Università degli Studi di Cagliari

DOTTORATO DI RICERCA

Matematica e Informatica

Ciclo XXIX

TITOLO TESI

Verification of contract-oriented systems

Settore scientifico disciplinare di afferenza

INF/01

Presentata da:	Maurizio Murgia
Coordinatore Dottorato:	Giuseppe Rodriguez
Tutor:	Massimo Bartoletti

Esame finale anno accademico 2015 – 2016

Tesi discussa nella sessione d'esame marzo – aprile 2017

Contents

List of Figures

List of Tables

Abstract

In this thesis 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. We develop 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. We develop a model-checking technique for this abstraction, we describe its implementation in Maude, and we discuss some experiments with it. We then introduce timed session types, an extension of binary session types, formalising timed communication protocols between two participants at the endpoints of a session. They feature a decidable compliance relation, which generalises to the timed setting the usual progress-based notion of compliance between untimed session types. We show a sound and complete technique to decide when a timed session type admits a compliant one, and if so, to construct the most precise session type compliant with a given one, according to the subtyping preorder induced by compliance. Decidability of subtyping follows from these results.

Chapter 1

Introduction

Motivation

Contract-oriented computing [28] 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 [84]).

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 7.1.8 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 as prescribed by *all* their advertised contracts, they are called *honest*. Instead, when services are *not* honest, they do *not* always respect the contracts they advertise, at least in some execution context. This may happen either unintentionally (because of errors in the service specification or in its implementation), or even because of malicious behaviour. Since discrepancies between the advertised and the actual behaviour can be sanctioned, a new kind of attacks becomes possible: if a service does not behave as promised, an attacker can induce it to a situation where the service is 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 paragraph can be formally modelled and studied in CO_2, a core process calculus for contract-oriented computing [28, 26]. 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 [41, 70, 71, 49], Petri nets [96, 14, 15], multi-player games [17, 16], logics [28, 26], *etc.* Among behavioural types, *session types* [69, 70] have been devoted a lot of attention in the last few years, both at the foundational level [71, 36, 48, 77, 58, 44, 98, 66, 46, 33, 24] and at the application level [54, 101, 87, 63]. 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 this thesis, we adopt CO_2 with binary session types as our contract model of choice. 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, a compliant contract needs to be advertised by another agent: compliance, which is strongly related to the standard notions of duality and subtyping [65, 64, 10], 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 [27] exploits the fact that the value-free fragment of CO_2 is Turing-powerful).

Time In order to fully take advantage of contract-oriented computing, the infrastructure should be able to detect (and possibly sanction) contract violations, and this should be accomplished inspecting *only* the state of opened sessions. The latter requirement seems impossible to realize with the contract model mentioned above: a participant is *always* able to recover from culpability by performing a finite number of actions (Lemma 6.1.10), making culpability a *transient* status. A fair infrastructure should sanction only those agents which are *permanently* culpable of contract violations. We believe that a natural approach to the above problem is to enrich the contract model with timing constraints, enabling the specification of deadlines. In this way, a participant becomes (definitively) culpable when she has not fulfilled her obligations within the time window prescribed by her contract.

Formal methods for time have already approached the realm of session types [38, 86]. However, these approaches introduce time into an already sophisticated framework, featuring multiparty session types with asynchronous communication (via unbounded buffers). While on the one hand this has the advantage of extending to the timed setting type techniques which enable compositional verification [71], on the other hand it seems that some of the key notions of the untimed setting (e.g., compliance, duality) have not been explored yet in the timed case.

We think that studying timed session types in a basic setting (synchronous communication between two endpoints, as in the seminal untimed version) is worthy of attention.

From a theoretical point of view, the objective is to lift to the timed case some decidability results, like those of compliance and subtyping. Some intriguing problems arise: unlike in the untimed case, a timed session type not always admits a compliant; hence, besides deciding if two session types *are compliant*, it becomes a relevant problem whether a session type *has a compliant*. From a more practical perspective, decision procedures for timed session types, like those for compliance and for dynamic verification, enable the implementation of programming tools and infrastructures for the development of safe communication-oriented distributed applications. For instance, the message-oriented middleware in [13] exploits timed session types to allow disciplined interactions between mutually distrusting services.

Contributions

Verification of honesty In Part II 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 devise an abstract notion of honesty (α-*honesty*, Definition 8.1.12), which approximates the execution context. The main technical result is Theorem 8.1.14, 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 thesis 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 8.1.7. 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 [27] (see Section 9.1.2). The Maude implementation of our technique has an important role in the tool-chain Diogenes [4, 6].

Timed contracts In Part III we present a theory of binary timed session types (TSTs), and we explore its viability as a foundation for programming tools to leverage the complexity of developing distributed applications.

The semantics of TSTs is a conservative extension of the synchronous semantics of untimed session types [10], adding *clock valuations* to associate each clock with a positive real. We also extend to the timed setting the standard semantic notion of *compliance*, which relates two session types whenever they enjoy progress until reaching success.

Despite the semantics of TSTs being infinite-state (while it is finite-state in the untimed case), we develop a sound and complete decision procedure for verifying compliance (Theorem 10.2.7). To do that, we reduce this problem to that of model-checking deadlock freedom in timed automata [3], which is decidable, and we implement our technique using the Uppaal model checker [31]. We detail the encoding of TSTs into timed automata in Section 12.1.

We develop a procedure to detect whether a TST admits a compliant. This takes the form of a kind system which associates, to each TST p, a set of clock valuations under which p admits a compliant. The kind system is sound and complete (Theorem 11.1.6 and Theorem 11.1.8), and it can be used to define the *canonical compliant* of a given TST (Definition 11.1.5).

In Section 11.2 we prove that kind inference is decidable (Theorem 11.2.4), and from this we infer a decidable (sound and complete) procedure for the existence of compliant, and the computability of the canonical compliant construction (Theorem 11.2.5).

In Section 11.3 we study the semantic subtyping preorder [10] for TSTs. We then show that the canonical compliant of p is the *greatest* TST compliant with p, according to the subtyping preorder (Theorem 11.3.2). Decidability of subtyping (Theorem 11.3.3) follows from that of compliance and kind inference. This provides us with an effective way of checking if a service with type p can be replaced by one with a subtype p' of p, guaranteeing that all the services which interacted correctly with the old one will do the same with the new one.

In Section 13.1 we address the problem of dynamically monitoring interactions regulated by TSTs. To do that, we will provide TSTs with a *monitoring semantics*, which detects when a participant is not respecting its TST. This semantics enjoys some desirable properties: it is deterministic, and it guarantees that in each state of an interaction, either we have reached success, or someone is in charge of a move, or not respecting its TST.

Timed session types are used as the contract model of choice in the contract-oriented middleware described in [13, 5], which implements the contractual primitives of CO_2.

Part of the material presented in this thesis is borrowed by the papers where our results were originally presented. In particular, the material in Part II borrows from [21, 20], while the material in Part III borrows from [11, 12].

Part I

Background

Chapter 2

Order theory

In this chapter we introduce partial orders, complete lattices, and order preserving functions between them. We illustrate some of their properties, in particular the Knaster-Tarski fixed point theorem, which will be useful in the subsequent development of this thesis. The material in this chapter is taken from [89, 2].

2.1 Partial orders

Definition 2.1.1 *A partially ordered set (poset for short) is a set together with a partial order relation, that is, a structure $(D, \leq)$, where D is a set and $\leq \subseteq D \times D$ is a relation satisfying:*

1. $\forall d \in D : d \leq d$ *(reflexivity)*
2. $\forall d, d', d'' \in D : d \leq d' \wedge d' \leq d'' \implies d \leq d''$ *(transitivity)*
3. $\forall d, d' \in D : d \leq d' \wedge d' \leq d \implies d = d'$ *(antisymmetricity)*

Definition 2.1.2 *Let $(D, \leq)$ be a poset and let $X \subseteq D$ and element $d \in D$ is said:*

- *An upper bound of X iff, for all $x \in X$, $x \leq d$.*
- *The least upper bound of X, denoted as $\mathrm{lub}(X)$, iff d is an upper bound of X and, for any d' upper bound of X, it holds that $d \leq d'$.*
- *A lower bound of X iff, for all $x \in X$, $d \leq x$.*
- *The greatest lower bound of X, denoted as $\mathrm{glb}(X)$, iff d is a lower bound of X and, for any d' lower bound of X, it holds that $d' \leq d$.*

Note that lubs and glbs not always exist, but when they exist, they are unique.

Definition 2.1.3 *An infinite sequence $\vec{d} = d_0 d_1 \ldots$ of elements in a poset $(D, \leq)$ is said* decreasing *iff $d_{i+1} \leq d_i$ for any $i \in \mathbb{N}$. We refer to infinite decreasing sequences with the term ω-*chains*. We write $\mathrm{lub}(\vec{d})$ (resp. $\mathrm{glb}(\vec{d})$) for the least upper bound (resp. greatest lower bound) of the set of elements appairing in $\vec{d}$. Given a function $f : D \to D$, we write $f(\vec{d})$ for the sequence resulting from the application of f to the elements of $\vec{d}$, i.e. $f(\vec{d}) = f(d_0)f(d_1)\ldots$, and we write $f^n(d)$ for the n-th iteration of f starting from d, formally:*

$$\begin{aligned} f^0(x) &= x \\ f^{n+1}(x) &= f(f^n(x)) \end{aligned}$$

We write $\vec{f}(x)$ for the (not necessarly decreasing) sequence $f^0(x)f^1(x)\ldots$.

2.2 Monotonic functions and lattices

Definition 2.2.1 *Let $(D, \leq)$ be a poset and let f be a function from D to D.*

- *We say that f is monotonic iff $d \leq d' \implies f(d) \leq f(d')$.*
- *We say that f is cocontinuous iff, for any ω-chain $\vec{d}$ of elements in D, $f(\mathrm{glb}(\vec{d})) = \mathrm{glb}(f(\vec{d}))$*

Definition 2.2.2 *Let $(D, \leq)$ be a poset, and let f be a monotonic function from D to D. Let $d \in D$, then:*

- *d is a pre-fixed point of f iff $f(d) \leq d$.*
- *d is a post-fixed point of f iff $d \leq f(d)$.*
- *d is a fixed point of f iff $d = f(d)$. We write $\mathrm{gfp}(f)$ and $\mathrm{lfp}(f)$ referring to, respectively, the greatest and the least fixed point of f (if they exist).*

Definition 2.2.3 *A complete lattice is a poset $(D, \leq)$ such that, for every $X \subseteq D$, both $\mathrm{lub}(X)$ and $\mathrm{glb}(X)$ exist. The top and bottom elements of $(D, \leq)$, $\mathrm{lub}(D)$ and $\mathrm{glb}(D)$, are denoted, respectively, by the symbols $\top$ and $\bot$ when D is clear from the context.*

We now state the fundamental Knaster-Tarski fixed point theorem, which basically says that the set of fixed points of a monotonic endofunction f over a complete lattice is a complete lattice as well (ordered with the relation of the starting lattice). Hence both $\mathrm{gfp}(f)$ and $\mathrm{lfp}(f)$ exist (as the top and bottom elements of F). Furthermore, top and bottom elements correspond, respectively, to the greatest pre-fixed point and to the least post-fixed point of f.

Theorem 2.2.4 (Knaster-Tarski fixed point theorem [93]) *Let $(D, \leq)$ be a complete lattice, and let f be a monotonic function from D to D. Then $(F, \leq)$, where F is the set of fixed points of f, is a complete lattice. In particular, we have:*

$$\mathrm{lub}(F) = \mathrm{lub}(\{x \mid x \leq f(x)\}) \qquad \mathrm{glb}(F) = \mathrm{glb}(\{x \mid f(x) \leq x\})$$

The following lemma recalls some well known facts about functions over complete lattices and their fixed points. These facts will be used to prove that kind inference in Chapter 11 is decidable. Note that, by Theorem 2.2.4, $\mathrm{gfp}(f)$ exists in Items (b) and (c) below.

Lemma 2.2.5 *Let $(D, \leq)$ be a complete lattice, and let f be a monotonic function from D to D. Then:*

(a) if D is finite, then f is cocontinuous.

(b)

$$\mathrm{gfp}(f) \leq \mathrm{glb}(\vec{f}(\top))$$

(c) if f is cocontinuous, then

$$\mathrm{gfp}(f) \;=\; \mathrm{glb}(\vec{f}(\top))$$

Proof. For item (a), let $d_0, d_1, \ldots$ be a decreasing sequence of elements in D. By finiteness of D, there must exist some n such that, for all n', $d_n = d_{n+n'}$. Clearly, $f(\bigsqcap_i d_i) = f(d_n)$. By monotonicity, the sequence $f(d_0), f(d_1), \ldots$ is decreasing, and its meet has to be $f(d_n)$.

For item (b), we show, by induction on n, that, for all $n \geq 0$, $\mathrm{gfp}(f) \leq f^n(\top)$. The base case is trivial, since $f^0(\top) = \top$. For the induction case, suppose that $\mathrm{gfp}(f) \leq f^n(\top)$. By monotonicity of f we have that: $\mathrm{gfp}(f) = f(\mathrm{gfp}(f)) \leq f(f^n(\top)) = f^{n+1}(\top)$.

For item (c), see [89].

Chapter 3

Labelled Transition Systems

In this chapter we illustrate Labelled Transition Systems (LTS), and we show how to use them to give operational semantics to a language for concurrent systems. The material in this chapter is taken from [89, 2].

3.1 Labelled transition systems

LTS are a very general formalism which can be used to model a variety of concrete systems (e.g., computing machines). Basically, the idea is that a system can be abstractly described by a set of *states*, plus a set of *transitions* from state to state, which model its behaviour.

Definition 3.1.1 (LTS) *A* labelled transition system *(LTS) is a triple* $(Q, \Sigma, \rightarrow)$*, where:*

- Q *is a set (called the set of* states*),*
- Σ *is a set (called the set of* labels*), and*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is a relation (called* transition relation*).*

We often write $q \xrightarrow{a} q'$ *as a shorthand for* $(q, a, q') \in \rightarrow$*. An* initial LTS *is a tuple* $(Q, \Sigma, \rightarrow, q_0)$*, where* $(Q, \Sigma, \rightarrow)$ *is an LTS, and* $q_0 \in Q$ *is the* initial state*.*

We now introduce the concept of *deterministic* LTS, namely those LTSs for which the next state is *uniquely* determined by the transition label.

Definition 3.1.2 (Deterministic LTS) *An LTS is* deterministic *when:*

$$\forall q, q', q'' \in Q \;:\; \forall a \in \Sigma \;:\; \quad q \xrightarrow{a} q' \;\wedge\; q \xrightarrow{a} q'' \implies q' = q''$$

3.2 Processes

We now introduce an idealized language for the representation of a simple class of LTSs. This kind of languages are usually referred as *Process Algebras* or *Process Calculi* in the literature [7]. The language in this section is basically the *Calculus of Communicating Systems* (CCS for short, [82]), although we omit recursion and relabelling for simplicity. Before giving the definition of the syntax of processes, we add some structure to the set of labels. In particular we will assume a special label τ, and an involutive unary operator on non-τ labels.

Definition 3.2.1 (Names, labels, and actions) *We define the following sets:*

- $\mathcal{A}$*, a countably infinite set of* names *(ranged over by* $a, b, \ldots$*);*
- $\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$*, the set of* co-names*;*
- $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$*, the set of labels; we extend complementation* $\bar{\cdot}$ *to all the elements of* $\mathcal{L}$*, and we impose* $\bar{\bar{a}} = a$ *for all* $a \in \mathcal{A}$*.*
- $\Sigma = \mathcal{L} \cup \{\tau\}$*, the set of* actions *(ranged over by* $\alpha, \beta, \ldots$*). The action* τ *models an* internal *operation of a process, which is not observable by the environment.*

Definition 3.2.2 (Processes) *The set* Proc *of* processes *is inductively defined as follows:*

$$
\begin{array}{rcll}
P, Q & ::= & \mathbf{0} & \textit{(Stuck process)} \\
 & \mid & \alpha.P & \textit{(Prefixed process, } \alpha \in \Sigma\textit{)} \\
 & \mid & P + Q & \textit{(Choice)} \\
 & \mid & P \mid Q & \textit{(Parallel composition)} \\
 & \mid & P \setminus L & \textit{(Restriction, } L \subseteq \mathcal{L}\textit{)}
\end{array}
$$

Intuitively, 0 represents a stuck process, which does nothing; the process $\alpha.P$ first "fires" the prefix α, and then proceeds as P; the process $P + Q$ may proceed either as P or as Q; the process $P \mid Q$ behaves as the interleaving of the actions of P and Q, or synchronise dual actions; the process $P \setminus L$ behaves as P except for actions (and their co-actions) in L, which are blocked.

Definition 3.2.3 (Semantics of processes) *The semantics of a process* P *is defined as the initial LTS* $(P, \mathsf{Proc}, \Sigma, \rightarrow)$*, where* $\rightarrow$ *is the least relation closed under the rules in Figure 3.1.*

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\ [\textsc{Pref}] \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ [\textsc{SumL}] \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}\ [\textsc{SumR}]$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ [\textsc{ParL}] \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\ [\textsc{ParR}]$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ [\textsc{Com}] \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha, \bar{\alpha} \notin L}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ [\textsc{Res}]$$

Figure 3.1: Operational semantics of processes.

Chapter 4

Kripke Structures, Linear Temporal Logic and Model checking

Model checking is a formal verification technique for assessing functional properties of systems [8]. Functional properties considered are usually combinations of safety properties (something bad never happens, [74]) and liveness properties (something good eventually happens, [74]). Given a model of the system under consideration, model checking automatically checks whether it satisfies a given property. This is accomplished trough an exaustive exploration of the state space of the model. In this chapter we will briefly illustrate Kripke Structures (KS for short, [73]) and Linear Temporal Logic (LTL for short, [88]), two mathematical tools for the specification of, respectively, the behaviour of systems and their properties.

4.1 Kripke Structures

The intuition behind KSs is very similar to that of LTSs, i.e. to describe the behaviour of systems trough states and transitions between them. However, the two formalisms promote two rather different points of view on what is observable about a system. In LTSs the basic observables are action labels, and the emphasis is (usually) on the interaction capabilities with an external agent, tipically the execution context or the user. KSs, instead, focus on the, possibly internal, properties of the system: in KSs transitions are unlabelled, while each state is associated with the set of *atomic propositions* which are satisfied by it. However, the two models can be related through appropriate mappings, see [57], or Definition 6.1.11 of this thesis for an ad hoc transformation of the LTS semantics of contracts into a KS.

Definition 4.1.1 (Kripke structure[51]) *Let* $\mathbb{AP}$ *be a set of* atomic propositions*. A* Kripke structure *is a triple* $(S, \rightarrow, L)$*, where:*

- S *is a set of states;*
- $\rightarrow \subseteq S \times S$ *is a transition relation such that, for all* $s \in S$*, there is* $s' \in S$ *with* $s \rightarrow s'$*;*

- $L : S \to 2^{\mathbb{AP}}$, *the labelling function, associates with each state the atomic propositions satisfied by it.*

Given a Kripke strucure $(S, \to, L)$ *and a state* $s_0 \in S$*, we define the set of* maximal traces *starting from* s_0, $\text{Paths}(s_0)$*, as follows:*

$$\{L(s_0)L(s_1)\cdots \mid \forall i > 0.\ s_{i-1} \to s_i\}$$

4.2 Linear temporal logic

LTL is a propositional *temporal* logic [62], able to specify *linear time* properties. Very roughly, temporal logics are logical formalisms used for the representation of assertions about time. Time here is dealt with abstractly, without explicit time-stamps, enabling the reasoning about the causal dependencies between events but not about numerical timing constraints. LTL, and linear time properties in general, are based on the assumption that time is an infinite line. Indeed, a linear time property is denoted by a set of *traces*, specifying all and only the allowed executions [8].

LTL can express statements like "the system will *eventually* leave the critical section", or "the system is *always* in a not deadlock state", with formulae $\Diamond\texttt{crit}$ and $\Box\neg\texttt{deadlock}$, where `crit` and `deadlock` are atomic propositions holding, respectively, in critical section states and deadlock states. Modalities $\Box$ and $\Diamond$ are not primitive in LTL, as they can be derived from connective $\texttt{U}$ (until). Roughly, the formula $\phi_1\ \texttt{U}\ \phi_2$ holds in those states which satisfy ϕ_1 for a finite numer of steps (possibly 0) and then satisfy ϕ_2 for ever. Another useful modality is $\bigcirc$ (next): $\bigcirc\phi$ is satisfied by those states which satisfies ϕ at the next step.

Definition 4.2.1 (LTL syntax) *The syntax of LTL formulae is given by the following productions:*

$$\phi ::= \texttt{true} \mid \texttt{a} \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc\phi \mid \phi_1\ \texttt{U}\ \phi_2$$

where $\texttt{a} \in \mathbb{AP}$*. The other boolean connectives can be derived in the usual way. The modalities* $\Diamond$ *and* $\Box$ *are derivable as follows:*

$$\Diamond\phi \stackrel{def}{=} \texttt{true}\ \texttt{U}\ \phi \qquad \Box\phi \stackrel{def}{=} \neg\Diamond\neg\phi$$

We now define when an infinite sequence of $\mathbb{AP}$ subsets satisfies an LTL formula. This is instrumental for the definition of the satisfaction relation over states of KSs.

Definition 4.2.2 (LTL semantics over traces) *Given a set* S*, we define the set* S^ω *as the set of infinite sequences of elements in* S*:*

$$S^\omega \stackrel{def}{=} \{\sigma_0\sigma_1\ldots \mid \forall i \in \mathbb{N} : \sigma_i \in S\}$$

Given an infinite sequence $\sigma = \sigma_0\sigma_1\ldots$*, we denote with* $\sigma[i\ldots]$ *the infinite suffix of* σ *starting from* i*, i.e.* $\sigma[i\ldots] = \sigma_i\sigma_{i+1}\ldots$*. We define the satisfaction relation over infinite traces as*

the least relation between $(2^{\mathbb{AP}})^\omega$ *and LTL formulae satisfying:*

$$
\begin{array}{lcll}
\sigma & \models & \texttt{true} & \\
\sigma & \models & \texttt{a} & \textit{iff} \quad \texttt{a} \in \sigma_0 \\
\sigma & \models & \phi_1 \wedge \phi_2 & \textit{iff} \quad \sigma \models \phi_1 \textit{ and } \sigma \models \phi_2 \\
\sigma & \models & \neg\phi & \textit{iff} \quad \sigma \not\models \phi \\
\sigma & \models & \bigcirc\phi & \textit{iff} \quad \sigma[1\ldots] \models \phi \\
\sigma & \models & \phi_1 \,\texttt{U}\, \phi_2 & \textit{iff} \quad \exists j \geq 0 \forall i < j : \sigma[i\ldots] \models \phi_1 \textit{ and } \sigma[j\ldots] \models \phi_2
\end{array}
$$

Definition 4.2.3 (LTL semantics over states) *Given a Kripke structure* $M = (S, \rightarrow, L)$*, we define the satisfaction relation over states* $s \in S$ *as follows:*

$$s \models \phi \qquad \textit{iff} \qquad \forall \sigma \in \mathrm{Paths}(s) : \sigma \models \phi$$

Chapter 5

Timed Automata

Timed automata are a classical formalism for modelling real-time systems, introduced by Alur and Dill in the early 90's [3], and extended in many subsequent works [99]. Since then, timed automata have become widely used both in the industry and in the academia, also thanks to successful tools (most notably Uppaal [31]), which enable the modelling and verification of realistic timed systems. Roughly, a timed automaton (TA) is a finite automaton, annotated with timing contraints and reset predicates using real valued *clocks*. We refer to [32, 99, 2, 8] for more details about the material in this chapter.

5.1 Basic definitions

Before formalising TA, we introduce some auxiliary notions and notation. Let $\mathbb{C}$ be a set of clocks, ranged over by $t, t', \ldots$ and let $d, d', \ldots$ range over the set $\mathbb{N}$ of natural numbers. We use $R, T, \ldots \subseteq \mathbb{C}$ to range over sets of clocks. Let $\mathtt{A}$ be a set of action names. We define the set of output action $\mathtt{A}^{!} = \{\mathtt{a}! \mid \mathtt{a} \in \mathtt{A}\}$ and the set of input actions $\mathtt{A}^{?} = \{\mathtt{a}? \mid \mathtt{a} \in \mathtt{A}\}$. We denote with $\mathtt{L}_{\tau} = \mathtt{A}^{!} \cup \mathtt{A}^{?} \cup \{\tau\}$ the set of labels, ranged over by $\ell_{\tau}, \ell_{\tau}', \ldots$.

Definition 5.1.1 (Guard) *We define the set $\mathcal{G}_{\mathbb{C}}$ of* guards *over clocks $\mathbb{C}$ as follows:*

$$g \ ::= \ \mathtt{true} \ \mid \ \neg g \ \mid \ g \wedge g \ \mid \ t \circ d \ \mid \ t - t' \circ d \qquad \text{where } \circ \in \{<, \leq, =, \geq, >\}$$

The semantics of guards is defined in terms of *clock valuations*: they are functions which associate each clock in $\mathbb{C}$ with a value in $\mathbb{R}_{\geq 0}$. These values are *not* associated with a particular unit of time (i.e., seconds, hours, ...); so, we will call clocks values generically *time units* (abbreviated *t.u.*).

Definition 5.1.2 (Clock valuation) *We denote with $\mathbb{V}[\mathbb{C}] : \mathbb{C} \to \mathbb{R}_{\geq 0}$ the set of* clock valuations *over $\mathbb{C}$. We write $\mathbb{V}$ as a shortcut for $\mathbb{V}[\mathbb{C}]$ when $\mathbb{C}$ is clear from the context. We use $\nu, \eta, \ldots$ to range over $\mathbb{V}$, and $\nu_0, \eta_0, \ldots$ to denote the valuations mapping each clock to 0. We use $\mathcal{K}, \mathcal{K}', \ldots$ to range over subsets of $\mathbb{V}$.*

Definition 5.1.3 (Time increment and reset) *We write $\nu + \delta$ for the clock valuation which increases ν by δ, i.e., for all $t \in \mathbb{C}$:*

$$(\nu + \delta)(t) \;=\; \nu(t) + \delta$$

For a set $R \subseteq \mathbb{C}$, we write $\nu[R]$ for the reset *of the clocks in R, i.e., for all $t \in \mathbb{C}$:*

$$\nu[R](t) \;=\; \begin{cases} 0 & \text{if } t \in R \\ \nu(t) & \text{otherwise} \end{cases}$$

When R is a singleton, e.g. $R = \{x\}$, we shall usually write $\nu[x]$ instead of $\nu[\{x\}]$.

Definition 5.1.4 *A set of clock valuations $\mathcal{K}$ is said* past closed *if and only if:*

$$\nu + \delta \in \mathcal{K} \implies \nu \in \mathcal{K}$$

Definition 5.1.5 (Semantics of guards) *For all guards g, we define the set of clock valuations $[\![g]\!]$ inductively as follows, where $\circ \in \{<, \leq, =, \geq, >\}$:*

$$[\![\texttt{true}]\!] = \mathbb{V} \qquad [\![\neg g]\!] = \mathbb{V} \setminus [\![g]\!] \qquad [\![g_1 \wedge g_2]\!] = [\![g_1]\!] \cap [\![g_2]\!]$$
$$[\![t \circ d]\!] = \{\nu \mid \nu(t) \circ d\} \qquad [\![t - t' \circ d]\!] = \{\nu \mid \nu(t) - \nu(t') \circ d\}$$

A guard g is said past closed when $[\![g]\!]$ is past closed.

5.2 Timed automata

We are now ready to give the definition of timed automata. They are composed by a finite set of *locations*, one of which is the *initial* location. Every location is associated with a past closed guard, called *invariant*, which specifies when the control can be in that location. A subset of locations, the *urgent* locations, are used to model states that "do not let time pass", i.e. force the next transition to be a discrete action. Urgent locations do not add expressive power to the model, since urgency can be specified through invariants [31], and often do not appair in definitions of TA. However, we prefer to have them explicitly, as this will simplify the technical threatment in section 12.1. Locations are connected by *edges*. Basically, edges have a similar role to transition functions of non-deterministic finite states automata, further enriched with a guard specifing when that transition is enabled, and with a set of clocks which are reset (exactly) when the transition is taken.

Definition 5.2.1 (Timed automaton) *A TA is a tuple $A = (Loc, Loc^u, l_0, E, I)$ where: Loc is a finite set of* locations*; $Loc^u \subset Loc$ is the set of* urgent *locations; $l_0 \in Loc$ is the* initial *location; $E \subseteq Loc \times \mathtt{L}_\tau \times \mathcal{G}_\mathbb{C} \times 2^\mathbb{C} \times Loc$ is a set of edges; and $I : Loc \to \mathcal{G}_\mathbb{C}$ is the* invariant *function. We require that, for all locations $l \in Loc$, $I(l)$ is past closed.*

The semantics of TA is defined in terms of *timed* LTSs. These are LTSs much alike those in Definition 3.1.1, except for the fact that, besides actions, labels also include time delays.

Definition 5.2.2 (Timed LTS) *A timed labelled transition system (TLTS) is a triple* $(Q, \mathtt{L}_\delta, \rightarrow)$*, where:*

- Q *is a set (called the set of* configurations*),*
- $\mathtt{L}_\delta \supseteq \mathbb{R}_{\geq 0}$ *is a set (called set of* labels*, and ranged over by* $\alpha, \beta, \ldots$*),*
- $\rightarrow \subseteq Q \times \mathtt{L}_\delta \times Q$ *is a relation (called* transition relation*).*

An initial TLTS *is a tuple* $(Q, \mathtt{L}_\delta, \rightarrow, q_0)$*, where* $(Q, \mathtt{L}_\delta, \rightarrow)$ *is a TLTS, and* $q_0 \in Q$ *is the* initial configuration*.*

We are now ready to define the semantics of TA. For the moment, we will assume a TA which runs *in isolation*, i.e. without interacting with other TA.

Definition 5.2.3 (Semantics of timed automata) *Let* $A = (Loc, Loc^u, l_0, E, I)$ *be a TA over a set of clocks* $\mathbb{C}$*. We define the initial TLTS* $[\![A]\!]$ *as follows:*

$$[\![A]\!] \;=\; (Loc \times \mathbb{V}[\mathbb{C}],\ \{\tau\} \cup \mathbb{R}_{\geq 0},\ \rightarrow,\ (l_0, \nu_0))$$

where the transition relation $\rightarrow$ *is specified by the following two rules:*

1. $(l, \nu) \xrightarrow{\ell_\tau} (l', \nu')$ *if* $(l, \ell_\tau, g, R, l') \in E \wedge \nu \in [\![g]\!] \wedge \nu' = \nu[R] \in [\![I(l')]\!]$
2. $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ *if* $\nu + \delta \in [\![I(l)]\!] \wedge l \notin Loc^u$

A configuration (l, ν) *is* reachable *when* $(l_0, \nu_0) \rightarrow^* (l, \nu)$*.*

We now comment the two rules in Definition 5.2.3:

- rule 1 allows to perform an action. This does not involve any time delay, but after the action has been performed, all the clocks in R are reset to zero. The action is permitted if the guard g on the edge is satisfied by the current clock valuation, and the invariant $I(l')$ of the target location is satisfied *after* the clock reset.
- rule 2 allows time to pass, provided that this does not break the invariant of the current (not urgent) location. Note that all the clocks progress with the same pace.

5.3 Networks of timed automata

We now introduce *networks* of TA, i.e. sets of TA which can interact by synchronizing on channels via input/output actions.

Definition 5.3.1 (Networks of TA) *A* network of TA *is a finite set of TA (over a given set of clocks* $\mathbb{C}$*). We denote with* $A_1 \mid \cdots \mid A_n$ *the network composed by* $A_1, \ldots, A_n$*.*

We now define the semantics of networks of TA. The configurations of a network $A_1 \mid \cdots \mid A_n$ are tuples of the form $(l_1, \ldots, l_n, \nu)$, where $l_1, \ldots, l_n$ represent the current locations in the automata, and ν is an evaluation of *all* the clocks in the network. Similarly to the semantics of isolated TA (Definition 5.2.3), the semantics of a network is a TLTS, whose states are configurations, and labels are internal actions, delays, and channels (for synchronisations).

Definition 5.3.2 (Semantics of networks of TA) *Let $A_i = (Loc_i, Loc_i^u, l_0^i, E_i, I_i)$ be TA over a set of clocks $\mathbb{C}$, for $i \in 1..n$. We define the behaviour of the network $N = A_1 \mid \cdots \mid A_n$ as the initial TLTS $[\![N]\!] = (Q, \mathtt{L}_\delta, \to, q_0)$, where:*

- $Q = Loc_1 \times \cdots \times Loc_n \times \mathbb{V}[\mathbb{C}]$
- $\mathtt{L}_\delta = \mathcal{C} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$
- $\to$ *is the relation defined in Figure 5.1*
- $q_0 = (l_0^1, \ldots, l_0^n, \nu_0)$.

$$\frac{(l_k, \tau, g, R, l'_k) \in E_k \qquad \nu \in [\![g]\!] \qquad \nu[R] \in [\![\textstyle\bigwedge_{i \in \{1..n\} \setminus \{k\}} I_i(l_i)]\!] \qquad \nu[R] \in [\![I_k(l'_k)]\!]}{(l_1, \ldots, l_k, \ldots, l_n, \nu) \xrightarrow{\tau} (l_1, \ldots, l'_k, \ldots, l_n, \nu[R])} \text{[Tau]}$$

$$\frac{\nu + \delta \in [\![\textstyle\bigwedge_{i \in \{i..n\}} I_i(l_i)]\!] \qquad l_1 \notin Loc_1^u \qquad \ldots \qquad l_n \notin Loc_n^u}{(l_1, \ldots, l_n, \nu) \xrightarrow{\delta} (l_1, \ldots, l_n, \nu + \delta)} \text{[Delay]}$$

$$\frac{\begin{array}{l}(l_h, \mathtt{a!}, g_h, R_h, l'_h) \in E_h \\ (l_k, \mathtt{a?}, g_k, R_k, l'_k) \in E_k\end{array} \qquad \nu \in [\![g_h \wedge g_k]\!] \qquad \nu[R_h \cup R_k] \in \begin{array}{c}[\![\textstyle\bigwedge_{i \in \{1..n\} \setminus \{h,k\}} I_i(l_i)]\!] \\ \cap\, [\![I_h(l'_h)]\!] \,\cap\, [\![I_k(l'_k)]\!]\end{array}}{(l_1, \ldots, l_h, \ldots, l_k, \ldots, l_n, \nu) \xrightarrow{\mathtt{a}} (l_1, \ldots, l'_h, \ldots, l'_k, \ldots, l_n, \nu[R_h \cup R_k])} \text{[Com]}$$

Figure 5.1: Transition relation of networks of TA.

Rule [Tau] states that one of the TA can take an internal edge, provided that its guard is satisfied, and that the invariants of *all* the locations in the target configuration are satisfied. Rule [Delay] allows time to elapse, at the same pace for all TA in the network, provided that the invariants at the current locations of all TA are satisfied after the delay. Rule [Com] allows two TA to synchronize on a channel `a`, provided that the following three conditions hold: (i) at their current locations, the two TA can fire complementary actions (such as `a!` and `a?`); (ii) the clock valuation satisfies the guards of those edges; (iii) the invariants of the target locations are satisfied after the clock reset.

If the current locations of a state have no outgoing edges, then such state is called *success*, while a state is called *deadlock* if it is not success and no *action*-transitions are possible (neither in the current clock valuation, nor in the future).

Definition 5.3.3 (Deadlock freedom) *A location of a TA is called success when it has not outgoing edges. Let s be a network state. Then, s is called success when all its locations are success. We say that s is* deadlock *whenever: (i) s is not* success*, and (ii)* $\nexists \delta \geq 0,\ \mathtt{a}_\tau \in \mathtt{A} \cup \{\tau\} : s \xrightarrow{\delta}_N \xrightarrow{\mathtt{a}_\tau}_N$*. A network is* deadlock-free *if none of its reachable states is deadlock.*

5.4 Regions and zones

This section illustrates the concepts of *regions* and *zones*, two fundamental tools for the algorithmic verification of clock-based timed systems. Regions are sets of clock valuations, parametric w.r.t. a natural number d (choosen as the maximal constant in the TA under analysis) which give rise to a *finite* partition of $\mathbb{V}$ [3]. The equivalence relation induced by this partiton (seen as a set of equivalence classes) is particularly meaningful in the context of timed automata. Indeed, given two clock valuations ν, ν' belonging to the same region, the configurations of any TA (l, ν) and (l, ν') are time abstract bisimilar [2] and satisfy the same TCTL assertions [8]. Without going into details, regions can be used to transform the infinite states LTS semantics of TA into untimed finite states LTSs, called *region graphs*, amenable of verification through standard model checking techniques. However, region graphs are usually too fine grained for the purposes of algorithmic analysis, as the number of regions grows exponentially in the number of clocks used. Thus, state of the art model checkers of TA are based on different tools, namely zones, which are basically unions of regions. The use of zones gives a more compact rappresentation of the symbolic state space, leading to more efficient (both in time and space) algorithms for the verification of TA, in the average case [32].

We now formally define the concepts discussed above. Regions were first introduced in [3], but without considering diagonal contraints. The definition below is an adaptation of the one in [68], which further introduced zones (called regions there) for the first time.

Definition 5.4.1 (Zones and regions) *For all $d \in \mathbb{N}$, a d-*zone[1] *is a set of clock valuations $\mathcal{K}$ such that there exists a guard g, with all constants bounded by d, and $[\![g]\!] = \mathcal{K}$. A d-region is a minimal nonempty d-zone, i.e. a nonempty d-zone $\mathcal{K}$ such that, for any nonempty d-zone $\mathcal{K}'$, $\mathcal{K}' \subseteq \mathcal{K} \implies \mathcal{K} = \mathcal{K}'$.*

Zones are closed under some useful operations on set of clocks valuations, *past* and *inverse reset*. The past and the inverse reset of a set $\mathcal{K}$ are composed by those clock valuations which will be part of $\mathcal{K}$, respectively, after some (possibly none) idling and after resetting a given set of clocks.

