\star(\mu) = \hat{\mu} \,\wedge\, \alpha^\star(S') = \hat{S}'$$

Proof. We proceed by induction on the rule used to derive $\hat{S} = \alpha^\star(S) \xrightarrow{\hat{\mu}}_\star \hat{S}'$ (excluding the case $\hat{\mu} = \mathsf{A} : \mathtt{if}$):

[TAU] Assume $S = \mathsf{A}[\tau.P + P' \mid Q]$. We have:

$$\alpha^\star(S) = \mathsf{A}[\tau.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \xrightarrow{\hat{\mu} = \mathsf{A}:\,\tau}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] = \hat{S}'$$

Then:

$$S \xrightarrow{\mu = \mathsf{A}:\,\tau} \mathsf{A}[P \mid Q] = S'$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[TELL] Assume $S = \mathsf{A}[\mathtt{tell}\downarrow_u c.P + P' \mid Q]$. We have:

$$\alpha^\star(S) = \mathsf{A}[\mathtt{tell}\downarrow_u \alpha^\star(c).\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \xrightarrow{\hat{\mu} = \mathsf{A}:\,\downarrow_u \alpha^\star(c)}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \mid \{\downarrow_u \alpha^\star(c)\}_{\mathsf{A}} = \hat{S}'$$

Then:

$$S \xrightarrow{\mu = \mathsf{A}:\,\downarrow_u c} \mathsf{A}[P \mid Q \mid \{\downarrow_u c\}_{\mathsf{A}}] = S'$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[Fuse] Assume $S = (x,y)(S'' \mid \{\downarrow_x c\}_{\mathsf{A}} \mid \{\downarrow_y d\}_{\mathsf{B}})$. We have:

$$\frac{\alpha^\star(c) \bowtie \alpha^\star(d) \qquad \hat{\gamma} = \mathsf{A} : \alpha^\star(c) \mid \mathsf{B} : \alpha^\star(d) = \alpha^\star(\gamma) \qquad \sigma = \{s/x,y\} \qquad s \text{ fresh}}{\alpha^\star(S) = (x,y)(\alpha^\star(S'') \ \mid\ \{\downarrow_x \alpha^\star(c)\}_{\mathsf{A}} \ \mid\ \{\downarrow_y \alpha^\star(d)\}_{\mathsf{B}}) \xrightarrow{\hat{\mu}=\mathsf{K}:\,\mathtt{fuse}}_\star (s)(\alpha^\star(S''\sigma) \ \mid\ s[\alpha^\star(\gamma)]) = \hat{S}'}$$

By Definition 2.3 and Lemma 2.6 (LHS of the double implication), we have $c \bowtie d$. Hence, by Lemma B.7:

$$\frac{c \bowtie d \qquad \gamma = \mathsf{A} : c \mid \mathsf{B} : d \qquad \sigma = \{s/x,y\} \qquad s \text{ fresh}}{S \xrightarrow{\mathsf{K}:\,\mathtt{fuse}} (s)(S''\sigma \ \mid\ s[\gamma]) = S'}$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[Do] We have:

$$\frac{\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:a}\!\!\!\!\twoheadrightarrow_\star \hat{\gamma}'}{\alpha^\star(S) = \mathsf{A}[\mathtt{do}_s\, a.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \ \mid\ s[\alpha^\star(\gamma)] \xrightarrow{\hat{\mu}=\mathsf{A}:\,\mathtt{do}_s\, a}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \ \mid\ s[\hat{\gamma}']}$$

There are two cases, according to the form of a:

- $a = (\mathtt{a}, \mathtt{T})!$. Then S has the form $\mathsf{A}[\mathtt{do}_s\, \mathtt{a}!e.P + P' \mid Q] \mid s[\gamma]$, with $e : \mathtt{T}$ and $[\![e]\!] = \mathtt{v}$. By item 2 of Lemma A.5, there exists γ' such that $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}!\mathtt{v}}\!\!\!\!\twoheadrightarrow \gamma'$ and $\alpha^\star(\gamma') = \hat{\gamma}'$. Then, by rule [Do!]:

$$\frac{[\![e]\!] = \mathtt{v} \qquad \gamma \xrightarrow{\mathsf{A}:\mathtt{a}!\mathtt{v}}\!\!\!\!\twoheadrightarrow \gamma'}{S = \mathsf{A}[\mathtt{do}_s\, \mathtt{a}!e.P + P' \mid Q] \ \mid\ s[\gamma] \xrightarrow{\mu=\mathsf{A}:\mathtt{do}_s\,\mathtt{a}!\mathtt{v}} \mathsf{A}[P \mid Q] \ \mid\ s[\gamma'] = S'}$$

  from which the thesis follows, because $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.
- $a = (\mathtt{a}, \mathtt{T})?$. Here S has the form $\mathsf{A}[\mathtt{do}_s\, \mathtt{a}?x : \mathtt{T}.P + P' \mid Q] \mid s[\gamma]$ (recall that S is assumed to be well typed). By item 2 of Lemma A.5, there exists γ' such that $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}?\mathtt{v}}\!\!\!\!\twoheadrightarrow \gamma'$ and $\alpha^\star(\gamma') = \hat{\gamma}'$. Then, by rule [Do?]:

$$\frac{\gamma \xrightarrow{\mathsf{A}:\mathtt{a}?\mathtt{v}}\!\!\!\!\twoheadrightarrow \gamma' \qquad \mathtt{v} : \mathtt{T}}{S \ = \ \mathsf{A}[\mathtt{do}_s\, \mathtt{a}?x : \mathtt{T}.\, P + P' \mid Q] \ \mid\ s[\gamma] \xrightarrow{\mu=\mathsf{A}:\mathtt{do}_s\,\mathtt{a}?\mathtt{v}} \mathsf{A}[P\{\mathtt{v}/x\} \mid Q] \ \mid\ s[\gamma'] \ = \ S'}$$

  Hence we have $\alpha^\star(\mu) = \hat{\mu}$, and by lemma B.7, we also have $\alpha^\star(S') = \hat{S}'$.

[Ask] Assume $S = \mathsf{A}[\mathtt{ask}_s\, \phi.P + P' \mid Q] \mid s[\gamma]$. We have:

$$\frac{\alpha^\star(\gamma) \vdash \phi}{\alpha^\star(S) = \mathsf{A}[\mathtt{ask}_s\, \phi.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \ \mid\ s[\alpha^\star(\gamma)] \xrightarrow{\hat{\mu}=\mathsf{A}:\,\mathtt{ask}_s\,\phi}_\star \mathsf{A}[\alpha^\star(P) \mid \alpha^\star(Q)] \ \mid\ s[\alpha^\star(\gamma)] = \hat{S}'}$$

By Corollary B.5(RHS of the double implication) we have:

$$\frac{\gamma \vdash \phi}{S \xrightarrow{\mu=\mathsf{A}:\,\mathtt{ask}_s\,\phi} \mathsf{A}[P \mid Q] \ \mid\ s[\gamma] = S'}$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[Def] Assume $S = \mathsf{A}[\mathrm{X}(\boldsymbol{u}, \boldsymbol{e}) \mid Q] \mid S''$. By applying the induction hypothesis, we have that $\exists \mu, S'$ such that $S''' = \mathsf{A}[P\{\boldsymbol{u}/\boldsymbol{x}\}\{\boldsymbol{e}/\boldsymbol{y}\} \mid Q] \mid S'' \xrightarrow{\mu} S'$ and $\alpha^\star(S''') \xrightarrow{\hat{\mu}}_\star \hat{S}' = \alpha^\star(S')$, with $\hat{\mu} = \alpha^\star(\mu)$. Therefore, we have:

$$\frac{\mathrm{X}(\boldsymbol{u}, \star) \stackrel{\text{def}}{=} \alpha^\star(P) \qquad \alpha^\star(S''') = \mathsf{A}[\alpha^\star(P)\{\boldsymbol{v}/\boldsymbol{u}\} \mid \alpha^\star(Q)] \mid \alpha^\star(S'') \xrightarrow{\hat{\mu}}_\star \hat{S}' = \alpha^\star(S')}{\alpha^\star(S) = \mathsf{A}[\mathrm{X}(\boldsymbol{v}, \star) \mid \alpha^\star(Q)] \mid \alpha^\star(S'') \xrightarrow{\hat{\mu}}_\star \hat{S}'}$$

Then, by Lemma B.7:

$$\frac{\mathsf{X}(\boldsymbol{x},\boldsymbol{y}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{{}^{u}/_{x}\}\{{}^{e}/_{y}\} \mid Q] \mid S'' \xrightarrow{\mu} S'}{S \xrightarrow{\mu} S'}$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[Par] Assume $S = S_0 \mid S''$. By applying the induction hypothesis, we have that $\exists \mu, S'_0$ such that $S_0 \xrightarrow{\mu} S'_0$ and $\alpha^\star(S_0) \xrightarrow{\hat{\mu}} \alpha^\star(S'_0)$, with $\hat{\mu} = \alpha^\star(\mu)$. Therefore, we have:

$$\frac{\alpha^\star(S_0) \xrightarrow{\hat{\mu}} \alpha^\star(S'_0)}{\alpha^\star(S) = \alpha^\star(S_0) \mid \alpha^\star(S'') \xrightarrow{\hat{\mu}} \alpha^\star(S'_0) \mid \alpha^\star(S'') = \hat{S}'}$$

Then:

$$\frac{S_0 \xrightarrow{\mu} S'_0}{S \xrightarrow{\mu} S'_0 \mid S'' = S'}$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

[Del] Assume $S = (u)S_0$. By applying the induction hypothesis and Lemma B.8, we have that $\exists \pi, S'_0$ such that $S_0 \xrightarrow{\mathsf{A}:\,\pi} S'_0$ and $\alpha^\star(S_0) \xrightarrow{\mathsf{A}:\,\hat{\pi}}_\star \alpha^\star(S'_0)$, with $\hat{\pi} = \alpha^\star(\pi)$. Therefore, we have:

$$\frac{\alpha^\star(S_0) \xrightarrow{\hat{\mu}=\mathsf{A}:\,\hat{\pi}}_\star \alpha^\star(S'_0)}{\alpha^\star(S) = (u)\alpha^\star(S_0) \xrightarrow{\mathsf{A}:\,\mathrm{del}_u(\hat{\pi})}_\star (u)\alpha^\star(S'_0) = \hat{S}'}$$

Then:

$$\frac{S_0 \xrightarrow{\mu=\mathsf{A}:\,\pi} S'_0}{S \xrightarrow{\mathsf{A}:\,\mathrm{del}_u(\pi)} (u)S'_0 = S'}$$

and therefore $\alpha^\star(\mu) = \hat{\mu}$ and $\alpha^\star(S') = \hat{S}'$.

Lemma B.11. $\alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \implies \exists S' : S \xrightarrow{\mathsf{A}:\mathtt{if}} S' \wedge \alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \alpha^\star(S')$.

Proof. Suppose $\alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star$. We proceed by rule induction.