Definition 5.4.2 (Past and inverse reset) *For all sets $\mathcal{K}$ of clock valuations, the set of clock valuations $\downarrow \mathcal{K}$ (the* past *of $\mathcal{K}$) and $\mathcal{K}[T]^{-1}$ (the* inverse reset *of $\mathcal{K}$) are defined as:*

$$\downarrow \mathcal{K} \ = \ \{\nu \,|\, \exists \delta \geq 0 : \nu + \delta \in \mathcal{K}\} \qquad\qquad \mathcal{K}[T]^{-1} \ = \ \{\nu \,|\, \nu[T] \in \mathcal{K}\}$$

[1]The term "zone" is often referred to convex sets of clocks, while our definition includes non-convex ones.

Part II

Honesty in contract-oriented computing

Chapter 6

Contracts

6.1 Session types as contracts

In this chapter we describe the contract model used in this thesis, namely an adaptation of binary session types [70]. We introduce a novel operational semantics (Definition 6.1.2) for them, more suited to contract-oriented computing with respect to the standard semantics of [10]. In particular, our semantics simplifies the threatment of monitoring and culpability, which are crucial notions of contract-oriented computing. Session types 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 6.1.3) ensures their progress, until a successful state is reached. We show that compliance can be decided by model-checking finite-state systems (Lemma 6.1.6), 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 (Lemma 6.1.9). Furthermore, a participant can always recover from culpability in a bounded number of steps (Lemma 6.1.10).

6.1.1 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. `int`, `bool`, `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 6.1.1 (Contracts) *Contracts are* binary session types*, i.e. terms defined by the grammar:*

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

where

1. *the index set $\mathcal{I}$ is finite,*

2. *the labels* $\mathtt{a}_i$ *in the prefixes of each summation are pairwise distinct, and*

3. *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 $\mathtt{a}!\mathtt{T} \,.\, c' = \mathtt{a}_i!\mathtt{T}_i \,.\, c_i$. Note that prefixing operator . has more precedence than sum operators $\oplus$ and $+$. Hereafter, we will omit the `unit` sort and the trailing occurrences of $\mathtt{0}$, and we will only consider contracts without free occurrences of recursion variables X.

6.1.2 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 with, respectively, contracts c and d is modelled by the contract configuration $\mathsf{A} : c \mid \mathsf{B} : d$. We specify in Definition 6.1.2 an operational semantics of such contract configurations, where the two participants alternate in firing actions. To do that, we extend the syntax of Definition 6.1.1 with the term $\mathtt{rdy}\ \mathtt{a?v} \,.\, c$, which models a participant ready to input a value `v` in a branch with label `a`, and then to continue as c. In other words, `rdy a?v` acts as a one-position buffer shared between the two participants.

Definition 6.1.2 (Semantics of contract configurations) *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* $\mathtt{rdy}\ \mathtt{a?v} \,.\, c$*. We postulate that at most one occurrence of* `rdy` *is present, and if so* `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 contract configurations is modelled by the labelled transition relation* $\twoheadrightarrow$*, which is the smallest relation closed under the rules in Figure 6.1 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* $\gamma_0 \twoheadrightarrow \gamma_1 \twoheadrightarrow \cdots$*.*

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 `rdy a?v`, while discarding all the other branches; the transition label $\mathsf{A} : \mathtt{a!v}$ models A selecting the branch with label `a`, and passing value `v`. Participant B can then perform his action in the subsequent step, by rule [Rdy]. Note that this semantics causes an *alternation* between

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

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

Figure 6.1: Semantics of contract configurations (symmetric rules for B actions omitted)

output and input actions, not present in other semantics of sessions types (for instance, the one in [10]). 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 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 B's turn in receiving.

6.1.3 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 0.

Definition 6.1.3 (Compliance) *We say that a configuration γ is* safe *iff either:*

$$\textit{(i)} \quad \gamma = \mathsf{A} : \bigoplus_{i \in I} \mathtt{a}_i\mathtt{!T}_i \,.\, c_i \mid \mathsf{B} : \sum_{j \in J} \mathtt{a}_j\mathtt{?T}_j \,.\, d_j \quad \textit{with} \quad \emptyset \neq I \subseteq J$$

$$\textit{or} \quad \textit{(ii)} \quad \gamma = \mathsf{A} : \mathtt{rdy\ a?v}.\, c \mid \mathsf{B} : d$$

$$\textit{or} \quad \textit{(iii)} \quad \gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$$

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

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

We observe that the notion of compliance in Definition 6.1.3 is equivalent to that of *progress* in [18, 9, 24]. This can be proved as in [16], 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 [16]).

Example 6.1.4 *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* a*, then γ will take a transition to $\mathsf{A} : c_1 \mid \mathsf{B} :$ $\mathtt{rdy\ a?v}.\, d_1$, for some* v*. Suppose instead that* A *chooses* b*, which is not offered by* B *in his external choice. In this case, for all* v*, $\gamma \not\xrightarrow{\mathsf{A}:\mathtt{b!v}}\!\!\!\!\rightarrow\!\!\!\!\rightarrow$, and indeed γ is* not *safe according to Definition 6.1.3. Therefore, c and d are* not *compliant.*

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

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

Figure 6.2: Semantics of value-abstract contract configurations (symmetric rules for B actions omitted)

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

Lemma 6.1.5 *For all contracts c, there exists some d such that $c \bowtie d$.*

Proof. See appendix A.1 on page 104.

Definition 6.1.3 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 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 7.1). Therefore, for the sake of checking compliance we can consider an alternative semantics of contracts, where we abstract from values (Figure 6.2). 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 rdy. For instance, $\mathtt{a!T}.\, c$ is abstracted as $(\mathtt{a},\mathtt{T})\mathtt{!}.\, \alpha^\star(c)$, while $\mathtt{rdy\ a?v}.\, c$ 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 Appendix A.2. In the following we will range over vale-abstract contracts and configurations with the same symbols of contracts and configurations, namely $c, d, \ldots$ and $\gamma, \gamma', \ldots$. When refering to non value-abstract contracts, we will often write *concrete* contracts.

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 6.1.6 for some implementation details).

Lemma 6.1.6 *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. See appendix A.1 on page 106.

Example 6.1.7 (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:*

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

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

$$\begin{aligned} c_{\mathsf{B}} &= \texttt{rec}\, Z \,.\, \big(\texttt{addToCart!int} \,.\, Z \;\oplus\; \texttt{checkout!} \,.\, c'_{\texttt{pay}}\big) \\ \textit{where}\ c'_{\texttt{pay}} &= \texttt{pay!string} \,.\, \big(\texttt{ok?} \;+\; \texttt{no?} \,.\, \texttt{cancel!} \;+\; \texttt{abort?}\big) \end{aligned}$$

The above contracts are not compliant: in fact, c_{B} can choose to perform the branch checkout *before doing an* addToCart. *Instead, the contract $c'_{\mathsf{B}} = \texttt{addToCart!int} \,.\, c_{\mathsf{B}}$ is compliant with c_{A}.*

6.1.4 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 Lemma 6.1.10), a participant can always move out from this state. Note that definitions and results of this section refer to not value abstract contracts, altough they can be easily lifted to the value abstract case.

Definition 6.1.8 (Culpability) *A participant* A *is culpable in γ ($\mathsf{A} \frown \gamma$ in symbols) iff $\gamma \xrightarrow{\mathsf{A}:\mathsf{aov}} \twoheadrightarrow$ for some $\mathtt{a}, \mathtt{v}$ and $\circ \in \{!, ?\}$. When* A *is* not *culpable in γ we write $\mathsf{A} \smile \gamma$.*

Lemma 6.1.9 below establishes that, when starting from a configuration of compliant contracts, exactly one participant is culpable in all subsequent configurations (the culpable participant can change during the interaction). The only exception is $\mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$, which represents a successfully terminated interaction, where nobody is culpable.

Lemma 6.1.9 (Unique culpable) *Let $c \bowtie d$. If $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow^* \gamma$, then either $\gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$, or there exists a unique culpable in γ.*

Proof. See appendix A.1 on page 106.

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

Lemma 6.1.10 (Contractual exculpation) *Let $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$, with $c \bowtie d$, and let $\gamma \twoheadrightarrow^* \gamma'$. Then:*

1. *$\gamma' \not\twoheadrightarrow \implies \mathsf{A} \smile \gamma'$ and $\mathsf{B} \smile \gamma'$*

$$\textit{2. } \mathsf{A} \frown \gamma' \implies \forall \gamma'' : \gamma' \twoheadrightarrow \gamma'' \implies \begin{cases} \mathsf{A} \smile \gamma'', \textit{ or} \\ \forall \gamma''' : \gamma'' \twoheadrightarrow \gamma''' \implies \mathsf{A} \smile \gamma''' \end{cases}$$

Proof. Item *1* follows directly from Definition 6.1.8.
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 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 $\gamma'' \xrightarrow{\mathsf{A:b!v}} \gamma'''$ for some b and v, then γ''' will have the form $\mathsf{A} : c''' \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{b?v}.\, d'''$, for some c''', d'''. Therefore, $\mathsf{A} \smile \gamma'''$.

Item (1) of Lemma 6.1.10 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.

6.1.5 Kripke structure semantics of contracts

In this section we briefly describe a simple Kripke structure semantics of contract configurations, useful for assessing properties of them through model checking, thus enabling participants to infer informations about the sessions they are engaged with. This knowledge can then be used to make decisions at runtime. Properties are expressed as LTL formulae, with atomic propositions composed of value abstract labels. The inuition is that an atomic proposition holds in those states which are reached firing that label (in the value abstract semantics). In order to achieve the above, we simply append the last fired label to each contract configuration (or the dummy label ε to the initial state).

More formally, fix $\mathbb{AP}$ as the set whose elements are terms of the form $(\mathtt{a}, \mathtt{T})\circ$, where a is a branch label, T is a sort, and $\circ \in \{\mathtt{!}, \mathtt{?}\}$. We let $\ell, \ell', \ldots$ range over $\mathbb{AP} \cup \{\varepsilon\}$, where ε is a special label not in $\mathbb{AP}$.

Definition 6.1.11 *We define the kripke structure* $\mathrm{TS} = (\Sigma, \to, L)$ *as follows:*

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

$$\begin{array}{lcll} (\ell, \gamma) & \to & (\ell, \gamma) & \textit{if } \gamma \not\twoheadrightarrow \\ (\ell, \gamma) & \to & ((\mathtt{a}, \mathtt{T})\circ, \gamma') & \textit{if } \gamma \xrightarrow{\mathsf{A:a}\circ\mathtt{v}} \gamma' \textit{ and } \mathtt{v} : \mathtt{T} \end{array}$$

- *the labelling function* $L : \Sigma \to 2^{\mathbb{AP}}$ *is defined as:*

$$L((\ell, \gamma)) = \begin{cases} \{\ell\} & \text{if } \ell \in \mathbb{AP}; \\ \emptyset & \text{otherwise.} \end{cases}$$

Then, we write $\gamma \vdash \phi$ *whenever* $(\varepsilon, \gamma) \models \phi$ *holds in LTL.*

The last line of Definition 6.1.11 explains the use of the symbol ε: it is a dummy label, needed to decorate the initial state of a trace. Note that we have added a self loop to terminal states. This transformation is standard, as terminal states are forbidden in Kripke Structures.

6.1.6 Maude implementation

In this section we describe our executable specification of (value abstract) contracts in Maude, aimed to be a practical tool for compliance and culpability checking. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [52]. 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 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 6.1.1; 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`). They are interpreted by the Maude engine as simplification rules (applied left-to-right). In order to be computationally meaningfull, they are required to be Church-Rosser and terminating.

For instance, the involution co(·) 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 [61], and with the Maude search capabilities.

The semantics of contract configurations is an almost-literal translation of that in Figure 6.1 using rewriting rules. A minor difference is that, since transitions in rewritings are unlabelled, we decorate states with labels, as done in [97]. Concretely, this is done by specifying the labelled transition $\gamma \overset{\mu}{\twoheadrightarrow}_\star \gamma'$ 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 [97]: 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 6.1.11). 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 6.1.6, 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 A : c | 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 predicate `{l} g |= --A-->>` holds whenever `{l} g` has been reached by some transitions of `A` (i.e. `l` is of the form `A says a`). Therefore, according to Definition 6.1.8, participant `A` is culpable in a configuration `g` if proposition `--A-->>` is satisfied by an immediate successor state of `g`. However, the value-abstract semantics of contracts (Figure 6.2) guarantees that this holds for all immediate successor state or none fo them. So, 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). The Maude implementation of the above is as follows:

```
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 .
```

We conclude this section with an example.

Example 6.1.12 *The contracts from Example 6.1.7 can be specified as follows:*

```
ops Y Z : -> Var .
ops addToCart checkout pay ok no abort cancel : -> ActName [ctor] .
ops CA CPay : -> UniContract .
ops CB CB' : -> UniContract .

eq CPay = rec Y . (pay ? string . ( ok ! unit . 0 (+)
                                    no ! unit . Y (+)
                                    abort ! unit . 0) +
                  cancel ? unit . 0) .

eq CA = addToCart ? int . (rec Z . (addToCart ? int . Z +
                                    checkout ? unit . CPay)) .

eq CB = rec Z . ( addToCart ! int . Z (+)
                  checkout ! unit . pay ! string . (ok ? unit . 0 +
                                                    no ? unit .
                                                      (cancel ! unit . 0) +
                                                    abort ? unit . 0) ) .

eq CB' = addToCart ! int . CB .
```

In order to check compliance, we can ask Maude to reduce the term `CA |X| CB` *(to a boolean). This is accomplished through the following command:*

```
red CA |X| CB .
```

The Maude output is:

```
==========================================
reduce in ONLINE-STORE : CA |X| CB .
rewrites: 180 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
==========================================
```

The contracts are not compliant, as anticipated in Example 6.1.7. We can check compliance between `CA` *and* `CB'` *(which are compliant, according to Example 6.1.7) in a similar way:*

```
red CA |X| CB' .
```

The Maude output is:

```
==========================================
reduce in ONLINE-STORE : CA |X| CB' .
rewrites: 751 in 0ms cpu (1ms real) (~ rewrites/second)
result Bool: true
==========================================
```

Chapter 7

Contract oriented computing and CO_2

7.1 Contract-oriented computing & Honesty

In this chapter we introduce the contract-oriented computing paradigm and the related notion of honesty. We model agents and systems in the process calculus CO_2 [28, 27, 22], which we instantiate with the contracts introduced in Section 6.1. In Section 7.1.1 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 7.1.2 we define the semantics of CO_2, and in Section 7.1.3 we formalise the concept of honesty.

7.1.1 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 compositions of them through 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 $\vec{u}, \vec{v}, \ldots$ to range over sequences of variables/names, and $\vec{e}$ to range over sequences of expressions. To make symbols lookup easier, we have summarised the syntactic categories and some notation in Table 7.1.

Definition 7.1.1 *The syntax of CO_2 is defined as follows:*

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

If $\vec{u} = u_0, \ldots, u_n$, we will use $(\vec{u})S$ and $(\vec{u})P$ 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:

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

Table 7.1: Summary of notation.

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

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

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

Figure 7.1: Structural equivalence for CO_2.

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

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 $\mathtt{do}_u\, \mathtt{a?}x : \mathtt{T}.\, P$, 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 $\mathtt{do}_u\, \mathtt{a?}x.\, \mathtt{do}_x\, \mathtt{b!v}$ and $(x)\, \mathtt{do}_u\, \mathtt{a!}x$.

Free *session* names/variables in a prefix are defined as follows: $\mathrm{fnv}(\tau) = \emptyset$, and $\mathrm{fnv}(\mathtt{tell}\downarrow_u c) = \{u\} = \mathrm{fnv}(\mathtt{do}_u\,\mathtt{a!}\,e) = \mathrm{fnv}(\mathtt{do}_u\,\mathtt{a?}x : \mathtt{T})$. 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 $\mathrm{X}(x_1,\ldots,x_j) \stackrel{\mathrm{def}}{=} P$ such that $\mathrm{fv}(P) \subseteq \{x_1,\ldots,x_j\} \subseteq \mathcal{V}$. We will sometimes omit the arguments of $\mathrm{X}(\vec{u},\vec{e})$ when they are clear from the context. As usual, we omit trailing occurrences of $\mathbf{0}$ in processes.

7.1.2 Semantics

The operational semantics of CO_2 systems is formalised by the labelled transition relation $\rightarrow$ in Figure 7.2, where we consider processes and systems up-to the congruence relation $\equiv$ in Figure 7.1. The axioms for $\equiv$ are fairly standard — except the bottom-rightmost one, which collects garbage terms possibly arising from variable substitutions. This may happen when telling two different contracts in the same variable (see Section 9.1.3 for a meaningful process which exploits this possibility). The labels μ of the transition relation can be of the following forms: $\mathsf{A} : \pi$ (where π is not $\mathtt{do}$), $\mathsf{A} : \mathtt{if}$, $\mathsf{A} : \mathtt{do}_s\,\mathtt{a!v}$, $\mathsf{A} : \mathtt{do}_s\,\mathtt{a?v}$, or $\mathsf{K} : \mathtt{fuse}$. Participant K plays the role of the *contract broker*. We postulate that participant K does not appair in systems (i.e. systems in the forms $\mathsf{K}[P]$, $s[\mathsf{K} : c \mid \ldots]$, and $\{\downarrow_x c\}_\mathsf{K}$ are forbidden).

We now briefly discuss the rules in Figure 7.2. 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 $\mathtt{v}$, resulting from a [Rdy] transition of the contract configuration γ within session s, and to bind to $\mathtt{v}$ 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 6.1.11). 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.:

1. the syntactic constraints required in Section 7.1.1 are respected;
2. the guards in conditionals have sort $\mathtt{bool}$;

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}:\tau} \mathsf{A}[P \mid Q] \quad [\textsc{Tau}]$$

$$\mathsf{A}[\mathtt{tell} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}:\mathtt{tell}\downarrow_u c} \mathsf{A}[P \mid Q] \mid \{\downarrow_u c\}_{\mathsf{A}} \quad [\textsc{Tell}]$$

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

$$\mathsf{A}[(\mathtt{if}\ e\ \mathtt{then}\ P_{\mathtt{true}}\ \mathtt{else}\ P_{\mathtt{false}}) \mid Q] \xrightarrow{\mathsf{A}:\mathtt{if}} \mathsf{A}[P_{[\![e]\!]} \mid Q] \quad [\textsc{If}]$$

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

$$\frac{\gamma \xrightarrow{\mathsf{A}:\mathtt{a}?\mathtt{v}} \gamma' \qquad \mathtt{v} : \mathtt{T}}{\mathsf{A}[\mathtt{do}_s\, \mathtt{a}?x : \mathtt{T}.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}:\mathtt{do}_s\, \mathtt{a}?\mathtt{v}} \mathsf{A}[P\{^{\mathtt{v}}/_x\} \mid Q] \mid s[\gamma']} \quad [\textsc{Do?}]$$

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

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

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

Figure 7.2: Reduction semantics of CO_2.

3. 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 7.1.2 (Online store) *Recall c_{A} from Example 6.1.7. A possible specification of the store is:*

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

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 $P_{\mathtt{add}}$, wherein it accepts two user choices:* addToCart*, which is followed by a recursive call to $P_{\mathtt{add}}$, and* checkout*, which passes the control to process $P_{\mathtt{pay}}$. 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 $P_{\mathtt{pay}}$, after the payment is received, the store internally chooses (through expression $e(t)$, which implements the store internal policy) whether to accept it or not: in the first case, it terminates; in the second case, it proceeds with $P_{\mathtt{pay}}$, thus allowing the user to retry the payment. In more concrete implementations (see e.g. Example 7.1.8), 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 $d_{\mathsf{B}} =$* `addToCart?int.`c_{B} *as in Example 6.1.7, and let:*

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

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}[P_{\mathsf{A}}{}'] \mid \mathsf{B}[Y]\big) \quad P_{\mathsf{A}}{}' = \mathtt{do}_x\,\mathtt{addToCart?}n.\,P_{\mathtt{add}}(x,n)\\
&\to (s)\ \big(\mathsf{A}[P_{\mathsf{A}}{}'\{^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}[P_{\mathsf{A}}{}'\{^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_{\mathtt{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 c'_{\mathsf{A}} = \mathtt{rec}\,Z.\cdots\\
&\to^*(s)\ \big(\mathsf{A}[P_{\mathtt{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 c'_{\mathsf{B}} = \mathtt{pay!string}.\cdots\\
&\to^*(s)\ \big(\mathsf{A}[P_{\mathtt{pay}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\,\mathtt{pay!}\,cc.\,\mathtt{do}_s\,\mathtt{ok?}] \mid s[\mathsf{A} : c_{\mathtt{pay}} \mid \mathsf{B} : c'_{\mathsf{B}}]\big)\\
&\to (s)\ \big(\mathsf{A}[P_{\mathtt{pay}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\,\mathtt{ok?}] \mid s[\mathsf{A} : \mathtt{rdy}\,\mathtt{pay?}cc\,.\cdots \mid \mathsf{B} : c''_{\mathsf{B}}]\big) \quad c''_{\mathsf{B}} = \mathtt{ok?} + \cdots\\
&\to (s)\ \big(\mathsf{A}[P_{\mathtt{ack}}(s,51)] \mid \mathsf{B}[\mathtt{do}_s\,\mathtt{ok?}] \mid s[\mathsf{A} : c_{\mathtt{ack}} \mid \mathsf{B} : c''_{\mathsf{B}}]\big) \quad c_{\mathtt{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.*

7.1.3 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 $\mathrm{O}_s^{\mathsf{A}}(S)$ 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 $\mathrm{O}_s^{\mathsf{A}}(S)$ to exculpate herself (possibly in two steps, according to Lemma 6.1.10).

Definition 7.1.3 (Obligations) *We define the set of branch labels $\mathrm{O}_s^{\mathsf{A}}(S)$ as:*

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

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

A participant A is *ready* in a system S if, whenever A has some 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. Note that CO_2 semantics (Figure 7.2) guarantees that for every label $\mathsf{A} : \mathtt{do}_s\, \mathtt{a} \circ \mathtt{v}$ fired by any system S, $\mathtt{a} \in \mathrm{O}_s^{\mathsf{A}}(S)$. The set $\mathrm{Rdy}_s^{\mathsf{A}}$ 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 7.1.4 (Readiness) *We define the predicates:*

$$S \xrightarrow{\mathsf{A}:\mathtt{do}_s} \iff \exists \mathtt{a}, \mathtt{v}, \circ, S' : S \xrightarrow{\mathsf{A}:\mathtt{do}_s\ \mathtt{a}\circ\mathtt{v}} S'$$
$$S \xrightarrow{\mathsf{A}:\neq\mathtt{do}_s} S' \iff \exists \pi : S \xrightarrow{\mathsf{A}:\pi} S' \wedge \pi \neq \mathtt{do}_s\, \mathtt{a} \circ \mathtt{v} \textit{ for all } \mathtt{a}, \circ, \mathtt{v}$$

We define the set of systems $\mathrm{Rdy}_s^{\mathsf{A}}$ *as the smallest set such that:*

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

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* $\forall s, S', \vec{u} \ :\ S \equiv (\vec{u})\, S' \ \wedge\ \mathrm{O}_s^{\mathsf{A}}(S') \neq \emptyset \implies S' \in \mathrm{Rdy}_s^{\mathsf{A}}$

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 7.1.5 (Honesty) *We say that:*

1. S is A-free *iff it has no variable binded latent contracts of* A*, opened sessions involving* A*, nor processes of* A
2. P *is* honest in S *iff* $\forall \mathsf{A} : (S$ *is* A*-free* $\wedge\ \mathsf{A}[P] \mid S \to^* S') \implies \mathsf{A}$ *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 7.1.6 (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. $P_6 = (x)\,\mathtt{tell}\downarrow_x \mathtt{a!}.\mathrm{X}(x) \qquad \mathrm{X}(x) \stackrel{def}{=} \tau.\tau.\mathrm{X}(x) + \tau.\mathtt{do}_x\,\mathtt{a!}$
7. $P_7 = (x)\,\mathtt{tell}\downarrow_x \mathtt{a!}.\mathrm{X}(x) \qquad \mathrm{X}(x) \stackrel{def}{=} \mathtt{if\ true\ then}\ \tau.\mathrm{X}(x)\ \mathtt{else}\ \mathtt{do}_x\,\mathtt{a!}$

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

- *P_1 is honest: it is advertising an internal choice between* a! *and* b!*, and then it is doing* a!*.*
- *P_2 is dishonest: if the rightmost τ is fired, then the process cannot do the promised* 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* 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* b!*.*
- *P_4 is honest: note that the branch* 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* a!*, then $\mathtt{do}_y\,\mathtt{b!}$ cannot be reached. Note that P_5 tries to behave correctly (in a naive way), and indeed the failure is caused by the context. However, in our framework, an agent is honest when she takes into account also the possible misbehaviour of the context, which may be unknown at development time.*
- *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 τs, which is enabled infinitely often, will be performed. We now show that P_6 is honest in $S = (y)(\{\downarrow_y \mathtt{a?}\}_{\mathsf{B}})$, 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. $(x,y)(\mathsf{A}[\mathrm{X}(x)] \mid \{\downarrow_x \mathtt{a!}\}_{\mathsf{A}} \mid \{\downarrow_y \mathtt{a?}\}_{\mathsf{B}})$. *Here* A *is vacuously ready, as in the previous item.*
3. $(s)(\mathsf{A}[\mathtt{do}_s\,\mathtt{a!}] \mid s[\mathsf{A} : \mathtt{a!} \mid \mathsf{B} : \mathtt{a?}])$. *Here* A *is ready, because* $\mathsf{A}[\mathtt{do}_s\,\mathtt{a!}] \mid s[\mathsf{A} : \mathtt{a!} \mid \mathsf{B} : \mathtt{a?}] \in \mathrm{Rdy}_s^{\mathsf{A}}$*, by item 1 of Definition 7.1.4.*
4. $(s)(\mathsf{A}[\mathrm{X}(s)] \mid s[\mathsf{A} : \mathtt{a!} \mid \mathsf{B} : \mathtt{a?}])$. *Since* $\mathsf{A}[\mathrm{X}(s)] \mid s[\mathsf{A} : \mathtt{a!} \mid \mathsf{B} : \mathtt{a?}] \xrightarrow{\mathsf{A}:\tau} \mathsf{A}[\mathtt{do}_s\,\mathtt{a!}] \mid s[\mathsf{A} : \mathtt{a!} \mid \mathsf{B} : \mathtt{a?}]$*, which is ready for the previous item, then* A *is ready.*
5. $(s)(\mathsf{A}[\mathbf{0}] \mid s[\mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{rdy\ a?}])$. *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 7.1.6 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 7.1.7 (Online store) *The specification of the online store* P_{A} *in Example 7.1.2 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:*

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

By Definition 7.1.3, we have that $\mathrm{O}_s^{\mathsf{B}}(S'') = \{\mathtt{no}\}$*, and by Definition 7.1.4 it follows that* $S'' \notin \mathrm{Rdy}_s^{\mathsf{B}}$*. Therefore,* B *is* not *ready in* S'*, and so by Definition 7.1.5 we conclude that* Q_{B} *is not honest.*

Example 7.1.8 (Online store with bank) *We now refine the specifications of the online store provided in Example 7.1.2, 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:*

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

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:

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

We now comment the differences with respect to Example 7.1.2. 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* $P_{\texttt{ack}}$*, then the buyer will never receive the expected* `ok`/`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* $P_{\texttt{bank}}$ *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 9.1.1 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 7.1.9 (Unfair scheduler) *Consider the following processes:*

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

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

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

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* $\texttt{do}_s\,\texttt{a!}$ *is enabled), and therefore* P *is honest according to Definition 7.1.5. 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.*

Chapter 8

Verification of honesty

8.1 Model checking honesty

We now address the problem of automatically verifying honesty. As mentioned in Chapter 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 [27] (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 7.1.5 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 8.1.1 we devise a *value abstraction* $\alpha^\star$ of systems (Definition 8.1.1), which replaces each expression e with a special value $\star$. In Theorem 8.1.5 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 `if`-free, then $\alpha^\star(P)$ is honest, too.

2. Second, in Sections 8.1.2 and 8.1.3 we provide a *context abstraction* α_{A} of contracts and systems (Definitions 8.1.6 and 8.1.11, 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 8.1.13 states that this abstraction is sound, too, and it is also complete for `ask`-free processes.

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 8.1.14). 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 8.1.5 we outline an implementation in Maude of our verification technique.

8.1.1 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 8.1.1). We provide *value-abstract systems* with a semantics (Figure 8.1), and in Definition 8.1.4 we accordingly refine the notion of honesty. Theorem 8.1.5 states the soundness (and, for `if`-free processes, also the completeness) of value abstraction.

Definition 8.1.1 (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:*

$$
\begin{array}{rcl}
\alpha^\star_\Gamma(\sum_i \pi_i.P_i) & = & \sum_i \alpha^\star_\Gamma(\pi_i.P_i) \\
\alpha^\star_\Gamma(\pi.P) & = & \begin{cases} \mathtt{do}_u\,(\mathtt{a},\mathtt{T})!.\alpha^\star_\Gamma(P) & \text{if } \pi = \mathtt{do}_u\,\mathtt{a}!e \text{ and } e:\mathtt{T} \text{ in } \Gamma \\ \mathtt{do}_u\,(\mathtt{a},\mathtt{T})?.\alpha^\star_{\Gamma,x:\mathtt{T}}(P) & \text{if } \pi = \mathtt{do}_u\,\mathtt{a}?x:\mathtt{T} \\ \pi.\alpha^\star_\Gamma(P) & \text{if } \pi \neq \mathtt{do} \end{cases} \\
\alpha^\star(\mathrm{X}(\vec{u},\vec{e})) & = & \mathrm{X}(\vec{u},\star) \\
\alpha^\star_\Gamma(\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q) & = & \mathtt{if}\ \star\ \mathtt{then}\ \alpha^\star_\Gamma(P)\ \mathtt{else}\ \alpha^\star_\Gamma(Q) \\
\alpha^\star_\Gamma((u)\,P) & = & (u)\,\alpha^\star_\Gamma(P) \\
\alpha^\star_\Gamma(P \mid Q) & = & \alpha^\star_\Gamma(P) \mid \alpha^\star_\Gamma(Q)
\end{array}
$$

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 8.1.3 for details).

Example 8.1.2 (Online store) *The value-abstract counterpart of the process P_{A} in Example 7.1.2 is:*

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

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

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

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

Figure 8.1: Reduction semantics of value-abstract systems (full set of rules in Appendix A.2.1).

Definition 8.1.3 (Value abstraction of systems) *For all systems S, we define the value abstract system $\alpha^\star(S)$ as follows (see Definition A.1.4 for the specification of $\alpha^\star(\gamma)$):*

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

The semantics of value-abstract systems is given by a set of rules, most of which are similar to those in Figure 7.2. Hence, in Figure 8.1 we only show those which differ substantially from the rules for concrete systems (the full set of rules is provided in Appendix A.2.1). 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 6.1.3), 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 A.2.1).

The notion of honesty for value-abstract systems (Definition 8.1.4 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 approximation of readiness: for instance, if $S = \mathsf{A}[\texttt{if true then do}_s\, \texttt{a!v else } \mathbf{0}]$, then A is ready at s in S according to Definition 7.1.4, while A is *not* $\alpha^\star$-ready at s in $\alpha^\star(S)$ according to Definition 8.1.4, because item 3 is not satisfied.

Definition 8.1.4 (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. $\left(S \xrightarrow{\mathsf{A}:\neq\{\mathtt{do}_s,\mathtt{if}\}}_\star S' \ \wedge\ S' \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}\right) \implies S \in \alpha^\star\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$
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 7.1.5, except for using the value-abstract notion of readiness instead of the concrete one.

Theorem 8.1.5 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 8.1.5 *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.*

Proof. See appendix A.2.1 on page 115.

8.1.2 Context abstraction of contracts

After having defined a value abstraction for CO_2 systems (Section 8.1.1), 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 8.1.6 and Figure 8.2), we will prove its main properties (Theorem 8.1.7), and finally ensure that such an abstraction is sound with respect to the semantics of the CO_2 `ask` primitive (Definition 8.1.8, Definition 8.1.9 and Lemma 8.1.10).

Let γ be a value-abstract contract configuration (Section 6.1.3): the *context-abstraction* $\alpha_{\mathsf{A}}(\gamma)$ (Definition 8.1.6) maintains only the contract of A, while recording whether the opponent contract has a `rdy`.

Definition 8.1.6 (Context abstraction of contracts) *For all value-abstract contract configurations γ, we define the context-abstract contract $\alpha_{\mathsf{A}}(\gamma)$ as follows:*

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

For all participants A, the LTS $\twoheadrightarrow_{\mathsf{A}}$ on context-abstract contracts is defined by the rules in Figure 8.2. 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

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

Figure 8.2: Semantics of context-abstract contracts.

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 A.2.18). In the following, we will say that a contract configuration $\gamma = \mathsf{A} : c \mid \mathsf{B} : d$ is compliant whenever $c \bowtie d$.

Theorem 8.1.7 *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* $\gamma_c : \big(\alpha_{\mathsf{A}}(\gamma_c) = c \implies \exists \gamma_{c'} \,:\, \gamma_c \xrightarrow{\mathsf{A}:a}\!\!\!\!\rightarrow_{\star} \gamma_{c'} \wedge \alpha_{\mathsf{A}}(\gamma_{c'}) = c'\big)$

Proof. See appendix A.2.2 on page 118.

Another desirable property of context abstraction of contracts is that also the satisfaction relation $\vdash_\star$ used in rule [$\alpha^\star$Ask] of the value-abstract semantics of CO_2 (Figure 8.1) is safely abstracted, in the sense formalised in Definition 8.1.8 below. Intuitively, given two relations $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ between context-abstract contracts and formulae (of some temporal logic):

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

In Section 8.1.3 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 8.3). In Definition 8.1.9 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 6.1. Lemma 8.1.10 will then show that these context-abstract relations are sound according to Definition 8.1.8. Recall from Definition 6.1.11 that $\mathbb{AP}$ is the set of terms $(\mathtt{a}, \mathtt{T})\circ$, where $\mathtt{a}$ is a branch label.

Definition 8.1.9 (Context abstraction of entailment) *For all participants* A*, we define the Kripke structure* $\mathrm{TS}_{\mathsf{A}} = (\Sigma, \to, L)$ *as follows:*

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

$$\begin{array}{ll} (\ell, c) \to (a, c') & \textit{if } c \overset{a}{\twoheadrightarrow}_{\mathsf{A}} c' \ \vee \ c \overset{ctx:a}{\twoheadrightarrow}_{\mathsf{A}} c' \\ (\ell, c) \to (\ell, c) & \textit{if } c \not\twoheadrightarrow_{\mathsf{A}} \end{array}$$

- *the labelling function* $L : \Sigma \to 2^{\mathbb{AP}}$ *is defined as:*

$$L((\ell, c)) = \begin{cases} \{\ell\} & \textit{if } \ell \in \mathbb{AP}; \\ \emptyset & \textit{otherwise.} \end{cases}$$

We write $c \vdash \phi$ *whenever* $(\varepsilon, c) \models \phi$ *holds in LTL. Then, we define the relations:*

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

Note that $c \nvdash \neg\phi$ is not equivalent to $c \vdash \phi$ in LTL. Lemma 8.1.10 below guarantees the soundness of our instantiation of $\vdash_{\mathsf{A}}$ and $\vdash_{ctx}$ with respect to Definition 8.1.8. Note that item 1 of Lemma 8.1.10 ensures a stronger condition than item 1 of Definition 8.1.8: this improves the accuracy of the analysis.

Lemma 8.1.10 *For all context-abstract contracts* c *and for all LTL formulae* ϕ*:*

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

Proof. See appendix A.2.3 on page 122.

8.1.3 Context abstraction of systems

After having defined a value abstraction for CO_2 systems (Section 8.1.1), and a context abstraction for contracts (Section 8.1.2), 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 8.1.12), whose properties (Theorem 8.1.13) will allow us to verify CO_2 systems via model checking (Section 8.1.5).

Definition 8.1.11 (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:*

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

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 8.3. Labels of $\rightarrow_{\mathsf{A}}$ are either *ctx* or they have the form $\mathsf{A} : \pi$, where A is the participant in $\rightarrow_{\mathsf{A}}$, and π is a CO_2 prefix.

The rules in Figure 8.3 can be arranged in two groups:

Rules for A. Rule [α-If] is substantially unchanged w.r.t. value-abstract semantics (Figure 8.1). 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 8.1.8 guarantees that such an $\mathtt{ask}\,\phi$ 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 $\mathtt{ask}\,\phi$ to fire a *ctx* transition, whenever c entails ϕ according to a sound context-abstract entailment relation $\vdash_{ctx}$. Item 2 of Definition 8.1.8 ensures that, if $\mathtt{ask}\,\phi$ 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 8.1.12), named *α-honesty*, follows the lines of that of honesty in Definition 7.1.5. As expected, we use a notion of readiness for context-abstract systems (named *α-readiness*): this is just a minor revisitation of Definition 8.1.4, where we use the context-abstract semantics instead of the value-abstract one (see Appendix A.2.3 for details).

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

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

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

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

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

$$\frac{c \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} c'}{s[c] \xrightarrow{ctx}_{\mathsf{A}} s[c']}\ [\alpha\text{-DoCtx}] \qquad S \xrightarrow{ctx}_{\mathsf{A}} S\ [\alpha\text{-Ctx}] \qquad \frac{S \xrightarrow{ctx}_{\mathsf{A}} S'}{(u)S \xrightarrow{ctx}_{\mathsf{A}} (u)S'}\ [\alpha\text{-DelCtx}]$$

Figure 8.3: Reduction semantics of context-abstract contracts and systems (full set of rules in Appendix A.2.3).

Definition 8.1.12 (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.*

Theorem 8.1.13 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 8.1.5), Theorem 8.1.13 is the final step for guaranteeing the soundness (and completeness) of our verification technique (see Section 8.1.5).

Theorem 8.1.13 *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. See appendix A.2.3 on page 131.

We now sketch the proof of Theorem 8.1.13 (full details are available in Appendix A.2.3). Correctness of α-honesty (first part of Theorem 8.1.13) 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 8.1.7). More precisely, to show that P is $\alpha^\star$-honest we have to prove that, for all A-free value-abstract S:

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

By Theorem A.2.24 in Appendix A.2.3 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 A.2.25 in Appendix A.2.3 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 8.1.13) follows by a similar argument. Theorem A.2.28 in Appendix A.2.3 ensures that, in the `ask`-free fragment, each context-abstract transition is mimicked by a value-abstract one. Lemma A.2.23 is then used to prove that $\alpha^\star$-readiness implies α-readiness.

8.1.4 Main result

We now put together the results of Sections 8.1.1 to 8.1.3, to devise a model checking technique for honesty. The main result of Part II 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 8.1.14 *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 8.1.5 and 8.1.13. 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 8.1.15 below we show two counterexamples to completeness, in case the process under observation is not `ask`-free and `if`-free.

8.1.5 Maude implementation

In order to implement α-honesty in Maude, we proceed similarly to Section 6.1.6: 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 A.2.30 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 A.2.30 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

$\tilde{S} \xrightarrow{\mathsf{A:if}}_{\mathsf{A}}$. 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 9.1.2).

```
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 8.1.15 *Consider the following processes:*

$$P_{\texttt{if}} = (x)\,\texttt{tell}\downarrow_x \texttt{a!}.\ \texttt{if true then do}_x\,\texttt{a! else } \mathbf{0}$$
$$P_{\texttt{ask}} = (x)\,\texttt{tell}\downarrow_x \texttt{a?} + \texttt{b?}.\ \left(\texttt{ask}_x\,\square\neg\texttt{b?}.\texttt{do}_x\,\texttt{a?}\ +\ \texttt{ask}_x\,\square\neg\texttt{a?}.\texttt{do}_x\,\texttt{b?}\right)$$

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.*

Chapter 9

Experiments

9.1 Experiments

In this chapter we validate our verification technique through some experiments. We consider five case studies: the online store with bank (Section 9.1.1), a voucher distribution system (Section 9.1.2), a car purchase financed with a loan (Section 9.1.3), an online casino featuring blackjack (Section 9.1.4), and a travel agency (Section 9.1.5). We specify each of these case studies in CO_2, and we (successfully) verify their honesty using our tool. Finally, in Section 9.1.6 we evaluate the performance of our model checking tool.

The full Maude implementation of all our experiments is available at `http://tcs.unica.it/software/co2-maude`.

9.1.1 Online store with bank

Our first experiment concerns the online store with bank introduced in Example 7.1.8. 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 7.1.8 is modified as follows:

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

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.

9.1.2 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:

$$\begin{array}{rcl} c_{\mathsf{B}} & \stackrel{\text{def}}{=} & \texttt{clickPay?.pay?string} \;+\; \texttt{clickVoucher?.} \\ & & (\texttt{reject!.pay?string} \;\oplus\; \texttt{accept!.voucher?string}) \\ c_{\mathsf{V}} & \stackrel{\text{def}}{=} & \texttt{ok?} \;+\; \texttt{no?} \end{array}$$

In [27] a CO_2 process for A was specified as follows:

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

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 [27]. 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 8.1.14) 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:

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

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 8.1.14 we deduce that the P' is honest.

9.1.3 Car loan

In this example, we show how the `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., `loan10k!`), or not (e.g., `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 ok? or no? as an answer from the car dealer), or abort the transaction. Her CO_2 process is the following:

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

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

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

In P, Alice tells 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 $\texttt{ask}_x$: by rule [Ask] on Figure 7.2, 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 loan10K! or 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 loan200K! or abort200K!.

The *first* $\texttt{ask}_x$ 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 $P_{\texttt{used}}(y)$: there, she tries to buy an used car on y, and if the car dealer's answer is ok?, she selects loan10K! on x.

The *second* $\texttt{ask}_x$-guarded branch is similar — except that its $\texttt{ask}_x$ 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 $P_{\texttt{ferrari}}(y)$; there, she tries to buy a Ferrari on y, and if the car dealer's answer is ok?, she selects 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.

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

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 `tell` prefixes of $\mathsf{A}[P]$ are fired, we have a system of the form:

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

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

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

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 7.1. 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 $\mathtt{ask}_x$-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 8.3: this allows the model checker to establish that P is α-honest, and hence honest.

9.1.4 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 (`hit`) or to terminate his turn (`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:

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

Players can choose between taking a card (`hit`) or passing the turn (`stand`). In the first case, the dealer either gives a `card` to the player (and returns to the beginning of the contract), or it notifies that the player `lose`s (or it may `abort` the game). In the second case (`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:

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

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

We specify the dealer as the following process P:

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

The first `tell` in P advertises the contract for the deck of cards. The dealer waits (via the `ask` prefix) that such contract is fused, and then it advertises the contract for the player (with the second `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 `hit` then the dealer asks the deck for the next card, and the control passes to $P_{\texttt{deck}}$. Instead, if the player chooses `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.

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

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.

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

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.

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

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.

9.1.5 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 7.1.8 for a more concrete treatment of payments). Then, the agency may decide whether to commit the transaction or to abort it:

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

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:

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

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

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

In addition to the contracts above, the agency should respect the following constraints:

1. the agency commits the transaction with U iff both the transactions with F and H are committed;
2. A pays the ticket and the hotel reservation only after it has received the payment from U;

3. 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:

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

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.

$$P_{\mathtt{quote}}(x, x_1, x_2, y) \stackrel{\text{def}}{=} P_{\mathtt{quote1}}(x, x_1, x_2, y) + P_{\mathtt{quote1}}(x, x_2, x_1, y) + \tau.\,P_{\mathtt{abort}}(x, x_1, x_2)$$

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

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

In the continuation $P_{\mathtt{quote}}$, the agency receives the quotes from F and H. In the left-most invocation $P_{\mathtt{quote1}}(x_u, x_1, x_2, y)$ the quote from F is received first, while in the right-most 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.

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

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 $P_{\mathtt{pay}}$.

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

In process $P_{\mathtt{pay}}$, 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 $P_{\mathtt{confirm1}}$ it waits the confirmation from F, and after that, in $P_{\mathtt{confirm2}}$ waits the confirmation from H. As above, waiting can always be terminated by a timeout, which is followed by $P_{\mathtt{abort}}$. Finally, in process $P_{\mathtt{commit}}$ the agency commits all the transactions.

The Maude honesty checker correctly determines that P is honest.

Example	Ref.	Rewritings	Avg. time (ms)	Std. dev.
Online store (dishonest)	7.1.8	18478	36.8	1.03
Online store (honest)	9.1.1	223379	147.6	2.27
Voucher (dishonest)	9.1.2	36668	49.8	1.03
Voucher (honest)	9.1.2	51201	54.8	2.69
Car loan	9.1.3	110804	114.8	5.00
Blackjack	9.1.4	125720	140.9	3.14
Travel Agency	9.1.5	15143028	6118	80.95

Table 9.1: Benchmarks for the honesty checker.

9.1.6 Benchmarks

To empirically valuate the effectiness of our verification technique, we have applied the Maude honesty checker on all the case studies in Section 9.1. 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 9.1.

Part III

A timed contract model

Chapter 10

Timed session types

10.1 Timed session types: syntax and semantics

In this chapter we introduce binary timed session types, giving their syntax and semantics.

We refer to Chapter 5 for syntax and semantics of guards, and notation. A TST p models the behaviour of a single participant involved in an interaction (Definition 10.1.1). To give some intuition, we consider two participants, Alice (A) and Bob (B), who want to interact. A advertises an *internal choice* $\bigoplus_i \mathsf{a}_i!\{g_i, R_i\}.p_i$ when she wants to do one of the outputs $\mathsf{a}_i!$ in a time window where g_i is true; further, the clocks in R_i will be reset after the output is performed. Dually, B advertises an *external choice* $\sum_i \mathsf{a}_i?\{g_i, R_i\}.q_i$ to declare that he is available to receive each message a_i in *any instant* within the time window defined by g_i (and the clocks in R_i will be reset after the input).

Definition 10.1.1 (Timed session type) Timed session types $p, q, \ldots$ *are terms of the following grammar:*

$$p \;::=\; \mathbf{1} \;\mid\; \bigoplus_{i\in I} \mathsf{a}_i!\{g_i, R_i\}.p_i \;\mid\; \sum_{i\in I} \mathsf{a}_i?\{g_i, R_i\}.p_i \;\mid\; \mathtt{rec}\, X.\, p \;\mid\; X$$

where (i) the set I is finite and non-empty, (ii) the actions in internal/external choices are pairwise distinct, (iii) recursion is guarded (e.g., we forbid both $\mathtt{rec}\, X.\, X$ *and* $\mathtt{rec}\, X.\, \mathtt{rec}\, Y.\, p$*).*

Except where stated otherwise, we consider TSTs up-to unfolding of recursion. A TST is *closed* when it has no recursion variables. If $q = \bigoplus_{i\in I} \mathsf{a}_i!\{g_i, R_i\}.p_i$ and $0 \notin I$, we write $\mathsf{a}_0!.p_0 \oplus q$ for $\bigoplus_{i\in I\cup\{0\}} \mathsf{a}_i!\{g_i, R_i\}.p_i$ (the same for external choices). True guards, empty resets, and trailing occurrences of the *success state* $\mathbf{1}$ can be omitted.

Example 10.1.2 (Simplified PayPal) *Along the lines of PayPal User Agreement [1], we specify the protection policy for buyers of a simple on-line payment platform, called PayNow (see Section 10.3 for the full version). PayNow helps customers in on-line purchasing, providing protection against seller misbehaviours. In case a buyer has not received what he has paid for, he can open a dispute within* 180 *days from the date the buyer made the payment.*

After opening of the dispute, the buyer and the seller may try to come to an agreement. If this is not the case, within 20 *days, the buyer can escalate the dispute to a claim. However, the buyer must wait at least* 7 *days from the date of payment to escalate a dispute. Upon not reaching an agreement, if still the buyer does not escalate the dispute to a claim within* 20 *days, the dispute is considered aborted. During a claim procedure, PayNow will ask the buyer to provide documentation to certify the payment, within* 3 *days of the date the dispute was escalated to a claim. After that, the payment will be refunded within 7 days. PayNow's agreement is described by the following TST* p*:*

$$
\begin{aligned}
p &= \texttt{pay?}\{t_{pay}\}.\,(\texttt{ok?} + \texttt{dispute?}\{t_{pay} < 180, t_d\}.\,p') && \text{where}\\
p' &= \texttt{ok?}\{t_d < 20\} + \\
&\quad \texttt{claim?}\{t_d < 20 \wedge t_{pay} > 7, t_c\}.\texttt{rcpt?}\{t_c < 3, t_c\}.\texttt{refund!}\{t_c < 7\} + \\
&\quad \texttt{abort?}
\end{aligned}
$$

Semantics

To define the behaviour of TSTs we use *clock valuations*, which associate each clock with its value. The state of the interaction between two TSTs is described by a *configuration* $(p, \nu) \mid (q, \eta)$, where the clock valuations ν and η record (keeping the same pace) the time of the clocks in p and q, respectively. The dynamics of the interaction is formalised as a transition relation between configurations (Definition 10.1.3). This relation describes all and only the *correct* interactions: e.g., we do not allow time passing to make unsatisfiable all the guards in an internal choice, since doing so would prevent a participant from respecting her protocol. In Section 13.1 we will study another semantics, which can also describe the behaviour of *dishonest* participants who do not respect their protocols.

Definition 10.1.3 (Semantics of TSTs) *A* configuration *is a term of the form* $(p, \nu) \mid (q, \eta)$*, where* p, q *are TSTs extended with* committed choices $[\texttt{a!}\{g, R\}]\,p$*. The semantics of TSTs is a labelled relation* $\to$ *over configurations (Figure 10.1), whose labels are either silent actions* τ*, delays* δ*, or branch labels, and where we define the set of clock valuations* $\mathrm{rdy}(p)$ *as:*

$$
\mathrm{rdy}(p) \;=\; \begin{cases} \downarrow \bigcup [\![g_i]\!] & \text{if } p = \bigoplus_{i \in I} \mathtt{a}_i\,!\{g_i, R_i\}\,.\,p_i \\ \mathbb{V} & \text{if } p = \sum \cdots \text{ or } p = \mathbf{1} \\ \emptyset & \text{otherwise} \end{cases}
$$

As usual, we write $p \xrightarrow{\alpha} p'$ *as a shorthand for* $(p, \alpha, p') \in \to$*, with* $\alpha \in \mathtt{L} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$*. We denote with* $\to^*$ *the reflexive and transitive closure of the relation* $\xrightarrow{\tau} \cup \xrightarrow{\delta}$*.*

We now comment the rules in Figure 10.1. The first four rules describe the behaviour of a TST in isolation. Rule [⊕] allows a TST to commit to the branch `a!` of her internal choice, provided that the corresponding guard is satisfied in the clock valuation ν. This results in the term $[\texttt{a!}\{g, R\}]\,p$, which represents the fact that the endpoint has committed to branch

$$(\mathtt{a!}\{g,R\}.\,p \oplus p',\ \nu) \xrightarrow{\tau} ([\mathtt{a!}\{g,R\}]\,p,\ \nu) \qquad \text{if } \nu \in [\![g]\!] \qquad [\oplus]$$

$$([\mathtt{a!}\{g,R\}]\,p,\ \nu) \xrightarrow{\mathtt{a!}} (p,\ \nu[R]) \qquad [!]$$

$$(\mathtt{a?}\{g,R\}.\,p + p',\ \nu) \xrightarrow{\mathtt{a?}} (p,\ \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad [?]$$

$$(p,\ \nu) \xrightarrow{\delta} (p,\ \nu+\delta) \qquad \text{if } \delta > 0 \ \wedge\ \nu+\delta \in \mathrm{rdy}(p) \qquad [\textsc{Del}]$$

$$\frac{(p,\nu) \xrightarrow{\tau} (p',\nu')}{(p,\nu) \mid (q,\eta) \xrightarrow{\tau} (p',\nu') \mid (q,\eta)}\ [\text{S-}\oplus] \qquad \frac{(p,\nu) \xrightarrow{\delta} (p,\nu') \quad (q,\eta) \xrightarrow{\delta} (q,\eta')}{(p,\nu) \mid (q,\eta) \xrightarrow{\delta} (p,\nu') \mid (q,\eta')}\ [\text{S-}\textsc{Del}]$$

$$\frac{(p,\nu) \xrightarrow{\mathtt{a!}} (p',\nu') \quad (q,\eta) \xrightarrow{\mathtt{a?}} (q',\eta')}{(p,\nu) \mid (q,\eta) \xrightarrow{\tau} (p',\nu') \mid (q',\eta')}\ [\text{S-}\tau]$$

Figure 10.1: Semantics of timed session types (symmetric rules omitted).

a! in a specific time instant: this term can only fire a! through rule [!] (which also resets the clocks in R), while time cannot pass. Rule [?] allows an external choice to fire any of its input actions whose guard is satisfied. Rule [Del] allows time to pass; this is always possible for external choices and success term, while for an internal choice we require that at least one of the guards remains satisfiable; this is obtained through the function rdy. The last three rules deal with configurations. Rule [S-$\oplus$] allows a TST to commit in an internal choice. Rule [S-τ] is the standard synchronisation rule *à la* CCS; note that B is assumed to read a message as soon as it is sent, so A never blocks on internal choices. Rule [S-Del] allows time to pass, equally for both endpoints.

Example 10.1.4 *Let* $p = \mathtt{a!} \oplus \mathtt{b!}\{t > 2\}$*, let* $q = \mathtt{b?}\{t > 5\}$*, and consider the computations:*

$$(p,\nu_0) \mid (q,\eta_0) \xrightarrow{7}\xrightarrow{\tau} ([\mathtt{b!}\{t>2\}]\,,\nu_0+7) \mid (q,\eta_0+7) \xrightarrow{\tau} (\mathbf{1},\nu_0+7) \mid (\mathbf{1},\eta_0+7) \qquad (10.1)$$

$$(p,\nu_0) \mid (q,\eta_0) \xrightarrow{\delta}\xrightarrow{\tau} ([\mathtt{a!}]\,,\nu_0+\delta) \mid (q,\eta_0+\delta) \qquad (10.2)$$

$$(p,\nu_0) \mid (q,\eta_0) \xrightarrow{3}\xrightarrow{\tau} ([\mathtt{b!}\{t>2\}]\,,\nu_0+3) \mid (q,\eta_0+3) \qquad (10.3)$$

The computation in (10.1) *reaches success, while the other two computations reach a deadlock state. In* (10.2)*, p commits to the choice* a! *after some delay δ; at this point, time cannot pass (because the leftmost endpoint is a committed choice), and no synchronisation is possible (because the other endpoint is not offering* a?*). In* (10.3)*, p commits to* b! *after* 3 *time units; here, the rightmost endpoint would offer* b?*, but not in the time chosen by the leftmost endpoint. Note that, were we allowing time to pass in committed choices, then we would have obtained e.g. that* $(\mathtt{b!}\{t>2\},\nu_0) \mid (q,\eta_0)$ *never reaches deadlock — contradicting our intuition that these endpoints should not be considered compliant.*

Note that, even when p and q have shared clocks, the rules in Figure 10.1 ensure that there is no interference between them. For instance, if a transition of (p,ν) resets some clock

t, this has no effect on a clock with the same name in q, i.e. on a transition of $(p, \nu) \mid (q, \eta)$. Thus, w.l.o.g. hereafter we will assume that the clocks in p and in q are disjoint.

10.2 Compliance between TSTs

We extend to the timed setting the standard progress-based compliance between (untimed) session types [10, 18], related to the compliance relations studied for the more general formalism of contracts for web services [75, 49]. If p is compliant with q, then whenever an interaction between p and q becomes stuck, it means that both participants have reached the success state. Intuitively, when two TSTs are compliant and participants behave honestly (according to their TSTs), then the interaction will progress, until both of them reach the success state. We recall that ν_0, η_0 are initial clock valuations, i.e. functions associating 0 to each clock (Definition 5.1.2).

Definition 10.2.1 (Compliance) *We say that* $(p, \nu) \mid (q, \eta)$ *is* deadlock *whenever* (i) *it is not the case that both* p *and* q *are* $\mathbf{1}$*, and* (ii) *there is no* δ *such that* $(p, \nu + \delta) \mid (q, \eta + \delta) \xrightarrow{\tau}$*. We then write* $(p, \nu) \bowtie (q, \eta)$ *whenever:*

$$(p, \nu) \mid (q, \eta) \rightarrow^* (p', \nu') \mid (q', \eta') \quad \textit{implies} \quad (p', \nu') \mid (q', \eta') \textit{ not deadlock}$$

We say that p *and* q *are* compliant *whenever* $(p, \nu_0) \bowtie (q, \eta_0)$ *(in short,* $p \bowtie q$*).*

Note that item (ii) of the definition of deadlock can be equivalently phrased as follows: $(p, \nu) \mid (q, \eta) \not\xrightarrow{\tau}$ (i.e., the configuration cannot do a τ-move in the *current* clock valuation), and there does not exist any $\delta > 0$ such that $(p, \nu) \mid (q, \eta) \xrightarrow{\delta} \xrightarrow{\tau}$.

Example 10.2.2 *The TSTs* $p = \mathtt{a?}\{t < 5\}.\mathtt{b!}\{t < 3\}$ *and* $q = \mathtt{a!}\{t < 2\}.\mathtt{b?}\{t < 3\}$ *are compliant, but* p *is* not *compliant with* $q' = \mathtt{a!}\{t < 5\}.\mathtt{b?}\{t < 3\}$*. Indeed, if* q' *outputs* `a` *at, say, time* 4*, the configuration will reach a state where no actions are possible, and time cannot pass: a deadlock state, according to Definition 10.2.1.*

Example 10.2.3 *Consider a customer of PayNow (Example 10.1.2) who is willing to wait 10 days to receive the item she has paid for, but after that she will open a claim. Further, she will instantly provide PayNow with any documentation required. The customer contract is described by the following TST, which is compliant with PayNow's* p *in Example 10.1.2:*

$$\begin{aligned}\mathtt{pay!}\{t_{pay}\}.(\ & \mathtt{ok!}\{t_{pay} < 10\} \oplus \\ & \mathtt{dispute!}\{t_{pay} = 10\}.\mathtt{claim!}\{t_{pay} = 10\}.\mathtt{rcpt!}\{t_{pay} = 10\}.\mathtt{refund?})\end{aligned}$$

Compliance between TSTs is more liberal than the untimed notion, as it can relate terms which, when cleaned from all the time annotations, would not be compliant in the untimed setting. For instance, the following example shows that a recursive internal choice can be compliant with a *non*-recursive external choice — which can never happen in untimed session types.

Example 10.2.4 *Let* $p = \mathtt{rec}\, X.\big(\mathtt{a!} \oplus \mathtt{b!}\{x \leq 1\}.\, \mathtt{c?}.\, X\big)$, $q = \mathtt{a?} + \mathtt{b?}\{y \leq 1\}.\, \mathtt{c!}\{y > 1\}.\, \mathtt{a?}$. *We have that* $p \bowtie q$. *Indeed, if* p *chooses the output* `a!`, *then* q *has the corresponding input, and they both succeed; instead, if* p *chooses* `b!`, *then it will read* `c?` *when* $x > 1$, *and so at the next loop it is forced to choose* `a!`, *since the guard of* `b!` *has become unsatisfiable.*

Definition 10.2.5 and Lemma 10.2.6 below coinductively characterise compliance between TSTs, by extending to the timed setting the coinductive compliance for untimed session types in [9]. Intuitively, an internal choice p is compliant with q when (i) q is an external choice, (ii) for each output `a!` that p can fire after δ time units, there exists a corresponding input `a?` that q can fire after δ time units, and (iii) their continuations are coinductively compliant. The case where p is an external choice is symmetric.

Definition 10.2.5 *We say* $\mathcal{R}$ *is a* coinductive compliance *iff* $(p, \nu)\mathcal{R}(q, \eta)$ *implies:*

1. $p = \mathbf{1} \iff q = \mathbf{1}$
2. $p = \bigoplus_{i \in I} \mathtt{a}_i!\{g_i, R_i\}.\, p_i \implies \nu \in \mathrm{rdy}(p) \ \wedge\ q = \sum_{j \in J} \mathtt{a}_j?\{g_j, R_j\}.\, q_j \ \wedge$
   $\forall \delta, i:\ \nu + \delta \in [\![g_i]\!] \implies \exists j : \mathtt{a}_i = \mathtt{a}_j \wedge \eta + \delta \in [\![g_j]\!] \wedge (p_i, \nu + \delta[R_i])\mathcal{R}(q_j, \eta + \delta[R_j])$
3. $p = \sum_{j \in J} \mathtt{a}_j?\{g_j, R_j\}.\, p_j \implies \eta \in \mathrm{rdy}(q) \ \wedge\ q = \bigoplus_{i \in I} \mathtt{a}_i!\{g_i, R_i\}.\, q_i \ \wedge$
   $\forall \delta, i:\ \eta + \delta \in [\![g_i]\!] \implies \exists j : \mathtt{a}_i = \mathtt{a}_j \wedge \nu + \delta \in [\![g_j]\!] \wedge (p_j, \nu + \delta[R_j])\mathcal{R}(q_i, \eta + \delta[R_i])$

Lemma 10.2.6 $p \bowtie q \iff \exists \mathcal{R} \textit{coinductive compliance} : (p, \nu_0)\mathcal{R}(q, \eta_0)$

Proof. See appendix B.1 on page 135.

The following theorem establishes decidability of compliance. To prove it, we reduce the problem of checking $p \bowtie q$ to that of model-checking deadlock freedom in a network of timed automata constructed from p and q. The translation of TSTs into timed automata is quite involved, and is explained in detail in Section 12.1. We anticipate here the statement about decidabiliy for presentationl reasons.

Theorem 10.2.7 *Compliance between TSTs is decidable.*

Proof. Straightforward consequence of Theorem 12.1.12 in Section 12.1 and by the decidability of deadlock freedom in timed automata [3, 95].

10.3 Case study: Paypal User Agreement

As a case study, we formalise as a TST (part of) the "protection for buyers" section of the PayPal User Agreement [1], which regulates the interaction between Paypal and buyers in trouble during online purchases. When a buyer has not received the item they have paid for (`inr`), or if they have received something significantly different from what was described (`snad`), they can open a *dispute*. The dispute can be opened within 180 days ($t_{pay} < 180$) of the payment date (`pay`). After opening the dispute, the buyer and the seller may try

to solve the problem, or it might be the case that the item finally arrives; otherwise, if an agreement (ok) is not found within 20 days ($t_{inr} < 20$), the buyer can escalate the dispute to a claim (claimINR,claimSNAD). However, in case of an item not received, the buyer must wait at least 7 days from the date of payment to escalate the dispute ($t_{pay} > 7$). Upon not reaching an agreement, if still the buyer does not escalate the dispute to a claim within 20 days ($t_{pay} > 20$), PayPal will close the dispute (close).

During the claim process, PayPal may require the buyer to provide documentation to support the claim, for instance receipts (rcpt) or photos (photo), and the buyer must comply in a *timely manner* to what they are required to do. For SNAD claims, if the claim is accepted, PayPal may require the buyer to ship the item back to the Seller, to PayPal, or to a third party and to provide proof of delivery. In case the item is counterfeit, the item will be destroyed (destroy) and not shipped back to the seller(sendBack). After that, the buyer will be refunded. In some cases, the buyer is not eligible for a refund (notEligible).

We can formalise this agreement as the following TST:

$$
\begin{aligned}
&\texttt{pay?}\{\texttt{true}, t_{pay}\}.(\ \texttt{ok?}\\
&\quad + \texttt{inr?}\{t_{pay} < 180, t_{inr}\}.(\texttt{ok?}\{t_{inr} < 20\} + \ \texttt{close?}\{t_{pay} \geq 20\}\\
&\qquad\quad + \texttt{claimINR?}\{t_{inr} < 20 \wedge t_{pay} > 7, t_c\}.\texttt{rcpt?}.(\texttt{refund!} \oplus \texttt{notEligible!})\\
&\quad + \texttt{snad?}\{t_{pay} < 180, t_{snad}\}.(\texttt{ok?}\{t_{snad} < 20\} + \ \texttt{close?}\{t_{pay} \geq 20\}\\
&\qquad\quad + \texttt{claimSNAD?}\{t_{snad} < 20, t_c\}.\texttt{photo?}.\\
&\qquad\quad\ (\ \texttt{sendBack!}.\texttt{ackSendBack?}.\texttt{refund!} \oplus \ \texttt{destroy!}.\texttt{ackDestroy?}.\texttt{refund!} \oplus \texttt{notEligible!}))
\end{aligned}
$$

Let us consider a possible buyer Alice, who wants to see if she may entrust PayPal for her transactions. Alice is willing to wait 10 days to receive the item she has paid for, but after that she will open a claim. She will readily provide PayPal with every documentation they may need in order to issue the refund. In case she receives an item significantly different from what she has paid for, she will complain to PayPal by opening a claim as soon as the item is received. Alice will timely comply to do whatever PayPal requires (either to destroy the item or to send it back) in order to be refunded. Alice's requirements can be formalised as the following TST:

$$
\begin{aligned}
&\texttt{pay!}\{\texttt{true}, t_{pay}\}.(\ \texttt{ok!}\{t_{pay} < 10\}\\
&\quad \oplus \texttt{inr!}\{t_{pay} = 10\}.\texttt{claimINR!}\{t_{pay} = 10\}.\texttt{rcpt!}\{t_{pay} = 10\}.(\texttt{refund?} + \texttt{notEligible?})\\
&\quad \oplus \texttt{snad!}\{t_{pay} < 10, t_{snad}\}.\texttt{claimSNAD!}\{t_{snad} = 0\}.\texttt{photo!}\{t_{snad} = 0\}.\\
&\qquad (\ \texttt{sendBack?}\{t_c\}.\texttt{ackSendBack!}\{t_c < 3\}.\texttt{refund?}\\
&\qquad + \texttt{destroy?}\{t_c\}.\texttt{ackDestroy!}\{t_c < 3\}.\texttt{refund?} + \texttt{notEligible?}\ ))
\end{aligned}
$$

Alice's and PayPal's TSTs are compliant, according to Definition 10.2.1. However, we can see that PayPal's TST lacks some important details: what does it means *timely comply to what is required*? And, most importantly: how long will it take for a buyer to be refunded? Without a deadline on the refund! action, Alice may possibly wait forever.

Chapter 11

Admissibility of a compliant and subtyping

11.1 Admissibility of a compliant

In the untimed setting, each session type p admits a compliant, i.e. there exists some q such that $p \bowtie q$. For instance, we can compute q by simply swapping internal choices with external ones (and inputs with outputs) in p (this q is called the *canonical dual* of p in some papers [67]). A naïve attempt to extend this construction to TSTs can be to swap internal with external choices, as in the untimed case, and leave guards and resets unchanged. This construction does not work as expected, as shown by the following example.

Example 11.1.1 *Consider the following TSTs:*

$$p_1 = \mathtt{a!}\{x \leq 2\}.\,\mathtt{b!}\{x \leq 1\} \qquad p_2 = \mathtt{a!}\{x \leq 2\} \oplus \mathtt{b!}\{x \leq 1\}.\,\mathtt{a?}\{x \leq 0\}$$
$$p_3 = \mathtt{rec}\, X\,.\,\mathtt{a?}\{x \leq 1 \wedge y \leq 1\}.\,\mathtt{a!}\{x \leq 1, \{x\}\}.\,X$$

The TST p_1 is not compliant with its naïve dual $q_1 = \mathtt{a?}\{x \leq 2\}.\,\mathtt{b?}\{x \leq 1\}$: even though q_1 can do the input `a?` *in the required time window, p_1 cannot perform* `b!` *if* `a!` *is performed after 1 time unit. For this very reason, no TST is compliant with p_1. Note instead that $q_1 \bowtie \mathtt{a!}\{x \leq 1\}.\,\mathtt{b!}\{x \leq 1\}$, which is* not *its naïve dual. In p_2, a similar deadlock situation occurs if the* `b!` *branch is chosen, and so also p_2 does not admit a compliant. The reason why p_3 does not admit a compliant is more subtle: actually, p_3 can loop until the clock y reaches the value 1; after this point, the guard $y \leq 1$ can no longer be satisfied, and then p_3 reaches a deadlock.*

To establish when a TST admits a compliant, we define a kind system which associates to each p a set of clock valuations $\mathcal{K}$ (called *kind of* p). The kind of a TST is unique, and each closed TST is kindable (Theorem 11.1.4). If p has kind $\mathcal{K}$, then there exists some q such that, for all $\nu \in \mathcal{K}$, the configuration $(p, \nu) \mid (q, \nu)$ never reaches a deadlock (Theorem 11.1.6). Also the *vice versa* holds: if, for some q, $(p, \nu) \mid (q, \nu)$ never reaches a deadlock, then $\nu \in \mathcal{K}$

$$\Gamma \vdash \mathbf{1} : \mathbb{V} \qquad \text{[T-1]}$$

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash \sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i : \bigcup_{i\in I} {\downarrow}\left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1}\right)} \qquad \text{[T-+]}$$

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i : \left(\bigcup_{i\in I} {\downarrow}[\![g_i]\!]\right) \setminus \left(\bigcup_{i\in I} {\downarrow}([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1})\right)} \qquad \text{[T-}\oplus\text{]}$$

$$\Gamma, X : \mathcal{K} \vdash X : \mathcal{K} \qquad \text{[T-Var]}$$

$$\frac{\Gamma, X : \mathcal{K} \vdash p : \mathcal{K}'}{\Gamma \vdash \mathtt{rec}\, X\,.\,p : \bigcup \{\mathcal{K}_0 \mid \exists \mathcal{K}_1 :\ \Gamma, X : \mathcal{K}_0 \vdash p : \mathcal{K}_1 \wedge \mathcal{K}_0 \subseteq \mathcal{K}_1\}} \qquad \text{[T-Rec]}$$

Figure 11.1: Kind system for TSTs.

(Theorem 11.1.8). Therefore, p admits a compliant whenever the initial clock valuation ν_0 belongs to $\mathcal{K}$. We give a constructive proof of the correctness of the kind system, by showing a TST $\mathrm{co}(p)$ which we call the *canonical compliant* of p.

Definition 11.1.2 (Kind system for TSTs) *Kind judgements $\Gamma \vdash p : \mathcal{K}$ are defined in Figure 11.1, where Γ is a partial function which associates kinds to recursion variables.*

Rule [T-1] says that the success TST $\mathbf{1}$ admits a compliant in every ν: indeed, $\mathbf{1}$ is compliant with itself. The kind of an exernal choice is the union of the kinds of its branches (rule [T-+]), where the kind of a branch is the past of those clock valuations which satisfy both the guard and, after the reset, the kind of their continuation. Internal choices are dealt with by rule [T-$\oplus$], which computes the difference between the union of the past of the guards and a set of error clock valuations. The error clock valuations are those which can satisfy a guard but not the kind of its continuation. Rule [T-Var] is standard. Rule [T-Rec] looks for a kind which is preserved by unfolding of recursion (hence a fixed point).

Example 11.1.3 *Recall $p_1 = \mathtt{a}!\{x \le 2\}.\mathtt{b}!\{x \le 1\}$, $q_1 = \mathtt{a}?\{x \le 2\}.\mathtt{b}?\{x \le 1\}$, and $p_2 = \mathtt{a}!\{x \le 2\} \oplus \mathtt{b}!\{x \le 1\}.\mathtt{a}?\{x \le 0\}$ from Example 11.1.1. We have the following kinding derivations:*

$$\dfrac{\dfrac{\vdash \mathbf{1} : \mathbb{V}}{\vdash \mathtt{b}!\{x \le 1\} : {\downarrow}[\![x \le 1]\!] \setminus {\downarrow}([\![x \le 1]\!] \setminus \mathbb{V}) = [\![x \le 1]\!] \setminus \emptyset = [\![x \le 1]\!]}\ \text{[T-}\oplus\text{]}}{\vdash p_1 : {\downarrow}[\![x \le 2]\!] \setminus {\downarrow}([\![x \le 2]\!] \setminus [\![x \le 1]\!]) = [\![x \le 2]\!] \setminus [\![x \le 2]\!] = \emptyset}\ \text{[T-}\oplus\text{]}$$

As noted in Example 11.1.1, intuitively p_1 has no compliant; this will be asserted by Theorem 11.1.8 below, as a consequence of the fact that $\nu_0 \notin \emptyset$.