[If] Assume:

$$\mathsf{A}[(\mathtt{if}\ \star\ \mathtt{then}\ \alpha^\star(P_0)\ \mathtt{else}\ \alpha^\star(P_1)) \mid \alpha^\star(Q)] \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star$$

Then, we necessarily have $S = (\mathtt{if}\ e\ \mathtt{then}\ P_0\ \mathtt{else}\ P_1) \mid Q$ for some e, and either:

$$\mathsf{A}[S] \xrightarrow{\mathsf{A}:\,\mathtt{if}} \mathsf{A}[P_0] = S' \qquad \text{or} \qquad \mathsf{A}[S] \xrightarrow{\mathsf{A}:\,\mathtt{if}} \mathsf{A}[P_1] = S'$$

In both cases, we have $\alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \alpha^\star(S')$.

[Par] Assume $S = S'' \mid S'''$, and:

$$\frac{\alpha^\star(S'') \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star}{\alpha^\star(S) = \alpha^\star(S'' \mid S''') = \alpha^\star(S'') \mid \alpha^\star(S''') \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star}$$

Then, by applying the induction hypothesis, there exists S''_1 such that $S'' \xrightarrow{\mathsf{A}:\,\mathtt{if}} S''_1$ and $\alpha^\star(S'') \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \alpha^\star(S''_1)$. Hence:

$$\frac{S'' \xrightarrow{\mathsf{A}:\,\mathtt{if}} S''_1}{S = S'' \mid S''' \xrightarrow{\mathsf{A}:\,\mathtt{if}} S''_1 \mid S''' = S'}$$

and we have $\alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \alpha^\star(S')$.

The cases for rule [Def] and [Del] are similar to [Par], using, respectively, Lemma B.8 and Lemma B.7.

Lemma B.12. *For all systems S, participant A and session names s:*

$$\mathsf{A} \textit{ ready at } s \textit{ in } \alpha^\star(S) \implies \mathsf{A} \textit{ ready at } s \textit{ in } S$$

Proof. Suppose A is ready at s in $\alpha^\star(S)$. We proceed by induction on the item/rule of Definition 4.2 which guarantees $S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$:

- item 1. By Lemma B.10, item 1 of Definition 3.5 follows trivially.
- item 2. Then $\alpha^\star(S) \xrightarrow{\mathsf{A}:\hat{\pi}}_\star \hat{S}'$, with $\hat{\pi} \notin \{\mathtt{do}_s, \mathtt{if}\}$, and A is ready at s in $\hat{S}'$.

  By Lemma B.10, it follows that $S \xrightarrow{\mathsf{A}:\pi} S'$, with $\alpha^\star(\mathsf{A} : \pi) = \mathsf{A} : \hat{\pi}$ and $\alpha^\star(S') = \hat{S}'$. By applying the induction hypothesis, A is ready at s in S'. Hence, item 2 of Definition 3.5 holds for S.
- item 3. By Lemma B.11 $S \xrightarrow{\mathsf{A}:\mathtt{if}} S'$ — because of an underlying $\mathtt{if}$ prefix chosing one of its branches. By item 3 of Definition 4.2 it follows A that is ready at s in $\alpha^\star(S')$ (actually, A is ready in *both* branches of the abstracted $\mathtt{if}$). By applying the induction hypothesis, A is ready at s in S', and hence item 2 of Definition 3.5 holds for S.

Lemma B.13. *For all $\mathtt{if}$-free systems S, participant A and session names s:*

$$\mathsf{A} \textit{ ready at } s \textit{ in } S \implies \mathsf{A} \textit{ ready at } s \textit{ in } \alpha^\star(S)$$

Proof. Direct consequence of Lemma B.9.

Lemma B.14. *Let S be a concrete system and $\hat{S}$ be the value-abstract system such that $\alpha^\star(\hat{S}) = S$. Then, if $\hat{S}$ is $\alpha^\star$−ready then S is ready. Conversely, if S is ready and $\mathtt{if}$-free then $\hat{S}$ is $\alpha^\star$−ready.*

Proof. Direct consequence of Lemma B.12 and Lemma B.13.
Theorem 4.3. *Let P be a concrete process. If $\alpha^\star(P)$ is $\alpha^\star$-honest, then P is honest. Conversely, if P is honest and $\mathtt{if}$-free, then $\alpha^\star(P)$ is $\alpha^\star$-honest.*
Proof. Direct conseguence of Lemma B.9, Lemma B.10 and Lemma B.14

Remark B.15 (On "safe" execution contexts). *When defining honesty, we consider contexts that cannot delimit the free variables of a participant*[1]*. This is necessary in order to rule out tricky cases in which a context with properly crafted delimitations would make a participant trivially dishonest. A notion of "safe" handling of delimitation, called* A-safety*, is formalised in Definition B.16 below.*

We now give an intuition about the problems at hand. Consider, for instance, the process $P = \mathtt{tell} \downarrow_x c$, where $c = \mathtt{a!}\,.\,0$.

P above is trivially honest: in fact, it advertises the contract c using a free variable x; and since rule [Fuse] *requires x to be delimited, c will never be stipulated if $\mathsf{A}[P]$ is composed in parallel with some A-free system, as per Definition 3.6.*

However, if $\mathsf{A}[P]$ is put in a context which delimits x, then c may indeed be stipulated. Consider, for instance, the system $S = (x)(\mathsf{A}[P] \mid \mathsf{B}[\mathtt{tell} \downarrow_x d])$, where $d = \mathtt{a?}\,.\,0$: we have $S \to^* (s)(\mathsf{A}[\mathbf{0}] \mid \mathsf{B}[\mathbf{0}] \mid s[\mathsf{A} : c \mid \mathsf{B} : d]) = S'$, and hence A is not ready in S'.

We can rule out these situations by only focusing on A-safe systems, which intuitively can be split in two parts:

[1] Almost equivalently, we could limit our treatment to *closed* participants (i.e., participants whose processes have no free variables).

1. A-solo *system* S_{A} *containing the process of* A*, the contracts advertised by* A *and all the sessions containing contracts of* A*;*

2. *an* A-free *system* S_{ctx}*.*

For instance, we have that the system S above is not A*-safe: since the delimitation (x) includes both* A*'s and* B*'s processes, then S cannot be rewritten (via congruence rules) in the form $(\boldsymbol{s})(S_{\mathsf{A}} \mid S_{ctx})$ with S_{A}* A*-solo and S_{ctx}* A*-free.*

In the following lemmata we will show that the A*-safety property is preserved both by abstractions and reductions.*

Definition B.16 (A-solo and A-safe systems). *We say that S is* A-solo *iff one of the following holds:*

$$S \equiv \mathbf{0} \qquad S \equiv \mathsf{A}[P] \qquad S \equiv s[\mathsf{A} : c \mid \mathsf{B} : d] \qquad S \equiv \{\downarrow_x c\}_{\mathsf{A}}$$
$$S \equiv S' \mid S'' \quad \text{where } S' \text{ and } S'' \text{ A-solo} \qquad S \equiv (u)S' \quad \text{where } S' \text{ A-solo}$$

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

Remark B.17. *We naturally extend the* A*-solo/safe/free definitions to value- and context-abstract systems.*

Proposition B.18 (A-safety abstraction). *Let S be* A*-safe. Then, both $\alpha^{\star}(S)$ and $\alpha_{\mathsf{A}}(S)$ are* A*-safe.*

Proof. Trivial.

In Lemma B.19 below we show that A-safety is preserved by transitions. Consequently, since the initial contexts where honesty is defined are themselves A-safe, w.l.o.g. we can consider A-safe systems only.

Lemma B.19. *For all systems S, S' such that $S \rightarrow_{\star} S'$:*

(1) If S is A*-solo, then S' is* A*-solo.*

(2) If S is A*-free, then S' is* A*-free.*

(3) If S is A*-safe, then S' is* A*-safe.*

Proof. Items (1) and (2) are straightforward by Definition B.16 and by induction on the semantic rules of value-abstract systems.

For item (3), we proceed by induction on the depth of the derivation of $S \rightarrow_{\star} S'$. Let S be A-safe. Then, by Definition B.16, $S \equiv (\boldsymbol{s})(S_{\mathsf{A}} \mid S_{ctx})$, for some A-solo S_{A} and A-free S_{ctx}.

By repeatedly applying rule [Del] backwards, we can remove the top-level delimited sessions and obtain:

$$\dfrac{\dfrac{\dfrac{\vdots}{S_{\mathsf{A}} \mid S_{ctx} \rightarrow_{\star} S'_0}}{\vdots}\;\text{[Del]}}{S \equiv (\boldsymbol{s})(S_{\mathsf{A}} \mid S_{ctx}) \rightarrow_{\star} (\boldsymbol{s})S'_0 = S'}\;\text{[Del]}$$

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

[Do] There are the following three subcases.

- The `do` is fired by A. Then, $S'_0 = S'_{\mathsf{A}} \mid S_{ctx}$. By item (1), S'_{A} is A-solo, hence S' is A-safe.
- The `do` is fired by some $\mathsf{B} \neq \mathsf{A}$, in a session not involving A. Then, $S'_0 = S_{\mathsf{A}} \mid S'_{ctx}$. By item (2), S'_{ctx} is A-free, hence S' is A-safe.

- The $\mathtt{do}_s\, a$ is fired by some $\mathsf{B} \neq \mathsf{A}$, in a session s involving A. We have that $S_{\mathsf{A}} \equiv S''_{\mathsf{A}} \mid s[\gamma]$, with $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, and $S_{ctx} = \mathsf{B}[\mathtt{do}_s\, a.P + Q \mid R] \mid S'_{ctx}$. Then, by rule [Do]:

$$\frac{\gamma \xrightarrow{\mathsf{B}:a}\!\!\twoheadrightarrow_\star \gamma'}{S_{\mathsf{A}} \mid S_{ctx} \rightarrow_\star S''_{\mathsf{A}} \mid s[\gamma'] \mid \mathsf{B}[P \mid R] \mid S'_{ctx}}$$

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

[Ask] Similar to the case [Do].

[Fuse] We have $S_{\mathsf{A}} \equiv (x)(S'_{\mathsf{A}} \mid \{\downarrow_x c\}_{\mathsf{A}})$, $S_{ctx} \equiv (y)(S'_{ctx} \mid \{\downarrow_y d\}_{\mathsf{B}})$, and:

$$\frac{c \bowtie d \qquad \gamma = \mathsf{A} : c \mid \mathsf{B} : d \qquad \sigma = \{^s/_{x,y}\} \qquad s \text{ fresh}}{S_{\mathsf{A}} \mid S_{ctx} \equiv (x,y)(S'_{\mathsf{A}} \mid \{\downarrow_x c\}_{\mathsf{A}} \mid S'_{ctx} \mid \{\downarrow_y d\}_{\mathsf{B}}) \rightarrow_\star (s)(S'_{\mathsf{A}}\sigma \mid s[\gamma] \mid S'_{ctx}\sigma) = S'_0}$$

We have that $S'_{\mathsf{A}}\sigma \mid s[\gamma]$ is A-solo, and $S'_{ctx}\sigma$ is A-free, then by Definition B.16, S'_0 is A-safe. Therefore, $S' = (\boldsymbol{s})S'_0$ is A-safe as well.

[Par] We have the following two subcases.