$$\dfrac{\dfrac{\vdash \mathbf{1} : \mathbb{V}}{\vdash \mathtt{b}?\{x \le 1\} : {\downarrow}([\![x \le 1]\!] \cap \mathbb{V}) = [\![x \le 1]\!]}\ \text{[T-+]}}{\vdash q_1 : {\downarrow}([\![x \le 2]\!] \cap [\![x \le 1]\!]) = [\![x \le 1]\!]}\ \text{[T-+]}$$

From Example 11.1.1, q_1 has compliants, and indeed $\nu_0 \in [\![x \le 1]\!]$.

$$
\begin{array}{rcll}
\mathrm{co}_\Gamma(\mathbf{1}) & = & \mathbf{1} & \\
\mathrm{co}_\Gamma\big(\sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i\big) & = & \bigoplus_{i\in I} \mathtt{a}_i!\{g_i \wedge \mathcal{K}_i[T_i]^{-1}, T_i\}\,.\,\mathrm{co}_\Gamma(p_i) & \text{if } \Gamma \vdash p_i : \mathcal{K}_i \\
\mathrm{co}_\Gamma\big(\bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i\big) & = & \sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,\mathrm{co}_\Gamma(p_i) & \\
\mathrm{co}_\Gamma(X) & = & X & \text{if } \Gamma(X) \text{ defined} \\
\mathrm{co}_\Gamma(\mathtt{rec}\,X\,.\,p) & = & \mathtt{rec}\,X\,.\,\mathrm{co}_{\Gamma\{\mathcal{K}/X\}}(p) & \text{if } \Gamma \vdash \mathtt{rec}\,X\,.\,p : \mathcal{K}
\end{array}
$$

Figure 11.2: Canonical compliant of a TST.

$$
\dfrac{\vdash \mathbf{1} : \mathbb{V} \qquad \dfrac{\vdash \mathbf{1} : \mathbb{V}}{\vdash \mathtt{a}!\{x \le 0\} : \downarrow [\![x \le 0]\!] \cap \mathbb{V} = [\![x \le 0]\!]}\ [\text{T-}+]}{\vdash p_2 : \big(\downarrow [\![x \le 2]\!] \cup \downarrow [\![x \le 1]\!]\big) \setminus \big(\downarrow [\![x \le 2]\!] \setminus \mathbb{V}) \cup \downarrow [\![x \le 1]\!] \setminus [\![x \le 0]\!]\big) = \mathcal{K}}\ [\text{T-}\oplus]
$$

where $\mathcal{K} = [\![(x > 1) \wedge (x \le 2)]\!]$. *From Example 11.1.1,* p_2 *has no compliant. However, since* $\mathcal{K}$ *is non-empty, Theorem 11.1.6 guarantees that there exists* q *such that* $(p_2, \nu) \bowtie (q, \nu)$*, for all clock valuations* $\nu \in \mathcal{K}$.

The following theorem states that *every* closed TST is kindable, as well as uniqueness of kinding. We stress that being kindable does not imply admitting a compliant: this holds if and only if the initial clock valuation ν_0 belongs to the kind. Note that uniqueness of kinding holds at the *semantic* level, but the same kind can be represented syntactically in different ways. In Section 11.2 we show that uniqueness of kinding may be obtained also at the *syntactic* level, by representing kinds as guards in normal form [32].

Theorem 11.1.4 (Uniqueness of kinding) *For all* p *and* Γ *with* $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$*, there exists unique* $\mathcal{K}$ *such that* $\Gamma \vdash p : \mathcal{K}$.

Proof. See appendix B.2 on page 136.

By exploiting the kind system we define the *canonical compliant* of kindable TSTs. Roughly, we turn internal choices into external ones (without changing guards nor resets), and external into internal, changing the guards so that the kind of continuations is preserved.

Definition 11.1.5 (Canonical compliant) *For all kinding environments* Γ *and* p *kindable in* Γ*, we define the TST* $\mathrm{co}_\Gamma(p)$ *in Figure 11.2. We will abbreviate* $\mathrm{co}_\Gamma(p)$ *as* $\mathrm{co}(p)$ *when* $\Gamma = \emptyset$.

The following theorem states the soundness of the kind system: is particular, if the initial clock valuation ν_0 belongs to the kind of p, then p admits a compliant.

Theorem 11.1.6 (Soundness) *If* $\vdash p : \mathcal{K}$ *and* $\nu \in \mathcal{K}$*, then* $(p, \nu) \bowtie (\mathrm{co}(p)\,, \nu)$.

Proof. See appendix B.2 on page 140.

$$\Gamma \vdash_I \mathbf{1} : \mathbb{V} \quad \text{[I-1]}$$

$$\frac{\Gamma \vdash_I p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash_I \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\} \,.\, p_i : \left(\bigcup_{i\in I}\downarrow [\![g_i]\!]\right) \setminus \left(\bigcup_{i\in I}\downarrow ([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1})\right)} \text{[I-}\oplus\text{]}$$

$$\Gamma, X : \mathcal{K} \vdash_I X : \mathcal{K} \quad \text{[I-Var]}$$

$$\frac{\Gamma \vdash_I p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash_I \sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\} \,.\, p_i : \bigcup_{i\in I}\downarrow \left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1}\right)} \text{[I-}+\text{]}$$

$$\Gamma \vdash_I \mathtt{rec}\, X \,.\, p : \textstyle\bigcap_{i\geq 0} \hat{F}^i_{\Gamma,X,p}(\mathbb{V}) \quad \text{[I-Rec]} \quad \textit{where } \hat{F}_{\Gamma,X,p}(\mathcal{K}) = \mathcal{K}' \text{ iff } \Gamma, X : \mathcal{K} \vdash_I p : \mathcal{K}'$$

Figure 11.3: Kind inference rules.

Example 11.1.7 *Recall* $q_1 = \mathtt{a}?\{x \leq 2\}.\,\mathtt{b}?\{x \leq 1\}$ *from Example 11.1.1. We have* $\mathrm{co}(q_1) = \mathtt{a}!\{x \leq 1\}.\,\mathtt{b}!\{x \leq 1\}$. *Since* $\vdash q_1 : \mathcal{K} = [\![x \leq 1]\!]$ *and* $\nu_0 \in \mathcal{K}$, *by Theorem 11.1.6 we have that* $q_1 \bowtie \mathrm{co}(q_1)$, *as anticipated in Example 11.1.1.*

The following theorem states the kind system is also *complete*: in particular, if p admits a compliant, then the initial clock valuation ν_0 belongs to the kind of p.

Theorem 11.1.8 (Completeness) *If* $\vdash p : \mathcal{K}$ *and* $\exists q, \eta.\ (p, \nu) \bowtie (q, \eta)$, *then* $\nu \in \mathcal{K}$.

Proof. See appendix B.2 on page 142.

Compliance is not transitive, in general: however, Theorem 11.1.10 below states that transitivity holds when passing through the canonical compliant.

Lemma 11.1.9 *For all* p, q, ν, η *and* p', ν' *such that* $\vdash p' : \mathcal{K}$ *and* $\nu' \in \mathcal{K}$:

$$(p, \nu) \bowtie (p', \nu') \ \wedge\ (\mathrm{co}(p'), \nu') \bowtie (q, \eta) \implies (p, \nu) \bowtie (q, \eta)$$

Proof. See appendix B.2 on page 142.

Theorem 11.1.10 *If* $p \bowtie p'$ *and* $\mathrm{co}(p') \bowtie q$, *then* $p \bowtie q$.

Proof. Straightforward after Lemma 11.1.9.

11.2 Computability of the canonical compliant

In this section we show that the canonical compliant construction is computable. To prove this, we first show the decidability of kind inference (Theorem 11.2.4). This fact is not completely obvious, because the cardinality of the set of kinds is $2^{2^{\aleph_0}}$; however, the kinds constructed by our inference rules can always be represented syntactically by guards.

The following lemma recalls some well known facts about functions over complete lattices and their fixed points. These facts will be used to prove that kind inference is decidable.

We start by showing that the kind obtained by rule [T-Rec] in Figure 11.1 is the greatest fixed point (over the lattice $(2^{\mathbb{V}}, \subseteq)$) of the functional $F_{\Gamma,X,p}$ defined as follows:

$$F_{\Gamma,X,p}(\mathcal{K}) = \mathcal{K}' \quad \text{whenever} \quad \Gamma, X : \mathcal{K} \vdash p : \mathcal{K}' \tag{11.1}$$

where we will omit the subscript from $F_{\Gamma,X,p}$ when clear from the context.

Note that, by uniqueness of kinding, F is a function; further, Theorem 11.1.4 ensures that F is total when $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma) \cup \{X\}$. The following lemma states that F is monotonic. Then, the Knaster-Tarski fixed point theorem [93] ensures that [T-Rec] yields the gfp of F.

Lemma 11.2.1 *The function $F_{\Gamma,X,p}$ is monotonic, for all Γ, X, p with $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma) \cup \{X\}$.*

Proof. See appendix B.3 on page 143.

To prove decidability of the kinding relation, we introduce in Figure 11.3 an alternative set of rules, with judgements of the form $\Gamma \vdash_I p : \mathcal{K}$. Lemma 11.2.3 below shows that the kind relations $\vdash$ and $\vdash_I$ are equivalent. The new set of rules can be exploited as a kind inference algorithm: in particular, rule [I-Rec] allows for computing the kind of a recursive TST $\mathtt{rec}\, X\,.\, p$ by evaluating the non-increasing sequence $\hat{F}^i_{\Gamma,X,p}(\mathbb{V})$ until it stabilizes. The following lemma states that the kinds inferred through the relation $\vdash_I$ are zones. By [32] it follows that the kind of a TST can always be represented as a guard.

Lemma 11.2.2 *If $\Gamma \vdash_I p : \mathcal{K}$, for some Γ which maps variables to zones, then $\mathcal{K}$ is a zone.*

Proof. Easy, by induction on the structure of p and inspection of the kind inference rules.

Lemma 11.2.3 *For all Γ mapping variables to zones, and for all p such that $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$:*

$$\Gamma \vdash p : \mathcal{K} \iff \Gamma \vdash_I p : \mathcal{K}$$

Proof. By structural induction on p. The only non-trivial case is when $p = \mathtt{rec}\, X\,.\, p'$.
For the ($\Rightarrow$) direction, suppose that:

$$\frac{\Gamma, X : \mathcal{K}'' \vdash p' : \mathcal{K}'''}{\Gamma \vdash \mathtt{rec}\, X\,.\, p' : \bigcup \{\mathcal{K}_0 \mid \exists \mathcal{K}_1 \supseteq \mathcal{K}_0 :\ \Gamma, X : \mathcal{K}_0 \vdash p' : \mathcal{K}_1\} = \mathcal{K}}\ \text{[T-Rec]}$$

and recall that $\mathcal{K} = \mathrm{gfp}(F_{\Gamma,X,p})$. By the induction hypothesis, for all zones $\mathcal{Z}$:

$$F_{\Gamma,X,p'}(\mathcal{Z}) = \hat{F}_{\Gamma,X,p'}(\mathcal{Z}) \tag{11.2}$$

Hence, Lemma 11.2.1 implies that $\hat{F}_{\Gamma,X,p'}$ is monotonic (on the lattice of zones). Since this lattice is finite, then $\hat{F}_{\Gamma,X,p'}$ is cocontinuous (item (a) of Lemma 2.2.5). By (11.2) and Theorem 11.1.4, $\hat{F}^i_{\Gamma,X,p}(\mathcal{K}_0)$ is defined for all $\mathcal{K}_0$ and i. Hence, by rule [I-Rec] and item (c) of Lemma 2.2.5:

$$\Gamma \vdash_I \mathtt{rec}\, X\,.\, p' : \textstyle\bigsqcap_{i \geq 0} \hat{F}^i_{\Gamma,X,p'}(\mathbb{V}) \;=\; \hat{\mathcal{K}} \;=\; \mathrm{gfp}(\hat{F}_{\Gamma,X,p'}) \tag{11.3}$$

To conclude that $\mathcal{K} = \hat{\mathcal{K}}$ it is enough to show that $\mathrm{gfp}(F_{\Gamma,X,p'}) = \hat{\mathcal{K}}$. By (11.2) and (11.3) it follows that $\hat{\mathcal{K}}$ is a fixed point of $F_{\Gamma,X,p'}$, and so by definition of gfp, $\hat{\mathcal{K}} \subseteq \mathrm{gfp}(F_{\Gamma,X,p'})$. By (11.2) and item (b) of Lemma 2.2.5 we conclude that $\mathrm{gfp}(F_{\Gamma,X,p'}) \subseteq \hat{\mathcal{K}}$.
For the ($\Leftarrow$) direction, suppose that by rule [I-Rec] we have:

$$\Gamma \vdash_I \mathtt{rec}\, X \,.\, p' : \textstyle\bigsqcap_{i\geq 0} \hat{F}^i_{\Gamma,X,p'}(\mathbb{V})$$

Since $\bigsqcap_{i\geq 0} \hat{F}^i_{\Gamma,X,p'}(\mathbb{V})$ is defined, the induction hypothesis gives $\Gamma, X : \mathbb{V} \vdash p' : F_{\Gamma,X,p'}(\mathbb{V})$, hence the premise of rule [T-Rec] is satisfied. We can conclude with the same argument as in the previous case.

Theorem 11.2.4 *Kind inference is decidable.*

Proof. By Lemma 11.2.3, kinds of closed TSTs can be inferred by the rules in Figure 11.3. All the operations between zones used in Figure 11.3 (except $\hat{F}$) are well know to be computable [68, 32]. By finiteness of the set of zones, also $\hat{F}$ is computable.

The following theorem states that the canonical compliant construction is computable.

Theorem 11.2.5 *The function* $\mathrm{co}(\cdot)$ *is computable.*

Proof. It follows by the fact that all the operations in Definition 11.1.5 are computable.

11.3 Subtyping

In this section we study the semantic subtyping preorder, which is a sound and complete model of the Gay and Hole subtyping relation (in reverse order) for untimed session types [10]. Intuitively, p is subtype of q if every q' compliant with q is compliant with p, too.

Definition 11.3.1 (Semantic subtyping) *For all TSTs p, we define the set $(p,\nu)^{\bowtie}$ as:*

$$(p,\nu)^{\bowtie} \;=\; \{(q,\eta) \,|\, (p,\nu) \bowtie (q,\eta)\}$$

Then, we define the preorder $\sqsubseteq$ between TSTs as follows:

$$p \sqsubseteq q \qquad \textit{whenever} \qquad (p,\nu_0)^{\bowtie} \supseteq (q,\nu_0)^{\bowtie}$$

The following theorem states that, as in the untimed setting, the canonical compliant of p is the maximum (i.e., the most "precise") in the set of TSTs compliant with p.

Theorem 11.3.2 $q \bowtie p \implies q \sqsubseteq \mathrm{co}(p)$

Proof. Assume that $q \bowtie p$ and $\vdash p : \mathcal{K}$, and let $(r,\eta) \in (\mathrm{co}(p)\,,\nu_0)^{\bowtie}$. Since $(q,\nu_0) \bowtie (p,\nu_0)$, by Theorem 11.1.8 we obtain $\nu_0 \in \mathcal{K}$. Therefore, by $(\mathrm{co}(p)\,,\nu_0) \bowtie (r,\eta)$ and Lemma 11.1.9 we conclude that $(q,\nu_0) \bowtie (r,\eta)$. So, $(r,\eta) \in (q,\nu_0)^{\bowtie}$, from which the thesis follows.

The following theorem reduces the problem of deciding $p \sqsubseteq q$ to that of checking compliance between p and $\mathrm{co}(q)$, when q admits a compliant (otherwise $(q,\eta_0)^{\bowtie} = \emptyset$, so q is supertype of every p). Since compliance, the canonical compliant construction, and checking the admissibility of a compliant are all decidable (Theorem 10.2.7, Theorem 11.2.5), this implies decidability of subtyping.

Theorem 11.3.3 *For all TSTs p, q:*

$$p \sqsubseteq q \iff \begin{cases} p \bowtie \mathrm{co}(q) & \text{if } q \text{ admits a compliant} \\ \texttt{true} & \text{otherwise} \end{cases}$$

Proof. If q does not admit a compliant then the thesis is trivial, so assume that $(q, \eta_0)^{\bowtie} \neq \emptyset$. For the ($\Rightarrow$) direction, assume that $p \sqsubseteq q$. Since q admits a compliant, by Theorem 11.1.8 there exists some $\mathcal{K}$ such that $\vdash q : \mathcal{K} \ni \nu_0$. By Theorem 11.1.6, it follows that $\mathrm{co}(q) \bowtie q$. Then, by Definition 11.3.1 we conclude that $p \bowtie \mathrm{co}(q)$. For the ($\Leftarrow$) direction, assume that $p \bowtie \mathrm{co}(q)$, and let q' be such that $q' \bowtie q$. Then, by Theorem 11.1.10 we conclude that $q' \bowtie p$, from which the thesis follows.

Theorem 11.3.4 (Decidability of subtyping) *Subtyping between TSTs is decidable.*

Proof. Immediate consequence of Theorem 10.2.7 and Theorem 11.3.3.

Unlike in the untimed case, the canonical compliant construction is not involutive, i.e. $\mathrm{co}(\mathrm{co}(p))$ is not equal to p, in general. However, p and $\mathrm{co}(\mathrm{co}(p))$ are still strongly related, as they have the same set of compliant TSTs, in every ν in the kind of p (Theorem 11.3.5). By Definition 11.3.1, this implies that $p \sqsubseteq\sqsupseteq \mathrm{co}(\mathrm{co}(p))$, for all kindable p.

Theorem 11.3.5 *Let $\vdash p : \mathcal{K}$ and $\nu \in \mathcal{K}$. Then: $(p, \nu)^{\bowtie} = (\mathrm{co}(\mathrm{co}(p)), \nu)^{\bowtie}$.*

Proof. Suppose that $\vdash p : \mathcal{K}$ and $\nu \in \mathcal{K}$. By Theorem 11.1.6:

$$(p, \nu) \bowtie (\mathrm{co}(p), \nu) \tag{11.4}$$

Assume that $\vdash \mathrm{co}(p) : \mathcal{K}'$. By (11.4) and Theorem 11.1.8 it follows that $\nu \in \mathcal{K}'$. By repeating the same argument twice, we also obtain that:

$$(\mathrm{co}(p), \nu) \bowtie (\mathrm{co}(\mathrm{co}(p)), \nu) \tag{11.5}$$

$$(\mathrm{co}(\mathrm{co}(p)), \nu) \bowtie (\mathrm{co}(\mathrm{co}(\mathrm{co}(p))), \nu) \tag{11.6}$$

To prove $(p, \nu)^{\bowtie} \subseteq (\mathrm{co}(\mathrm{co}(p)), \nu)^{\bowtie}$, let $(q, \eta) \in (p, \nu)^{\bowtie}$. By applying Lemma 11.1.9 on $(q, \eta) \bowtie (p, \nu)$ and on (11.5), we obtain $(q, \eta) \bowtie (\mathrm{co}(\mathrm{co}(p)), \nu)$.
To prove $(p, \nu)^{\bowtie} \supseteq (\mathrm{co}(\mathrm{co}(p)), \nu)^{\bowtie}$, let $(q, \eta) \in (\mathrm{co}(\mathrm{co}(p)), \nu)^{\bowtie}$. By applying Lemma 11.1.9 on (11.4) and (11.6) (and using commutativity of compliance), we obtain:

$$(\mathrm{co}(\mathrm{co}(\mathrm{co}(p))), \nu) \bowtie (p, \nu) \tag{11.7}$$

Finally, by applying Lemma 11.1.9 on $(q, \eta) \bowtie (\mathrm{co}(\mathrm{co}(p)), \nu)$ and on (11.7), we conclude that $(q, \eta) \bowtie (\mathrm{co}(\mathrm{co}(p)), \nu)$.

Chapter 12

Encoding timed session types into timed automata

12.1 Encoding TSTs into Timed Automata

In this chapter we define a semantic-preserving encoding of TSTs into timed automata (Definition 12.1.9), and we exploit it to devise an effective procedure to decide compliance (Theorem 12.1.12).

We start introducing some combinators of TA. The *union* of a set of TA collects all the locations and all the edges; the invariant on a location is the conjunction of all the invariants defined on that location; the initial location is specified as a parameter. The *Idle* pattern creates a TA with only a location, used to model succesful states; the *Pfx* pattern adds a starting location to a TA; finally, the *Br* pattern prefixes a location to a set of TA, using guarded edges.

Definition 12.1.1 (Union) *For all $i \in I$, let $A_i = (Loc_i, Loc_i^u, l_i^0, E_i, I_i)$, and let $l \in \bigcup_i Loc_i$. We define $\bigsqcup_{i\in I}^{l} A_i = (\bigcup_i Loc_i, \bigcup_i Loc_i^u, l, \bigcup_i E_i, I)$, where $I(l_j) = \bigwedge_i I_i(l_j)$ for all $l_j \in \bigcup_i Loc_i$.*

Definition 12.1.2 (Idle pattern) *Let l^0 be a location. Then, $Idle(l^0) = (\{l^0\}, \emptyset, l^0, \emptyset, \emptyset)$.*

Definition 12.1.3 (Prefix pattern) *Let $A = (Loc_1, Loc_1^u, l_1^0, E_1, I_1)$ be a TA, let $\ell_\tau \in \mathtt{L}_\tau$, $l^0 \notin Loc_1$, and $R \subseteq \mathbb{C}$. Then, $Pfx(l^0, \ell_\tau, R, A) = (Loc_1 \cup \{l^0\}, Loc_1^u \cup \{l^0\}, l^0, E, I_1\{^{\mathtt{true}}/_{l^0}\})$, where $E = E_1 \cup \{(l^0, \ell_\tau, \mathtt{true}, R, l_1^0)\}$.*

Definition 12.1.4 (Branch pattern) *For all $i \in I$, let $A_i = (Loc_i, Loc_i^u, l_i^0, E_i, I_i)$, and let $g_i, g \in \mathcal{G}_\mathbb{C}$, $R_i \subseteq \mathbb{C}$, $\ell_{\tau_i} \in \mathtt{L}_\tau$. Then, $Br(l^0, g, \{(\ell_{\tau_i}, g_i, R_i, A_i) \mid i \in I\}) = (Loc, Loc^u, l^0, E, I)$, where: $Loc = \bigcup_i Loc_i \cup \{l^0\}$, $Loc^u = \bigcup_i Loc_i^u$, $E = \bigcup_i E_i \cup \bigcup_i \{(l^0, \ell_{\tau_i}, g_i, R_i, l_i^0)\}$, and $I = \bigcup_i I_i \cup \{(l^0, g)\}$.*

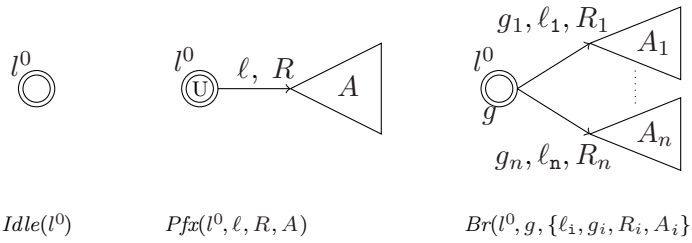

Figure 12.1: Patterns for TA composition, represented as in [60]. Circles denote locations (those marked with U are urgent), and arrows denote edges. Internal actions, `true` guards/invariants and empty resets are left blank. A TA is depicted as a triangle, whose left vertex represents its initial location (a double circle). An arrow from l^0 to triangle A represents an edge from l^0 to the initial location of A.

12.1.1 Defining equations

The first step of our encoding from TSTs to TA is to put TSTs in a normal form where recursive terms $\mathtt{rec}\, X\,.\, p$ are replaced by *defining equations*. This alternative representation of infinite-state processes is common in concurrency theory [83], hence we will defer some standard technicalities to Appendix B.4. In our normal form (called DE-TST) each process is represented as a pair, composed of a recursion variable and a set of defining equations of the form $X_i \triangleq p_i$, where X_i is a recursion variable and p_i is a term where every recursion variable is guarded by exactly one action (Definition 12.1.5).

Definition 12.1.5 (Defining-equation normal form) *A DE-TST is a pair (X, D), where D is a set of defining equations of the form $X' \triangleq p$, where p has the following syntax:*

$$p \;::=\; \mathbf{1} \;\mid\; \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, R_i\}\,.\,X_i \;\mid\; \sum_{i\in I} \mathtt{a}_i?\{g_i, R_i\}\,.\,X_i$$

and (i) the index set I is finite and non-empty, and (ii) the actions in internal/external choices are pairwise distinct

Given a DE-TST (X, D), we denote with $\mathrm{uv}(D)$ the set of recursion variables *used* in D, and with $\mathrm{dv}(D)$ the set of recursion variables *defined* in D. We say that (X, D) is *closed* when $\{X\} \cup \mathrm{uv}(D) \subseteq \mathrm{dv}(D)$, and each $X \in \mathrm{uv}(D)$ is defined exactly once. Every TST p can be translated into a DE-TST $\langle p \rangle_V$ (where V is a set of fresh recursion variables, see Definition B.4.2) in a way that preserves closedness (Lemma B.4.4) and compliance (Lemma 12.1.8). Hereafter, we will always assume closed DE-TST.

To define the semantics of DE-TST, we use a new set $\mathcal{S}$ of terms (Definition 12.1.6). Intuitively, in the state τX the process has performed an internal move (without choosing the branch), while in $[\mathtt{a}!\{g, R\}]\, X$ it has committed to the branch $\mathtt{a}!$.

Definition 12.1.6 (Semantics of DE-TST) *Let $\mathcal{S}$ be a set of terms of the form:*

$$t \;::=\; \tau X \;\mid\; [\mathtt{a}!\{g, R\}]\, X \;\mid\; X \qquad \text{(where } \mathtt{a} \in \mathtt{A},\ g \in \mathcal{G}_{\mathbb{C}}, \text{ and } R \subseteq \mathbb{C})$$

$$\frac{(X \triangleq p) \in D \quad p = \bigoplus \ldots \quad \nu \in \mathrm{rdy}(p)}{(X,\ \nu) \xrightarrow{\tau}_D (\tau X,\ \nu)}\ [\text{DE-}\tau] \qquad \frac{(X \triangleq \mathtt{a!}\{g,R\}.Y \oplus p) \in D \quad \nu \in [\![g]\!]}{(\tau X,\ \nu) \xrightarrow{\tau}_D ([\mathtt{a!}\{g,R\}]Y,\ \nu)}\ [\text{DE-}\oplus]$$

$$\frac{}{([\mathtt{a!}\{g,R\}]Y,\ \nu) \xrightarrow{\mathtt{a!}}_D (Y,\ \nu[R])}\ [\text{DE-!}] \qquad \frac{(X \triangleq \mathtt{a?}\{g,R\}.Y + p) \in D \quad \nu \in [\![g]\!]}{(X,\ \nu) \xrightarrow{\mathtt{a?}}_D (Y,\ \nu[R])}\ [\text{DE-?}]$$

$$\frac{(X \triangleq \textstyle\sum \ldots) \in D \ \vee\ (X \triangleq \mathbf{1}) \in D}{(X,\ \nu) \xrightarrow{\delta}_D (X, \nu+\delta)}\ [\text{DE-Del1}] \qquad \frac{X \triangleq p \in D \quad \nu + \delta \in \mathrm{rdy}(p)}{(\tau X,\ \nu) \xrightarrow{\delta}_D (\tau X,\ \nu+\delta)}\ [\text{DE-Del2}]$$

$$\frac{(s,\nu) \xrightarrow{\tau}_D (s',\nu)}{(s,\nu) \mid (t,\eta) \xrightarrow{\tau}_D (s',\nu) \mid (t,\eta)}\ [\text{DES-}\oplus] \qquad \frac{(s,\nu) \xrightarrow{\delta}_D (s,\nu') \quad (t,\eta) \xrightarrow{\delta}_D (t,\eta')}{(s,\nu) \mid (t,\eta) \xrightarrow{\delta}_D (s,\nu') \mid (t,\eta')}\ [\text{DES-Del}]$$

$$\frac{(s,\nu) \xrightarrow{\mathtt{a!}}_D (s',\nu') \quad (t,\eta) \xrightarrow{\mathtt{a?}}_D (t',\eta')}{(s,\nu) \mid (t,\eta) \xrightarrow{\mathtt{a}}_D (s',\nu') \mid (t',\eta')}\ [\text{DES-}\tau]$$

Figure 12.2: Semantics of DE-TST (symmetric rules omitted).

and let D be a set of defining equations. Then, the relation $\rightarrow_D$ is inductively defined by the set of rules in Figure 12.2, where we use $t, s \ldots$ to range over $\mathcal{S}$.

There is almost a one-to-one correspondence between the rules of $\rightarrow$ and $\rightarrow_D$, aside from the syntactic differences between TST and DE-TST. Rules [DE-Del1] and [DE-Del2] allow time to pass in the same way as [Del] does: they have been split in two only to accommodate with the new term τX. Rule [DE-τ] forces every internal choice to perform an internal step before committing to a branch. However, this is done only if $\nu \in \mathrm{rdy}(p)$, i.e. if at least one of the branches is available. Note that before the internal step has been performed, time cannot pass, hence the only possible move is via [DE-$\oplus$] (so, a sequence of rules [DE-τ] and [DE-$\oplus$] corresponds to rule [$\oplus$] in $\rightarrow$).

Definition 12.1.7 (Compliance for DE-TST) *A state $(X,\nu) \mid (Y,\eta)$ in $\rightarrow_D$ is* deadlock *whenever (i) it is not the case that both $X \triangleq \mathbf{1}$ and $Y \triangleq \mathbf{1}$ are in D, and (ii) there is no δ and no $\mathtt{a}_\tau \in \mathtt{A} \cup \{\tau\}$ such that $(X,\nu+\delta) \mid (Y,\eta+\delta) \xrightarrow{\mathtt{a}_\tau}_D$. We write $(x,\nu) \bowtie_D (y,\eta)$ when:*

$$(x,\nu) \mid (y,\eta) \rightarrow_D^* (x',\nu') \mid (y',\eta') \quad \textit{implies} \quad (x',\nu') \mid (y',\eta') \textit{ not deadlock}$$

and we write $(x,D') \bowtie (y,D'')$ whenever $(x,\nu_0) \bowtie_{D' \cup D''} (y,\eta_0)$.

Lemma 12.1.8 *Let p and q be two closed TSTs with no shared clocks. Let V_1 and V_2 be two sets of recursion variables not occurring in p and q and such that $V_1 \cap V_2 = \emptyset$. Then:*

$$p \bowtie q \iff \langle p \rangle_{V_1} \bowtie \langle q \rangle_{V_2}$$

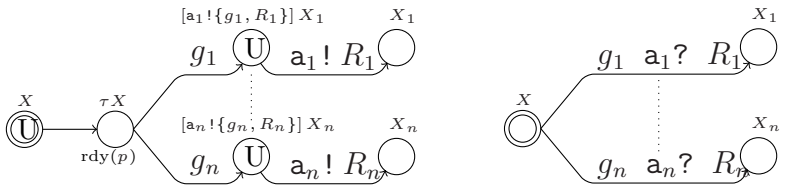

Figure 12.3: Encoding of an internal choice (left) and of an external choice (right).

Proof.(Sketch). Rules in $\rightarrow$ and $\rightarrow_D$ have a strong correspondence: first of all, all the terms in configurations but one are the same, secondly every move but one in one system corresponds to one move in the other. The only exception concerns rule [⊕] in $\rightarrow$ which corresponds to pair ([DE-τ], [DE-⊕]) in $\rightarrow_D$, and vice-versa. Hence, the proof is done by contradiction: if only one of the systems were deadlock and the other not, the other could still move and since rules and configurations are the same, also could the first one, leading to a contradiction.

12.1.2 Encoding DE-TST into TA

In Definition 12.1.9 we transform DE-TST into TA: first, we use the function $[\![\cdot]\!]$ to transform each defining equation into a TA; then, we compose all the resulting TA with the union operator $\bigsqcup$ (introduced in Definition 12.1.1). Our encoding produces a location for every term in $\mathcal{S}$ reachable in $\rightarrow_D$; it creates an edge for each move that can be performed by a TST, so that, in the end, the moves performed by the network associated to (X, D) coincide with the moves of X in $\rightarrow_D$. To avoid some technicalities in proofs, we assume that disjunctions never occur in guards (while they can occur in invariants).

Definition 12.1.9 (Encoding of DE-TST into TA) *For all closed DE-TST (X, D), we define the function $\mathcal{T}(X, D) = \bigsqcup_{d\in D}^{X} [\![d]\!]$, where:*

$$[\![X \triangleq p]\!] = \begin{cases} \mathit{Idle}(X) & \text{if } p = \mathbf{1} \\ \mathit{Br}(X, \mathrm{rdy}(p), \{(\mathtt{a}_i?, g_i, R_i, \mathit{Idle}(X_i))\}_i) & \text{if } p = \sum_{i\in I} \mathtt{a}?\{g_i, R_i\}.X_i \\ \mathit{Pfx}(X, \tau, \emptyset, \mathit{Br}(\tau X, \mathrm{rdy}(p), \{(\tau, g_i, \emptyset, A_i)\}_i), \text{where} \\ A_i = \mathit{Pfx}([\mathtt{a}_i!\{g_i, R_i\}]\, X_i, \mathtt{a}_i!, R_i, \mathit{Idle}(X_i)) & \text{if } p = \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, R_i\}.X_i \end{cases}$$

The encoding of $X \triangleq \mathbf{1}$ produces an *idle* TA (Definition 12.1.2), with a single success location X. The encoding of an *external choice* $X \triangleq \sum_{i\in I} \mathtt{a}_i?\{g_i, R_i\}.X_i$ generates a location X (with `true` invariant), and outgoing edges towards all the locations X_i: these edges have guards g_i, reset sets R_i, and synchronization labels $\mathtt{a}_i?$ (see Figure 12.3, right). Basically, the TA is listening on all its channels $\mathtt{a}_i?$, and since the location X is not urgent, time can pass forever while in there. The encoding of $X \triangleq p$ when p is an *internal choice* $\bigoplus_{i\in I} \mathtt{a}_i!\{g_i, R_i\}.X_i$ is a bit more complex (see Figure 12.3, left). First, we generate an urgent location X, and an edge leading to a non-urgent location τX with invariant $\mathrm{rdy}(p)$.

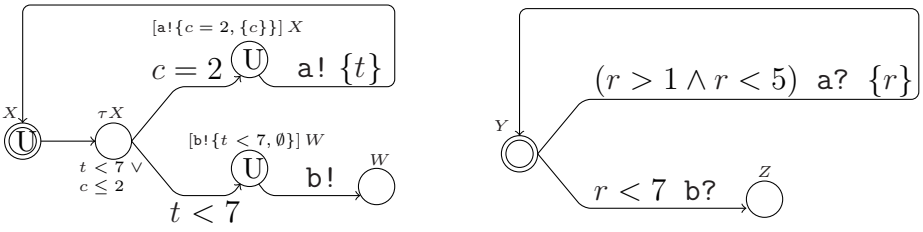

Figure 12.4: Encoding of the TSTs in Example 12.1.10.

Note that, although $\mathrm{rdy}(p)$ is a semantic object (a set of clock valuations), it can always be represented syntactically as a guard [68]. Second, we connect the latter location to i urgent locations (named $[\mathtt{a}_i!\{g_i, R_i\}]\, X_i$) via internal edges with guards g_i; each of these locations is connected to X_i through an edge with reset set R_i and label $\mathtt{a}_i!$. The resulting TA can wait some time before deciding on which branch committing, since time can pass in location τX; however, time passing cannot make all the guards on the branches unsatisfiable, because the invariant $\mathrm{rdy}(p)$ on τX ensures that the location is left on time. As soon as this happens, since the arriving location is urgent, time cannot pass anymore, and a synchronization is performed (if possible). The reason we use both locations X and τX is that, in some executions, all the guards of an internal choice may have *already* expired. In this case, the invariant $\mathrm{rdy}(p)$ on location τX would be false, and the system could not enter it, so preventing a previous action (if any) to be done. To avoid this problem, we have put an extra location (X) before τX.

Since location names are in $\mathcal{S}$, every defined/used variable X has a location called X, in *every* TA which defines/uses it. When the TA obtained from different equations are composed with $\bigsqcup$, locations with the same name collapse, therefore connecting together the call of a recursion variable with its definition.

Example 12.1.10 *Let* $p_1 = \mathtt{rec}\, X_1.\, (\mathtt{a!}\{c = 2, \{c\}\}.X_1 \oplus \mathtt{b!}\{t < 7\})$*, and let* V_1 *be a set of recursion variables not in* p_1*. The DE-TST of* p_1 *is* $\langle p_1 \rangle_{V_1} = (X, D)$*, where:*

$$D \;=\; \{\;\; X \triangleq p,\;\; W \triangleq \mathbf{1} \;\;\} \qquad\qquad \text{where } p = \mathtt{a!}\{c = 2, \{c\}\}.X \oplus \mathtt{b!}\{t < 7\}.W$$

Figure 12.4 (left) shows the TA $\mathcal{T}(X, D)$*. All the locations but* τX *and* W *are urgent; all the invariants are* `true`*, except for* $I(\tau X) = \mathrm{rdy}(p) = \downarrow (\llbracket c = 2 \rrbracket \cup \llbracket t < 7 \rrbracket)$ *(represented by the guard* $c \leq 2 \vee t < 7$*).*

Now, let $q_1 = \mathtt{rec}\, Y_1.\, \mathtt{a?}\{r > 1 \wedge r < 5, \{r\}\}.Y_1 + \mathtt{b?}\{r < 7\}$*, and let* V_2 *be a set of recursion variables not in* q_1*. The DE-TST of* q_1 *is* $\langle q_1 \rangle_{V_2} = (Y, F)$*, where:*

$$F \;=\; \{\;\; Y \triangleq q,\;\; Z \triangleq \mathbf{1} \;\;\} \qquad\qquad \text{where } q = \mathtt{a?}\{r > 1 \wedge r < 5, \{r\}\}.Y + \mathtt{b?}\{r < 7\}.Z$$

Figure 12.4 (right) shows the TA $\mathcal{T}(Y, F)$*. Its locations are* Y *and* Z *(both non-urgent), with invariants* $I(Z) =$ `true` *and* $I(Y) = \mathrm{rdy}(q) = \mathbb{V}$ *(represented by the guard* `true`*).*

Lemma 12.1.11 shows a strict correspondence between the timed LTSs $\rightarrow_D$ and $\rightarrow_N$: matching states are strongly bisimilar. To make explicit that some state q belongs to a LTS $\rightarrow$, we write it as a pair $(q, \rightarrow)$.

Lemma 12.1.11 *Let (X, D') and (Y, D'') be DE-TST such that $\mathrm{dv}(D') \cap \mathrm{dv}(D'') = \emptyset$. Let $N = \mathcal{T}(X, D') \mid \mathcal{T}(Y, D'')$. Then:*

$$((X, \nu_0) \mid (Y, \eta_0), \rightarrow_{D' \cup D''}) \sim ((X, Y, \nu_0 \sqcup \eta_0), \rightarrow_N)$$

Proof. See appendix B.4 on page 146.

12.1.3 Decidability of compliance

To prove that compliance between TSTs is decidable, we reduce such problem to that of checking if a network of TA is deadlock-free — which is known to be decidable [3].

Theorem 12.1.12 *Let p and q be two closed TSTs. Let V_1 and V_2 be two sets of recursion variables not occurring in p and q and such that $V_1 \cap V_2 = \emptyset$. Then:*

$$p \bowtie q \iff \mathcal{T}\langle p \rangle_{V_1} \mid \mathcal{T}\langle q \rangle_{V_2} \text{ is deadlock-free}$$

Proof. Let $(X, D') = \langle p \rangle_{V_1}$ and $(Y, D'') = \langle q \rangle_{V_2}$. We show that:

$$(X, D') \bowtie (Y, D'') \iff \mathcal{T}(X, D') \mid \mathcal{T}(Y, D'') \text{ is deadlock-free} \tag{12.1}$$

For the ($\Rightarrow$) direction, assume by contradiction that $(X, D') \bowtie (Y, D'')$ but the associated network N is *not* deadlock-free. By Definition 5.3.3, there exist a reachable deadlock state s, i.e. $(X, Y, \nu_0 \sqcup \eta_0) \rightarrow_N^* s = (x, y, \nu \sqcup \eta)$ and there are no $\delta \geq 0$ and $\mathtt{a}_\tau \in \mathtt{A} \cup \{\tau\}$ such that $s \xrightarrow{\delta}_N \xrightarrow{\mathtt{a}_\tau}_N$. By Lemma 12.1.11, $(X, \nu_0) \mid (Y, \eta_0) \rightarrow_{D' \cup D''}^* (x, \nu) \mid (y, \eta)$, and the last state is bisimilar to s. However, since $(X, D') \bowtie (Y, D'')$, the state $(x, \nu) \mid (y, \eta)$ is not deadlock according to Definition 12.1.7. Here we have two cases. 1. If $x = \mathbf{1}$ and $y = \mathbf{1}$, then by Definition 12.1.9 x and y are success locations — contradiction, because s is not success; 2. $(x, \nu + \delta) \mid (y, \eta + \delta) \xrightarrow{\mathtt{a}_\tau}_{D' \cup D''}$ for some δ and no $\mathtt{a}_\tau \in \mathtt{A} \cup \{\tau\}$. Then, by Lemma 12.1.11 also s can fire that moves — contradiction. The direction ($\Leftarrow$) is similar. The main statement follows from (12.1) and from Lemma 12.1.8.

Example 12.1.13 *Let us consider the two TSTs in Example 12.1.10. Since the associated network of TA (Figure 12.4) is deadlock-free, by Theorem 12.1.12 we conclude that $p_1 \bowtie q_1$.*

Our implementation of TSTs (`co2.unica.it`) uses *Uppaal* [31] to check compliance. *Uppaal* can verify deadlock-freedom of a network of TA through its query language, which is a simplified version of Time Computation Tree Logic. In *Uppaal*, the special state formula `deadlock` is satisfied by all deadlock states; hence, a network is deadlock-free if none of its reachable states satisfies `deadlock`. Note that checking deadlock-freedom with the path formula `A[] not deadlock` would not be correct, because Definition 5.3.3 does not consider as deadlock the success states. The actual *Uppaal* query we use takes into account success states. E.g., let $N = A_1 \mid A_2$ be a network of TA, with l_1 success location of A_1, and l_2, l_3 success locations of A_2. The query `A[] deadlock imply (A1.l1 && (A2.l2 || A2.l3))` correctly checks deadlock freedom according to Definition 5.3.3.

Chapter 13

Monitoring timed session types

13.1 Runtime monitoring of TSTs