- $S_{\mathsf{A}} \rightarrow_\star S'_{\mathsf{A}}$, and $S_0 = S'_{\mathsf{A}} \mid S_{ctx}$. By item (1), S'_{A} is A-solo, hence S' is A-safe.
- $S_{ctx} \rightarrow_\star S'_{ctx}$. By item (2), S'_{ctx} is A-free, hence S' is A-safe.

[Def] We have two subcases, according to which participant has moved.

A **has moved.** We have that $S_{\mathsf{A}} = \mathsf{A}[\mathrm{X}(\boldsymbol{v}) \mid Q] \mid S'_{\mathsf{A}}$, for some A-solo S'_{A}, and:

$$\frac{\mathrm{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{^{\boldsymbol{v}}/_{\boldsymbol{u}}\} \mid Q] \mid S'_{\mathsf{A}} \mid S_{ctx} \xrightarrow{\mu}_\star S'_0}{S_{\mathsf{A}} \mid S_{ctx} \xrightarrow{\mu}_\star S'_0}\ \text{[Def]}$$

Since $\mathsf{A}[P\{^{\boldsymbol{v}}/_{\boldsymbol{u}}\} \mid Q] \mid S'_{\mathsf{A}}$ is A-solo, then by applying the induction hypothesis it follows that S'_0 is A-safe. Therefore, $S' = (\boldsymbol{s})S'_0$ is A-safe as well.

$\mathsf{B} \neq \mathsf{A}$ **has moved.** Similar to the previous case, but considering that $\mathsf{A}[P\{^{\boldsymbol{v}}/_{\boldsymbol{u}}\} \mid Q] \mid S'_{\mathsf{A}}$ remains unchanged (and A-solo), while (by item (2)) the context reduces remaining A-free.

B.2. Proofs for Section 4.2

We shall sometimes use metavariables $\tilde{c}, \tilde{d}, \ldots$ to range over context-abstract contracts. We will drop the decoration when there is no ambiguity; similar notational conventions apply for context-abstract systems.

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

Theorem 4.5. *For all value-abstract contract configurations $\hat{\gamma}$, $\hat{\gamma}'$:*

(1) $\gamma \xrightarrow{\mathsf{A}:a}\!\!\twoheadrightarrow_\star \gamma' \implies \alpha_{\mathsf{A}}(\gamma) \xrightarrow{a}\!\!\twoheadrightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\gamma')$

(2) $\gamma \xrightarrow{\mathsf{B}:a}\!\!\twoheadrightarrow_\star \gamma' \implies \alpha_{\mathsf{A}}(\gamma) \xrightarrow{ctx:a}\!\!\twoheadrightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\gamma') \qquad (\mathsf{B} \neq \mathsf{A})$

(3) $c \xrightarrow{a}\!\!\twoheadrightarrow_{\mathsf{A}} c' \implies \forall$ *compliant* $\gamma_c : \big(\alpha_{\mathsf{A}}(\gamma_c) = c \implies \exists \gamma_{c'} : \gamma_c \xrightarrow{\mathsf{A}:a}\!\!\twoheadrightarrow_\star \gamma_{c'} \wedge \alpha_{\mathsf{A}}(\gamma_{c'}) = c'\big)$

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

[AbsIntExt] By the semantics of concrete and abstract contracts and by Definition 4.4, we have:

$$\begin{array}{llll} \hat{\gamma} = \mathsf{A} : a \,.\, \hat{c} \oplus \hat{c}' \mid \mathsf{B} : \mathrm{co}(a) \,.\, \hat{d} + \hat{d}' & \xrightarrow{\mathsf{A}:a}\!\!\twoheadrightarrow_\star & \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ \mathrm{co}(a).\hat{d} & = \hat{\gamma}' \\ \alpha_{\mathsf{A}}(\hat{\gamma}) = a \,.\, \hat{c} \oplus \hat{c}' & \xrightarrow{a}\!\!\twoheadrightarrow_{\mathsf{A}} & ctx\ \mathrm{co}(a)\,.\hat{c} & = \alpha_{\mathsf{A}}(\hat{\gamma}') \end{array}$$

[AbsRdy] Since $\hat{d}$ is ready-free, we have:

$$\begin{array}{rlll} \hat{\gamma} = \mathsf{A} : \mathtt{rdy}\ a.\hat{c} \mid \mathsf{B} : \hat{d} & \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star & \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d} & = \hat{\gamma}' \\ \alpha_{\mathsf{A}}(\hat{\gamma}) = \mathtt{rdy}\ a.\hat{c} & \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} & \hat{c} & = \alpha_{\mathsf{A}}(\hat{\gamma}') \end{array}$$

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

[AbsIntExt] We have:

$$\begin{array}{rlll} \hat{\gamma} = \mathsf{A} : (\mathrm{co}(a)\ .\ \hat{c} + \hat{c}') \mid \mathsf{B} : (a\ .\ \hat{d} \oplus \hat{d}') & \xrightarrow{\mathsf{B}:a}\!\!\!\!\to_\star & \mathsf{A} : \mathtt{rdy}\ \mathrm{co}(a).c \mid \mathsf{B} : d & = \hat{\gamma}' \\ \alpha_{\mathsf{A}}(\hat{\gamma}) = \mathrm{co}(a)\ .\ \hat{c} + \hat{c}' & \xrightarrow{ctx:a}\!\!\!\!\to_{\mathsf{A}} & \mathtt{rdy}\ \mathrm{co}(a).\hat{c} & = \alpha_{\mathsf{A}}(\hat{\gamma}') \end{array}$$

[AbsRdy] We have:

$$\begin{array}{rlll} \hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ a.\hat{d} & \xrightarrow{\mathsf{B}:a}\!\!\!\!\to_\star & \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d} & = \hat{\gamma}' \\ \alpha_{\mathsf{A}}(\hat{\gamma}) = ctx\ a.\hat{c} & \xrightarrow{ctx:a}\!\!\!\!\to_{\mathsf{A}} & \hat{d} & = \alpha_{\mathsf{A}}(\hat{\gamma}') \end{array}$$

For item 3, we have two cases:

- For $\tilde{c} = a\ .\ \hat{c}' \oplus \hat{c}''$ we have that $\tilde{c} \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} ctx\ \mathrm{co}(a)\ .\hat{c}' = \tilde{c}'$. Let $\hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d}$, with $\hat{c} = \tilde{c}$, $\hat{d} = \mathrm{co}(a)\ .\ \hat{d}' + \hat{d}''$. Clearly $\hat{\gamma} \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star \mathsf{A} : \hat{c}' \mid \mathsf{B} : \mathtt{rdy}\ \mathrm{co}(a).\hat{d}' = \hat{\gamma}'$. Of course $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$ and $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$.
- For $\tilde{c} = \mathtt{rdy}\ a.\hat{c}'$ we have that $\tilde{c} \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} \hat{c}' = \tilde{c}'$. Let $\hat{\gamma} = \mathsf{A} : \tilde{c} \mid \mathsf{B} : \hat{d}$, with $\hat{d}$ ready-free and $\hat{d} \bowtie \hat{c}'$. $\hat{\gamma} \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star \mathsf{A} : \hat{c}' \mid \mathsf{B} : \hat{d} = \hat{\gamma}'$. Clearly $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$ and $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$.

Definition B.20. *Let γ_{B} be a function defined as follows:*

$$\begin{array}{rcll} \gamma_{\mathsf{B}}(ctx\ a.\hat{c}) & = & \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ a.\mathrm{dual}(\hat{c}) & \\ \gamma_{\mathsf{B}}(\mathtt{rdy}\ a.\hat{c}) & = & \mathsf{A} : \mathtt{rdy}\ a.\hat{c} \mid \mathsf{B} : \mathrm{dual}(\hat{c}) & \\ \gamma_{\mathsf{B}}(\hat{c}) & = & \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathrm{dual}(\hat{c}) & \textit{otherwise} \end{array}$$

Lemma B.21. *For all abstract contracts $\tilde{c}$:*

(1) $\tilde{c} \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} \tilde{c}' \implies \gamma_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star \gamma_{\mathsf{B}}(\tilde{c}')$

(2) $\tilde{c} \xrightarrow{ctx:a}\!\!\!\!\to_{\mathsf{A}} \tilde{c}' \implies \gamma_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{B}:a}\!\!\!\!\to_\star \gamma_{\mathsf{B}}(\tilde{c}')$

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

Proof.

(1) There are two cases:

- $\tilde{c} = a\ .\ \hat{c} \oplus \hat{c}' \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} ctx\ \mathrm{co}(a)\ .\ \hat{c} = \tilde{c}'$
  $\gamma_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : a\ .\ \hat{c} \oplus \hat{c}' \mid \mathsf{B} : \mathrm{co}(a)\ .\ \mathrm{dual}(\hat{c}) + \mathrm{dual}(\hat{c}')$
  $\gamma_{\mathsf{B}}(\tilde{c}') = \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ \mathrm{co}(a)\ .\ \mathrm{dual}(\hat{c})$
  Clearly: $\gamma_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star \gamma_{\mathsf{B}}(\tilde{c}')$
- $\tilde{c} = \mathtt{rdy}\ a\ .\ \hat{c}' \xrightarrow{a}\!\!\!\!\to_{\mathsf{A}} \hat{c}' = \tilde{c}'$
  $\gamma_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \mathtt{rdy}\ a\ .\ \hat{c}' \mid \mathsf{B} : \mathrm{dual}(\hat{c}') \xrightarrow{\mathsf{A}:a}\!\!\!\!\to_\star \mathsf{A} : \hat{c}' \mid \mathsf{B} : \mathrm{dual}(\hat{c}') = \gamma_{\mathsf{B}}(\tilde{c}')$

(2) There are two cases:

- $\tilde{c} = \text{co}(a) \,.\, \hat{c} + \hat{c}' \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \texttt{rdy}\ \text{co}(a) \,.\, \hat{c} = \tilde{c}'$
  $\gamma_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \text{co}(a) \,.\, \hat{c} + \hat{c}' \mid \mathsf{B} : a \,.\, \text{dual}(\hat{c}) \oplus \text{dual}(\hat{c}')$
  $\gamma_{\mathsf{B}}(\tilde{c}') = \mathsf{A} : \texttt{rdy}\ \text{co}(a) \,.\hat{c} \mid \mathsf{B} : \text{dual}(\hat{c})$
  Clearly: $\gamma_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_{\star} \gamma_{\mathsf{B}}(\tilde{c}')$
- $\tilde{c} = ctx\ a \,.\, \hat{c} \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \hat{c} = \tilde{c}'$
  $\gamma_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \hat{c} \mid \mathsf{B} : \texttt{rdy}\ a \,.\, \text{dual}(\hat{c}) \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_{\star} \mathsf{A} : \hat{c} \mid \mathsf{B} : \text{dual}(\hat{c}) = \gamma_{\mathsf{B}}(\tilde{c}')$

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

- $\alpha_{\mathsf{A}}(\gamma_{\mathsf{B}}(ctx\ a.\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : c \mid \mathsf{B} : \texttt{rdy}\ a.\text{dual}(\hat{c})) = ctx\ a.\hat{c} = \tilde{c}$
- $\alpha_{\mathsf{A}}(\gamma_{\mathsf{B}}(\texttt{rdy}\ a.\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : \texttt{rdy}\ a.\hat{c} \mid \mathsf{B} : \text{dual}(\hat{c})) = \texttt{rdy}\ a.\hat{c} = \tilde{c}$
- $\alpha_{\mathsf{A}}(\gamma_{\mathsf{B}}(\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : \hat{c} \mid \mathsf{B} : \text{dual}(\hat{c})) = \hat{c} = \tilde{c}$, with $\hat{c}$ $\texttt{rdy}$-free.