In this chapter we study runtime monitoring based on TSTs. The setting is the following: two participants A and B want to interact according to two (compliant) TSTs p_{A} and p_{B}, respectively. This interaction happens through a server, which monitors all the messages exchanged between A and B, while keeping track of the passing of time. If a participant (say, A) sends a message not expected by her TST, then the monitor classifies A as *culpable* of a violation. There are other two circumstances where A is culpable: (i) p_{A} is an internal choice, but A loses time until all the branches become unfeasible, or (ii) p_{A} is an external choice, but A does not readily receive an incoming message sent by B.

Note that the semantics in Figure 10.1 cannot be directly exploited to define such a runtime monitor, for two reasons. First, the synchronisation rule is purely symmetric, while the monitor outlined above assumes an asymmetry between internal and external choices. Second, in the semantics in Figure 10.1 all the transitions (both messages and delays) must be allowed by the TSTs: for instance, $(\mathtt{a}!\{t \leq 1\}, \nu)$ cannot take any transitions (neither $\mathtt{a}!$ nor δ) if $\nu(t) > 1$. In a runtime monitor we want to avoid such kind of situations, where no actions are possible, and the time is frozen. More specifically, our desideratum is that the runtime monitor acts as a *deterministic* automaton, which reads a *timed trace* (a sequence of actions and time delays) and it reaches a unique state γ, which can be inspected to find which of the two participants (if any) is culpable.

To reach this goal, we define the semantics of the runtime monitor on two levels. The first level, specified by the relation $\twoheadrightarrow$, deals with the case of honest participants; however, differently from the semantics in Section 10.1, here we decouple the action of sending from that of receiving. More precisely, if A has an internal choice and B has an external choice, then we postulate that A must move first, by doing one of the outputs in her choice, and then B must be ready to do the corresponding input. The second level, called *monitoring semantics* and specified by the relation $\twoheadrightarrow_M$, builds upon the first one. Each move accepted by the first level is also accepted by the monitor. Additionally, the monitoring semantics defines transitions for actions not accepted by the first level, for instance unexpected input/output actions, and improper time delays. In these cases, the monitoring semantics signals which

$$(\mathtt{a!}\{g,R\}.\, p \oplus p', [], \nu) \parallel (q, [], \eta) \xrightarrow{\mathsf{A}:\mathtt{a!}}\!\!\!\!\rightarrow (p, [\mathtt{a!}], \nu[R]) \parallel (q, [], \eta) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-}\oplus\text{]}$$

$$(p, [\mathtt{a!}], \nu) \parallel (\mathtt{a?}\{g,R\}.\, q + q', [], \eta) \xrightarrow{\mathsf{B}:\mathtt{a?}}\!\!\!\!\rightarrow (p, [], \nu) \parallel (q, [], \eta[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-+]}$$

$$\frac{\nu + \delta \in \mathrm{rdy}(p) \qquad \eta + \delta \in \mathrm{rdy}(q)}{(p, [], \nu) \parallel (q, [], \eta) \xrightarrow{\delta}\!\!\!\!\rightarrow (p, [], \nu + \delta) \parallel (q, [], \eta + \delta)} \qquad \text{[M-Del]}$$

$$\frac{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\lambda}\!\!\!\!\rightarrow (p', c', \nu') \parallel (q', d', \eta')}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\lambda}\!\!\!\!\rightarrow_M (p', c', \nu') \parallel (q', d', \eta')} \qquad \text{[M-Ok]}$$

$$\frac{(p, c, \nu) \parallel (q, d, \eta) \not\xrightarrow{\mathsf{A}:\ell}\!\!\!\!\rightarrow}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\mathsf{A}:\ell}\!\!\!\!\rightarrow_M (\mathtt{0}, c, \nu) \parallel (q, d, \eta)} \qquad \text{[M-FailA]}$$

$$\frac{(d = [] \wedge \nu + \delta \notin \mathrm{rdy}(p)) \ \vee \ d \neq []}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\delta}\!\!\!\!\rightarrow_M (\mathtt{0}, c, \nu + \delta) \parallel (q, d, \eta + \delta)} \qquad \text{[M-FailD]}$$

Figure 13.1: Monitoring semantics (symmetric rules omitted).

of the two participants is culpable.

Definition 13.1.1 (Monitoring semantics of TSTs) Monitoring configurations $\gamma, \gamma', \ldots$ *are terms of the form* $P \parallel Q$*, where* P *and* Q *are triples* (p, c, ν)*, such that* p *is either a TST or* $\mathtt{0}$*, and* c *is a one-position buffer (either empty or containing an output label). The transition relations* $\twoheadrightarrow$ *and* $\twoheadrightarrow_M$ *over monitoring configurations, with labels* $\lambda, \lambda', \ldots \in (\{\mathsf{A}, \mathsf{B}\} \times \mathtt{L}) \cup \mathbb{R}_{\geq 0}$*, are defined in Figure 13.1.*

In the rules in Figure 13.1, we always assume that the leftmost TST is governed by A, while the rightmost one is governed by B. In rule [M-$\oplus$], A has an internal choice, and she can fire one of her outputs $\mathtt{a!}$, provided that its buffer is empty, and the guard g is satisfied. When this happens, the message $\mathtt{a!}$ is written to the buffer, and the clocks in R are reset. Then, B can read the buffer, by firing $\mathtt{a?}$ in an external choice through rule [M-+]; this requires that the buffer of B is empty, and the guard g of the branch $\mathtt{a?}$ is satisfied. Rule [M-Del] allows time to pass, provided that the delay δ is permitted for both participants, and both buffers are empty. The last three rules specify the runtime monitor. Rule [M-Ok] says that any move accepted by $\twoheadrightarrow$ is also accepted by the monitor. Rule [M-FailA] is used when participant A attempts to do an action not permitted by $\twoheadrightarrow$: this makes the monitor evolve to a configuration where A is culpable (denoted by the term $\mathtt{0}$). Rule [M-FailD] makes A culpable when time passes, in two cases: either A has an internal choice, but the guards are no longer satisfiable; or she has an incoming message. The latter case is motivated by the fact that we are studying synchronous TSTs, and hence dalayed inputs are forbidden.

The following lemma establishes that the monitoring semantics is deterministic, i.e., if $\gamma \xrightarrow{\lambda}\!\!\!\!\rightarrow_M \gamma'$ and $\gamma \xrightarrow{\lambda}\!\!\!\!\rightarrow_M \gamma''$, then $\gamma' = \gamma''$. This is a very desirable property indeed, because it

ensures that the culpability of a participant at any given time is uniquely determined by the past actions. Further, for all finite timed traces $\vec{\lambda}$ (i.e., sequences of actions $\mathsf{A} : \ell$ or time delays δ), there exists some configuration γ reachable from the initial one.

Lemma 13.1.2 *Let $\gamma_0 = (p, [], \nu_0) \parallel (q, [], \eta_0)$. If $p \bowtie q$, then $(\twoheadrightarrow_M, \gamma_0)$ is deterministic, and for all finite timed traces $\vec{\lambda}$ there exists (unique) γ such that $\gamma_0 \xrightarrow{\vec{\lambda}}\!\!\!\!\twoheadrightarrow_M \gamma$.*

Proof. A simple inspection of the rules in Figure 13.1 and an induction on the length of the timed traces $\vec{\lambda}$.

The goal of the runtime monitor is to detect, at any state of the execution, which of the two participants is culpable (if any). Further, we want to identify who is in charge of the next move. This is formalised by the following definition.

Definition 13.1.3 (Duties & culpability) *Let $\gamma = (p, c, \nu) \parallel (q, d, \eta)$. We say that* A *is* culpable *in γ iff $p = \mathtt{0}$. We say that* A *is* on duty *in γ if (i)* A *is not culpable in γ, and (ii) either c is empty and p is an internal choice, or d is not empty.*

Lemma 13.1.4 states that, in each reachable configuration, only one participant can be on duty; and if no one is on duty nor culpable, then both participants have reached success.

Lemma 13.1.4 *If $p \bowtie q$ and $(p, [], \nu_0) \parallel (q, [], \eta_0) \twoheadrightarrow_M^* \gamma$, then:*

1. *there exists at most one participant on duty in γ,*
2. *if there exists some culpable participants in γ, then no one is on duty in γ,*
3. *if no one is on duty in γ, then γ is success, or someone is culpable in γ.*

Proof. Straightforward by Definition 13.1.3 and by the rules in Figure 13.1.

Note that both participants may be culpable in a configuration. For instance, let $\gamma = (\mathtt{a!\{true\}}, [], \eta_0) \parallel (\mathtt{a?\{true\}}, [], \eta_0)$. By applying [M-FailA] twice, we obtain:

$$\gamma \xrightarrow{\mathsf{A:b?}}\!\!\!\!\twoheadrightarrow_M (\mathtt{0}, [], \nu_0) \parallel (\mathtt{a?\{true\}}, [], \eta_0) \xrightarrow{\mathsf{B:b?}}\!\!\!\!\twoheadrightarrow_M (\mathtt{0}, [], \nu_0) \parallel (\mathtt{0}, [], \eta_0)$$

and in the final configuration both participants are culpable.

The following example shows three scenarios in which culpability arises.

Example 13.1.5 *Let $p = \mathtt{a!}\{2 < t < 4\}$ and $q = \mathtt{a?}\{2 < t < 5\} + \mathtt{b?}\{2 < t < 5\}$ be the TSTs of participants* A *and* B*, respectively. Participant* A *declares that she will send* $\mathtt{a}$ *between 2 and 4 time unit (abbr. t.u.), while* B *declares that he is willing to receive* $\mathtt{a}$ *or* $\mathtt{b}$ *if they are sent within 2 and 5 t.u. We have that $p \bowtie q$. Let $\gamma_0 = (p, [], \nu_0) \parallel (q, [], \nu_0)$.*

A correct interaction is given by the timed trace $\vec{\lambda} \doteq 1.2,\ \mathsf{A} : \mathtt{a!},\ \mathsf{B} : \mathtt{a?}$. Indeed, $\gamma_0 \xrightarrow{\vec{\lambda}}\!\!\!\!\twoheadrightarrow_M (\mathbf{1}, [], \nu_0) \parallel (\mathbf{1}, [], \nu_0)$. On the contrary, things may go awry in three cases:

(i) a participant does something not permitted. E.g., if A *fires* $\mathtt{a}$ *at 1 t.u., by* [M-FailA]*: $\gamma_0 \xrightarrow{1}\!\!\!\!\twoheadrightarrow_M \xrightarrow{\mathsf{A:a!}}\!\!\!\!\twoheadrightarrow_M (\mathtt{0}, [], \nu_0 + 1) \parallel (q, [], \eta_0 + 1)$, where* A *is culpable.*

(ii) a participant avoids to do something she is supposed to do. E.g., assume that after 6 *t.u.,* A *has not yet fired* a*. By rule* [M-FAILD]*,* $\gamma_0 \xrightarrow{6}\!\!\!\!\rightarrow_M (0, [], \nu_0 + 6) \parallel (q, [], \eta_0 + 6)$*, where* A *is culpable.*

(iii) a participant does not receive a message as soon as it is sent. For instance, after a *is sent at* 1.2 *t.u., at* 5.2 *t.u.* B *has not yet fired* a?*. By* [M-FAILD]*,* $\gamma_0 \xrightarrow{1.2}\!\!\!\!\rightarrow_M \xrightarrow{\mathsf{A:a!}}\!\!\!\!\rightarrow_M \xrightarrow{4}\!\!\!\!\rightarrow_M (\mathbf{1}, [\mathtt{a!}], \nu_0 + 5.2) \parallel (0, [], \eta_0 + 5.2)$*, where* B *is culpable.*

To relate the monitoring semantics in Figure 13.1 with the one in Figure 10.1 we use the notion *turn-simulation* of [16]. This relation is between states of two arbitrary LTSs $\rightarrow_1$ and $\rightarrow_2$, and it is parameterised over two sets S_1 and S_2 of success states. A state $(s_2, \rightarrow_2)$ turn-simulates $(s_1, \rightarrow_1)$ whenever each move of s_1 can be matched by a sequence of moves of s_2 (ignoring the labels), and stuckness of s_1 implies that s_2 will get stuck in at most one step. Further, turn-simulation must preserve success.

Definition 13.1.6 (Turn-simulation [16]) *For $i \in \{1, 2\}$, let $\rightarrow_i$ be an LTS over a state space Z_i, and let S_i be a set of states of $\rightarrow_i$. We say that a relation $\mathcal{R} \subseteq Z_1 \times Z_2$ is a* turn-simulation *iff $s_1 \mathcal{R} s_2$ implies:*

(a) $s_1 \rightarrow_1 s_1' \implies \exists s_2' : s_2 \rightarrow_2^* s_2'$ *and* $s_1' \mathcal{R} s_2'$

(b) $s_2 \rightarrow_2 s_2' \implies s_1 \rightarrow_1$ *or* $(s_1 \mathcal{R} s_2'$ *and* $s_2' \not\rightarrow_2)$

(c) $s_2 \in S_2 \implies s_1 \in S_1$

If there is a turn-simulation between s_1 and s_2 (written $s_1 \mathcal{R} s_2$), we say that s_2 turn-simulates s_1. We denote with $\preccurlyeq$ the greatest turn-simulation. We say that $\mathcal{R}$ is a turn-bisimulation *iff both $\mathcal{R} \subseteq Z_1 \times Z_2$ and $\mathcal{R}^{-1} \subseteq Z_2 \times Z_1$ are turn-simulations.*

The following lemma establishes that the two semantics of TSTs (Definitions 10.1.3 and 13.1.1) are turn-bisimilar.

Lemma 13.1.7 $((p, \nu) \mid (q, \eta), \rightarrow)$ *is turn-bisimilar to* $((p, [], \nu) \parallel (q, [], \eta), \twoheadrightarrow)$*.*

Proof. See appendix B.5 on page 151.

When both participants behave honestly, i.e., they never take [M-FAIL*] moves, the monitoring semantics preserves compliance (Theorem 13.1.9). The *monitoring compliance* relation $\bowtie_M$ is the straightforward adaptation of that in Definition 10.2.1, except that $\twoheadrightarrow$ transitions are used instead of $\rightarrow$ ones (Definition 13.1.8).

Definition 13.1.8 (Monitoring Compliance) *We say that a monitoring configuration γ is* deadlock *whenever (i) it is not the case that both p and q in γ are $\mathbf{1}$, and (ii) there is no λ such that $\gamma \xrightarrow{\lambda}\!\!\!\!\rightarrow$. We then write $(p, c, \nu) \bowtie_M (q, d, \eta)$ whenever:*

$$(p, c, \nu) \parallel (q, d, \eta) \twoheadrightarrow^* \gamma \quad \text{implies} \quad \gamma \text{ not deadlock}$$

We write $p \bowtie_M q$ whenever $(p, [], \nu_0) \bowtie_M (q, [], \eta_0)$.

Theorem 13.1.9 $\bowtie = \bowtie_M$.

Proof. For the inclusion $\subseteq$, assume that $p \bowtie q$. By Lemma 13.1.7, the states $(p, \nu_0) \mid (q, \eta_0)$ and $(p, [], \nu_0) \parallel (q, [], \eta_0)$ are turn-bisimilar. By $(p, [], \nu_0) \parallel (q, [], \eta_0) \preccurlyeq (p, \nu_0) \mid (q, \eta_0)$ and $(p, \nu_0) \bowtie (q, \nu_0)$, Lemma 5.9 in [16] implies that $(p, [], \nu_0) \bowtie_M (q, [], \eta_0)$, hence $p \bowtie_M q$. The other inclusion is similar.

Part IV

Concluding remarks

Chapter 14

Related work

Maude The development of the tool presented in this thesis 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 [80] 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 [97], the π-calculus [94], Petri nets [91], Erlang [85], Klaim [100], adaptive systems [42], *etc.* A more comprehensive list of calculi, programming languages, tools and applications implemented in Maude is collected in [81].

CO_2 and contract-oriented computing To the best of our knowledge, the concept of *contract-oriented computing* (in the meaning used in this thesis) has been introduced in [28]. CO_2, a contract-agnostic calculus for contract-oriented computing has been instantiated with several contract models — both bilateral [27, 22] and multiparty [26, 76, 19]. Here we have instantiated it with binary session types (Section 6.1). A minor difference w.r.t. [27, 22, 76] 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. [27] 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 7.1.4) in all possible contexts, similarly to [22, 76]. We conjecture that these two notions are equivalent in the contract model based on binary session types considered in this thesis.

In [23], which extends [22], 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". The type system of [23] allows to type check some kinds of infinite-state processes, which are not dealt with in the implementation described in this thesis (based on the direct model checking of the abstract semantics in Section 12.1.3). Furthermore, in [23] only the core fragment of CO_2 is addressed, which roughly corresponds to the CO_2

subset in which α-honesty is also complete.

The programming model envisioned by CO_2 has been implementated as a *contract-oriented middleware* [13], which uses *timed* session types [12] 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 [84]): as a consequence, its chances of being involved in further sessions are reduced.

Comparison with other calculi CO_2 takes inspiration from Concurrent Constraint Programming (CCP [90]): 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 [26].

In cc-pi [43], 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, 35]; 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 [79], the monitors of [50, 35] implement *suppression*, whereas CO_2's monitoring is closer to *truncation*.

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 deadlock-freedom: this depends on its execution context. In fact, if an honest agent $\mathsf{A}[P]$ is involved in 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). Deadlock freedom holds for systems where *all* processes are honest [30]: specifically, all open sessions can be carried forward until their successful termination.

The problem of ensuring safe communication is tackled in the session types literature, whose origins date back to [69, 92, 70] (and from which the contract model adopted in Part II 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:

$$P \;=\; (x,y)\,\mathtt{tell}\downarrow_x \mathtt{a?}.\mathtt{tell}\downarrow_y \mathtt{b!}.\mathtt{do}_x\,\mathtt{a?}.\mathtt{do}_y\,\mathtt{b!} \tag{14.1}$$

$$Q \;=\; (y,x)\,\mathtt{tell}\downarrow_y \mathtt{b?}.\mathtt{tell}\downarrow_x \mathtt{a!}.\mathtt{do}_x\,\mathtt{b?}.\mathtt{do}_y\,\mathtt{a!} \tag{14.2}$$

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 [59, 34, 47, 53], 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 (14.1) and (14.2) 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 [47] 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,

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

is honest, despite it would not be typeable according to [47].

In [45], deadlock-freedom is obtained by using global types [71] (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.

Timed session types Compliance between TSTs is loosely related to the notion of compliance between *untimed* session types (in symbols, $\bowtie_u$). Let $u(p)$ be the session type obtained by erasing from p all the timing annotations. It is easy to check that the semantics of $(u(p), \nu_0) \mid (u(q), \nu_0)$ in Section 10.1 coincides with the semantics of $u(p) \mid u(q)$ in [10]. Therefore, if $u(p) \bowtie u(q)$, then $u(p) \bowtie_u u(q)$. Instead, *semantic conservation* of compliance does not hold, i.e. it is not true in general that if $p \bowtie q$, then $u(p) \bowtie_u u(q)$. E.g., let $p = \mathtt{a!}\{t < 5\} \oplus \mathtt{b!}\{t < 0\}$, and let $q = \mathtt{a?}\{t < 7\}$. We have that $p \bowtie q$ (because the branch $\mathtt{b!}$ can never be chosen), whereas $u(p) = \mathtt{a!} \oplus \mathtt{b!} \not\bowtie_u \mathtt{a?} = u(q)$. Note that, for every p, $u(\mathrm{co}(p)) = \mathrm{co}(u(p))$.

In the context of session types, time has been originally introduced in [38]. However, the setting is different than ours (multiparty and asynchronous, while ours is bi-party and synchronous), as well as its objectives: while we have focussed on primitives for the bottom-up approach to service composition [26], [38] extends to the timed case the *top-down* approach. There, a *choreography* (expressing the overall communication behaviour of a set of participants) is projected into a set of *local session types*, which in turn are refined as processes, to be type-checked against their session type in order to make service composition preserve the properties enjoyed by the choreography. Also a mapping from timed local types to communicating timed automata [72] is presented, which extends to the timed setting the transformation in [58].

Our approach is a conservative extension of untimed session types, in the sense that a participant which performs an output action chooses not only the branch, but the time of writing too; dually, when performing an input, one has to passively follow the choice of the other participant. Instead, in [38] external choices can also delay the reading time. Extending our semantics to an asynchronous one would make compliance undecidable (as it is for untimed asynchronous session types [58]).

Note that our notion of compliance does not imply progress with the semantics of [38] (adapted to the binary case). For instance, consider the TSTs:

$$p \;=\; \mathtt{a?}\{x \leq 2\}.\,\mathtt{a!}\{x \leq 1\} \qquad\qquad q \;=\; \mathtt{a!}\{y \leq 1\}.\,\mathtt{a?}\{y \leq 1\}$$

These TSTs are compliant according to Definition 10.2.1, while using the semantics of [38]:

$$(\nu_0, (p, q, \vec{w_0})) \rightarrow^* (\nu, (\mathtt{a!}\{x \leq 1\}, \mathtt{a?}\{y \leq 1\}, \vec{w_0})) \qquad \text{with } \nu(x) = \nu(y) > 1$$

This is a deadlock state, hence p and q do not enjoy progress according to [38]. The notion of correct interaction studied in [38] is called *feasibility*: a choreography is feasible iff all its reducts can reach the success state. This property implies progress, but it is undecidable in general, as shown by [72] in the context of communicating timed automata. However, feasibility is decidable for the subclass of *infinitely satisfiable* choreographies [38]. The problem of deciding if, given a local type T, there exists a choreography G such that T is in the projection of G and G enjoys (global) progress is not addressed in [38]. A possible way to deal with this problem would be to adapt our kind system (in particular, rule [T-+]).

Dynamic verification of timed multiparty session types is addressed by [86], where the top-down approach to service composition is pursued [71]. Our middleware instead composes and monitors services in a bottom-up fashion [26].

The work [37] studies *communicating timed automata*, a timed version of communicating finite-state machines [40]. In this model, participants in a network communicate asynchronously through bi-directional FIFO channels; similarly to [38], clocks, guards and resets are used to impose time constraints on when communications can happen. A system enjoys *progress* when no deadlock state is reachable, as well as no orphan messages, unsuccessful receptions, and unfeasible configurations. Since deadlock-freedom is undecidable in the untimed case, then *a fortiori* progress is undecidable for communicating timed automata. So, the authors propose an approximated (sound, but not complete) decidable technique to check when a system enjoys progress. This technique is based on *multiparty compatibility*, a condition which guarantees deadlock-freedom of untimed systems [78]. A classical property of timed systems addressed by [37] is *zenoness*, i.e. the situation in which all the paths from a reachable state are infinite and time-convergent. Again, multiparty compatibility is exploited by [37] to devise an approximated decidable technique which guarantees non-zenoness of communicating timed automata. In our setting, an example of zenoness is given by the following TSTs:

$$p \;=\; \mathtt{rec}\, X \,.\, \mathtt{a!}\{x \leq 1\}.\, X \qquad\qquad q \;=\; \mathtt{rec}\, X \,.\, \mathtt{a?}\{x \leq 1, x\}.\, X$$

Although $p \bowtie q$, the total elapsed time cannot exceed 1. This implies that, in order to respect p, a participant should have to perform *infinitely* many writings in *a single* time unit. This problem can be solved imposing some restrictions to TSTs, in order to have the property that composition of TSTs is always non-zeno. For instance, this is guaranteed by the notion of *strong non-zenoness* of [95], which can be computed efficiently but is not complete. Another possibility is to check non-zenoness directly in the network of timed automata constructed for checking compliance, using one of the techniques appeared in the literature [68, 95, 39].

In [55] timed specifications are studied in the setting of *timed I/O transition systems* (TIOTS). They feature a notion of correct composition, called *compatibility*, following the *optimistic approach* pursued in [56]: roughly, two systems are compatible whenever there exists an environment which, composed with them, makes "undesirable" states unreachable. A

notion of *refinement* is coinductively formalised as an alternating timed simulation. Refinement is a preorder, and it is included in the semantic subtyping relation (using compatibility instead of $\bowtie$). Because of the different assumptions (open systems and broadcast communications in [55], closed binary systems in TSTs), compatibility/refinement seem unrelated to our notions of compliance/subtyping. Despite the main notions in [55] are defined on semantic objects (TIOTS), they can be decided on timed I/O automata, which are finite representations of TIOTS. With respect to TSTs, timed I/O automata are more liberal: e.g., they allow for *mixed choices*, while in TSTs each state is either an input or an output. However, this increased expressiveness does not seem appropriate for our purposes: first, it makes the concept of culpability unclear (and it breaks one of the main properties of ours, i.e. that at most one participant is on duty at each execution step); second, it seems to invalidate any dual construction. This is particularly unwelcome, since this construction is one of the crucial primitives of contract-oriented interactions.

Chapter 15

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 [27]. Several case studies (e.g. Section 9.1.5 and Example 7.1.8) 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 6.1 and 7.1). 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 8.1.14). Finally, we have provided an implementation of this technique in Maude, and we have validated our technique against a set of case studies (Sections 8.1.5 and 9.1).

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 7.1.8 and Sections 9.1.4 and 9.1.5). 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 8.1.7 (for the `ask`-free fragment), and in Definition 8.1.8 (for the `ask` statement). While in this thesis 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 [76]. For instance, the conditions in Definition 8.1.8 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 thesis we have used a more precise abstraction; see Definition 8.1.9 and Lemma 8.1.10). 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 8.1.7 and the conditions in Definition 8.1.8.

We have then studied a theory of session types (TSTs), featuring timed synchronous communication between two endpoints. We have defined a decidable notion of compliance between TSTs, a decidable procedure to detect when a TST admits a compliant, a computable canonical compliant construction, a decidable subtyping relation, and a decidable runtime monitoring of interactions based on TSTs.

We remark that the work in this thesis is part of a larger picture, namely the resarch on contract-oriented computing. This research direction has led to several fundational studies [28, 29, 25, 27, 26, 22, 19, 30, 23] and tools based on them. For instance, the programming model envisioned by CO_2 has been implemented as a *contract-oriented middleware* [13, 5], which uses timed session types 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 [84]): as a consequence, its chances of being involved in further sessions are reduced. The tool-chain Diogenes [4, 6] supports the design and development of contract-oriented applications running on top of the middleware above, offering the following features:

1. writing CO 2 specifications of services within an Eclipse plugin;
2. verifying honesty of these specifications;
3. generating from them skeletal Java programs which use the contract-oriented APIs of the middleware;
4. verifying the honesty of Java programs upon refinement.

Diogenes uses the Maude-based tools described in Section 8.1.5 as its core verification engine.

Bibliography

[1] PayPal buyer protection. `https://www.paypal.com/us/webapps/mpp/ua/useragreement-full#13`. Accessed: December 31, 2015.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] Nicola Atzei and Massimo Bartoletti. Developing honest java programs with diogenes. In *FORTE*, pages 52–61, 2016.

[5] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Contract-oriented programming with timed session types, Submitted, 2016. `http://tcs.unica.it/papers/co2-middleware-tutorial.pdf`.

[6] Nicola Atzei, Massimo Bartoletti, Maurizio Murgia, Emilio Tuosto, and Roberto Zunino. Contract-oriented design of distributed applications: a tutorial, Submitted, 2016. `http://tcs.unica.it/papers/diogenes-tutorial.pdf`.

[7] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2):131 – 146, 2005.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[9] Franco Barbanera and Ugo de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, pages 155–164, 2010.

[10] Franco Barbanera and Ugo de'Liguoro. Sub-behaviour relations for session-based client/server systems. *Math. Struct. in Comp. Science*, pages 1–43, 1 2015.

[11] Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. Timed session types. Accepted for publication in Logical Methods in Computer Science, 2016.

[12] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *FORTE*, pages 161–177, 2015.

[13] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. A contract-oriented middleware. In *FACS*, LNCS. Springer, 2015. `http://co2.unica.it`.

[14] Massimo Bartoletti, Tiziana Cimoli, and G. Michele Pinna. Lending Petri nets and contracts. In *FSEN*, volume 8161 of *LNCS*, pages 66–82. Springer, 2013.

[15] Massimo Bartoletti, Tiziana Cimoli, and G. Michele Pinna. Lending Petri nets. *Science of Computer Programming*, 2015. (to appear).

[16] Massimo Bartoletti, Tiziana Cimoli, G. Michele Pinna, and Roberto Zunino. Contracts as games on event structures. *J. Log. Algebr. Meth. Program.*, 85(3):399–424, 2016.

[17] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. A theory of agreements and protection. In *POST*, volume 7796 of *LNCS*, pages 186–205. Springer, 2013.

[18] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Compliance in behavioural contracts: a brief survey. In *PLABS*, volume 9465 of *LNCS*. Springer, 2015.

[19] Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. Choreographies in the wild. *Science of Computer Programming*, 109:36–60, 2015.

[20] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, and Roberto Zunino. Modelling and verifying contract-oriented systems in Maude. In *WRLA*, volume 8663 of *LNCS*, pages 130–146, 2014.

[21] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, and Roberto Zunino. Verifiable abstractions for contract-oriented systems. *J. Log. Algebr. Meth. Program.*, 86(1):159–207, 2017.

[22] Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. Honesty by typing. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 305–320, 2013.

[23] Massimo Bartoletti, Alceste Scalas, Emilio Tuosto, and Roberto Zunino. Honesty by typing. *Logical Methods in Computer Science*, 2017. Pre-print available as: `https://arxiv.org/abs/1211.2609`.

[24] Massimo Bartoletti, Alceste Scalas, and Roberto Zunino. A semantic deconstruction of session types. In *CONCUR*, pages 402–418, 2014.

[25] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. Contracts in distributed systems. In *ICE*, 2011.

[26] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1):5–60, 2012.

[27] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION*, volume 7274 of *LNCS*, pages 245–260, 2012.

[28] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. In *LICS*, pages 332–341, 2010.

[29] Massimo Bartoletti and Roberto Zunino. Primitives for contract-based synchronization. In *ICE*, 2010.

[30] Massimo Bartoletti and Roberto Zunino. On the decidability of honesty and of its variants. In *WSFM-BEAT*, LNCS. Springer, 2015.

[31] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on Uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[32] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *ACPN*, pages 87–124, 2003.

[33] Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types. In *CONCUR*, pages 387–401, 2014.

[34] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.

[35] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FORTE*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

[36] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, pages 162–176, 2010.

[37] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, pages 283–296, 2015.

[38] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR*, pages 419–434, 2014.

[39] Howard Bowman and Rodolfo Gómez. How to stop time stopping. *Formal Asp. Comput.*, 18(4):459–493, 2006.

[40] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[41] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, pages 34–50, 2007.

[42] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In *WRLA*, volume 7571 of *LNCS*, pages 118–138, 2012.

[43] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP*, pages 18–32, 2007.

[44] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[45] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.

[46] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62, 2014.

[47] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of session types. In *PPDP*, pages 219–230, 2009.

[48] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Comp. Sci.*, 8(1), 2012.

[49] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5):19:1–19:61, 2009.

[50] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In Roberto Bruni and Vladimiro Sassone, editors, *Trustworthy Global Computing*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer Berlin Heidelberg, 2012.

[51] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[52] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *TCS*, 2001.

[53] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION*, pages 45–59, 2013.

[54] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5), 2008.

[55] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, Louis-Marie Traonouez, and Andrzej Wasowski. Real-time specifications. *STTT*, 17(1):17–45, 2015.

[56] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ACM SIGSOFT*, pages 109–120, 2001.

[57] Rocco De Nicola and Frits W. Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[58] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, pages 174–186, 2013.

[59] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, pages 257–275, 2007.

[60] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Trans. Software Eng.*, 34(6):844–859, 2008.

[61] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electr. Notes Theor. Comput. Sci.*, 71:162–187, 2002.

[62] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. North-Holland Pub. Co./MIT Press, 1990.

[63] Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In *SEFM Workshops: BEAT2*, pages 15–28, 2013.

[64] Simon Gay and Malcolm Hole. Subtyping for session types in the Pi calculus. *Acta Inf.*, 42(2), 2005.

[65] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP*, pages 74–90, 1999.

[66] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[67] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[68] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[69] Kohei Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.

[70] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138, 1998.

[71] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[72] Pavel Krcál and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *CAV*, pages 249–262, 2006.

[73] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[74] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[75] Cosimo Laneve and Luca Padovani. The *must* Preorder Revisited. In *CONCUR*, pages 212–225, 2007.

[76] Julien Lange and Alceste Scalas. Choreography synthesis as contract agreement. In *ICE*, pages 52–67, 2013.

[77] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In *CONCUR*, pages 225–239, 2012.

[78] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232, 2015.

[79] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[80] José Meseguer. Rewriting as a unified model of concurrency. In *CONCUR*, volume 458 of *LNCS*, pages 384–400, 1990.

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

[82] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[83] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[84] A. Mukhija, Andrew Dingwall-Smith, and D.S. Rosenblum. Qos-aware service composition in Dino. In *ECOWS*, pages 3–12, 2007.

[85] Martin Neuhäußer and Thomas Noll. Abstraction and model checking of core Erlang programs in Maude. *ENTCS*, 176(4), 2007.

[86] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*, pages 19–26, 2014.

[87] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146, 2014.

[88] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[89] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2012.

[90] Vijay A. Saraswat and Martin C. Rinard. Concurrent constraint programming. In *POPL*, pages 232–245, 1990.

[91] Mark-Oliver Stehr, José Meseguer, and Peter Csaba Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, pages 250–303, 2001.

[92] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.

[93] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[94] Prasanna Thati, Koushik Sen, and Narciso Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. *ENTCS*, 71, 2002.

[95] Stavros Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop*, pages 299–314, 1999.

[96] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106, 2010.

[97] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. *Electr. Notes Theor. Comput. Sci.*, 71:282–300, 2002. WRLA 2002, Rewriting Logic and Its Applications.

[98] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[99] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1 – 26, 2013.

[100] Martin Wirsing, Jonas Eckhardt, Tobias Mühlbauer, and José Meseguer. Design and analysis of cloud-based architectures with KLAIM and Maude. In *WRLA*, volume 7571 of *LNCS*, pages 54–82, 2012.

[101] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, pages 22–41, 2013.

Appendix A

Appendix for Part II

A.1 Proofs for Section 6.1

Lemma A.1.1 *If* $\gamma \twoheadrightarrow \mathsf{A} : \mathtt{rdy}\ \mathtt{a?v}.\, c \mid \mathsf{B} : d$*, then* d *is* `rdy`*-free.*

Proof. By Definition 6.1.1, only one `rdy` can occur in γ, and if so it must occur at top-level of a contract. This properties are preserved by transitions (Definition 6.1.2). The thesis then follows by straightforward analysis of the rules in Figure 6.1.

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.1.2 (Determinism) *For all transition labels* μ *of* $\twoheadrightarrow$*:*

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

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.1.1.

[IntExt] The thesis follows by the assumption that the branch labels in c and d are pairwise distinct (item 2 of Definition 6.1.1).

Lemma A.1.3 *If* $\gamma \equiv \gamma'$*, then* γ *is safe iff* γ' *is safe.*

Proof. Straightforward by Definition 6.1.3 and by the definition of the equivalence $\equiv$ (Definition 6.1.2).

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.

Proof of Lemma 6.1.5: For all contracts c, there exists some d such that $c \bowtie d$.

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

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

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

$$(c, \mathrm{co}(c)) \in \mathcal{R} \tag{A.1}$$
$$(c, \mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(c)) \in \mathcal{R} \tag{A.2}$$
$$(\mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(c)\,, c) \in \mathcal{R} \tag{A.3}$$

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 A.2 and A.3, while for A.1 it can be easily proved by cases on the structure of c. We now prove that (c,d) satisfies the second requirement of F. We have the following three cases, according to the equation used to prove $(c,d) \in \mathcal{R}$.

(A.1) $d = \mathrm{co}(c)$. We have two subcases, according to the form of c. Assume first that c is an external choice, i.e. $c = (\mathtt{a?T} \,.\, c') + c''$. Then $d = \mathtt{a!T} \,.\, \mathrm{co}(c') \oplus \mathrm{co}(c'')$. By rule [IntExt], $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow \mathsf{A} : c' \mid \mathsf{B} : \mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(c')$. We obtain the thesis by observing that $(c', \mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(c')) \in \mathcal{R}$ follows by (A.2). The case where c is an internal choice is similar.

(A.2) $d = \mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(c)$. By rule [Rdy], $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow \mathsf{A} : c \mid \mathsf{B} : \mathrm{co}(c)$. The thesis follows by observing that $(c,\, \mathrm{co}(c)) \in \mathcal{R}$ follows by (A.1).

(A.3) $c = \mathtt{rdy}\ \mathtt{a?v}.\, \mathrm{co}(d)$. Similar to the previous case.

Summing up, $\mathcal{R}$ is a compliance relation, hence by item (A.1) it follows that $c \bowtie \mathrm{co}(c)$.

Definition A.1.4 (Value abstraction of contracts) *Value-abstract contracts are terms of the grammar:*

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

where

1. *the index set $\mathcal{I}$ is finite,*

2. *the labels* $\mathtt{a}_i$ *in the prefixes of each summation are pairwise distinct,*
3. *recursion variables* X *are guarded, and*
4. `rdy` *occurs at the top-level, only.*

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

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

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

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

Proof. For item 1 we proceed by cases on the rule used to deduce $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}\circ\mathtt{v}}\!\!\!\!\rightarrow\, \gamma'$. By the semantics of concrete and value-abstract contracts and by Definition A.1.4, we have:

- [IntExt]

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

- [Rdy]

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

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

For item 2 we proceed by cases on the rule used to deduce $\alpha^\star(\gamma) \xrightarrow{\mathsf{A}:(\mathtt{a},\mathtt{T})\circ}\!\!\!\!\rightarrow_\star\, \hat{\gamma}'$. By the semantics of concrete and value-abstract contracts and by Definition A.1.4, we have:

- [AbsIntExt]

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

  Hence by Definition A.1.4 we conclude that $\alpha^\star(\gamma') = \hat{\gamma}'$.

- [AbsRdy]

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

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

  Hence by Definition A.1.4 we conclude that $\alpha^\star(\gamma') = \hat{\gamma}'$.

Proof of Lemma 6.1.6:
For all contracts c, d:

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

Straightforward consequence of Lemma A.1.5 and of the fact that γ is safe iff $\alpha^\star(\gamma)$ is such.

Lemma 6.1.9. *Let $c \bowtie d$. If $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow^* \gamma$, then either $\gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$, or there exists a unique culpable in γ.*

Proof of Lemma 6.1.9:
Let $c \bowtie d$. If $\mathsf{A} : c \mid \mathsf{B} : d \twoheadrightarrow^* \gamma$, then either $\gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$, or there exists a unique culpable in γ.

Let $\gamma = \mathsf{A} : c' \mid \mathsf{B} : d'$. By Definition 6.1.3 it follows that γ is safe. Hence, by Definition 6.1.3 we have the following three cases (symmetric ones are omitted):

- $\gamma = \mathsf{A} : \mathtt{0} \mid \mathsf{B} : \mathtt{0}$, from which the thesis follows directly.
- $\gamma = \mathsf{A} : \mathtt{rdy}\ \mathtt{a}?\mathtt{v}.\,c' \mid \mathsf{B} : d'$, with d' rdy-free (Lemma A.1.1). We have only one transition from γ, given by rule [Rdy], which prescribes a move of A. By Definition 6.1.8, we have that $\mathsf{A} \ddot{\frown} \gamma$ and $\mathsf{B} \ddot{\smile} \gamma$.
- $\gamma = \mathsf{A} : \bigoplus_{i\in\mathcal{I}} \mathtt{a}_i!\mathtt{T}_i\,.\,c_i \mid \mathsf{B} : \sum_i \mathtt{a}_{i\in\mathcal{I}}?\mathtt{T}_i\,.\,d_i + \sum_j \mathtt{b}_j?\mathtt{T}_j\,.\,d_j$, with $\mathcal{I} \neq \emptyset$. Only rule [IntExt] can be applied, and it prescribes a move of A. By Definition 6.1.8, we conclude that $\mathsf{A} \ddot{\frown} \gamma$ and $\mathsf{B} \ddot{\smile} \gamma$.

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

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

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

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

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

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

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

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

Figure A.1: Reduction semantics of value-abstract systems.

A.2 Proofs for Section 8.1

A.2.1 Proofs for Section 8.1.1

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.

We now formalise the meaning of the relation $\vdash$ used in rule [Ask]. We denote with $\mathbb{AP}$ the 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 $\circ \in \{!, ?\}$. Let $\ell, \ell', \cdots$ range over $\mathbb{AP} \cup \{\varepsilon\}$.

Definition A.2.1 *We define the Kripke structure* $\mathrm{TS}_\star = (\Sigma, \to, L)$ *as follows:*

- $\Sigma = \{(\ell, \hat{\gamma}) \mid \ell \in \mathbb{AP} \cup \{\varepsilon\},$ *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:*

$$(\ell, \hat{\gamma}) \to (a, \hat{\gamma}') \quad \text{if } \hat{\gamma} \xrightarrow{\mathsf{A}:a}\!\!\!\!\rightarrow_\star \hat{\gamma}'$$
$$(\ell, \hat{\gamma}) \to (\ell, \hat{\gamma}) \quad \text{if } \hat{\gamma} \not\xrightarrow{}\!\!\!\!\rightarrow_\star$$

- *the labelling function* $L : \Sigma \to 2^{\mathbb{AP}}$ *is defined as:*

$$L((\ell, \hat{\gamma})) = \begin{cases} \{\ell\} & \text{if } \ell \in \mathbb{AP}; \\ \emptyset & \text{otherwise.} \end{cases}$$

We write $\hat{\gamma} \vdash_\star \phi$ *whenever* $(\varepsilon, \hat{\gamma}) \models \phi$ *holds in LTL.*

Lemma A.2.2 *For all compliant* γ*:*

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

Proof. Straightforward consequence of Lemma A.1.5

Corollary A.2.3 *For all compliant* γ*:*

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

Definition A.2.4 (Value-abstract labels) *For all transition labels* μ *of* $\to$ *(CO₂ semantics, Figure 7.2), 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 A.2.5 *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 A.2.6 *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 7.2.*

Proof. Trivial.

Lemma A.2.7 $S \xrightarrow{\mu} S' \implies \alpha^\star(S) \xrightarrow{\alpha^\star(\mu)}_\star \alpha^\star(S')$

Proof. Suppose $S \xrightarrow{\mu} S'$. We proceed by rule induction, rewriting the transitions according to Definition A.2.4 and Figure A.1:

[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:

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

We have that:

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

Then:

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

[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 6.1.6 (RHS of the double implication) and Definition 6.1.3, with an obvious overloading of $\bowtie$ we have $\alpha^\star(c) \bowtie \alpha^\star(d)$. We have that:

$$\alpha^\star(S) = (x,y)(\alpha^\star(S'') \mid \{\downarrow_x \alpha^\star(c)\}_\mathsf{A} \mid \{\downarrow_y \alpha^\star(d)\}_\mathsf{B})$$

Then, by Lemma A.2.5:

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

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

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

We have that:

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

Then:

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

[Do!] Suppose:

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

Assuming $\mathtt{v} : \mathtt{T}$, we have that:

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

Then, by item 1 of Lemma A.1.5:

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

[Do?] Suppose:

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

Assuming $\mathtt{v} : \mathtt{T}$, we have that:

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

Then, by item 1 of Lemma A.1.5 and the "furthermore..." part of lemma A.2.5:

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

[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 have that:

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

We can now notice that, by Definition 8.1.9, 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.1.4), with an obvious overloading of $\vdash$ we obtain $\gamma \vdash \phi \iff \alpha^\star(\gamma) \vdash \phi$. Hence, we have:

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

[Def] Suppose:

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

Then, by applying the induction hypothesis, and by Lemma A.2.5:

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

[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 A.2.6:

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

Lemma A.2.8 *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:

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

And:

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

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:

$$\alpha^\star(S) = (x, y)(\alpha^\star(S'') \;\mid\; \{\downarrow_x \alpha^\star(c)\}_\mathsf{A} \;\mid\; \{\downarrow_y \alpha^\star(d)\}_\mathsf{B})$$

And:

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

By Definition 6.1.3 and Lemma 6.1.6 (LHS of the double implication), we have $c \bowtie d$. Hence, by Lemma A.2.5:

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

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

[Do] We have:

$$\alpha^\star(S) = \mathsf{A}[\mathtt{do}_s\, a.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \;\mid\; s[\alpha^\star(\gamma)]$$

And:

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

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.1.5, there exists γ' such that $\gamma \xrightarrow{\mathsf{A}:\mathtt{a}!\mathtt{v}}\!\!\!\!\rightarrow \gamma'$ and $\alpha^\star(\gamma') = \hat{\gamma}'$. Then, by rule [Do!]:

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

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.1.5, there exists γ' such that $\gamma \xrightarrow{\mathsf{A:a?v}}\!\!\!\!\rightarrow \gamma'$ and $\alpha^\star(\gamma') = \hat{\gamma}'$. Then, by rule [Do?]:

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

Hence we have $\alpha^\star(\mu) = \hat{\mu}$, and by lemma A.2.5, 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:

$$\alpha^\star(S) = \mathsf{A}[\mathtt{ask}_s\ \phi.\alpha^\star(P) + \alpha^\star(P') \mid \alpha^\star(Q)] \;\mid\; s[\alpha^\star(\gamma)]$$

And:

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

By Corollary A.2.3(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 $S = \mathsf{A}[\mathrm{X}(\vec{u}, \vec{e}) \mid Q] \mid S''$. By applying the induction hypothesis, we have that $\exists \mu, S'$ such that $S''' = \mathsf{A}[P\{^{\vec{u}}/_{\vec{x}}\}\{^{\vec{e}}/_{\vec{y}}\} \mid Q] \mid S'' \xrightarrow{\mu} S'$ and $\alpha^\star(S''') \xrightarrow{\hat{\mu}}_\star \hat{S}' = \alpha^\star(S')$, with $\hat{\mu} = \alpha^\star(\mu)$. Therefore, we have:

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

Then, by Lemma A.2.5:

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

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 A.2.6we 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 A.2.9 $\alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \implies \exists S' : S \xrightarrow{\mathsf{A}:\mathtt{if}} S' \wedge \alpha^\star(S) \xrightarrow{\mathsf{A}:\mathtt{if}}_\star \alpha^\star(S')$.

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 A.2.6 and Lemma A.2.5.

Lemma A.2.10 *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 8.1.4 which guarantees $S \in \alpha^\star\text{-}\,\mathrm{Rdy}_s^{\mathsf{A}}$:

- item 1. By Lemma A.2.8, item 1 of Definition 7.1.4 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 A.2.8, 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 7.1.4 holds for S.
- item 3. By Lemma A.2.9 $S \xrightarrow{\mathsf{A}:\mathtt{if}} S'$ — because of an underlying `if` prefix chosing one of its branches.

  By item 3 of Definition 8.1.4 it follows A that is ready at s in $\alpha^\star(S')$ (actually, A is ready in *both* branches of the abstracted `if`). By applying the induction hypothesis, A is ready at s in S', and hence item 2 of Definition 7.1.4 holds for S.

Lemma A.2.11 *For all* `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 A.2.7.

Lemma A.2.12 *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* `if`*-free then* $\hat{S}$ *is* $\alpha^\star-$*ready.*

Proof. Direct consequence of Lemma A.2.10 and Lemma A.2.11.

Proof of Theorem 8.1.5: 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.

Direct conseguence of Lemma A.2.7, Lemma A.2.8 and Lemma A.2.12

When defining honesty, we consider contexts that cannot delimit the free variables of a participant. 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 A.2.13 below.

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

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 7.1.5.

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 $d = \mathtt{a?}\,.\,\mathtt{0}$: 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. 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 $(\vec{s})(S_{\mathsf{A}} \mid S_{ctx})$ 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 A.2.13 (A-solo and A-safe systems) *We say that S is* A-solo *iff one of the following holds:*

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

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

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

Proposition A.2.14 (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 A.2.15 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 A.2.15 *For all systems S, S' such that $S \to_\star S'$:*

(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 A.2.13 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 $S \to_\star S'$. Let S be A-safe. Then, by Definition A.2.13, $S \equiv (\vec{s})(S_\mathsf{A} \mid S_{ctx})$, 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:

$$\dfrac{\dfrac{\vdots}{\dfrac{S_\mathsf{A} \mid S_{ctx} \to_\star S'_0}{\vdots}\ [\textsc{Del}]}}{S \equiv (\vec{s})(S_\mathsf{A} \mid S_{ctx}) \to_\star (\vec{s})S'_0 = S'}\ [\textsc{Del}]$$

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

[Do] There are the following three subcases.

- The $\mathtt{do}$ 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 $\mathtt{do}$ 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]:

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

  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:

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

We have that $S'_\mathsf{A}\sigma \mid s[\gamma]$ is A-solo, and $S'_{ctx}\sigma$ is A-free, then by Definition A.2.13, S'_0 is A-safe. Therefore, $S' = (\vec{s})S'_0$ is A-safe as well.

[Par] We have the following two subcases.

- $S_\mathsf{A} \to_\star S'_\mathsf{A}$, and $S_0 = S'_\mathsf{A} \mid S_{ctx}$. By item (1), S'_A is A-solo, hence S' is A-safe.
- $S_{ctx} \to_\star S'_{ctx}$. 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 $S_{\mathsf{A}} = \mathsf{A}[\mathrm{X}(\vec{v}) \mid Q] \mid S'_{\mathsf{A}}$, for some A-solo S'_{A}, and:

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

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

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

A.2.2 Proofs for Section 8.1.2

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.

Proof of Theorem 8.1.7:
For all value-abstract contract configurations γ, γ', for all context-abstract contracts c, c':

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

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

[AbsIntExt] By the semantics of concrete and abstract contracts and by Definition 8.1.6, we have:

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

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

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

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

[AbsIntExt] We have:

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

[AbsRdy] We have:

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

For item 3, we have two cases:

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

Definition A.2.16 *Let $\gamma^{\mathsf{A}}_{\mathsf{B}}$ be a function defined as follows:*

$$\begin{array}{rcll} \gamma^{\mathsf{A}}_{\mathsf{B}}(ctx\ a.\hat{c}) & = & \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ a.\,\mathrm{co}(\hat{c}) & \\ \gamma^{\mathsf{A}}_{\mathsf{B}}(\mathtt{rdy}\ a.\hat{c}) & = & \mathsf{A} : \mathtt{rdy}\ a.\hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c}) & \\ \gamma^{\mathsf{A}}_{\mathsf{B}}(\hat{c}) & = & \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c}) & \textit{otherwise} \end{array}$$

Lemma A.2.17 *For all abstract contracts $\tilde{c}$:*

(1) $\tilde{c} \xrightarrow{a}_{\mathsf{A}} \tilde{c}' \implies \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{A}:a}_{\star} \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$

(2) $\tilde{c} \xrightarrow{ctx:a}_{\mathsf{A}} \tilde{c}' \implies \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{B}:a}_{\star} \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$

(3) $\alpha_{\mathsf{A}}(\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c})) = \tilde{c}$

Proof.

(1) There are two cases:

- $\tilde{c} = a\,.\,\hat{c} \oplus \hat{c}' \xrightarrow{a}_{\mathsf{A}} ctx\ \mathrm{co}(a)\ .\,\hat{c} = \tilde{c}'$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : a\,.\,\hat{c} \oplus \hat{c}' \mid \mathsf{B} : \mathrm{co}(a)\,.\,\mathrm{co}(\hat{c}) + \mathrm{co}(\hat{c}')$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}') = \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ \mathrm{co}(a)\ .\mathrm{co}(\hat{c})$
  Clearly: $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{A}:a}_{\star} \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$
- $\tilde{c} = \mathtt{rdy}\ a\ .\ \hat{c}' \xrightarrow{a}_{\mathsf{A}} \hat{c}' = \tilde{c}'$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \mathtt{rdy}\ a\ .\ \hat{c}' \mid \mathsf{B} : \mathrm{co}(\hat{c}') \xrightarrow{\mathsf{A}:a}_{\star} \mathsf{A} : \hat{c}' \mid \mathsf{B} : \mathrm{co}(\hat{c}') = \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$

(2) There are two cases:

- $\tilde{c} = \mathrm{co}(a) \,.\, \hat{c} + \hat{c}' \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \mathtt{rdy}\ \mathrm{co}(a) \,.\, \hat{c} = \tilde{c}'$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \mathrm{co}(a) \,.\, \hat{c} + \hat{c}' \mid \mathsf{B} : a \,.\, \mathrm{co}(\hat{c}) \oplus \mathrm{co}(\hat{c}')$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}') = \mathsf{A} : \mathtt{rdy}\ \mathrm{co}(a) \,.\hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c})$
  Clearly: $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_{\star} \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$
- $\tilde{c} = ctx\ a \,.\, \hat{c} \xrightarrow{ctx:a}\!\!\!\!\rightarrow_{\mathsf{A}} \hat{c} = \tilde{c}'$
  $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}) = \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathtt{rdy}\ a \,.\ \mathrm{co}(\hat{c}) \xrightarrow{\mathsf{B}:a}\!\!\!\!\rightarrow_{\star} \mathsf{A} : \hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c}) = \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}')$

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

- $\alpha_{\mathsf{A}}(\gamma^{\mathsf{A}}_{\mathsf{B}}(ctx\ a.\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : c \mid \mathsf{B} : \mathtt{rdy}\ a.\mathrm{co}(\hat{c})) = ctx\ a.\hat{c} = \tilde{c}$
- $\alpha_{\mathsf{A}}(\gamma^{\mathsf{A}}_{\mathsf{B}}(\mathtt{rdy}\ a.\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : \mathtt{rdy}\ a.\hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c})) = \mathtt{rdy}\ a.\hat{c} = \tilde{c}$
- $\alpha_{\mathsf{A}}(\gamma^{\mathsf{A}}_{\mathsf{B}}(\hat{c})) = \alpha_{\mathsf{A}}(\mathsf{A} : \hat{c} \mid \mathsf{B} : \mathrm{co}(\hat{c})) = \hat{c} = \tilde{c}$, with $\hat{c}$ rdy-free.

Lemma A.2.18 *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 A.2.17, the thesis holds, and in particular $\gamma = \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c})$.

A.2.3 Proofs for Section 8.1.3

Lemma A.2.19 *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 A.2.20 *For all value-abstract systems $\hat{S}$:* $\mathrm{fv}(\alpha_{\mathsf{A}}(\hat{S})) \cup \mathrm{fn}(\alpha_{\mathsf{A}}(\hat{S})) \subseteq \mathrm{fv}(\hat{S}) \cup \mathrm{fn}(\hat{S})$

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 $\alpha_{\mathsf{A}}(\hat{S})$.

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

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

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

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

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

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

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

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

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

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

Figure A.2: Reduction semantics of context-abstract systems (full set of rules).

- $\hat{S}' \mid \hat{S}''$. 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 A.2.21 *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 $\tilde{c}$:*

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

Proof. $\supseteq$: Suppose that:

$$a_0a_1... \in \bigcup_{\hat{\gamma} \in \Gamma_{\tilde{c}}} \text{Paths}((a, \hat{\gamma}))$$

i.e.:

$$\exists \hat{\gamma} \in \Gamma_{\tilde{c}}.\ a_0a_1... \in \text{Paths}((a, \hat{\gamma}))$$

Let $s_0 = (a_0, \hat{\gamma_0})$, with $a_0 = a$ and $\hat{\gamma_0} = \hat{\gamma}$. By Definition 6.1.11, 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 8.1.9, $\alpha_{\mathsf{A}}(\hat{\gamma_i}) \xrightarrow{\tilde{\mu}}\!\!\!\!\rightarrow_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma_{i+1}})$ 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 6.1.11 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 8.1.7, the thesis follows. $\subseteq$: Suppose that $a_0a_1... \in \text{Paths}((a, \tilde{c}))$. Let $\tilde{s}_0 = (a, \tilde{c})$. By Definition 6.1.11, 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 $s_i = (a_i, \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}_i))$, for all i. Clearly, $L(s_i) = L(\tilde{s_i})$. We have to show that, for all $i > 0$, $\gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}_{i-1}) \xrightarrow{\mu}\!\!\!\!\rightarrow \gamma^{\mathsf{A}}_{\mathsf{B}}(\tilde{c}_i)$, with $\mu = \mathsf{A} : a_i$ or $\mathsf{B} : a_i$. This follows by Definition 8.1.9 and items (1) and (2) of Lemma A.2.17.

Proof of Lemma 8.1.10:
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** $\gamma : \alpha_{\mathsf{A}}(\gamma) = c \ \wedge\ \gamma \vdash_\star \phi)$

(1) For the $\implies$ direction, suppose that $\tilde{c} \vdash \phi$. By definition of $\vdash$, it follows that $\forall \lambda \in \text{Paths}(\varepsilon, \tilde{c})\ .\ \lambda \vdash \phi$. Let $\hat{\gamma}$ be such that $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$. Suppose $\lambda \in \text{Paths}(\varepsilon, \hat{\gamma})$. By Lemma A.2.21 we can conclude that $\lambda \in \text{Paths}(\varepsilon, \tilde{c})$ and then $\lambda \vdash \phi$.

For the $\impliedby$ direction, suppose that $\tilde{c} \nvdash \phi$. By definition of $\vdash$, it follows that $\exists \lambda \in \text{Paths}(\varepsilon, \tilde{c})\ .\ \lambda \nvdash \phi$. By Lemma A.2.21 we know that $\exists \hat{\gamma}\ .\ \lambda \in \text{Paths}(\varepsilon, \hat{\gamma})$. 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.

Lemma A.2.22 *For all* A*-safe value-abstract systems* $\hat{S}$*, whenever* $\hat{S} \equiv (x_1, ..., x_n, \vec{s})(\hat{S}_{\mathsf{A}} \mid \hat{S}_{ctx})$*:*

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

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 7.1 allows to put $\hat{S}$ in the form $(\vec{s})(\hat{S}'_{\mathsf{A}} \mid \hat{S}'_{ctx})$, and so $\hat{S}$ is not A-safe.

Lemma A.2.23 *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* $(\mu = \mathsf{A} : \pi \ \wedge\ \hat{S}$ `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 $\hat{\gamma} \xrightarrow{\mathsf{A}:\mathsf{a}} \hat{\gamma}'$. By item 1 of Theorem 8.1.7 we have $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{\mathsf{a}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma}')$. 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 $\hat{\gamma} \xrightarrow{\mathsf{B}:\mathsf{a}} \hat{\gamma}'$ 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 [α-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 $\hat{\gamma} \xrightarrow{\mathsf{B}:\mathsf{a}} \hat{\gamma}'$ and $\hat{\gamma} = \mathsf{A} : \hat{c} \mid \mathsf{B} : \hat{d}$. By item 2 of Theorem 8.1.7, $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{ctx:\mathsf{a}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{\gamma}')$. 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 [α-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 [α-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 $\alpha_{\mathsf{A}}(\hat{S}) = (\vec{u})\alpha_{\mathsf{A}}(\hat{S}_0)$ and $\alpha_{\mathsf{A}}(\hat{S}') = (\vec{u})\alpha_{\mathsf{A}}(\hat{S}'_0)$. 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 $\hat{S}$ `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 $\alpha_{\mathsf{A}}(\hat{S}) = (\vec{u})\alpha_{\mathsf{A}}(\hat{S}_0)$ and $\alpha_{\mathsf{A}}(\hat{S}') = (\vec{u})\alpha_{\mathsf{A}}(\hat{S}'_0)$. 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:

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

Using the induction hypothesis for thesis (2):

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

[Def] We can have two subcases:

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

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

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

By Lemma A.2.22 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:

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

By Lemma A.2.22 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]: $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{ctx}_{\mathsf{A}} (s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}})\{^{s}/_{x}\} \mid s[\hat{c}]) = \tilde{S}'$.

$$\begin{aligned}
\alpha_{\mathsf{A}}(\hat{S}') &= \\
(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}\sigma) \mid s[\hat{c}]) &\equiv \\
(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}\{^{s}/_{x}\}) \mid s[\hat{c}]) &\equiv && \text{because } y \text{ is not free in } \alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}}) \\
(s)(\alpha_{\mathsf{A}}(\hat{S}'_{\mathsf{A}})\{^{s}/_{x}\} \mid s[\hat{c}]) &= && \text{by Lemma A.2.19} \\
\tilde{S}'. &
\end{aligned}$$

[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 8.1.8, $\hat{c} \vdash_{ctx} \phi$. 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 A.2.24 *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* 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:

$$\exists \tilde{\eta}' \,:\, \alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\tilde{\eta}'}_{\mathsf{A}}{}^{n} \alpha_{\mathsf{A}}(\hat{S}'') \tag{A.4}$$