Lemma B.22. *For all abstract contracts $\tilde{c}$:*

$$\tilde{c} \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}' \implies \exists \hat{\gamma}, \hat{\gamma}' \,.\, \hat{\gamma} \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_{\star} \hat{\gamma}' \ \wedge\ \alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c} \ \wedge\ \alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$$

Proof. We already know, by items (2) and (3) of Lemma B.21, the thesis holds, and in particular $\gamma = \gamma_{\mathsf{B}}(\tilde{c})$.

B.3. Proofs for Section 4.3

Lemma B.23. *For all substitutions σ and value-abstract systems $\hat{S}$ $(\alpha_{\mathsf{A}}(\hat{S}))\sigma = \alpha_{\mathsf{A}}(\hat{S}\sigma)$.*

Proof. Straightforward structural induction.

Lemma B.24. *For all value-abstract systems $\hat{S}$:* $\text{fv}(\alpha_{\mathsf{A}}(\hat{S})) \cup \text{fn}(\alpha_{\mathsf{A}}(\hat{S})) \subseteq \text{fv}(\hat{S}) \cup \text{fn}(\hat{S})$

Proof. We proceed by induction on the structure of $\hat{S}$. There are the following cases:

- $\alpha_{\mathsf{A}}(\hat{S}) = \mathbf{0}$. The thesis holds trivially.
- $\mathsf{A}[\tilde{P}]$. Obvious, by the fact that $\alpha_{\mathsf{A}}(\mathsf{A}[\tilde{P}]) = \mathsf{A}[\tilde{P}]$.
- $s[\mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d}]$. We have $\alpha_{\mathsf{A}}(\hat{S}) = s[\hat{c}]$. Clearly u not free in $\hat{S}$ only if $u \neq s$. But this holds also for $\alpha_{\mathsf{A}}(\hat{S})$.
- $\{\downarrow_x \hat{c}\}_{\mathsf{A}}$. We have $\alpha_{\mathsf{A}}(\hat{S}) = \{\downarrow_x \hat{c}\}$. Clearly u is not free in $\hat{S}$ only if $x \neq s$. But this holds also for $\alpha_{\mathsf{A}}(\hat{S})$.
- $\hat{S}' \mid \hat{S}''$. We have $\alpha_{\mathsf{A}}(\hat{S}) = \alpha_{\mathsf{A}}(\hat{S}') \mid \alpha_{\mathsf{A}}(\hat{S}'')$. By induction hypothesis we have that (u not free in $\hat{S}' \implies u$ not free in $\alpha_{\mathsf{A}}(\hat{S}')) \wedge (u$ not free in $\hat{S}'' \implies u$ not free in $\alpha_{\mathsf{A}}(\hat{S}''))$. Assume that u is not free in $\hat{S}$, so u must be not free in $\hat{S}'$ and in $\hat{S}''$, but then, by induction hypotesis, we have that u is not free in $\alpha_{\mathsf{A}}(\hat{S}')$ and in $\alpha_{\mathsf{A}}(\hat{S}'')$, which implies u not free in $\alpha_{\mathsf{A}}(\hat{S})$.
- $(u')\hat{S}'$. We have $\alpha_{\mathsf{A}}(\hat{S}) = (u')(\alpha_{\mathsf{A}}(\hat{S}'))$. Under the assumption that u is not free in $\hat{S}$, we have that $u = u'$ or u is not free in $\hat{S}'$. If the first holds we are done. If u is not free in $\hat{S}'$ the thesis follows by induction hypothesis.

Lemma B.25. *Let $\Gamma_{\tilde{c}} = \{\hat{\gamma} \mid \alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}\}$. Then, for all $a \in \mathbf{A} \cup \{\varepsilon\}$, and for all $\tilde{c}$:*

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

$$\mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\tau}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \qquad [\alpha\text{-Tau}]$$

$$\mathsf{A}[(\texttt{if} \star \texttt{then}\ \tilde{P}_0\ \texttt{else}\ \tilde{P}_1) \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{if}}_{\mathsf{A}} \mathsf{A}[\tilde{P}_i \mid \tilde{Q}] \quad (i \in \{0,1\}) \qquad [\alpha\text{-If}]$$

$$\mathsf{A}[\texttt{tell}\downarrow_x \tilde{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{tell}\downarrow_x\tilde{c}}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \tilde{c}\}_{\mathsf{A}} \qquad [\alpha\text{-Tell}]$$

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

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

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

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

$$\frac{\mathrm{X}(\boldsymbol{x}) \stackrel{\text{def}}{=} \tilde{P} \qquad \mathsf{A}[\tilde{P}\{\boldsymbol{u}/\boldsymbol{x}\} \mid \tilde{Q}] \mid \tilde{S} \xrightarrow{\mu}_{\mathsf{A}} \tilde{S}'}{\mathsf{A}[\mathrm{X}(\boldsymbol{u}) \mid \tilde{Q}] \mid \tilde{S} \xrightarrow{\mu}_{\mathsf{A}} \tilde{S}'} \ [\alpha\text{-Def}] \qquad \frac{\tilde{S} \xrightarrow{\mu}_{\mathsf{A}} \tilde{S}'}{\tilde{S} \mid \tilde{S}'' \xrightarrow{\mu}_{\mathsf{A}} \tilde{S}' \mid \tilde{S}''} \ [\alpha\text{-Par}]$$

$$\frac{\tilde{S} \xrightarrow{\mathsf{A}:\pi}_{\mathsf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{\mathsf{A}:\mathrm{del}_u(\pi)}_{\mathsf{A}} (u)\tilde{S}'} \ [\alpha\text{-Del}] \qquad \frac{\tilde{S} \xrightarrow{ctx}_{\mathsf{A}} \tilde{S}'}{(u)\tilde{S} \xrightarrow{ctx}_{\mathsf{A}} (u)\tilde{S}'} \ [\alpha\text{-DelCtx}]$$

$$\frac{\tilde{c} \xrightarrow{ctx}\!\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}'}{s[\tilde{c}] \xrightarrow{ctx}_{\mathsf{A}} s[\tilde{c}']} \ [\alpha\text{-DoCtx}] \qquad \tilde{S} \xrightarrow{ctx}_{\mathsf{A}} \tilde{S} \ [\alpha\text{-Ctx}]$$

Figure 9: Reduction semantics of context-abstract systems (full set of rules).

Proof. $\supseteq$: Suppose that:

$$a_0 a_1 \ldots \in \bigcup_{\hat{\gamma} \in \Gamma_{\tilde{c}}} \mathrm{Paths}((a, \hat{\gamma}))$$

i.e.:

$$\exists \hat{\gamma} \in \Gamma_{\tilde{c}}.\ a_0 a_1 \ldots \in \mathrm{Paths}((a, \hat{\gamma}))$$

Let $s_0 = (a_0, \hat{\gamma}_0)$, with $a_0 = a$ and $\hat{\gamma}_0 = \hat{\gamma}$. By Definition 3.3, there exists a path $s_1, s_2, \ldots$ such that $L(s_i) = a_i$ and $s_{i-1} \to s_i$, for all i. For all i, $\tilde{s}_i = (a_i, \alpha_{\mathsf{A}}(\hat{\gamma}_i))$. $L(s_i) = L(\tilde{s}_i)$. We have to show that $\tilde{s}_i \to \tilde{s}_{i+1}$, i.e., by Definition 4.7, $\alpha_{\mathsf{A}}(\hat{\gamma}_i) \xrightarrow{\tilde{\mu}}\!\!\!\!\rightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma_{i+1}})$ with $\tilde{\mu} = a_{i+1}$ or $\tilde{\mu} = ctx : a_{i+1}$. Since $s_i \to s_{i+1}$ holds by hypothesis, by Definition 3.3 we have that $\hat{\gamma}_i \xrightarrow{\mu}\!\!\!\!\rightarrow \hat{\gamma_{i+1}}$, with $\mu = \mathsf{A} : a_{i+1}$ or $\mu = \mathsf{B} : a_{i+1}$. But then, by Theorem 4.5, the thesis follows. In case of finite paths, the proof above is still valid, except that $s_n \not\to$ for some n, and we have to show that $\tilde{s}_n \not\to$. By Definition 3.3 we have that $\hat{\gamma}_n \not\to$, which is only possible when $\hat{\gamma}_n = \mathsf{A} : \mathsf{0} \mid \mathsf{B} : \mathsf{0}$. But then $\alpha_{\mathsf{A}}(\hat{\gamma}_n) = \mathsf{0}$ and, clearly, $(a, \mathsf{0}) \not\to$.
$\subseteq$: Suppose that $a_0 a_1 \ldots \in \mathrm{Paths}((a, \tilde{c}))$. Let $\tilde{s}_0 = (a, \tilde{c})$. By Definition 3.3, there exists a path $\tilde{s}_1, \tilde{s}_2, \ldots$ such that $L(\tilde{s}_i) = a_i$ and $\tilde{s}_{i-1} \to \tilde{s}_i$ for all i. Let $s_i = (a_i, \gamma_{\mathsf{B}}(\tilde{c}_i))$, for all i. Clearly, $L(s_i) = L(\tilde{s}_i)$. We have to show that, for all $i > 0$, $\gamma_{\mathsf{B}}(\tilde{c}_{i-1}) \xrightarrow{\mu}\!\!\!\!\rightarrow \gamma_{\mathsf{B}}(\tilde{c}_i)$, with $\mu = \mathsf{A} : a_i$ or $\mathsf{B} : a_i$. This follows by Definition 4.7 and items (1) and (2) of Lemma B.21. In case of finite paths, we have to show that $(a_n, \tilde{c}_n) \not\to \implies (a_n, \gamma_{\mathsf{B}}(\tilde{c}_n)) \not\to$. By Definitions 3.3 and 4.7 it holds when $\tilde{c}_n \not\twoheadrightarrow_{\mathsf{A}} \implies \gamma_{\mathsf{B}}(\tilde{c}_n) \not\twoheadrightarrow$. But this follows by items (1) and (2) of Lemma B.21 and Theorem 4.5.

Lemma 4.8. *For all context-abstract contracts c and for all LTL formulae ϕ:*

1. $c \vdash \phi \iff (\forall$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \implies \gamma \vdash_\star \phi)$
2. $c \nvdash \neg\phi \impliedby (\exists$ *compliant* $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \wedge \gamma \vdash_\star \phi)$

Proof.