$$\eta' \text{ A-solo and } \hat{S} \text{ ask-free} \implies \eta' = \tilde{\eta}' \tag{A.5}$$

Now, assume that $\hat{S}'' \xrightarrow{\mu} \hat{S}'$, and let $\eta = \eta'\mu$. By Lemma A.2.23, it follows that:

$$\exists \tilde{\mu} \,:\, \alpha_{\mathsf{A}}(\hat{S}'') \xrightarrow{\tilde{\mu}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S}') \tag{A.6}$$

$$\mu = \mathsf{A} : \pi \,\wedge\, \hat{S} \text{ ask-free} \implies \mu = \tilde{\mu} \tag{A.7}$$

By (A.4) and (A.6), 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 ask-free. Therefore, by (A.5) and (A.7) we conclude that: $\eta = \eta'\,\mu = \tilde{\eta}'\,\tilde{\mu} = \tilde{\eta}$.

Lemma A.2.25 $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A}:\pi}_{\mathsf{A}} \tilde{S}' \implies \exists \hat{S}'$ 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 $\hat{S} \equiv \mathsf{A}[(\mathtt{if}\ e\ \mathtt{then}\ \tilde{P}\ \mathtt{else}\ \tilde{P}') \mid \tilde{Q}] \mid \hat{S}_{\,ctx}$, and let $\alpha_{\mathsf{A}}(\hat{S}) \equiv \mathsf{A}[(\mathtt{if}\ e\ \mathtt{then}\ \tilde{P}\ \mathtt{else}\ \tilde{P}') \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\mathtt{if}}_{\mathsf{A}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \tilde{S}'$. We have $\hat{S} \xrightarrow{\mathsf{A}:\mathtt{if}} \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{\,ctx} = \hat{S}' = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$. The case in which the "false" branch is taken is similar.

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

[α-DO] Let $\alpha_{\mathsf{A}}(\hat{\gamma}) \xrightarrow{\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}'$ — which, by item 3 of Theorem 8.1.7, implies $\hat{\gamma} \xrightarrow{\mathsf{A}:\mathsf{a}}\!\!\!\!\rightarrow \hat{\gamma}'$, with $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$. Furthermore, let $\hat{S} \equiv s[\hat{\gamma}] \mid \mathsf{A}[\mathtt{do}_s\,\mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \mid \hat{S}_{ctx}$, and $\hat{S} \xrightarrow{\mathsf{A}:\mathtt{do}_s\,\mathsf{a}} s[\hat{\gamma}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] \mid \hat{S}_{ctx} = \hat{S}'$. We have $\alpha_{\mathsf{A}}(\hat{S}) = s[\hat{c}] \mid \mathsf{A}[\mathtt{do}_s\,\mathsf{a}.\tilde{P} + \tilde{P}' \mid \tilde{Q}] \xrightarrow{\mathsf{A}:\mathtt{do}_s\,\mathsf{a}}_{\mathsf{A}} s[\hat{c}'] \mid \mathsf{A}[\tilde{P} \mid \tilde{Q}] = \alpha_{\mathsf{A}}(\hat{S}') = \tilde{S}'$.

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

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

By item 1 of Definition 8.1.8, since $\hat{c} \vdash_{\mathsf{A}} \phi$ and $\alpha_{\mathsf{A}}(\hat{\gamma}) = \hat{c}$, then $\hat{\gamma} \vdash \phi$. Then, by rule [ASK]:

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

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

Lemma A.2.26 *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 A.2.25 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 A.2.27 *For all* A*-safe systems* $\hat{S}'$*, and for all* `ask`*-free abstract systems* $\tilde{S}$*:*

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

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

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

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

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

[α-Do] Let $\tilde{c}, \tilde{c}'$ be abstract contracts such that $\tilde{c} \overset{\mathsf{a}}{\twoheadrightarrow}_{\mathsf{A}} \tilde{c}'$, 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 $\alpha_{\mathsf{A}}(\hat{\gamma}') = \tilde{c}'$. Let $\hat{\gamma}$ be a contract configuration such that $\alpha_{\mathsf{A}}(\hat{\gamma}) = \tilde{c}$. By Lemma A.2.18 and item 3 of Theorem 8.1.7 $\hat{\gamma} \twoheadrightarrow \hat{\gamma}'$. So $\hat{S} = s[\hat{\gamma}] \mid \hat{S}_{ctx} \to s[\hat{\gamma}'] \mid \hat{S}_{ctx}$.

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

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

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

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

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

$$\begin{aligned}
\alpha_{\mathsf{A}}(\hat{S}) &= \\
(x,y)(\tilde{S}_0 \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}) &= \\
(x)(\tilde{S}_0 \mid \{\downarrow_x \hat{c}\}_{\mathsf{A}}) &= && \text{since } y \text{ not free in } \hat{S}_{\mathsf{A}}, \text{ by Lemma A.2.20, } y \text{ is not free in } \hat{S}_0 \\
\tilde{S}. &
\end{aligned}$$

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

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

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:

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

[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 $\tilde{c} \xrightarrow{ctx:\mathsf{a}}\!\!\!\!\rightarrow_{\mathsf{A}} \tilde{c}'$, 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 A.2.18 and item 3 of Theorem 8.1.7 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 A.2.28 *For all* ask*-free context-abstract systems* $\tilde{S}$*:*

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

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 A.2.27 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 A.2.29 (Context-abstract obligations) *We define the following set:*

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

Definition A.2.30 (Context-abstract readiness) *Given a session name s and participant A, we define the set of context-abstract systems $\alpha\text{-}\mathrm{Rdy}_s^{\mathsf{A}}$ as the smallest set such that:*

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

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

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

Lemma A.2.31 *For all A-safe $\hat{S}$, and for all s:*

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

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

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

Proof.

- (1) We have that $\mathrm{O}_s^{\mathsf{A}}(\hat{S}) \subseteq \mathrm{O}_s^{\mathsf{A}}(\alpha_{\mathsf{A}}(\hat{S}))$ follows by item 1 of Theorem 8.1.7. The reverse inclusion holds by item 3 of Theorem 8.1.7.
- (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 A.2.30:
  - 1 By Lemma A.2.25 trivially follows that item 1 of Definition 8.1.4 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 A.2.25 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 8.1.4 holds for $\hat{S}$.

  - 3 By Lemma A.2.25 $\hat{S} \xrightarrow{\mathsf{A:if}}_{\star}$. Let $\tilde{S}'$ be an abstract system such that $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A:if}}_{\mathsf{A}} \tilde{S}'$. Again, Lemma A.2.25 guarantees $\hat{S} \xrightarrow{\mathsf{A:if}}_{\star} \hat{S}'$, 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 8.1.4 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 8.1.4:

  - 1 By Lemma A.2.23 trivially follows that item 1 of Definition A.2.30 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

  - 2 Let $\hat{S}'$ be the system such that $\hat{S} \xrightarrow{\mathsf{A:}\neq s\mathsf{,if}}_{\star} \hat{S}'$, with A $\alpha^{\star}$-ready at s in $\hat{S}'$. By Lemma A.2.23 $\alpha_{\mathsf{A}}(\hat{S}) \xrightarrow{\mathsf{A:}\neq s\mathsf{,if}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\tilde{S}')$, with A α-ready at s in $\alpha_{\mathsf{A}}(\hat{S}')$ by induction hypothesis. Clearly item 2 of Definition A.2.30 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

  - 3 Let $\hat{S}'$ be a system such that $\hat{S} \xrightarrow{\mathsf{a:if}}_{\star} \hat{S}'$. A must be ready at s in $\hat{S}'$. By Lemma A.2.23 $\alpha_{\mathsf{A}}(\tilde{S}) \xrightarrow{\mathsf{A:}\neq s\mathsf{,if}}_{\mathsf{A}} \alpha_{\mathsf{A}}(\hat{S})$, with A α-ready at s in $\alpha_{\mathsf{A}}(\tilde{S}')$ by induction hypothesis. Clearly item 3 of Definition A.2.30 holds for $\alpha_{\mathsf{A}}(\hat{S})$.

Lemma A.2.32 *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 A.2.31

Proof of Theorem 8.1.13:
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.

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 A.2.15 it follows that $\hat{S}$ is A-safe as well. Let $\tilde{S} = \alpha_{\mathsf{A}}(\hat{S})$. By Theorem A.2.24 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 A.2.32 guarantees that A is $\alpha^{\star}$-ready in $\hat{S}$. Hence, by Definition 7.1.5 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 A.2.28 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 A.2.32 it follows that A is α-ready in $\tilde{S}'$. Hence, by Definition 8.1.12 we conclude that $\tilde{P}$ is α-honest.

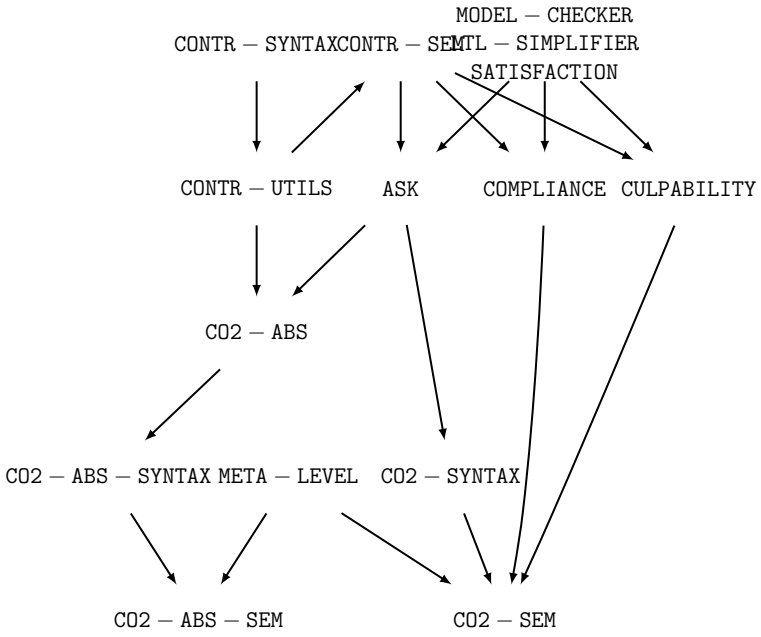

Figure A.3: Graph of module importation.

The diagram in Figure A.4 illustrates the dependencies among the proofs. The diagram Figure A.3 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`.

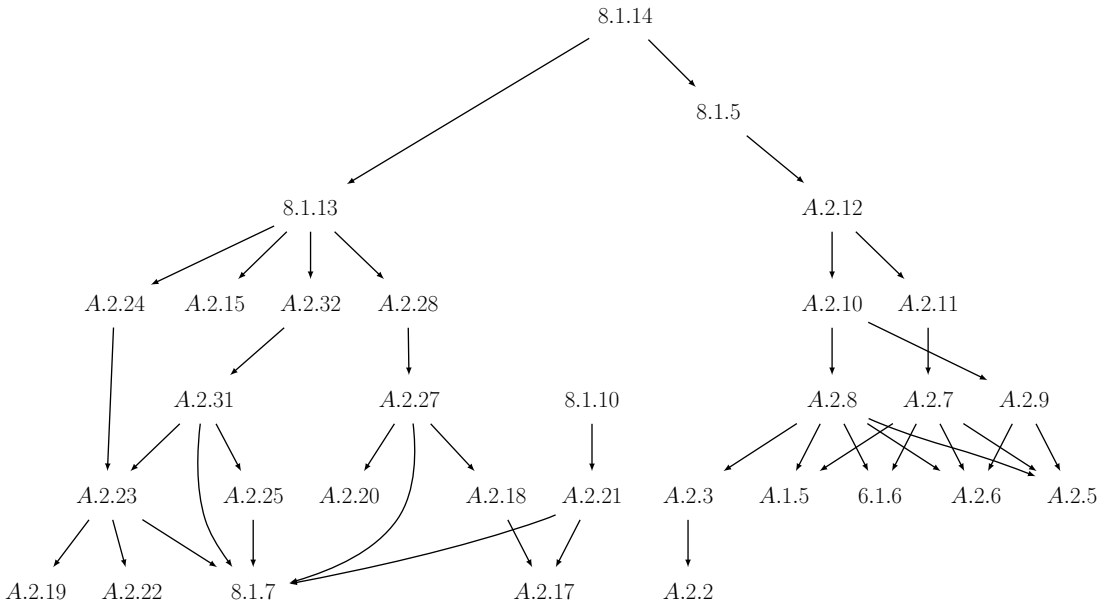

Figure A.4: Dependencies among the proofs.

Appendix B

Appendix for Part III

B.1 Proofs for Section 10.2

Lemma B.1.1 *For all $(p,\nu),(q,\eta)$ such that $(p,\nu) \bowtie (q,\eta)$:*

$$p = \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, R_i\}\,.\,p_i \implies q = \sum_{j\in J} \mathtt{a}_j?\{g_j, R_j\}\,.\,p_j$$
$$p = \sum_{i\in I} \mathtt{a}_i?\{g_i, R_i\}\,.\,p_i \implies q = \bigoplus_{j\in J} \mathtt{a}_j!\{g_j, R_j\}\,.\,p_j$$

Proof. Trivial.

Lemma B.1.2 *Let $\mathcal{R}$ be a coinductive compliance relation, and let $(p,\nu)\mathcal{R}(q,\eta)$. Then:*

1. *$(p,\nu) \mid (q,\eta)$ not deadlock*
2. *$(p,\nu) \mid (q,\eta) \xrightarrow{\tau} \gamma \implies \exists!\, p',q',\nu',\eta' : \gamma \xrightarrow{\tau} (p',\nu') \mid (q',\eta') \;\wedge\; (p',\nu')\mathcal{R}(q',\eta')$*
3. *$(p,\nu) \mid (q,\eta) \xrightarrow{\delta} (p',\nu') \mid (q',\eta') \implies (p',\nu')\mathcal{R}(q',\eta')$*

Proof. Assume that $(p,\nu)\mathcal{R}(q,\eta)$. We proceed by cases on the form of p, modulo unfolding of recursion (note that $\mathcal{R}$ does not talk about committed choices, i.e. terms of the form $[\mathtt{a}!\{g,R\}]\,p$). If $p = \mathbf{1}$, then by Definition 10.2.5 we have $q = \mathbf{1}$, and so all items are trivial. If p is an internal choice, i.e. $p = \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i$, by Definition 10.2.5 we have that $q = \sum_{j\in J} \mathtt{a}_j?\{g_j, T_j\}\,.\,q_j$. Take some δ and i such that $\nu + \delta \in [\![g_i]\!]$ (their existence is guaranteed by Definition 10.2.5). Let $\nu' = \nu + \delta$ and $\eta' = \eta + \delta$. Then:

$$\frac{\dfrac{\nu' \in [\![g_i]\!]}{(\bigoplus_{i\in I} \mathtt{a}_i!\{g_i,T_i\}\,.\,p_i, \nu') \xrightarrow{\tau} ([\mathtt{a}_i!\{g_i,T_i\}]\,p_i, \nu')}\,[\oplus]}{(\bigoplus_{i\in I} \mathtt{a}_i!\{g_i,T_i\}\,.\,p_i, \nu') \mid (q,\eta') \xrightarrow{\tau} ([\mathtt{a}_i!\{g_i,T_i\}]\,p_i,\ \nu') \mid (q,\eta')}\,[\text{S-}\oplus] \tag{B.1}$$

Hence, $(p,\nu) \mid (q,\eta)$ is not deadlock, which proves item 1.

For item 2, assume $(p,\nu) \mid (q,\eta) \xrightarrow{\tau} \gamma$. The derivation of such step must be as in (B.1), with $\delta = 0$ and $\gamma = ([\mathtt{a}_i!\{g_i, T_i\}]\, p_i,\ \nu) \mid (q,\eta)$. By Definition 10.2.5, there exists $j \in J$ such that $\mathtt{a}_i = \mathtt{a}_j$, $\eta \in [\![g_j]\!]$ and $p_i \mathcal{R} q_j$. Hence:

$$\dfrac{\dfrac{}{([\mathtt{a}_i!\{g_i,T_i\}]\, p_i,\ \nu) \xrightarrow{\mathtt{a}_i!} (p_i, \nu[T_i])}\,[!] \qquad \dfrac{\eta \in [\![g_j]\!]}{(q,\eta) \xrightarrow{\mathtt{a}_j?} (q_j, \eta[T_j])}\,[?]}{\gamma \xrightarrow{\tau} (p_i, \nu[T_i]) \mid (q_j, \eta[T_j])}\,[\text{S-}\tau]$$

Note that the target state is unique, because branch labels in choices are pairwise distinct.

For item 3, assume $(p,\nu) \mid (q,\eta) \xrightarrow{\delta} (p',\nu') \mid (q',\eta')$. Its derivation must be as follows:

$$\dfrac{\dfrac{\nu+\delta \in \mathrm{rdy}(p) = \downarrow \bigcup [\![g_i]\!]}{(p,\nu) \xrightarrow{\delta} (p, \nu+\delta)}\,[\textsc{Del}] \qquad \dfrac{\eta+\delta \in \mathrm{rdy}(q) = \mathbb{V}}{(q,\eta) \xrightarrow{\delta} (q, \eta+\delta)}\,[\textsc{Del}]}{(p,\nu) \mid (q,\eta) \xrightarrow{\delta} (p, \nu+\delta) \mid (q, \eta+\delta)}\,[\text{S-}\textsc{Del}] \tag{B.2}$$

Let $\nu' = \nu + \delta$ and $\eta' = \eta + \delta$. We have to show $(p,\nu')\mathcal{R}(q,\eta')$. By (B.2) we have that $\nu' \in \mathrm{rdy}(p)$. It remains to show that, whenever $\nu' + \delta' \in [\![g_i]\!]$, there exist j such that $\mathtt{a}_i = \mathtt{a}_j$ and $\eta' + \delta' \in [\![g_j]\!]$ and $(p_i, \nu' + \delta'[T_i])\mathcal{R}(q_j, \eta' + \delta'[T_j])$. This follows by the assumption $(p,\nu)\mathcal{R}(q,\eta)$. The case where p is an external choice is similar.

Proof of Lemma 10.2.6:
$p \bowtie q \iff \exists \mathcal{R}$coinductive compliance $:\ (p,\nu_0)\mathcal{R}(q,\eta_0)$

We prove the more general statement:

$$(p,\nu) \bowtie (q,\eta) \iff \exists \mathcal{R}\text{coinductive compliance} \ :\ (p,\nu)\mathcal{R}(q,\eta)$$

For the ($\Rightarrow$) direction we proceed by showing that $\bowtie$ is a coinductive compliance relation. Assume $(p,\nu) \bowtie (q,\eta)$. We show the case where $p = \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i$ (the case of external choice is similar, and the case $p = \mathbf{1}$ is trivial). By Lemma B.1.1, $q = \sum_{j\in J} \mathtt{a}_j?\{g_j, T_j\}\,.\,q_j$. Since $(p,\nu) \mid (q,\eta)$ is not deadlock, it must be $\nu \in \mathrm{rdy}(p)$. By Definition 10.2.5 it remains to show that, for all δ, i:

$$\nu+\delta \in [\![g_i]\!] \implies \exists j :\ \mathtt{a}_i = \mathtt{a}_j \ \wedge\ \eta+\delta \in [\![g_j]\!] \ \wedge\ (p_i, (\nu+\delta)[T_i]) \bowtie (q_j, (\eta+\delta)[T_j])$$

By contradiction, suppose this is not the case, and take a δ and an i such that:

$$\nu+\delta \in [\![g_i]\!] \wedge (\forall j :\ \mathtt{a}_i \neq \mathtt{a}_j \vee \eta+\delta \notin [\![g_j]\!] \vee (p_i, (\nu+\delta)[T_i]) \not\bowtie (q_j, (\eta+\delta)[T_j])$$

There are two cases. If $\delta = 0$, then:

$$(p,\nu) \mid (q,\eta) \xrightarrow{\tau} ([\mathtt{a}_i!\{g_i,T_i\}]\, p_i, \nu) \mid (q,\eta) = \gamma$$

If, for all j, $\mathtt{a}_i \neq \mathtt{a}_j \vee \nu+\delta \notin [\![g_j]\!]$, then γ is deadlock. Otherwise:

$$\gamma \xrightarrow{\tau} (p_i, (\nu+\delta)[T_i]) \mid (q_j, (\eta+\delta)[T_j]) = \gamma'$$

with $(p_i, (\nu + \delta)[T_i]) \not\bowtie (q_j, (\eta + \delta)[T_j])$, and so, by Definition 10.2.1, there exists some deadlock configuration γ'' such that $\gamma' \to^* \gamma''$. Hence $(p, \nu) \not\bowtie (q, \eta)$. If $\delta > 0$:

$$(p, \nu) \mid (q, \eta) \xrightarrow{\delta} (p, \nu + \delta) \mid (q, \eta + \delta)$$

The thesis follows by an argument similar to the case with $\delta = 0$.
The ($\Leftarrow$) direction is a straightforward consequence of Lemma B.1.2

B.2 Proofs for Section 11.1

Proof of Theorem 11.1.4:
For all p and Γ with $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$, there exists unique $\mathcal{K}$ such that $\Gamma \vdash p : \mathcal{K}$.

We have to prove that:

$$\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma) \implies \exists ! \mathcal{K} \ : \ \Gamma \vdash p : \mathcal{K}$$

Let Γ as in the statement. By induction on the structure of p, we have the following cases:

- $p = \mathbf{1}$. Trivial.
- p is a choice (internal or external). Straightforward by the induction hypothesis.
- $p = X$. Since $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$, the thesis follows by rule [T-Var].
- $p = \mathtt{rec}\, X \,.\, p'$. Since $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma)$, then for all $\mathcal{K}'$: $\mathrm{fv}(p') \subseteq \mathrm{fv}(p) \cup \{X\} \subseteq \mathrm{dom}(\Gamma) \cup \{X\} = \mathrm{dom}(\Gamma, X : \mathcal{K}')$. Hence, by the induction hypothesis we have that $\exists ! \mathcal{K}'' : \Gamma, X : \mathcal{K}' \vdash p' : \mathcal{K}''$. The thesis follows by rule [T-Rec].

Lemma B.2.1 *For all $\mathcal{K}, \mathcal{K}'$ such that $\mathcal{K} \subseteq \mathcal{K}'$, and for all sets of clocks T:*

$$\downarrow \mathcal{K} \subseteq \downarrow \mathcal{K}' \qquad \textit{and} \qquad \mathcal{K}[T]^{-1} \subseteq \mathcal{K}'[T]^{-1}$$

Proof. Straightforward by Definition 5.4.2.

Hereafter, we assume substitutions to be capture avoiding.

Lemma B.2.2 (Substitution) *Let $\Gamma \vdash p' : \mathcal{K}'$. Then, for all p:*

$$\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K} \iff \Gamma \vdash p\{^{p'}/_X\} : \mathcal{K}$$

Proof. We prove that, under the assumption $\Gamma \vdash p' : \mathcal{K}'$, the following items hold:

1. $\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K} \implies \exists \mathcal{K}'' \supseteq \mathcal{K} : \ \Gamma \vdash p\{^{p'}/_X\} : \mathcal{K}''$.
2. $\Gamma \vdash p\{^{p'}/_X\} : \mathcal{K} \implies \exists \mathcal{K}'' \supseteq \mathcal{K} : \ \Gamma, X : \mathcal{K}' \vdash p : \mathcal{K}''$.

Before proving the two items, we show that together they imply the thesis. Assume that $\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K}$. By item 1, $\Gamma \vdash p\{^{p'}/_X\} : \mathcal{K}''$ for some $\mathcal{K}'' \supseteq \mathcal{K}$. Then, by item 2, $\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K}'''$ for some $\mathcal{K}''' \supseteq \mathcal{K}''$. Therefore, by uniqueness of kinding, $\mathcal{K}'' \subseteq \mathcal{K}''' = \mathcal{K} \subseteq \mathcal{K}''$. The other direction follows by a similar argument.

To prove item 1, assume $\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K}$. We proceed by induction on the typing rules.

- [T-**1**], [T-Var]. Trivial.
- [T-$\oplus$]. We have:

$$\frac{\Gamma, X : \mathcal{K}' \vdash p_i : \mathcal{K}_i}{\Gamma, X : \mathcal{K}' \vdash \bigoplus \mathtt{a}_i \,!\, \{g_i, T_i\} \,.\, p_i : \left(\bigcup\!\downarrow [\![g_i]\!]\right) \setminus \left(\bigcup\!\downarrow ([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1})\right)}$$

  By the induction hypothesis:

$$\frac{\Gamma \vdash p_i\{^{p'}/_X\} : \tilde{\mathcal{K}}_i \supseteq \mathcal{K}_i}{\Gamma \vdash \bigoplus \mathtt{a}_i \,!\, \{g_i, T_i\} \,.\, (p_i\{^{p'}/_X\}) : \left(\bigcup\!\downarrow [\![g_i]\!]\right) \setminus \left(\bigcup\!\downarrow ([\![g_i]\!] \setminus \tilde{\mathcal{K}}_i[T_i]^{-1})\right)}$$

  The thesis follows by Lemma B.2.1.

- [T-+]. Similar to [T-$\oplus$].
- [T-Rec]. p must have the form $\mathtt{rec}\, Y \,.\, p''$. If $X = Y$ the thesis is trivial; otherwise:

$$\frac{\exists \mathcal{K}_0, \mathcal{K}'_0 : \ \Gamma, X : \mathcal{K}', Y : \mathcal{K}_0 \vdash p'' : \mathcal{K}'_0}{\Gamma, X : \mathcal{K}' \vdash \mathtt{rec}\, Y \,.\, p'' : \bigcup \left\{ \tilde{\mathcal{K}} \supseteq \tilde{\mathcal{K}}' \,\middle|\, \Gamma, X : \mathcal{K}', Y : \tilde{\mathcal{K}} \vdash p'' : \tilde{\mathcal{K}}' \right\} = \mathcal{K}}$$

  Since $X \neq Y$, then $p\{^{p'}/_X\} = \mathtt{rec}\, Y \,.\, (p''\{^{p'}/_X\})$, and for all $\mathcal{K}_0, \mathcal{K}'_0$:

$$\Gamma, X : \mathcal{K}', Y : \mathcal{K}_0 \vdash p' : \mathcal{K}'_0 \iff \Gamma, Y : \mathcal{K}_0, X : \mathcal{K}' \vdash p' : \mathcal{K}'_0 \tag{B.3}$$

  The kinding derivation for $p\{^{p'}/_X\}$ must be as follows:

$$\frac{\exists \mathcal{K}_0, \mathcal{K}'_0 : \ \Gamma, Y : \mathcal{K}_0 \vdash (p''\{^{p'}/_X\}) : \mathcal{K}'_0}{\Gamma \vdash p\{^{p'}/_X\} : \bigcup \left\{ \tilde{\mathcal{K}} \,\middle|\, \Gamma, Y : \tilde{\mathcal{K}} \vdash (p''\{^{p'}/_X\}) : \tilde{\mathcal{K}}' \wedge \tilde{\mathcal{K}} \subseteq \tilde{\mathcal{K}}' \right\} = \mathcal{K}''} \tag{B.4}$$

  We first show that the premise of (B.4) holds. Since substitutions are capture avoiding, Y is not free in p', and hence $\Gamma, Y : \mathcal{K} \vdash p' : \mathcal{K}'$ as well as Γ. Then, by Equation (B.3) together with the induction hypothesis, the thesis follows. It remains to show $\mathcal{K} \subseteq \mathcal{K}''$. If $\mathcal{K}$ is empty the thesis holds trivially. Otherwise, take some $\tilde{\mathcal{K}}, \tilde{\mathcal{K}}'$ such that:

$$\Gamma, X : \mathcal{K}', Y : \tilde{\mathcal{K}} \vdash p'' : \tilde{\mathcal{K}}' \wedge \tilde{\mathcal{K}} \subseteq \tilde{\mathcal{K}}'$$

  By (B.3), together with the induction hypothesis, we have that, for some $\tilde{\mathcal{K}}''$:

$$\Gamma, Y : \tilde{\mathcal{K}} \vdash p''\{^{p'}/_X\} : \tilde{\mathcal{K}}'' \supseteq \tilde{\mathcal{K}}'$$

  from which the thesis follows.

To prove item 2, assume $\Gamma \vdash p\{^{p'}/_X\} : \mathcal{K}$. We proceed by induction on the structure of p.

- $p = \mathbf{1}$. Trivial.
- $p = \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i$.

$$\frac{\Gamma \vdash p_i\{^{p'}/_X\} : \mathcal{K}_i}{\Gamma \vdash \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,(p_i\{^{p'}/_X\}) : \bigcup\!\downarrow [\![g_i]\!] \setminus \bigcup\!\downarrow ([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1})}\,{}^{[\text{T-}\oplus]}$$

  By induction hypothesis:

$$\frac{\Gamma\{^{\mathcal{K}_i}/_X\} \vdash p_i : \tilde{\mathcal{K}}_i \supseteq \mathcal{K}_i}{\Gamma, X : \mathcal{K}' \vdash \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i : \bigcup\!\downarrow [\![g_i]\!] \setminus \bigcup\!\downarrow ([\![g_i]\!] \setminus \tilde{\mathcal{K}}_i[T_i]^{-1})}\,{}^{[\text{T-}\oplus]}$$

  The thesis follows by Lemma B.2.1.

- $p = \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$. Similar to the internal choice case.
- $p = Y$. If $Y \neq X$ the thesis follows trivially. Otherwise, let $p = X$. Then $\Gamma \vdash p\{^{p'}/_X\} = p' : \mathcal{K}'$ and $\Gamma, X : \mathcal{K}' \vdash p = X : \mathcal{K}'$.
- $p = \mathtt{rec}\,Y\,.\,p''$. Assume $X \neq Y$ (otherwise the thesis holds trivially).

$$\frac{\exists \mathcal{K}_0, \mathcal{K}'_0 :\ \Gamma, Y : \mathcal{K}_0 \vdash (p''\{^{p'}/_X\}) : \mathcal{K}'_0}{\Gamma \vdash p\{^{p'}/_X\} : \bigcup \left\{ \tilde{\mathcal{K}} \;\middle|\; \Gamma, Y : \tilde{\mathcal{K}} \vdash (p''\{^{p'}/_X\}) : \tilde{\mathcal{K}}' \wedge \tilde{\mathcal{K}} \subseteq \tilde{\mathcal{K}}' \right\} = \mathcal{K}}$$

  As in the proof of (1), we have $\Gamma, Y : \mathcal{K} \vdash p' : \mathcal{K}'$. Then, by (B.3) and the induction hypothesis:

$$\frac{\exists \mathcal{K}_0, \mathcal{K}'_0 :\ \Gamma, X : \mathcal{K}', Y : \mathcal{K}_0 \vdash p'' : \mathcal{K}'_0}{\Gamma, X : \mathcal{K}' \vdash \mathtt{rec}\,Y\,.\,p'' : \bigcup \left\{ \tilde{\mathcal{K}} \;\middle|\; \Gamma, X : \mathcal{K}', Y : \tilde{\mathcal{K}} \vdash p'' : \tilde{\mathcal{K}}' \wedge \tilde{\mathcal{K}} \subseteq \tilde{\mathcal{K}}' \right\} = \mathcal{K}''}$$

  It remains to prove $\mathcal{K} \subseteq \mathcal{K}''$. If $\mathcal{K}$ is empty the thesis holds trivially. Otherwise, take some $\tilde{\mathcal{K}}, \tilde{\mathcal{K}}'$ such that:

$$\Gamma, Y : \tilde{\mathcal{K}} \vdash p''\{^{p'}/_X\} : \tilde{\mathcal{K}}' \wedge \tilde{\mathcal{K}} \subseteq \tilde{\mathcal{K}}'$$

  By (B.3), together with the induction hypothesis, we have that, for some $\tilde{\mathcal{K}}''$:

$$\Gamma, X : \mathcal{K}', Y : \tilde{\mathcal{K}} \vdash p'' : \tilde{\mathcal{K}}'' \supseteq \tilde{\mathcal{K}}'$$

  from which the thesis follows.

Lemma B.2.3 (Substitution under dual) *Let $\Gamma \vdash p' : \mathcal{K}$, with X not free in p'. Then, for all p such that p is kindable with environment $\Gamma, X : \mathcal{K}$:*

$$\mathrm{co}_{\Gamma, X:\mathcal{K}}(p)\,\{^{\mathrm{co}_\Gamma(p')}/_X\} = \mathrm{co}_\Gamma(p\{^{p'}/_X\})$$

Proof. By induction on the structure of p:

- $p = \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i$:

  $\mathrm{co}_{\Gamma,X:\mathcal{K}}(p)\,\{^{\mathrm{co}_\Gamma(p')}/_X\} =$

  $$\begin{array}{lll} = & \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,(\mathrm{co}_{\Gamma,X:\mathcal{K}}(p_i)\,\{^{\mathrm{co}_\Gamma(p')}/_X\}) & \\ = & \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,\mathrm{co}_\Gamma(p_i\{^{p'}/_X\}) & \text{by induction hypothesis} \\ = & \mathrm{co}_\Gamma(p\{^{p'}/_X\}) & \end{array}$$

- $p = \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$: Let $\Gamma, X:\mathcal{K} \vdash p_i : \mathcal{K}_i$ for all $i \in I$. Then:

  $\mathrm{co}_{\Gamma,X:\mathcal{K}}(p)\,\{^{\mathrm{co}_\Gamma(p')}/_X\} =$

  $$\begin{array}{lll} = & \bigoplus \mathtt{a}_i\,!\,\{g_i \wedge \mathcal{K}_i[T_i]^{-1}, T_i\}\,.\,(\mathrm{co}_{\Gamma,X:\mathcal{K}}(p_i)\,\{^{\mathrm{co}_\Gamma(p')}/_X\}) & \\ = & \bigoplus \mathtt{a}_i\,!\,\{g_i \wedge \mathcal{K}_i[T_i]^{-1}, T_i\}\,.\,\mathrm{co}_\Gamma(p_i\{^{p'}/_X\}) & \text{by induction hypothesis} \\ = & \mathrm{co}_\Gamma(p\{^{p'}/_X\}) & \text{by Lemma B.2.2} \end{array}$$

- $p = \mathbf{1}$: Trivial

- $p = Y$: Trivial, whether or not $Y = X$

- $p = \mathtt{rec}\,Y\,.\,p''$: If $Y = X$ the thesis is trivial. Otherwise, assume $\Gamma, X:\mathcal{K} \vdash \mathtt{rec}\,Y\,.\,p'' : \mathcal{K}'$. First note that, since we are assuming capture avoiding substitutions, Y must not be free in p'. Indeed, if Y were free in p', then Y would be captured in $p\{^{p'}/_X\}$. Then:

  $\mathrm{co}_{\Gamma,X:\mathcal{K}}(p)\,\{^{\mathrm{co}_\Gamma(p')}/_X\} =$

  $$\begin{array}{lll} = & \mathtt{rec}\,Y\,.\,(\mathrm{co}_{\Gamma,X:\mathcal{K},Y:\mathcal{K}'}(p'')\,\{^{\mathrm{co}_\Gamma(p')}/_X\}) & \text{by Definition 11.1.5} \\ = & \mathtt{rec}\,Y\,.\,(\mathrm{co}_{\Gamma,Y:\mathcal{K}',X:\mathcal{K}}(p'')\,\{^{\mathrm{co}_\Gamma(p')}/_X\}) & \text{since } X \neq Y \\ = & \mathtt{rec}\,Y\,.\,(\mathrm{co}_{\Gamma,Y:\mathcal{K}',X:\mathcal{K}}(p'')\,\{^{\mathrm{co}_{\Gamma,Y:\mathcal{K}'}(p')}/_X\}) & \text{since } Y \text{ is not free in } p' \\ = & \mathtt{rec}\,Y\,.\,\mathrm{co}_{\Gamma,Y:\mathcal{K}'}(p''\{^{p'}/_X\}) & \text{by induction hypothesis} \\ = & \mathrm{co}_{\Gamma,Y:\mathcal{K}'}(p\{^{p'}/_X\}) & \text{by Definition 11.1.5} \\ = & \mathrm{co}_\Gamma(p\{^{p'}/_X\}) & \text{since } Y \text{ is not free in } p\{^{p'}/_X\} \end{array}$$

Note that the induction hypothesis is trivially applicable: p is kindable in $\Gamma, X:\mathcal{K}$, necessarily by rule [T-Rec]. The premise of rule [T-Rec] implies p'' is kindable in $\Gamma, X:\mathcal{K}, Y:\mathcal{K}'$.

Lemma B.2.4 *For all kindable p and q, we have that: $p \equiv q \implies \mathrm{co}(p) \equiv \mathrm{co}(q)$.*

Proof. The thesis follows by the following more general statement:

$$\forall \Gamma \;:\; \forall p, q \text{ kindable in } \Gamma \;:\; p \equiv q \implies \mathrm{co}_\Gamma(p) \equiv \mathrm{co}_\Gamma(q)$$

where, as usual, $\mathtt{rec}\,X\,.\,p \equiv p\{^{\mathtt{rec}\,X.p}/_X\}$ does not hold when the substitution captures some variable free in $\mathtt{rec}\,X\,.\,p$. The proof is by easy induction on the structure of p, using Lemma B.2.3 in the case $p = \mathtt{rec}\,X\,.\,p'$.

Lemma B.2.5 *Every closed p is structurally equivalent to a TST of the following shape:*

$$\mathbf{1} \qquad \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i \qquad \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$$

Proof. Trivial.

Proof of Theorem 11.1.6: Soundness

Let:

$$\mathcal{R} = \{((p,\nu),(\mathrm{co}(p)\,,\nu)) \mid \exists \mathcal{K} :\ \Gamma \vdash p : \mathcal{K} \wedge \nu \in \mathcal{K}\}$$

By Lemma 10.2.6, it is enough to show that $\mathcal{R}$ is a coinductive compliance relation (Definition 10.2.5). Assume that $\Gamma \vdash p : \mathcal{K}$ and $\nu \in \mathcal{K}$. We proceed by cases on the shape of p. According to Lemma B.2.5, we have the following cases:

- $p \equiv \mathbf{1}$. By Lemma B.2.4 $\mathrm{co}(p) \equiv \mathbf{1}$, from which the thesis follows.
- $p \equiv \bigoplus \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i$. By Lemma B.2.4, $\mathrm{co}(p) \equiv \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,\mathrm{co}(p_i)$. By rule [T-⊕] it follows that $\nu \in \mathrm{rdy}(p)$. To conclude:

  $$\forall \delta, i :\ \nu + \delta \in [\![g_i]\!] \implies (p_i, (\nu+\delta)[T_i])\mathcal{R}(\mathrm{co}(p_i)\,,(\nu+\delta)[T_i])$$

  Let $\vdash p_i : \mathcal{K}_i$. By the typing rule [T-⊕] we have that:

  $$\begin{aligned}\mathcal{K} &= \big(\bigcup_{i\in I}\!\downarrow [\![g_i]\!]\big) \setminus \big(\bigcup_{i\in I}\!\downarrow ([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1})\big)\\ &= \big(\bigcup_{i\in I}\!\downarrow [\![g_i]\!]\big) \setminus \{\nu \mid \exists \delta, i : \nu+\delta \in [\![g_i]\!] \wedge \nu+\delta[T_i] \notin \mathcal{K}_i\}\end{aligned}$$

  from which the thesis follows.
- $p \equiv \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$. By Lemma B.2.4, $\mathrm{co}(p) = \bigoplus \mathtt{a}_i!\{g_i \wedge \mathcal{K}_i, T_i\}\,.\,\mathrm{co}(p_i)$ with $\vdash p_i : \mathcal{K}_i$. By rule [T-+]:

  $$\mathcal{K} = \bigcup\!\downarrow \big([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1}\big) = \{\nu \mid \exists \delta, i :\ \nu+\delta \in [\![g_i]\!] \wedge (\nu+\delta)[T_i] \in \mathcal{K}_i\}$$

  from which the thesis follows.

Definition B.2.6 *We define the function* Φ *from TSTs to sets of clock valuations as follows:*

$$\Phi(p) \stackrel{def}{=} \{\nu \mid \exists q, \eta : (p,\nu) \bowtie (q,\eta)\}$$

Lemma B.2.7 *Let* $\Gamma = X_1 : \Phi(p_1), \ldots, X_m : \Phi(p_m)$*, where all* p_i *are closed, and let* $\vec{X} = (X_1, \ldots, X_m), \vec{p} = (p_1, \ldots, p_m)$*. Then, for all* p *such that* $\mathrm{fv}(p) \subseteq \vec{X}$*:*

$$\Gamma \vdash p : \mathcal{K} \implies \Phi(p\{\vec{p}/\vec{X}\}) \subseteq \mathcal{K}$$

Proof. We start with an auxiliary definition. The recursion nesting level (RL) for a TST is inductively defined as follows:

$$\begin{array}{ll}\mathrm{RL}(\mathbf{1}) \stackrel{\mathrm{def}}{=} \mathrm{RL}(X) \stackrel{\mathrm{def}}{=} 0 & \mathrm{RL}(\sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i) \stackrel{\mathrm{def}}{=} max\ \{\mathrm{RL}(p_i)\}_{i\in I}\\ \mathrm{RL}(\mathtt{rec}\,X\,.\,p) \stackrel{\mathrm{def}}{=} 1 + \mathrm{RL}(p) & \mathrm{RL}(\bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i) \stackrel{\mathrm{def}}{=} max\ \{\mathrm{RL}(p_i)\}_{i\in I}\end{array}$$

$$\begin{array}{rcl} \mathrm{RL}(\mathbf{1}) & \stackrel{\text{def}}{=} & 0 \\ \mathrm{RL}(\sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i) & \stackrel{\text{def}}{=} & max\ \{\mathrm{RL}(p_i)\}_{i\in I} \\ \mathrm{RL}(\bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i) & \stackrel{\text{def}}{=} & max\ \{\mathrm{RL}(p_i)\}_{i\in I} \\ \mathrm{RL}(X) & \stackrel{\text{def}}{=} & 0 \\ \mathrm{RL}(\mathtt{rec}\,X\,.\,p) & \stackrel{\text{def}}{=} & 1 + \mathrm{RL}(p) \end{array}$$

We then define the relation $\prec$ as:

$$p \prec q \text{ whenever } \mathrm{RL}(p) < \mathrm{RL}(q) \vee (\mathrm{RL}(p) = \mathrm{RL}(q) \wedge p \text{ is a strict subterm of } q)$$

It is trivial to check that $\prec$ is a well-founded relation (exploiting the fact that the strict subterm relation is well-founded as well). We then proceed by well-founded induction on $\prec$. We have the following cases, according to the form of p:

- $p = \mathbf{1}$. Since $\mathcal{K} = \mathbb{V}$ (by kinding rule [T-1]), the thesis follows trivially.
- $p = X_i$, for some $i \in \{1, \ldots, m\}$. $\mathcal{K} = \Gamma(X_i) = \Phi(p_i) = \Phi(X_i\{\vec{p}/\vec{x}\})$.
- $p = \sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$.

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \text{for } i \in I}{\Gamma \vdash \sum_{i\in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i : \bigcup_{i\in I} \downarrow \left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1}\right) = \mathcal{K}} \text{ [T-+]}$$

Since, for all $i \in I$ it holds $p_i \prec p$, by the induction hypothesis:

$$\Phi(p_i\{\vec{p}/\vec{x}\}) \subseteq \mathcal{K}_i \tag{B.5}$$

Now suppose, by contradiction, $\Phi(p\{\vec{p}/\vec{x}\}) \not\subseteq \mathcal{K}$. Then there exist ν, η, q such that $\Phi(p\{\vec{p}/\vec{x}\}) \ni \nu \notin \mathcal{K}$ and $(p\{\vec{p}/\vec{x}\}, \nu) \bowtie (q, \eta)$. By Definition 10.2.5 we have $q = \bigoplus_{j\in J} \mathtt{a}_j!\{g_j, T_j\}\,.\,q_j$, with $\eta \in \mathrm{rdy}(q)$ and

$$\forall \delta, j :\ \eta + \delta \in [\![g_j]\!] \implies$$

$$\exists i : \mathtt{a}_i = \mathtt{a}_j \wedge \nu + \delta \in [\![g_i]\!] \wedge (p_i\{\vec{p}/\vec{x}\}, (\nu + \delta)[R_i]) \bowtie (q_j, (\eta + \delta)[R_j])$$

But then, by Definition B.2.6 and Equation (B.5):

$$(\nu + \delta)[R_i] \in \Phi(p_i\{\vec{p}/\vec{x}\}) \subseteq \mathcal{K}_i$$

Since $\mathcal{K}$ is past closed (i.e. for all ν, δ: $\nu + \delta \in \mathcal{K} \implies \nu \in \mathcal{K}$), it follows $\nu \in \mathcal{K}$, a contradiction.

- $p = \bigoplus_{i\in I} \mathtt{a}_i!\{g_i, T_i\}\,.\,p_i$. Similar to the external choice case.
- $p = \mathtt{rec}\,Y\,.\,p'$. By rule [T-Rec]:

$$\frac{\Gamma, Y : \mathcal{K}_0 \vdash p' : \mathcal{K}'_0}{\Gamma \vdash \mathtt{rec}\,Y\,.\,p' : \bigcup \{\mathcal{K} \mid \exists \mathcal{K}' \supseteq \mathcal{K} :\ \Gamma, Y : \mathcal{K} \vdash p' : \mathcal{K}'\} = \mathcal{K}}$$

To prove $\Phi(p\{\vec{p}/\vec{x}\}) \subseteq \mathcal{K}$, it is enough to show that, for some $\mathcal{K}' \supseteq \Phi(p\{\vec{p}/\vec{x}\})$:

$$\Gamma, Y : \Phi(p\{\vec{p}/\vec{x}\}) \vdash p' : \mathcal{K}'$$

Since compliance is preserved by unfolding of recursion (because TSTs are considered up-to $\equiv$ in Definition 10.1.3), by Definition B.2.6 it follows that:

$$\Phi(p\{\vec{p}/\vec{x}\}) = \Phi(p'\{\vec{p}/\vec{x}\setminus\{Y\}\}\{^{\mathtt{rec}\, Y.(p'\{\vec{p}/\vec{x}\setminus\{Y\}\})}/_{Y}\}) = \Phi(p'\{\vec{p'}/\vec{x'}\})$$