(1) For the $\implies$ direction, suppose that $\tilde{c} \vdash \phi$. By definition of $\vdash$, it follows that $\forall\lambda \in \mathrm{Paths}(\varepsilon, \tilde{c})\ .\ \lambda \vdash \phi$. Let $\hat{\gamma}$ be such that $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$. Suppose $\lambda \in \mathrm{Paths}(\varepsilon, \hat{\gamma})$. By Lemma B.25 we can conclude that $\lambda \in \mathrm{Paths}(\varepsilon, \tilde{c})$ and then $\lambda \vdash \phi$.

For the $\impliedby$ direction, suppose that $\tilde{c} \nvdash \phi$. By definition of $\vdash$, it follows that $\exists\lambda \in \mathrm{Paths}(\varepsilon, \tilde{c})\ .\ \lambda \nvdash \phi$. By Lemma B.25 we know that $\exists\hat{\gamma}\ .\ \lambda \in \mathrm{Paths}(\varepsilon, \hat{\gamma})$. But then, $\hat{\gamma} \nvdash \phi$.

(2) Assume that $\exists\hat{\gamma}\ .\ \alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c} \wedge \hat{\gamma} \vdash \phi$. Then it also holds that $\exists\hat{\gamma}\ .\ \alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c} \wedge \hat{\gamma} \nvdash \neg\phi$. By item 1 we obtain the thesis. $\square$

Lemma B.26. *For all* A*-safe value-abstract systems $\hat{S}$, whenever $\hat{S} \equiv (x_1, ..., x_n, \boldsymbol{s})(\hat{S}_{\mathsf{A}} \mid \hat{S}_{ctx})$:*

$$x_i \text{ is free in } \hat{S}_{\mathsf{A}} \implies x_i \text{ is not free in } \hat{S}_{ctx}$$

Proof. Suppose, by contradiction, that there exists i such that x_i is free in both $\hat{S}_{\mathsf{A}}$ and $\hat{S}_{ctx}$. Then no rule of Figure 3 allows to put $\hat{S}$ in the form $(\boldsymbol{s})(\hat{S}'_{\mathsf{A}} \mid \hat{S}'_{ctx})$, and so $\hat{S}$ is not A-safe. $\square$

Lemma B.27. *For all value-abstract systems $\hat{S}, \hat{S}'$, and for all labels μ:*

(1) $\hat{S} \xrightarrow{\mu}_\star \hat{S}' \implies \exists\tilde{\mu}\ .\ \alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}')$

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

Proof. By induction on the depth of the derivation of $\hat{S} \xrightarrow{\mu} \hat{S}'$. 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:

- $\hat{S} \equiv \mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\tau}_\star \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \hat{S}'$. Both the theses (1) and (2) follow because $\alpha_{\mathsf{A}}(\hat{S}) = \hat{S}$, $\alpha_{\mathsf{A}}(\hat{S}') = \hat{S}'$, and $\hat{S} \xrightarrow{\mathsf{A}:\tau}_{\mathsf{A}} \hat{S}'$ by rule [α-Tau].
- $\hat{S} \equiv \mathsf{B}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{B}:\tau}_\star \mathsf{B}[\tilde{P} \mid \tilde{Q}] = \hat{S}'$. The thesis (1) follows because, by rule [α-Ctx], $\alpha_{\mathsf{A}}(\hat{S}) = \mathbf{0} \xrightarrow{ctx}_{\mathsf{A}} \mathbf{0} = \alpha_{\mathsf{A}}(\hat{S}')$. The thesis (2) follows trivially.

[If] Similar to [Tau]

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

- $\hat{S} \equiv \mathsf{A}[\mathtt{tell} \downarrow_x \hat{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\,\mathtt{tell} \downarrow_x \hat{c}}_\star \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}} = \hat{S}'$. Both the theses (1) and (2) follow because $\alpha_{\mathsf{A}}(\hat{S}) = \hat{S}$, $\alpha_{\mathsf{A}}(\hat{S}') = \hat{S}'$, and $\hat{S} \xrightarrow{\mathsf{A}:\,\mathtt{tell} \downarrow_x \hat{c}}_{\mathsf{A}} \hat{S}'$ by rule [α-Tell].
- $\hat{S} \equiv \mathsf{B}[\mathtt{tell} \downarrow_x \hat{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{B}:\,\mathtt{tell} \downarrow_x \hat{c}}_\star \mathsf{B}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \hat{c}\}_{\mathsf{B}} = \hat{S}'$. The thesis (1) follows because, by rule [α-Ctx], $\alpha_{\mathsf{A}}(\hat{S}) = \mathbf{0} \xrightarrow{ctx}_{\mathsf{A}} \mathbf{0} = \alpha_{\mathsf{A}}(\hat{S}')$. The thesis (2) follows trivially.

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

- $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{A}[\mathtt{do}_s\, \mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\,\mathtt{do}_s\, \mathsf{a}} s[\hat{\gamma}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \hat{S}'$, where $\hat{\gamma} \xrightarrow{\mathsf{A}:\mathsf{a}}\!\!\!\!\rightarrow \hat{\gamma}'$. By item 1 of Theorem 4.5 we have $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma}')$. Therefore both the theses (1) and (2) follow because $\alpha_{\mathsf{A}}(\hat{S}) = s[\alpha_{\mathsf{A}}(\hat{\gamma})] \mid \mathsf{A}[\mathtt{do}_s\, \mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\,\mathtt{do}_s\, \mathsf{a}}_{\mathsf{A}} s[\alpha_{\mathsf{A}}(\hat{\gamma}')] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{\hat{S}}' = \alpha_{\mathsf{A}}(\hat{S}')$, by rule [α-Do].

- $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{B}[\mathtt{do}_s\, \mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{B}:\, \mathtt{do}_s\, \mathsf{a}} s[\hat{\gamma}'] \mid \mathsf{B}[\tilde{P} \mid \tilde{Q}] = \hat{S}'$, where $\hat{\gamma} \xrightarrow{\mathsf{B}:\mathsf{a}}\!\!\!\!\to \hat{\gamma}'$ and $\hat{\gamma} = \mathsf{B} : \hat{c} \mid \mathsf{B}' : \hat{d}$. The thesis (1) follows because $\alpha_{\mathsf{A}}(\hat{S}) = \mathbf{0} \xrightarrow{ctx}_{\mathsf{A}} \mathbf{0} = \alpha_{\mathsf{A}}(\hat{S}')$, by rule [$\alpha$-Ctx]. The thesis (2) holds trivially.
- $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{B}[\mathtt{do}_s\, \mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{B}:\, \mathtt{do}_s\, \mathsf{a}} s[\hat{\gamma}'] \mid \mathsf{B}[\tilde{P} \mid \tilde{Q}] = \hat{S}'$, where $\hat{\gamma} \xrightarrow{\mathsf{B}:\mathsf{a}}\!\!\!\!\to \hat{\gamma}'$ and $\hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d}$. By item 2 of Theorem 4.5, $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{ctx:\mathsf{a}}\!\!\!\!\to_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma}')$. The thesis (1) follows because $\alpha_{\mathsf{A}}(\hat{S}) = s[\alpha_{\mathsf{A}}(\hat{\gamma})] \xrightarrow{ctx}_{\mathsf{A}} s[\alpha_{\mathsf{A}}(\hat{\gamma}')] = \hat{S}' = \alpha_{\mathsf{A}}(\hat{S}')$, by rule [$\alpha$-DoCtx]. The thesis (2) holds trivially.

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

- $\hat{S} \equiv (u)\hat{S}_0 \xrightarrow{\mathsf{A}:\, \pi} (u)\hat{S}'_0 = \hat{S}'$, with $\hat{S}_0 \xrightarrow{\mathsf{A}:\, \pi'} \hat{S}'_0$ and $\pi = \mathrm{del}_u(\pi')$. By the induction hypothesis, there exists $\tilde{\mu}'$ such that $\alpha_{\mathsf{A}}(\hat{S}_0) \xrightarrow{\tilde{\mu}'}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'_0)$. There are two further subcases, according to the form of the label π:
  - $\pi = \mathtt{do}_s\, \mathsf{a}$. Then it must be $\pi = \pi'$. By the induction hypothesis of item (2), it follows that $\tilde{\mu}' = \mathsf{A} :\, \pi'$. Then, by rule [$\alpha$-Del]:

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

    The thesis (1) follows because $\alpha_{\mathsf{A}}(\hat{S}) = (\boldsymbol{u})\alpha_{\mathsf{A}}(\hat{S}_0)$ and $\alpha_{\mathsf{A}}(\hat{S}') = (\boldsymbol{u})\alpha_{\mathsf{A}}(\hat{S}'_0)$. The thesis (2) follows because $\tilde{\mu} = \mathsf{A} :\, \mathrm{del}_u(\pi') = \mathsf{A} :\, \mathrm{del}_u(\pi) = \mu$.
  - $\mu = \mathsf{A} :\, \pi$ and $\hat{S}$ $\mathtt{ask}$-free. Similar to the previous case.
  - Otherwise, by rule [α-DelCtx]:

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

    The thesis (1) follows because $\alpha_{\mathsf{A}}(\hat{S}) = (\boldsymbol{u})\alpha_{\mathsf{A}}(\hat{S}_0)$ and $\alpha_{\mathsf{A}}(\hat{S}') = (\boldsymbol{u})\alpha_{\mathsf{A}}(\hat{S}'_0)$. The thesis (2) follows trivially.
- $\hat{S} \equiv (u)\hat{S}_0 \xrightarrow{\mathsf{B}:\, \pi} (u)\hat{S}'_0 = \hat{S}'$. Then $\hat{S}_0 \xrightarrow{\mathsf{B}:\, \pi'} \hat{S}'_0$, with $\pi = \mathrm{del}_u(\pi')$. By the induction hypothesis $\alpha_{\mathsf{A}}(\hat{S}_0) \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'_0)$ for some $\tilde{\mu}$, so $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}')$.

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

$$\frac{\alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'''')}{\alpha_{\mathsf{A}}(\hat{S}'')|\alpha_{\mathsf{A}}(\hat{S}''') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'')|\alpha_{\mathsf{A}}(\hat{S}'''')} \, [\alpha\text{-Par}]$$

Using the induction hypothesis for thesis (2):

$$\frac{\alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\mathsf{A}:\, \pi}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'''')}{\alpha_{\mathsf{A}}(\hat{S}'')|\alpha_{\mathsf{A}}(\hat{S}''') \xrightarrow{\mathsf{A}:\, \pi'}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'')|\alpha_{\mathsf{A}}(\hat{S}'''')} \, [\alpha\text{-Par}]$$

[Def] We can have two subcases:

- Let $\mathrm{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} \tilde{P}$, $\hat{S} \equiv \mathsf{A}[\mathrm{X}(\boldsymbol{v}) \mid \tilde{Q}] \mid \hat{S}''$, and assume that $\mathsf{A}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \hat{S}'' \xrightarrow{\mathsf{A}:\, \pi} \hat{S}'$. Both the theses (1) and (2) hold, assuming they hold in the premise of the rule:

  $$\frac{\mathrm{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} \tilde{P} \qquad \mathsf{A}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \tilde{\hat{S}} \xrightarrow{\mathsf{A}:\, \pi}_{\mathsf{A}} \tilde{\hat{S}}'}{\mathsf{A}[\mathrm{X}(\boldsymbol{v}) \mid \tilde{Q}] \mid \tilde{\hat{S}} \xrightarrow{\mathsf{A}:\, \pi}_{\mathsf{A}} \tilde{\hat{S}}'} \, [\alpha\text{-Def}].$$

- Let $\mathsf{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} \tilde{P}$, $\hat{S} \equiv \mathsf{B}[\mathsf{X}(\boldsymbol{v}) \mid \tilde{Q}] \mid \hat{S}''$, and assume that $\mathsf{B}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \hat{S}'' \xrightarrow{\mu} \hat{S}'$. By the induction hypothesis $\alpha_{\mathsf{A}}(\mathsf{B}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \hat{S}'') = \alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}')$. But then both the theses (1) and (2) hold: $\alpha_{\mathsf{A}}(\hat{S}) = \alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}'') = \alpha_{\mathsf{A}}(\hat{S}')$.

[Fuse] There are two cases:

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

$$\frac{\hat{c} \bowtie \hat{d} \qquad \hat{\gamma} = \mathsf{B} : \hat{c} \mid \mathsf{B}' : \hat{d} \qquad \sigma = \{s/x,y\} \qquad s \text{ fresh}}{\hat{S} \xrightarrow{\mathsf{K}:\,\mathtt{fuse}} (s)((\hat{S}_{\mathsf{A}} \mid \hat{S}'_{ctx})\sigma \;\mid\; s[\hat{\gamma}]) = \hat{S}'}\text{[Fuse]}$$

By Lemma B.26 both x and y are not free in $\hat{S}_{\mathsf{A}}$, and then $\hat{S}' \equiv (s)(\hat{S}_{\mathsf{A}} \mid \hat{S}'_{ctx}\sigma \mid s[\hat{\gamma}])$. By rule [$\alpha$-Fuse]: $\alpha_{\mathsf{A}}(\hat{S}) = \alpha_{\mathsf{A}}(\hat{S}_{\mathsf{A}}) \xrightarrow{ctx}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}_{\mathsf{A}}) = \tilde{S}$.
$\alpha_{\mathsf{A}}(\hat{S}') = \alpha_{\mathsf{A}}((s)(\hat{S}_{A})) \equiv \alpha_{\mathsf{A}}(\hat{S}_{A}) = \tilde{S}'$.

ii. Let $\hat{S} \equiv (x,y)(\hat{S}_{\mathsf{A}} \mid \hat{S}_{ctx})$, with $\hat{S}_{\mathsf{A}} \equiv \hat{S}'_{\mathsf{A}} \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}$ and $\hat{S}_{ctx} \equiv \hat{S}'_{ctx} \mid \{\downarrow_y \hat{d}\}_{\mathsf{B}}$. Suppose:

$$\frac{\hat{c} \bowtie \hat{d} \qquad \hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d} \qquad \sigma = \{s/x,y\} \qquad s \text{ fresh}}{\hat{S} \xrightarrow{\mathsf{K}:\,\mathtt{fuse}} (s)((\hat{S}'_{\mathsf{A}} \mid \hat{S}'_{ctx})\sigma \;\mid\; s[\hat{\gamma}]) = \hat{S}'}\text{[Fuse]}$$

By Lemma B.26 y is not free in $\hat{S}_{\mathsf{A}}$ and then $\alpha_{\mathsf{A}}(\hat{S}) = (x,y)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}) \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}) \equiv (x)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}) \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}})$. By rule [$\alpha$-Fuse]: $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{ctx}_{\mathsf{A}} (s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}})\{s/x\} \mid s[\hat{c}]) = \tilde{S}'$.

$$\begin{aligned}
&\alpha_{\mathsf{A}}(\hat{S}') = \\
&(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}\sigma) \mid s[\hat{c}]) \equiv \\
&(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}\{s/x\}) \mid s[\hat{c}]) \equiv && \text{because } y \text{ is not free in } \alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}) \\
&(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}})\{s/x\} \mid s[\hat{c}]) = && \text{by Lemma B.23} \\
&\tilde{S}'.
\end{aligned}$$

[Ask] There are two cases, according to the participant which has moved:

- $\hat{S} \equiv \mathsf{A}[\mathtt{ask}_s\,\phi.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid s[\hat{\gamma}] \xrightarrow{\mathsf{A}:\,\mathtt{ask}_s\,\phi} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid s[\hat{\gamma}]$. Let $\hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d}$. By item 2 of Definition 4.6, $\hat{c} \vdash_{ctx} \phi$. The thesis (1) holds because, by rule [α-AskCtx]:

$$\frac{\hat{c} \not\vdash \neg\phi}{\alpha_{\mathsf{A}}(\hat{S}) = \mathsf{A}[\mathtt{ask}_s\,\phi.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid s[\alpha_{\mathsf{A}}(\hat{\gamma})] \xrightarrow{ctx}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \;\mid\; s[\alpha_{\mathsf{A}}(\hat{\gamma})]}$$

  The thesis (2) follows trivially.

- $\hat{S} \equiv \mathsf{B}[\mathtt{ask}_s\,\phi.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid s[\hat{\gamma}] \xrightarrow{\mathsf{B}:\,\mathtt{ask}_s\,\phi} \mathsf{B}[\tilde{P} \mid \tilde{Q}] \mid s[\hat{\gamma}]$. The thesis (1) follows by rule [α-Ctx]:

$$\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{ctx}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}) = \alpha_{\mathsf{A}}(\hat{S}')$$

  The thesis (2) follows trivially.

Theorem B.28. *For all* A*-safe value-abstract systems* S*, and for all traces* η*:*

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

Furthermore, if η *is* A*-solo and* S *is* $\mathtt{ask}$*-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 $\hat{S} \xrightarrow{\eta'}{}^{n} \hat{S}''$, for $n > 0$. By the induction hypothesis, we have:

$$\exists \tilde{\eta}' \;:\; \alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\tilde{\eta}'}_{\mathsf{A}}{}^{n} \alpha_{\mathsf{A}}(\hat{S}'') \tag{5}$$
$$\eta' \;\mathsf{A}\text{-solo and } \hat{S} \;\texttt{ask}\text{-free} \implies \eta' = \tilde{\eta}' \tag{6}$$

Now, assume that $\hat{S}'' \xrightarrow{\mu} \hat{S}'$, and let $\eta = \eta'\mu$. By Lemma B.27, it follows that:

$$\exists \tilde{\mu} \;:\; \alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}') \tag{7}$$
$$\mu = \mathsf{A} : \pi \;\wedge\; \hat{S} \;\texttt{ask}\text{-free} \implies \mu = \tilde{\mu} \tag{8}$$

By (5) and (7), it follows that:

$$\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\tilde{\eta}}_{\mathsf{A}}{}^{n} \alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}')$$

For the "furthermore" part, let $\tilde{\eta} = \tilde{\eta}'\tilde{\mu}$, and assume that η is A-solo and $\hat{S}$ is $\texttt{ask}$-free. Therefore, by (6) and (8) we conclude that: $\eta = \eta'\,\mu = \tilde{\eta}'\,\tilde{\mu} = \tilde{\eta}$.

Lemma B.29. $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\pi}_{\mathsf{A}} \tilde{S}' \implies \exists \hat{S}'$ A-*safe* . $\hat{S} \xrightarrow{\mathsf{A}:\pi} \hat{S}' \wedge \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$

Proof. Let $\hat{S}_{ctx}$ be an A-free system. Below, we proceed by rule induction, reconstructing the system $\hat{S}$ from $\alpha_{\mathsf{A}}(\hat{S})$ in each case.

[α-Tau] Let $\hat{S} \equiv \mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{ctx}$, and let $\alpha_{\mathsf{A}}(\hat{S}) \equiv \mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\tau}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{S}'$. We have $\hat{S} \xrightarrow{\mathsf{A}:\tau} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{ctx} = \hat{S}' = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

[α-If] Consider the case in which the "true" branch is taken. Then, let $\hat{S} \equiv \mathsf{A}[(\texttt{if } e \texttt{ then } \tilde{P} \texttt{ else } \tilde{P}') \mid \tilde{Q}] \mid \hat{S}_{ctx}$, and let $\alpha_{\mathsf{A}}(\hat{S}) \equiv \mathsf{A}[(\texttt{if } e \texttt{ then } \tilde{P} \texttt{ else } \tilde{P}') \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{if}}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{S}'$. We have $\hat{S} \xrightarrow{\mathsf{A}:\texttt{if}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{ctx} = \hat{S}' = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$. The case in which the "false" branch is taken is similar.

[α-Tell] Let $\hat{S} \equiv \mathsf{A}[\texttt{tell}\downarrow_x \hat{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{ctx}$, and also let $\alpha_{\mathsf{A}}(\hat{S}) \equiv \mathsf{A}[\texttt{tell}\downarrow_x \hat{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{tell}\downarrow_x \hat{c}}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}} = \tilde{S}'$. We have $\hat{S} \xrightarrow{\mathsf{A}:\texttt{tell}\downarrow_x \hat{c}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}} \mid \hat{S}_{ctx} = \hat{S}' = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

[α-Do] Let $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}'$ — which, by item 3 of Theorem 4.5, implies $\hat{\gamma} \xrightarrow{\mathsf{A}:\mathsf{a}}\!\!\!\!\rightarrow \hat{\gamma}'$, with $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$. Furthermore, let $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{A}[\texttt{do}_s\,\mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{ctx}$, and $\hat{S} \xrightarrow{\mathsf{A}:\texttt{do}_s\,\mathsf{a}} s[\hat{\gamma}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{ctx} = \hat{S}'$. We have $\alpha_{\mathsf{A}}(\hat{S}) = s[\hat{c}] \mid \mathsf{A}[\texttt{do}_s\,\mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{do}_s\,\mathsf{a}}_{\mathsf{A}} s[\hat{c}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

[α-Ask] We have $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{A}[\texttt{ask}_s\,\phi.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{ctx}$, $\hat{c} \vdash_{\mathsf{A}} \phi$ and:

$$\alpha_{\mathsf{A}}(\hat{S}) = s[\hat{c}] \mid \mathsf{A}[\texttt{ask}_s\,\phi.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\texttt{ask}_s\,\phi}_{\mathsf{A}} s[\hat{c}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$$

By item 1 of Definition 4.6, since $\hat{c} \vdash_{\mathsf{A}} \phi$ and $\alpha_{\mathsf{A}}(\hat{\gamma}) = \hat{c}$, then $\hat{\gamma} \vdash \phi$. Then, by rule [Ask]:

$$\hat{S} \xrightarrow{\mathsf{A}:\texttt{ask}_s\,\phi} s[\hat{\gamma}] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{ctx} = \hat{S}'$$

[α-Del] Let $\hat{S} \equiv (u)\hat{S}''$ and $\hat{S} \xrightarrow{\mathsf{A}:\pi} (u)\hat{S}''' = \hat{S}'$. Also, let $\alpha_{\mathsf{A}}(\hat{S}) \equiv (u)\alpha_{\mathsf{A}}(\hat{S}'')$. Assuming $\alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\mathsf{A}:\pi'}_{\mathsf{A}} \tilde{S}'''$, we have, by induction hypothesis, $\hat{S}'' \xrightarrow{\mathsf{A}:\pi'} \hat{S}'''$ and $\alpha_{\mathsf{A}}(\hat{S}''') = \tilde{S}'''$ (note that π and π' may differ, depending on whether $u \in \mathrm{fnv}(\pi')$). But then $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\pi}_{\mathsf{A}} (u)\alpha_{\mathsf{A}}(\hat{S}''') = \tilde{S}'$. Clearly $\alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

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

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

Lemma B.30. *For all* A*-safe systems* $\hat{S}$*, and for all* A*-solo traces* η*:*

$$\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\eta}_{\mathsf{A}}{}^{*} \tilde{S}' \implies \exists \hat{S}' \ \mathsf{A}\text{-}safe\ .\ \hat{S} \xrightarrow{\eta}{}^{*} \hat{S}' \wedge \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$$

Proof. By induction on the length of the derivation. Base case holds trivially. Suppose that $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\eta'}{}^{n} \tilde{S}''$, with η' A-solo. By inductive hypothesis $\hat{S} \xrightarrow{\eta'}{}^{n} \hat{S}'' \wedge \alpha_{\mathsf{A}}(\hat{S}'') = \tilde{S}''$. Let $\alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\mathsf{A}:\pi}_{\mathsf{A}} \tilde{S}'$. By Lemma B.29 it trivially follows that $\exists \hat{S}'$ A-safe . $\hat{S}'' \xrightarrow{\mathsf{A}:\pi} \hat{S}' \wedge \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

Lemma B.31. *For all* A*-safe systems* $\hat{S}'$*, and for all* `ask`*-free abstract systems* $\tilde{S}$*:*

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

Proof. In the following $\hat{S}_{\mathit{ctx}}$ is a generic A-free system. We show that lemma holds by rule induction:

[α-Tau] Let $\tilde{S} = \mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \rightarrow_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{S}'$. $\hat{S}'$ must be in the form $\mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{\mathit{ctx}}$. Then $\hat{S} = \mathsf{A}[\tau.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{\mathit{ctx}}$.

[α-If] Consider the case in which the "true" branch is taken. Then, let $\tilde{S} = \mathsf{A}[(\texttt{if}\ e\ \texttt{then}\ \tilde{P}\ \texttt{else}\ \tilde{P}') \mid \tilde{Q}] \rightarrow_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{S}'$. $\hat{S}'$ must be in the form $\mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{\mathit{ctx}}$. Then $\hat{S} = \mathsf{A}[(\texttt{if}\ e\ \texttt{then}\ \tilde{P}\ \texttt{else}\ \tilde{P}') \mid \tilde{Q}] \mid \hat{S}_{\mathit{ctx}}$. The case in which the "false" branch is taken is similar.

[α-Tell] Let $\tilde{S} = \mathsf{A}[\texttt{tell}^{x}\ \tilde{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \rightarrow_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \tilde{c}\}_{\mathsf{A}} = \tilde{S}'$. $\hat{S}'$ must be in the form $\mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}} \mid \hat{S}_{\mathit{ctx}}$. Then $\hat{S} = \mathsf{A}[\texttt{tell}^{x}\ \hat{c}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{\mathit{ctx}}$.

[α-Do] Let $\tilde{c}, \tilde{c}'$ be abstract contracts such that $\tilde{c} \xrightarrow{\mathsf{a}}\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}'$, and let $\tilde{S} = s[\tilde{c}]$ and $\tilde{S}' = s[\tilde{c}']$. We have that $\tilde{S} \rightarrow_{\mathsf{A}} s[\tilde{c}'] = \tilde{S}'$. Then $\hat{S}' = s[\hat{\gamma}'] \mid \hat{S}_{\mathit{ctx}}$, with $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$. Let $\hat{\gamma}$ be a contract configuration such that $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$. By Lemma B.22 and item 3 of Theorem 4.5 $\hat{\gamma} \twoheadrightarrow \hat{\gamma}'$. So $\hat{S} = s[\hat{\gamma}] \mid \hat{S}_{\mathit{ctx}} \rightarrow s[\hat{\gamma}'] \mid \hat{S}_{\mathit{ctx}}$.

[α-Fuse] Suppose: $\dfrac{s\ \mathit{fresh}}{\tilde{S} = (x)(\tilde{S}_0 \mid \{\downarrow_x \tilde{c}\}_{\mathsf{A}}) \xrightarrow{\mathit{ctx}}_{\mathsf{A}} (s)(s[\tilde{c}] \mid \tilde{S}_0\{s/x\}) = \tilde{S}'}$ [α-Fuse].

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

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

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

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

$$\begin{aligned}
\alpha_{\mathsf{A}}(\hat{S}) &= \\
(x,y)(\tilde{S}_0 \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}) &= \\
(x)(\tilde{S}_0 \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}) &= && \text{since } y \text{ not free in } \hat{S}_{\mathsf{A}}, \text{ by Lemma B.24, } y \text{ is not free in } \hat{S}_0 \\
\tilde{S}.
\end{aligned}$$

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

$$\frac{\mathrm{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} \tilde{P} \qquad \mathsf{A}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \tilde{S}_0 \to_{\mathsf{A}} \tilde{S}'}{\tilde{S} = \mathsf{A}[\mathrm{X}(\boldsymbol{v}) \mid \tilde{Q}] \mid \tilde{S}_0 \to_{\mathsf{A}} \tilde{S}'}$$

Let $\hat{S}_0$ be a system such that $\alpha_{\mathsf{A}}(\hat{S}_0) = \tilde{S}_0$. By the induction hypothesis applied to the premise of the rule above, we have:

$$\frac{\mathrm{X}(\boldsymbol{u}) \stackrel{\text{def}}{=} \tilde{P} \qquad \mathsf{A}[\tilde{P}\{\boldsymbol{v}/\boldsymbol{u}\} \mid \tilde{Q}] \mid \hat{S}_0 \to \hat{S}'}{\hat{S} = \mathsf{A}[\mathrm{X}(\boldsymbol{v}) \mid \tilde{Q}] \mid \hat{S}_0 \to \hat{S}'}\ [\text{Def}]$$

[α-Par] Assume $\alpha_{\mathsf{A}}(\hat{S}_1) = \tilde{S}_1$, $\alpha_{\mathsf{A}}(\hat{S}_2) = \tilde{S}_2$ and suppose:

$$\frac{\tilde{S}_0 \to \tilde{S}_1}{\tilde{S} = \tilde{S}_0 \mid \tilde{S}_2 \to_{\mathsf{A}} \tilde{S}_1 \mid \tilde{S}_2}$$

By induction hypothesis $\alpha_{\mathsf{A}}(\hat{S}_0) = \tilde{S}_0$. Then we have:

$$\frac{\hat{S}_0 \to \hat{S}_1}{\hat{S} = \hat{S}_0 \mid \hat{S}_2 \to \hat{S}_1 \mid \hat{S}_2}\ [\text{Par}]$$

[α-Del][α-DelCtx] These cases can be treated together, since we are ignoring labels. Assume $\alpha_{\mathsf{A}}(\hat{S}_1) = \tilde{S}_1$ and suppose:

$$\frac{\tilde{S}_0 \to_{\mathsf{A}} \tilde{S}_0}{(u)\tilde{S}_0 \to_{\mathsf{A}} (u)\tilde{S}_1}$$

By induction hypothesis $\alpha_{\mathsf{A}}(\hat{S}_0) = \tilde{S}_0$ and:

$$\frac{\hat{S}_0 \to \hat{S}_1}{(u)\hat{S}_0 \to (u)\hat{S}_1}$$

[α-DoCtx] Let $\tilde{c}, \tilde{c}'$ be abstract contracts such that $\tilde{c} \xrightarrow{ctx:\mathsf{a}}\!\!\!\twoheadrightarrow_{\mathsf{A}} \tilde{c}'$, and let $\tilde{S} = s[\tilde{c}]$ and $\tilde{S}' = s[\tilde{c}']$. We have that $\tilde{S} \to_{\mathsf{A}} s[\tilde{c}'] = \tilde{S}'$. Then $\hat{S}' = s[\hat{\gamma}'] \mid \hat{S}_{ctx}$, with $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$. By Lemma B.22 and item 3 of Theorem 4.5 we have that there exists a $\hat{\gamma}$ such that $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$ and $\hat{\gamma} \twoheadrightarrow \hat{\gamma}'$. So $\hat{S} = s[\hat{\gamma}] \mid \hat{S}_{ctx} \to s[\hat{\gamma}'] \mid \hat{S}_{ctx}$.

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

Theorem B.32. *For all* ask*-free context-abstract systems* $\tilde{S}$*:*

$$\tilde{S} \to_{\mathsf{A}}^{*} \tilde{S}' \implies \exists S, S' \ \mathsf{A}\text{-}safe : \alpha_{\mathsf{A}}(S) = \tilde{S} \ \wedge\ S \to_{\star}^{*} S' \ \wedge\ \alpha_{\mathsf{A}}(S') = \tilde{S}'$$

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 $\tilde{S} \to_{\mathsf{A}} \tilde{S}'' \to_{\mathsf{A}}^{n} \tilde{S}'$. By the induction hypothesis, there exist $\hat{S}''$, $\hat{S}'$ A-safe such that $\alpha_{\mathsf{A}}(\hat{S}'') = \tilde{S}''$, $\alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$, and $\hat{S}'' \to^{*} \hat{S}'$. Since $\tilde{S} \to_{\mathsf{A}} \tilde{S}''$, Lemma B.31 guarantees that there exists $\hat{S}$ A-safe such that $\alpha_{\mathsf{A}}(\hat{S}) = \tilde{S}$ and $\hat{S} \to^{*} \hat{S}''$. Summing up, we conclude that $\hat{S} \to^{*} \hat{S}'$.

Definition B.33 (Context-abstract obligations). *We define the following set:*

$$\mathrm{O}^{\mathsf{A}}_s(\tilde{S}) = \left\{ \mathsf{a} \,\middle|\, \tilde{S} \equiv s[\hat{c}] \mid \tilde{S}' \;\wedge\; \hat{c} \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} \right\}$$

Definition B.34 (Context-abstract readiness). *Given a session name s and participant A, we define the set of context-abstract systems $\alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s$ as the smallest set such that:*

1. $\tilde{S} \xrightarrow{\mathsf{A}:\,\mathtt{do}_s}_{\mathsf{A}} \implies \tilde{S} \in \alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s$

2. $(\tilde{S} \xrightarrow{\mathsf{A}:\,\neq\{\mathtt{do}_s,\mathtt{if}\}}_{\mathsf{A}} \tilde{S}' \wedge \tilde{S}' \in \alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s) \implies \tilde{S} \in \alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s$

3. $\tilde{S} \xrightarrow{\mathsf{A}:\,\mathtt{if}}_{\mathsf{A}} \wedge (\forall \tilde{S}' : \tilde{S} \xrightarrow{\mathsf{A}:\,\mathtt{if}}_{\mathsf{A}} \tilde{S}' \implies \tilde{S}' \in \alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s) \implies \tilde{S} \in \alpha\text{-}\mathrm{Rdy}^{\mathsf{A}}_s$

We say A *is* α-ready at s in $\tilde{S}$ *when* $\tilde{S} \in \mathrm{Rdy}^{\mathsf{A}}_s$*, and that* A *is* α-ready in $\tilde{S}$ *when*

$$\tilde{S} \equiv (\boldsymbol{u})\tilde{S}' \wedge \mathrm{O}^{\mathsf{A}}_s(\tilde{S}') \neq \emptyset \implies \mathsf{A}\ \alpha\textit{-ready at } s \textit{ in } \tilde{S}'$$

Lemma B.35. *For all* A*-safe* $\hat{S}$*, and for all* s*:*

(1) $\mathrm{O}^{\mathsf{A}}_s(\hat{S}) = \mathrm{O}^{\mathsf{A}}_s(\alpha_{\mathsf{A}}(\hat{S}))$,

(2) A *α-ready at s in* $\alpha_{\mathsf{A}}(\hat{S}) \implies \mathsf{A}$ *$\alpha^\star$-ready at s in* $\hat{S}$

(3) A *$\alpha^\star$-ready at s in* $\hat{S} \implies \mathsf{A}$ *α-ready at s in* $\alpha_{\mathsf{A}}(\hat{S})$*, if* $\hat{S}$ *is* `ask`*-free.*

Proof.

- (1) We have that $\mathrm{O}^{\mathsf{A}}_s(\hat{S}) \subseteq \mathrm{O}^{\mathsf{A}}_s(\alpha_{\mathsf{A}}(\hat{S}))$ follows by item 1 of Theorem 4.5. The reverse inclusion holds by item 3 of Theorem 4.5.

- (2) By induction: Assume A α-ready at s in $\alpha_{\mathsf{A}}(\hat{S})$. There are three cases according to the items of Definition B.34:

  - 1 By Lemma B.29 trivially follows that item 1 of Definition 4.2 holds for $\hat{S}$.

  - 2 Let $\tilde{S}'$ be the system such that $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\,\neq\{\mathtt{do}_s,\mathtt{if}\}}_{\mathsf{A}} \tilde{S}'$, with A α-ready at s in $\tilde{S}'$. By Lemma B.29 we have that exists $\hat{S}'$ such that $\hat{S} \xrightarrow{\mathsf{A}:\,\neq\{\mathtt{do}_s,\mathtt{if}\}}_\star \hat{S}'$ and $\alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$. By induction hypothesis A $\alpha^\star$-ready at s in $\hat{S}'$, and hence item 2 of Definition 4.2 holds for $\hat{S}$.

  - 3 By Lemma B.29 $\hat{S} \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star$. Let $\tilde{S}'$ be an abstract system such that $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\,\mathtt{if}}_{\mathsf{A}} \tilde{S}'$. Again, Lemma B.29 guarantees $\hat{S} \xrightarrow{\mathsf{A}:\,\mathtt{if}}_\star \hat{S}'$, with $\alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$. Since A is α-ready at s in $\tilde{S}'$, by induction hypothesis, A is ready at s in $\hat{S}'$, and hence item 3 of Definition 4.2 holds for $\hat{S}$.

- (3) By induction: Let $\hat{S}$ be an `ask`-free system, and assume A $\alpha^\star$−ready at s in $\hat{S}$. There are three cases according to the items of Definition 4.2:

  - 1 By Lemma B.27 trivially follows that item 1 of Definition B.34 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

  - 2 Let $\hat{S}'$ be the system such that $\hat{S} \xrightarrow{\mathsf{A}:\,\neq s,\mathtt{if}}_\star \hat{S}'$, with A $\alpha^\star$-ready at s in $\hat{S}'$. By Lemma B.27 $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\,\neq s,\mathtt{if}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\tilde{S}')$, with A α-ready at s in $\alpha_{\mathsf{A}}(\hat{S}')$ by induction hypothesis. Clearly item 2 of Definition B.34 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

  - 3 Let $\hat{S}'$ be a system such that $\hat{S} \xrightarrow{\mathsf{a}:\,\mathtt{if}}_\star \hat{S}'$. A must be ready at s in $\hat{S}'$. By Lemma B.27 $\alpha_{\mathsf{A}}(\tilde{S}) \xrightarrow{\mathsf{A}:\,\neq s,\mathtt{if}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S})$, with A α-ready at s in $\alpha_{\mathsf{A}}(\tilde{S}')$ by induction hypothesis. Clearly item 3 of Definition B.34 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

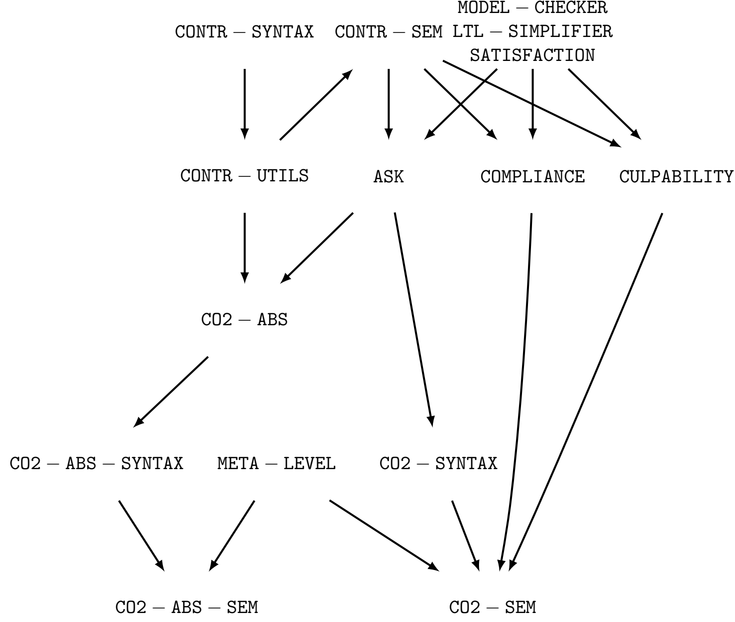

Figure 10: Graph of module importation.

Lemma B.36. *For all* A*-safe* $\hat{S}$*:*

(1) if A *is* α*-ready in* $\alpha_{\mathsf{A}}(\hat{S})$*, then* A *is* $\alpha^{\star}$*-ready in* $\hat{S}$*.*

(2) if A *is* $\alpha^{\star}$*-ready in* $\hat{S}$ *and* $\hat{S}$ *is* `ask`*-free, then* A *is* α*-ready in* $\alpha_{\mathsf{A}}(\hat{S})$*.*

Proof. Straightforward consequence of Lemma B.35

Theorem 4.11. *Let* P *be a context-abstract process. If* P *is* α*-honest, then* P *is* $\alpha^{\star}$*-honest. Conversely, if* P *is* $\alpha^{\star}$*-honest and* `ask`*-free, then* P *is* α*-honest.*

Proof. For the first part, let $\tilde{P}$ be an α-honest abstract process. Let $\hat{S}_{ctx}$ be an A-free value-abstract system, and assume that $\mathsf{A}[\tilde{P}] \mid \hat{S}_{ctx} \rightarrow^{*} \hat{S}$ for some $\hat{S}$. Since $\mathsf{A}[\tilde{P}] \mid \hat{S}_{ctx}$ is A-safe, then by item (3) of Lemma B.19 it follows that $\hat{S}$ is A-safe as well. Let $\tilde{S} = \alpha_{\mathsf{A}}(\hat{S})$. By Theorem B.28 it follows that $\mathsf{A}[\tilde{P}] \rightarrow_{\mathsf{A}}^{*} \tilde{S}$. Since $\tilde{P}$ is α-honest, then A is α-ready in $\tilde{S}$. Therefore, item (1) of Lemma B.36 guarantees that A is $\alpha^{\star}$-ready in $\hat{S}$. Hence, by Definition 3.6 we conclude that $\tilde{P}$ is $\alpha^{\star}$-honest.

For the second part, suppose that $\tilde{P}$ is $\alpha^{\star}$-honest and `ask`-free. Let $\tilde{S}$ be a context-abstract system such that $\mathsf{A}[\tilde{P}] \rightarrow_{\mathsf{A}}^{*} \tilde{S}$. By Theorem B.32 it follows that there exist $\hat{S}, \hat{S}'$ A-safe such that $\alpha_{\mathsf{A}}(\hat{S}) = A[\tilde{P}]$, $\alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}$, and $\hat{S} \rightarrow^{*} \hat{S}'$. Since $\tilde{P}$ is $\alpha^{\star}$-honest, then A is $\alpha^{\star}$-ready in $\hat{S}'$, and so by item (2) of Lemma B.36 it follows that A is α-ready in $\tilde{S}'$. Hence, by Definition 4.10 we conclude that $\tilde{P}$ is α-honest.

The diagram in Figure 11 illustrates the dependencies among the proofs. The diagram Figure 10 illustrates the graph of module importation used in our Maude implementation. The complete code of our verification tool is available at `http://tcs.unica.it/software/co2-maude`.

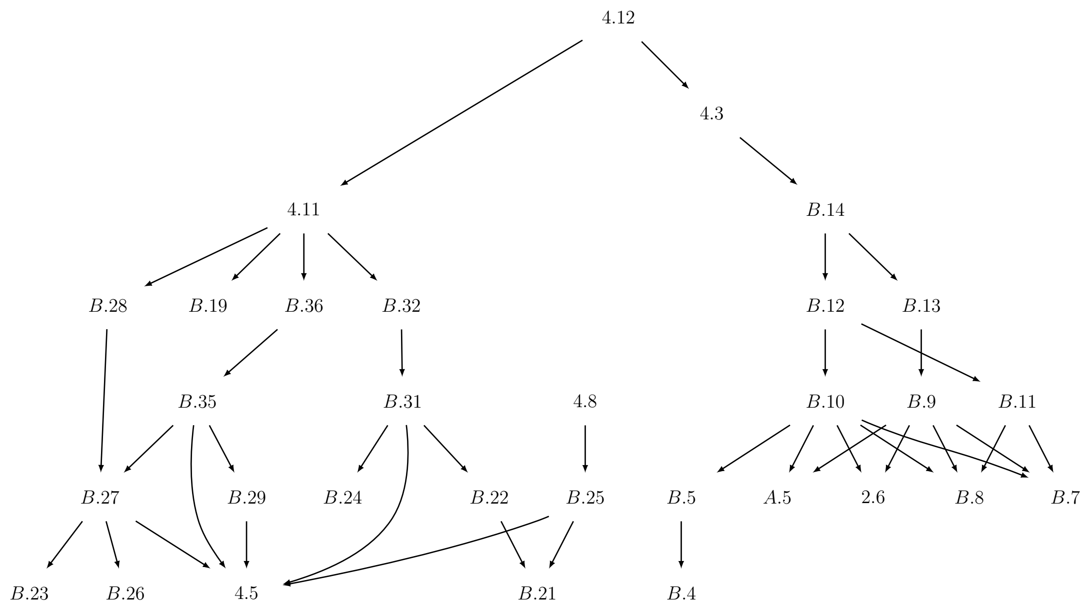

Figure 11: Dependencies among the proofs .