where $\vec{X}'$ and $\vec{p'}$ are defined as follows:

$$\vec{X}' = \begin{cases} \vec{X}Y & \text{if } Y \notin \vec{X} \\ \vec{X} & \text{otherwise} \end{cases} \qquad \vec{p'} = \begin{cases} (p_1, \ldots, p_{i-1}, p\{\vec{p}/\vec{x}\}, p_{i+1}, \ldots, p_m) & \text{if } Y = X_i \\ \vec{p}(p\{\vec{p}/\vec{x}\}) & \text{otherwise} \end{cases}$$

Since $\mathrm{RL}(p') < \mathrm{RL}(p)$, by the induction hypothesis we have:

$$\Gamma, Y : \Phi(p'\{\vec{p'}/\vec{x'}\}) \vdash p' : \mathcal{K}' \implies \Phi(p'\{\vec{p'}/\vec{x'}\}) \subseteq \mathcal{K}'$$

Since $\mathrm{fv}(p) \subseteq \vec{X} = \mathrm{dom}(\Gamma)$ it follows that $\mathrm{fv}(p') \subseteq \vec{X}' = \mathrm{dom}(\Gamma, Y : \Phi(p'\{\vec{p'}/\vec{x'}\}))$, and hence, by Theorem 11.1.4, the premise of the induction hypothesis is satisfied, and we can conclude $\Phi(p'\{\vec{p'}/\vec{x'}\}) \subseteq \mathcal{K}'$.

Proof of Theorem 11.1.8: Completeness

Suppose $\vdash p : \mathcal{K}$ and $\exists q, \eta.\ (p, \nu) \bowtie (q, \eta)$. By instantiating Lemma B.2.7 with the empty kinding environment, we obtain $\Phi(p) \subseteq \mathcal{K}$. Hence, by Definition B.2.6 we conclude that $\nu \in \mathcal{K}$.

Proof of Lemma 11.1.9: For all p, q, ν, η and p', ν' such that $\vdash p' : \mathcal{K}$ and $\nu' \in \mathcal{K}$:

$$(p, \nu) \bowtie (p', \nu') \ \wedge\ (\mathrm{co}(p'), \nu') \bowtie (q, \eta) \implies (p, \nu) \bowtie (q, \eta)$$

We prove that the relation:

$$\mathcal{R} \ = \ \{((p, \nu), (q, \eta)) \mid \exists p', \nu' : (p, \nu) \bowtie (p', \nu') \wedge (\mathrm{co}(p'), \nu') \bowtie (q, \eta)\}$$

is a coinductive compliance relation (Definition 10.2.5). We proceed by cases on the form of p:

- $p = \mathbf{1}$. Trivial.

- $p = \bigoplus \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i$. It must be:

$$p' = \sum \mathtt{a}_j?\{g_j, T_j\}\,.\,{p_j}' \quad \text{and} \quad \mathrm{co}(p') = \bigoplus \mathtt{a}_j\,!\,\{g_j \wedge \mathcal{K}_j[T_j]^{-1}, T_j\}\,.\,\mathrm{co}({p_j}')$$

  with $\vdash p_j : \mathcal{K}_j$ for all $j \in J$ and, by Definition 10.2.5, for all δ, i such that $\nu + \delta \in [\![g_i]\!]$, there exists j such that $\mathtt{a}_i = \mathtt{a}_j$, $\nu' + \delta \in [\![g_j]\!]$, $(p_i, (\nu + \delta)[T_i]) \bowtie ({p_j}', (\nu' + \delta)[T_j])$, and $\nu + \delta \in \mathrm{rdy}(p)$. Hence, $q = \sum \mathtt{a}_k?\{g_k, T_k\}\,.\,p_k$, and for all δ, j such that $\nu' + \delta \in [\![g_j]\!]$, there exists k such that $\mathtt{a}_k = \mathtt{a}_j$, $\eta + \delta \in [\![g_k]\!]$, $(\mathrm{co}({p_j}')\,, (\nu' + \delta)[T_j]) \bowtie (q_k, (\eta + \delta)[T_k])$, and $\nu' \in \mathrm{rdy}(\mathrm{co}(p'))$. Now, assume $\nu + \delta \in g_i$ for some δ, i, and suppose, by contradiction, $\nu' + \delta \notin \mathcal{K}_j[T_j]^{-1}$ for some j such that $\mathtt{a}_i = \mathtt{a}_j$. But then we should have $(p_i, (\nu + \delta)[T_i]) \bowtie ({p_j}', (\nu' + \delta)[T_j])$ and $(\nu' + \delta)[T_j] \notin \mathcal{K}_j$ — contradiction by Theorem 11.1.8.

- $p = \sum \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$. It must be:

$$p' = \bigoplus \mathtt{a}_j\,!\,\{g_j, T_j\}\,.\,{p_j}' \qquad \text{and} \qquad \mathrm{co}(p') = \sum \mathtt{a}_j?\{g_j, T_j\}\,.\,\mathrm{co}(p_j)$$

  and for all δ, j such that $\nu' + \delta \in [\![g_j]\!]$, there exists i such that $\mathtt{a}_j = \mathtt{a}_i$, $\nu + \delta \in [\![g_i]\!]$, $({p_j}', \nu' + \delta[T_j]) \bowtie (p_i, \nu + \delta[T_j])$, and $\nu' \in \mathrm{rdy}(p')$. Hence, $q = \bigoplus \mathtt{a}_k?\{g_k, T_k\}\,.\,p_k$, and for all δ, k such that $\eta + \delta \in [\![g_k]\!]$, there exists j such that $\mathtt{a}_k = \mathtt{a}_j$, $\nu' + \delta \in [\![g_j]\!]$, $(q_k, (\eta + \delta)[T_k]) \bowtie ({p_j}', (\nu + \delta)[T_i])$, and $\eta \in \mathrm{rdy}(q)$. The thesis follows by the composition of the above.

B.3 Proofs for Section 11.2

Proof of Lemma 11.2.1:
The function $F_{\Gamma,X,p}$ is monotonic, for all Γ, X, p with $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma) \cup \{X\}$.

We define the partial order $\sqsubseteq$ between environments as follows:

$$\Gamma \sqsubseteq \Gamma' \iff \forall X \in \mathrm{dom}(\Gamma) : \Gamma(X) \subseteq \Gamma'(X)$$

and we show that, for all Γ, Γ' such that $\Gamma \sqsubseteq \Gamma'$:

$$\Gamma \vdash p : \mathcal{K} \implies \exists \mathcal{K}' \supseteq \mathcal{K} :\ \Gamma' \vdash p : \mathcal{K}' \tag{B.6}$$

Suppose $\Gamma \sqsubseteq \Gamma'$ and $\Gamma \vdash p : \mathcal{K}$. By induction on the structure of p:

- $p = \mathbf{1}$: Trivial.
- $p = \sum_{i \in I} \mathtt{a}_i?\{g_i, T_i\}\,.\,p_i$: By rule [T-+], it must be:

$$\mathcal{K} = \bigcup_{i \in I} \downarrow \left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1} \right), \text{ with } \Gamma \vdash p_i : \mathcal{K}_i$$

Then, by the induction hypothesis, $\forall i \in I : \exists \mathcal{K}'_i \supseteq \mathcal{K}_i$ such that:

$$\Gamma' \vdash p_i : \mathcal{K}'_i$$

The thesis follows by Lemma B.2.1.

- $p = \bigoplus_{i \in I} \mathtt{a}_i\,!\,\{g_i, T_i\}\,.\,p_i$: Similar to the external choice case.
- $p = X$: Trivial by definition of $\sqsubseteq$ and kinding rule [T-Var].
- $p = \mathtt{rec}\,X\,.\,p'$: By rule [T-Rec], it must be, for some $\mathcal{K}_0, \mathcal{K}'_0$:

$$\frac{\Gamma, X : \mathcal{K}_0 \vdash p' : \mathcal{K}'_0}{\Gamma \vdash \mathtt{rec}\,X\,.\,p' : \bigcup \{\mathcal{K} \mid \exists \mathcal{K}' \supseteq \mathcal{K} :\ \Gamma, X : \mathcal{K} \vdash p' : \mathcal{K}'\} = \mathcal{K}} \tag{B.7}$$

By the induction hypothesis, for some $\mathcal{K}''_0$:

$$\frac{\Gamma', X : \mathcal{K}_0 \vdash p' : \mathcal{K}''_0}{\Gamma' \vdash \mathtt{rec}\,X\,.\,p' : \bigcup \{\mathcal{K} \mid \exists \mathcal{K}' \supseteq \mathcal{K} :\ \Gamma', X : \mathcal{K} \vdash p' : \mathcal{K}'\} = \mathcal{K}'} \tag{B.8}$$

It remains to show $\mathcal{K} \subseteq \mathcal{K}'$. To see this, take some $\nu \in \mathcal{K}$. By eq. (B.7), there exist some $\mathcal{K}_0, \mathcal{K}'_0$, with $\nu \in \mathcal{K}_0 \subseteq \mathcal{K}'_0$ such that:

$$\Gamma : X, \mathcal{K}_0 \vdash p' : \mathcal{K}'_0$$

By the induction hypothesis, there is some $\mathcal{K}'_1 \supseteq \mathcal{K}'_0$:

$$\Gamma', X : \mathcal{K}_0 \vdash p' : \mathcal{K}'_1$$

Then, by eq. (B.8), $\mathcal{K}_0 \subseteq \mathcal{K}'$, from which the thesis follows.

Back to the main statement, let $\mathrm{fv}(p) \subseteq \mathrm{dom}(\Gamma) \cup \{X\}$, and suppose $\mathcal{K} \subseteq \mathcal{K}'$. We have to show $F(\mathcal{K}) \subseteq F(\mathcal{K}')$. Expanding (11.1): $\Gamma, X : \mathcal{K} \vdash p : \mathcal{K}_0$ and $\Gamma, X : \mathcal{K}' \vdash p : \mathcal{K}_1$, for some $\mathcal{K}_0, \mathcal{K}_1$ such that $\mathcal{K}_0 \subseteq \mathcal{K}_1$. Such $\mathcal{K}_0, \mathcal{K}_1$ exist and are unique (Theorem 11.1.4); $\mathcal{K}_0 \subseteq \mathcal{K}_1$ follows by (B.6).

B.4 Proofs for Section 12.1

For readabiliy, we use a simplified version of the semantics of network of TA, which considers only networks composed by two automata.

Definition B.4.1 (Semantics of TA) *Let $A_i = (Loc_i, Loc_i^u, l_i^0, E_i, I_i)$ be TA, for $i \in \{1, 2\}$. We define the behaviour of the network $A_1 \mid A_2$ as the timed LTS $(S, s_0, Lab, \rightarrow_N)$, where: $S = Loc_1 \times Loc_2 \times \mathbb{V}$; $s_0 = (l_1^0, l_2^0, \nu_0)$; $Lab = \mathtt{A} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$; $\rightarrow_N$ is defined in Figure B.1.*

We now define a transformation from TST to $DE - TST$. Technically, it is a relation, parametric w.r.t. a set V of recursion variables. However, since the order in which the variables are chosen from V is imaterial (from a semantical point of view), we will regard the transformation as a function.

$$\frac{(l_1, \tau, g, R, l_1') \in E_1 \quad \nu \in [\![g]\!] \quad \nu[R] \in [\![I_1(l_1') \wedge I_2(l_2)]\!]}{(l_1, l_2, \nu) \xrightarrow{\tau}_N (l_1', l_2, \nu[R])} \text{[TA1]}$$

$$\frac{(l_1, \mathtt{a!}, g_1, R_1, l_1') \in E_1 \quad (l_2, \mathtt{a?}, g_2, R_2, l_2') \in E_2 \quad \nu \in [\![g_1 \wedge g_2]\!] \quad \nu[R_1][R_2] \in [\![I_1(l_1') \wedge I_2(l_2')]\!]}{(l_1, l_2, \nu) \xrightarrow{\mathtt{a}}_N (l_1', l_2', \nu[R_1][R_2])} \text{[TA2]}$$

$$\frac{\forall i \in \{1,2\} : (\nu + \delta) \in [\![I_i(l_i)]\!] \wedge l_i \notin Loc_i^u}{(l_1, l_2, \nu) \xrightarrow{\delta}_N (l_1, l_2, \nu + \delta)} \text{[TA3]}$$

Figure B.1: Semantics of networks of TA (symmetric rules omitted).

Definition B.4.2 (DE-TST transformation) *Let p be a TST according to Definition 10.1.1. Let V be an infinite set of recursion variables not occurring in p. Then, the DE-TST of p, denoted by $\langle p \rangle_V$, is given by:*

$$\begin{array}{lllll}
(a) & \langle \bigcirc_{i=1}^n \ell_{\mathtt{i}}\{g_i, R_i\}.p_i \rangle_V & = & (X_0,\ \bigcup_{i=1}^n D_i \cup \{X_0 \triangleq \bigcirc_{i=1}^n \ell_{\mathtt{i}}\{g_i, R_i\}.X_i\}) & \\
 & & & \quad \textit{for } \bigcirc \in \{\oplus, \sum\},\ \textit{with } X_0 \in V \textit{ and} & \\
 & & & \quad (X_i, D_i) = \langle p_i \rangle_{V \setminus W_i} \textit{ and} & \\
 & & & \quad W_i = (\{X_0\} \ \cup \ \bigcup_{j=1}^{i-1} \mathrm{dv}(D_j)) & \\
(b) & \langle \mathtt{rec}\, X \,.\, p' \rangle_V & = & (X_0, D[^{X_0}/_X]) \quad \textit{where } (X_0, D) = \langle p' \rangle_V & \\
(c) & \langle X \rangle_V & = & (X, \emptyset) & \\
(d) & \langle \mathbf{1} \rangle_V & = & (X_0, \{X_0 = \mathbf{1}\}) \quad \textit{where } X_0 \in V &
\end{array}$$

For short, we indicate the normal form of p, with $\langle p \rangle$.

Example B.4.3 *Consider the following translations of TSTs, where V is a set of recursion variables not occurring in any p_i (we omit guards and reset sets).*

$$\begin{aligned}
\langle \mathtt{a!} \oplus \mathtt{b!} \rangle &= (X_0, \{X_0 \triangleq \mathtt{a!}.X_1 \oplus \mathtt{b!}.X_2,\ X_1 \triangleq \mathbf{1},\ X_2 \triangleq \mathbf{1}\}) \\
\langle \mathtt{a!.rec}\, X \,.\, (\mathtt{b!}.X \oplus \mathtt{c!}) \rangle &= (X_0, \{X_0 \triangleq \mathtt{a!}.X_1,\ X_1 \triangleq \mathtt{b!}.X_1 \oplus \mathtt{c!}.X_2,\ X_2 \triangleq \mathbf{1}\}) \\
\langle \mathtt{rec}\, X \,.\, \mathtt{a!.rec}\, Y \,.\, (\mathtt{b!}.X \oplus \mathtt{c!}.Y) \rangle &= (X_0, \{X_0 \triangleq \mathtt{a!}.X_1,\ X_1 \triangleq \mathtt{b!}.X_0 \oplus \mathtt{c!}.X_1\}) \\
\langle \mathtt{rec}\, X \,.\, (\mathtt{a!.rec}\, X \,.\, \mathtt{b!}.X \ \oplus \ \mathtt{c!}.X) \rangle &= (X_0, \{X_0 \triangleq \mathtt{a!}.X_1 \oplus \mathtt{c!}.X_0,\ X_1 \triangleq \mathtt{b!}.X_1\})
\end{aligned}$$

Lemma B.4.4 *If p is closed, then $\langle p \rangle$ is closed.*

Proof.(Sketch). A DE-TST is closed if its used recursion variables plus the initial one are defined exactly once. In the transformation, we see by construction that all the new variable are first declared. The only open issue is caused by variables already presents in the TST. However, if a TST is closed, then all its used variables are in the scope of some rec declaration, and hence they will be correctly renamed by rule B.4.2(b).

Lemma B.4.5 *Let (X, D) be a DE-TST, and let $A = [\![D]\!]^X = (Loc, Loc^u, l^0, E, I)$. Then:*

$$I(l) = \begin{cases} \mathrm{rdy}(p) & \text{if } l = \tau Y, \text{ for some } Y \text{ and } X \triangleq p \in D \\ true & \text{otherwise} \end{cases} \qquad (\forall l \in Loc)$$

Proof. By Definition 12.1.1, the invariant of A is given by $I(l) = \bigwedge_i I_i(l)$ for all l, where each I_i is the invariant of the encoding of some defining equation in D. By Definition 12.1.9, the only location with invariant other than `true` has the form τY, which can only be obtained by encoding $Y \triangleq p$. Since (X, D) is closed, there is exactly one defining equation for Y. Hence, τY occurs only in one TA (say, in A_j), and so $I(l) = \bigwedge_i I_i(l) = I_j(l) = \mathrm{rdy}(p)$.

In Definition B.4.6 we recall the definition of strong bisimulation.

Definition B.4.6 (Strong bisimulation) *A binary relation $\mathcal{R}$ between states of an LTS is a bisimulation if whenever whenever $s_1 \mathcal{R} s_2$, and α is an action:*

1. *if $s_1 \xrightarrow{\alpha} s_1'$ then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' \mathcal{R} s_2'$*
2. *if $s_2 \xrightarrow{\alpha} s_2'$ then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $s_1' \mathcal{R} s_2'$*

Two states s_1 and s_2 are bisimilar, written $s_1 \sim s_2$, iff there is a bisimulation that relates them. Henceforth the relation $\sim$ will be referred to strong bisimulation equivalence or strong bisimilarity.

Proof of Lemma 12.1.11:

Let (X, D') and (Y, D'') be DE-TST such that $\mathrm{dv}(D') \cap \mathrm{dv}(D'') = \emptyset$. Let $N = \mathcal{T}(X, D') \mid \mathcal{T}(Y, D'')$. Then:

$$((X, \nu_0) \mid (Y, \eta_0), \rightarrow_{D' \cup D''}) \sim ((X, Y, \nu_0 \sqcup \eta_0), \rightarrow_N)$$

Let (X, D') and (Y, D'') as in the statement, and let $\mathcal{T}(X, D') = (Loc_1, Loc_1^u, l_1^0, E_1, I_1)$ and $\mathcal{T}(Y, D'') = (Loc_2, Loc_2^u, l_2^0, E_2, I_2)$. We show that:

$$\mathcal{R} = \{((x, \nu) | (y, \eta), \quad (x, y, \nu \sqcup \eta)) \mid x, y, z \in \mathcal{S} \text{ and } \nu \in [\![I_1(x)]\!] \text{ and } \eta \in [\![I_2(y)]\!]\} \qquad \text{(B.9)}$$

is a bisimulation. We denote with $\mathtt{ck}(D)$ the set of clocks used in D. First, we show that every possible move from $(x, \nu) | (y, \eta)$ in $\rightarrow_D$ is matched by a move of $(x, y, \nu \cup \eta)$ in $\rightarrow_N$, and that the resulting states are related by $\mathcal{R}$. We have the following cases, according to the rule used to move:

- [DES-$\oplus$].

$$\frac{(x, \nu) \xrightarrow{\tau}_D (x', \nu)}{(x, \nu) \mid (y, \eta) \xrightarrow{\tau}_D (x', \nu) \mid (y, \eta)}$$

According to this rule, we have two sub cases in which the premise can fire a τ move:

[DE-τ]: $(x,\nu) \xrightarrow{\tau}_D (\tau X,\nu)$ with $x = X$ and $X \triangleq p \in D$ with $p = \oplus \ldots$ and $\nu \in \mathrm{rdy}(p)$. According to Definition 12.1.9, the mapping of an internal choice is $[\![X \triangleq p]\!] = \mathit{Pfx}(X,\tau,\emptyset,\mathit{Br}(\tau X,\mathrm{rdy}(p),\{(\tau,g_i,\emptyset,A_i)\}_i)$ for some A_i and g_i. Hence, according to Definition 12.1.3 of *Pfx* pattern, there exists an edge in E_1 such as $(X,\tau,\mathtt{true},\emptyset,\tau X)$. According to Definition 12.1.4 of *Br* pattern and to Lemma B.4.5, $I_1(\tau X) = \mathrm{rdy}(p)$. In our case rule [TA1] of Definition B.4.1 states that:

$$\frac{(X,\tau,\mathtt{true},\emptyset,\tau X) \in E_1 \qquad \nu \sqcup \eta \in [\![\mathtt{true}]\!] \qquad (\nu \sqcup \eta)[\emptyset] \in [\![I_1(\tau X) \wedge I_2(y)]\!]}{(X,y,\nu \sqcup \eta) \xrightarrow{\tau}_N (\tau X,y,\nu \sqcup \eta)}$$

We must show that $\nu \sqcup \eta \in [\![\mathtt{true}]\!]$ and $\nu \sqcup \eta \in [\![I_1(x) \wedge I_2(y)]\!]$. The former holds trivially. For the second, according to Lemma B.4.5 we have $I_1(\tau X) = \mathrm{rdy}(p)$; by premises in [DES-$\oplus$] we have $\nu \in \mathrm{rdy}(p)$ and, since $\mathtt{ck}(D') \cap \mathtt{ck}(D'') = \emptyset$, we have $\nu \sqcup \eta \in \mathrm{rdy}(p)$. $\nu \sqcup \eta \in [\![I_2(y)]\!]$ holds by hypothesis in the definition of $\mathcal{R}$ in Equation (B.9). Hence $(X,y,\nu \sqcup \eta) \xrightarrow{\tau}_N (\tau X,y,\nu \sqcup \eta)$ and the resulting states belong to $\mathcal{R}$.

[DE-$\oplus$]: $(x,\nu)|(y,\eta) \xrightarrow{\tau}_D (x',\nu)|(y,\eta)$ with $x = \tau X$ and $X \triangleq \mathtt{a!}\{g,R\}.\, Y \oplus p'$ and $\nu \in [\![g]\!]$ and $x' = [\mathtt{a!}\{g,R\}]\, Y$. According to Definition 12.1.9, the mapping of an internal choice is

$$[\![X \triangleq p]\!] = \mathit{Pfx}(X,\tau,\emptyset,\mathit{Br}(\tau X,\mathrm{rdy}(p),\{(\tau,g,\emptyset,A)\} \cup S)$$

for some set S, and with $A = \mathit{Pfx}([\mathtt{a!}\{g,R\}]\, Y,\mathtt{a!},R,\mathit{Idle}(Y))$. According to Definition 12.1.4 of *Br* pattern and Lemma B.4.5, there exists an edge in E_1 such as $(\tau X,\tau,g,\emptyset,x')$ with $I_1(x') = \mathtt{true}$. In our case rule [TA1] of Definition B.4.1 states that:

$$\frac{(\tau X,\tau,g,\emptyset,x') \in E_1 \qquad \nu \sqcup \eta \in [\![g]\!] \qquad (\nu \sqcup \eta)[\emptyset] \in [\![I_1(x') \wedge I_2(y)]\!]}{(\tau X,y,\nu \sqcup \eta) \xrightarrow{\tau}_N (x',y,\nu \sqcup \eta)}$$

We must show that $\nu \sqcup \eta \in [\![g]\!]$ and $\nu \sqcup \eta \in [\![I_1(x') \wedge I_2(y)]\!]$. The former holds by hypothesis since $\nu \in [\![g]\!]$. For the second, according to Lemma B.4.5 we have $I_1(\tau X) = \mathtt{true}$– hence $\nu \sqcup \eta \in [\![I_1(x')]\!]$ is trivially true; and $\nu \sqcup \eta \in [\![I_2(y)]\!]$ holds by hypothesis in the definition of $\mathcal{R}$ in Equation (B.9). Hence $(\tau X,y,\nu \sqcup \eta) \xrightarrow{\tau}_N (x',y,\nu \sqcup \eta)$ and the resulting states belong to $\mathcal{R}$.

- [DES-Del].

$$\frac{(x,\nu) \xrightarrow{\delta}_D (x,\nu') \quad (y,\eta) \xrightarrow{\delta}_D (y,\eta')}{(x,\nu) \mid (y,\eta) \xrightarrow{\delta}_D (x,\nu') \mid (y,\eta')}$$

According to this rule, we have several possible combinations of the premises to fire a δ move:

  – Case [DE-Del2] applied twice:

$$\frac{\dfrac{X \triangleq p \in D \quad \nu + \delta \in \mathrm{rdy}(p)}{(\tau X,\ \nu) \xrightarrow{\delta}_D (\tau X,\ \nu + \delta)} \qquad \dfrac{Y \triangleq q \in D \quad \nu + \delta \in \mathrm{rdy}(q)}{(\tau Y,\ \nu) \xrightarrow{\delta}_D (\tau Y,\ \nu + \delta)}}{(\tau X,\nu) \mid (\tau Y,\eta) \xrightarrow{\delta}_D (\tau X,\nu + \delta) \mid (\tau Y,\eta + \delta)}$$

According to Definition 12.1.9 and Lemma B.4.5, there is only a case in which a location name has prefix τ, and in that case we have $I_1(\tau X) = \text{rdy}(p)$ and $\tau X \notin Loc_1^u$. Similarly, $I_2(\tau Y) = \text{rdy}(q)$ and $\tau Y \notin Loc_2^y$. In our case rule [TA3] of Definition B.4.1 states that:

$$\frac{((\nu \sqcup \eta) + \delta) \in [\![I_1(\tau X) \wedge I_2(\tau Y)]\!] \qquad \tau X \notin Loc_1^u \qquad \tau Y \notin Loc_2^y}{(\tau X, \tau Y, \nu \sqcup \eta) \xrightarrow{\delta}_N (\tau X, \tau Y, (\nu \sqcup \eta) + \delta)}$$

By [DE-DEL2] rule, we have $\nu + \delta \in \text{rdy}(p)$, hence, since $\texttt{ck}(D') \cap \texttt{ck}(D'') = \emptyset$ and so $(\nu \sqcup \eta) + \delta \in \text{rdy}(p)$. Similarly, since $\eta + \delta \in \text{rdy}(q)$, it follows $(\nu \sqcup \eta) + \delta \in \text{rdy}(q)$. We already noted that τY and τX are not urgent, so we conclude $(\tau X, y, \nu \sqcup \eta) \xrightarrow{\delta}_N (\tau X, y, \nu + \delta \sqcup \eta + \delta)$. The resulting states belong to $\mathcal{R}$.

– Case [DE-DEL2] and [DE-DEL1]:

$$\frac{\dfrac{(X \triangleq \ldots) \in D \vee (X \triangleq \mathbf{1}) \in D}{(X,\ \nu) \xrightarrow{\delta}_D (X,\ \nu + \delta)} \qquad \dfrac{Y \triangleq q \in D \quad \nu + \delta \in \text{rdy}(q)}{(\tau Y,\ \nu) \xrightarrow{\delta}_D (\tau Y,\ \nu + \delta)}}{(X, \nu) \mid (\tau Y, \eta) \xrightarrow{\delta}_D (X, \nu + \delta) \mid (\tau Y, \eta + \delta)}$$

According to Definition 12.1.9 and Lemma B.4.5, there is only a case in which a location name has prefix τ, and in that case we have $I_2(\tau Y) = \text{rdy}(q)$ and $\tau Y \notin Loc_2^y$. For X we have two possibilities: either (i) it is an internal choice or (i) it is a success term.

(i) In the first case, according to Definition 12.1.9 and Lemma B.4.5, the mapping for an external choice is: $[\![X \triangleq p]\!] = Br(X, \text{rdy}(p), S)$ for some set S. Since $\text{rdy}(p)$ is `true` for external choices, then there exists a location X with invariant $I_1(X) = \text{rdy}(p) = \texttt{true}$ and $\tau Y \notin Loc_1^u$. In this case rule [TA3] of Definition B.4.1 states that:

$$\frac{((\nu \sqcup \eta) + \delta) \in [\![\texttt{true} \wedge I_2(\tau Y)]\!] \qquad X \notin Loc_1^u \qquad \tau Y \notin Loc_2^y}{(X, \tau Y, \nu \sqcup \eta) \xrightarrow{\delta}_N (X, \tau Y, (\nu \sqcup \eta) + \delta)}$$

By [DE-DEL2] rule, we have $\eta + \delta \in \text{rdy}(q)$, it follows $(\nu \sqcup \eta) + \delta \in \text{rdy}(q)$. We already noted that τY and X are not urgent, so we conclude $(X, y, \nu \sqcup \eta) \xrightarrow{\delta}_N (X, y, \nu + \delta \sqcup \eta + \delta)$. The resulting states belong to $\mathcal{R}$.

(ii) In the second case, according to Definition 12.1.9, the mapping for the success termination is: $[\![X \triangleq \mathbf{1}]\!] = Idle(X)$. Hence, according to Definition 12.1.2 of *Idle* pattern and Lemma B.4.5, the location X is not urgent, with $I_1(X) = \texttt{true}$. Again $\tau Y \notin Loc_1^u$. In this case rule [TA3] of Definition B.4.1 states that:

$$\frac{((\nu \sqcup \eta) + \delta) \in [\![\texttt{true} \wedge I_2(\tau Y)]\!] \qquad X \notin Loc_1^u \qquad \tau Y \notin Loc_2^y}{(X, \tau Y, \nu \sqcup \eta) \xrightarrow{\delta}_N (X, \tau Y, (\nu \sqcup \eta) + \delta)}$$

By IDRULE rule, we have $\eta + \delta \in \text{rdy}(q)$, it follows $(\nu \sqcup \eta) + \delta \in \text{rdy}(q)$. We already noted that τY and X are not urgent, so we conclude $(X, y, \nu \sqcup \eta) \xrightarrow{\delta}_N (X, y, \nu + \delta \sqcup \eta + \delta)$. The resulting states belong to $\mathcal{R}$.

– Case [DE-DEL1] applied twice: Similar to previous cases.

- [DES-τ].

$$\frac{(x,\nu) \xrightarrow{\mathtt{a!}}_D (x',\nu') \quad (y,\eta) \xrightarrow{\mathtt{a?}}_D (y',\eta')}{(x,\nu) \mid (y,\eta) \xrightarrow{\mathtt{a}}_D (x',\nu') \mid (y',\eta')}$$

According to this rule, we can synchronize on a only if

$$\frac{([\mathtt{a!}\{g,R\}]\,X',\ \nu) \xrightarrow{\mathtt{a!}}_D (X',\ \nu[R]) \quad \dfrac{(Y \triangleq \mathtt{a?}\{f,T\}.\,Y' + q) \in D \quad \eta \in [\![f]\!]}{(Y,\ \nu) \xrightarrow{\mathtt{a?}}_D (Y',\ \eta[T])}}{([\mathtt{a!}\{g,R\}]\,X',\nu) \mid (Y,\eta) \xrightarrow{\mathtt{a}}_D (X',\nu[R]) \mid (Y',\eta[T])}$$

Let $x = [\mathtt{a!}\{g,R\}]\,X'$. According to Definition 12.1.9, the last part of the mapping for an internal choice is such as $\mathit{Pfx}(x,\mathtt{a!},R,\mathit{Idle}(X'))$. Hence, according to Definition 12.1.3 of *Pfx* pattern and Lemma B.4.5 , there exists an edge $(x,\mathtt{a!},\mathtt{true},R,X') \in E_1$ and $I_1(X') = \mathtt{true}$. According to Definition 12.1.9, the mapping for an external choice is: $[\![Y \triangleq \mathtt{a?}\{f,T\}.\,Y' + q]\!] = \mathit{Br}(Y,\mathtt{true},\{(\mathtt{a?},f,T,\mathit{Idle}(Y'))\} \cup S)$ for some S. Hence, according to Definition 12.1.4 of *Br* pattern and Lemma B.4.5, there exists an edge $(Y,\mathtt{a?},f,T,Y')$ and $I_1(Y') = \mathtt{true}$. In this case rule [TA2] of Definition B.4.1 states that:

$$\frac{\begin{array}{cc}(x,\mathtt{a!},\mathtt{true},R,X') \in E_1 & (\nu \sqcup \eta) \in [\![\mathtt{true}]\!] \\ (Y,\mathtt{a?},f,T,Y') \in E_2 & (\nu \sqcup \eta) \in [\![f]\!]\end{array} \qquad (\nu \sqcup \eta)[R][T] \in [\![\mathtt{true} \wedge \mathtt{true}]\!]}{(x,Y,\nu \sqcup \eta) \xrightarrow{\mathtt{a}}_N (X',Y',(\nu \sqcup \eta)[R][T])}$$

Since by hypothesis $\eta \in [\![f]\!]$, since $\mathtt{ck}(D') \cap \mathtt{ck}(D'') = \emptyset$ and we obtain $(\eta \sqcup \nu) \in [\![f]\!]$. Hence by rule [TA2] of Definition B.4.1 , we have $(x,Y,\nu \sqcup \eta) \xrightarrow{\mathtt{a}}_N (X',Y',\nu \sqcup \eta[R][T]$. The resulting states, belong to $\mathcal{R}$.

We now show that every possible move of $(x,y,\nu \sqcup \eta)$ in $\rightarrow_N$ is matched by a move of $(x,\nu)|(y,\eta)$ in $\rightarrow_D$, and that the resulting states are related by $\mathcal{R}$. We have the following cases, according to the rules of Definition B.4.1 used to justify the move:

- [TA1].

$$\frac{(x,\tau,g,R,x') \in E_1 \qquad (\nu \sqcup \eta) \in [\![g]\!] \qquad (\nu \sqcup \eta)[R] \in [\![I_1(x') \wedge I_2(y)]\!]}{(x,y,\nu \sqcup \eta) \xrightarrow{\tau}_N (x',y,\nu \sqcup \eta[R])}$$

According to Definition 12.1.9, there exist two possibilities for an edge in network N to display a τ label, and both derive from the mapping of an internal choice. Let $p = \mathtt{a!}\{g,R\}Y \oplus p'$, then $[\![X \triangleq p]\!] = \mathit{Pfx}(X,\tau,\emptyset,\mathit{Br}(\tau X,\mathrm{rdy}(p),\{(\tau,g,\emptyset,A)\} \cup S))$ for some set S and automaton A. Hence, according to Definition 12.1.3 and Definition 12.1.4 of patterns and with Lemma B.4.5, we have two edges with τ label in E_1: (i) $(X,\tau,\mathtt{true},\emptyset,\tau X)$ with $I_1(\tau X) = \mathrm{rdy}(p)$; and (ii) $(\tau X,\tau,g,\emptyset,Y)$ with $I_1(x') = \mathtt{true}$. So we have two sub-cases:

(i) Let assume we fire the first edge $(X, \tau, \texttt{true}, \emptyset, \tau X)$. Hence $x = X$, $x' = \tau X$ and $R = \emptyset$. By hypothesis $\nu \sqcup \eta \in [\![I_1(\tau X)]\!] = \mathrm{rdy}(p)$. Since $\texttt{ck}(D') \cap \texttt{ck}(D'') = \emptyset$, we derive $\nu \in \mathrm{rdy}(p)$. Hence, we can use [DE-τ] and [DES-$\oplus$] rule from Definition 12.1.6 to obtain:

$$\frac{(X \triangleq p) \in D \quad p = \oplus \ldots \quad \nu \in \mathrm{rdy}(p)}{\dfrac{(X,\ \nu) \xrightarrow{\tau}_D (\tau X,\ \nu)}{(X, \nu) \mid (y, \eta) \xrightarrow{\tau}_D (\tau X, \nu) \mid (y, \eta)}}$$

Since by [TA1], $(\nu \sqcup \eta)[\emptyset] \in [\![I_1(\tau X) \wedge I_2(y)]\!]$, the resulting states, belong to $\mathcal{R}$.

(ii) Let assume we fire the second edge $(\tau X, \tau, g, \emptyset, x')$. Hence $x = \tau X$, $x' = Y$ and $R = \emptyset$. By [TA1], $\nu \sqcup \eta \in [\![g]\!]$, since $\texttt{ck}(D') \cap \texttt{ck}(D'') = \emptyset$, we derive $\nu \in [\![g]\!]$. Hence, we can use [DE-$\oplus$] and [DES-$\oplus$] rules from Definition 12.1.6 to obtain:

$$\frac{(X \triangleq \texttt{a!}\{g, R\}Y \oplus p') \in D \quad \nu \in [\![g]\!]}{\dfrac{(\tau X,\ \nu) \xrightarrow{\tau}_D ([\texttt{a!}\{g, R\}]\, Y,\ \nu)}{(\tau X, \nu) \mid (y, \eta) \xrightarrow{\tau}_D ([\texttt{a!}\{g, R\}]\, Y, \nu) \mid (y, \eta)}}$$

By hypothesis $(\nu \sqcup \eta)[\emptyset] \in [\![I_1(x') \wedge I_2(y)]\!]$, so the resulting states belong to $\mathcal{R}$.

- [TA2].

$$\frac{\begin{array}{c}(x, \texttt{a!}, g, R, x') \in E_1 \quad (\nu \sqcup \eta) \in [\![g]\!] \quad (\nu \sqcup \eta)[R][T] \in [\![I_1(x)]\!] \\ (y, \texttt{a?}, f, T, y') \in E_2 \quad (\nu \sqcup \eta) \in [\![f]\!] \quad (\nu \sqcup \eta)[R][T] \in [\![I_2(y)]\!]\end{array}}{(x, y, \nu \sqcup \eta) \xrightarrow{\texttt{a}}_N (x', y', (\nu \sqcup \eta)[R][T])}$$

According to Definition 12.1.9, a synchronization happens only if an internal-external choice synchronize.

Hence $x = [\texttt{a!}\{g, R\}]\, X$ and $y = Y$ for some $Y \triangleq \texttt{a?}\{f, S\}.Y' + p$. According to Definition 12.1.9, the last part of the mapping for an internal choice is such as $\mathit{Pfx}(x, \texttt{a!}, R, \mathit{Idle}(X'))$. Hence, according to Definition 12.1.3 of *Pfx* pattern and to Lemma B.4.5, there exists an edge $(x, \texttt{a!}, \texttt{true}, R, X') \in E_1$ and $I_1(X') = \texttt{true}$. According to Definition 12.1.9, the mapping for an external choice is:
$[\![Y \triangleq \texttt{a?}\{f, T\}.\, Y' + q]\!] = \mathit{Br}(Y, \texttt{true}, \{(\texttt{a?}, f, T, \mathit{Idle}(Y'))\} \cup S)$ for some S. Hence, according to Definition 12.1.4 of *Br* pattern, there exists an edge $(Y, \texttt{a?}, f, T, Y')$ and $I_1(Y') = \texttt{true}$. So in this case [TA2] becomes:

$$\frac{\begin{array}{c}(x, \texttt{a!}, \texttt{true}, R, X') \in E_1 \quad (\nu \sqcup \eta) \in [\![\texttt{true}]\!] \quad (\nu \sqcup \eta)[R][T] \in [\![I_1(X')]\!] \\ (Y, \texttt{a?}, f, T, Y') \in E_2 \quad (\nu \sqcup \eta) \in [\![f]\!] \quad (\nu \sqcup \eta)[R][T] \in [\![I_2(Y')]\!]\end{array}}{(x, Y, \nu \sqcup \eta) \xrightarrow{\texttt{a}}_N (X', Y', (\nu \sqcup \eta)[R][T])}$$

Since $\texttt{ck}(D') \cap \texttt{ck}(D'') = \emptyset$ and since $(\nu \sqcup \eta) \in [\![f]\!]$, we derive $\eta \in [\![f]\!]$. Hence we can apply [DE-?][DE-!] and [DES-τ] to obtain:

$$\frac{([\texttt{a!}\{g, R\}]\, X',\ \nu) \xrightarrow{\texttt{a!}}_D (X',\ \nu[R]) \quad \dfrac{(Y \triangleq \texttt{a?}\{f, T\}.\, Y' + q) \in D \quad \eta \in [\![f]\!]}{(Y,\ \nu) \xrightarrow{\texttt{a?}}_D (Y',\ \eta[T])}}{([\texttt{a!}\{g, R\}]\, X', \nu) \mid (Y, \eta) \xrightarrow{\texttt{a}}_D (X', \nu[R]) \mid (Y', \eta[T])}$$

By hypothesis $(\nu \sqcup \eta)[R][T] \in [\![I_1(X) \wedge I_2(Y)]\!]$, so the resulting states belong to $\mathcal{R}$.

- [TA3].

$$\frac{((\nu \sqcup \eta) + \delta) \in [\![I_1(x)]\!] \quad x \notin Loc_1^u \quad ((\nu \sqcup \eta) + \delta) \in [\![I_2(y)]\!] \quad y \notin Loc_2^u}{(x, y, \nu \sqcup \eta) \xrightarrow{\delta}_N (x, y, (\nu \sqcup \eta) + \delta)}$$

According to [TA3], time can pass in a network only if all the locations of the current state are not urgent. According with Definition 12.1.9, this happens for: the second location in the mapping of an internal choice or the first location in the mapping of an external choice or a success location. Hence, we have several sub-cases:

(i) Let us assume that x derive from an internal choice and y from an external choice. According to Definition 12.1.9 and to Lemma B.4.5 , x must be of the form $x = I_1(\tau X)$ for some $X \triangleq p \in D$ with $p = \oplus \ldots$, obtaining $I_1(\tau X) = \mathrm{rdy}(p)$ and $\tau X \notin Loc_1^u$. According to Definition 12.1.9 and to Lemma B.4.5 , y must be of the form $y = Y$ for some $Y \triangleq q \in D$ with $q = + \ldots$, obtaining $I_2(Y) = \mathrm{rdy}(q) = \mathtt{true}$ and $\tau Y \notin Loc_2^u$. So in this case [TA2] becomes:

$$\frac{((\nu \sqcup \eta) + \delta) \in [\![I_1(\tau X)]\!] \quad (\tau X) \notin Loc_1^u \quad ((\nu \sqcup \eta) + \delta) \in [\![I_2(Y)]\!] \quad (Y) \notin Loc_2^u}{(l_1, l_2, \nu \sqcup \eta) \xrightarrow{\delta}_N (l_1, l_2, (\nu \sqcup \eta) + \delta)}$$

By hypothesis of [TA2], $((\nu \sqcup \eta) + \delta) \in [\![I_1(\tau X)]\!] = \mathrm{rdy}(p)$. Since $\mathtt{ck}(D') \cap \mathtt{ck}(D'') = \emptyset$, we derive $\nu + \delta \in \mathrm{rdy}(p)$. Hence, we have:

$$\frac{\dfrac{X \triangleq p \in D \quad \nu + \delta \in \mathrm{rdy}(p)}{(\tau X,\ \nu) \xrightarrow{\delta}_D (\tau X,\ \nu + \delta)} \quad \dfrac{(Y = + \ldots) \in D \vee (Y = \mathbf{1}) \in D}{(Y,\ \nu) \xrightarrow{\delta}_D (Y,\ \nu + \delta)}}{(\tau X, \nu) \mid (Y, \eta) \xrightarrow{\delta}_D (\tau X, \nu + \delta) \mid (Y, \eta + \delta)}$$

Since by hypothesis, $((\nu \sqcup \eta) + \delta) \in [\![I_1(\tau X) \wedge I_2(Y)]\!]$, the resulting states, belong to $\mathcal{R}$.

(ii) All the other combinations can be proved similarly to the previous one.

B.5 Proofs for Section 13.1

Proof of Lemma 13.1.7:
$((p, \nu) \mid (q, \eta), \rightarrow)$ is turn-bisimilar to $((p, [], \nu) \parallel (q, [], \eta), \twoheadrightarrow)$.

Let us consider the LTS $\rightarrow$ of Figure 10.1 and the set of success states
$S_1 = \{(\mathbf{1}, \nu) \mid (q, \eta) \mid q \text{ TST}, \nu, \eta \in \mathbb{V}\}$. Also, consider the LTS $\twoheadrightarrow_M$ of Figure 13.1 and the

set of success states $S_2 = \{(\mathbf{1}, [], \nu) \parallel (q, [], \eta) \mid q \text{ TST}, \nu, \eta \in \mathbb{V}\}$. Let $\mathcal{R}$ be the relation:

$$\begin{aligned}
\mathcal{R} &= \mathcal{R}[1] \cup \mathcal{R}[2] \cup \mathcal{R}[3] \cup \mathcal{R}[2S] \cup \mathcal{R}[3S] \\
\mathcal{R}[1] &= \{((p,\nu) \mid (q,\eta),\ \ (p,[],\nu) \parallel (q,[],\eta)) \mid p, q \text{ TST}, \nu, \eta \in \mathbb{V}\} \\
\mathcal{R}[2] &= \{(([\mathtt{a!}\{g,R\}]\, p, \nu) \mid (q,\eta),\ \ (p,[\mathtt{a!}],\nu[R]) \parallel (q,[],\eta)) \mid (p, q \text{ TST}, \nu \in [\![g]\!]\} \\
\mathcal{R}[3] &= \{(([\mathtt{a!}\{g,R\}]\, p, \nu) \mid ([\mathtt{b!}\{f,S\}]\, q, \eta),\ \ (p,[\mathtt{a!}],\nu[R]) \parallel (\mathtt{b!}\{f,S\}.q \oplus q', [], \eta)) \mid \cdots\} \\
\mathcal{R}[2S] &= \{((p,\nu) \mid ([\mathtt{a!}\{f,S\}]\, q, \eta),\ \ (p,[],\nu) \parallel (q,[\mathtt{a!}],\nu[S])) \mid p, q \text{ TST}, \eta \in [\![f]\!]\} \\
\mathcal{R}[3S] &= \{(([\mathtt{a!}\{g,R\}]\, p, \nu) \mid ([\mathtt{b!}\{f,S\}]\, q, \eta),\ \ (\mathtt{a!}\{g,R\}.p \oplus p', [], \nu) \parallel (q,[\mathtt{b!}],\eta[R]) \mid \cdots\}
\end{aligned}$$

Let $s_1 = (p,\nu) \mid (q,\eta)$, and let $s_2 = (p,[],\nu) \parallel (q,[],\eta)$. Clearly, $s_1 \mathcal{R} s_2$, hence to obtain the thesis we will prove that $\mathcal{R}$ is a turn-bisimulation. The proof is organised as follows. In **Part A** we show that s_2 turn-simulates s_1 via $\mathcal{R}$, and in **Part B** that s_1 turn-simulates s_2 via $\mathcal{R}$. Within each part, we proceed by cases on the form of s_1 and s_2: in **Case 1** we assume that $(s_1, s_2) \in \mathcal{R}[1]$, in **Case 2** that $(s_1, s_2) \in \mathcal{R}[2]$, and in **Case 3** that $(s_1, s_2) \in \mathcal{R}[3]$. We omit cases for $\mathcal{R}[2S]$ and $\mathcal{R}[3S]$, since they are specular to cases $\mathcal{R}[2]$ and $\mathcal{R}[3]$. For each case, we show that items ((a)), ((b)), and ((c)) of Definition 13.1.6 hold. We will only consider the moves of the LHS of a composition $P \circ Q$; all the symmetric cases will be omitted.

Part A: s_2 turn-simulates s_1 via $\mathcal{R}$.

Case 1: Let $s_1 = (p,\nu) \mid (q,\eta)$ and $s_2 = (p,[],\nu) \parallel (q,[],\eta)$.

(a) To prove item ((a)) of Definition 13.1.6, we consider the possible moves of s_1:

- [S-$\oplus$]. We have:
$$\frac{(p,\nu) \xrightarrow{\tau} ([\mathtt{a!}\{g,R\}]\, p', \nu)}{s_1 \xrightarrow{\tau} ([\mathtt{a!}\{g,R\}]\, p', \nu) \mid (q,\eta) = s_1'}$$
where the premise requires $p = \mathtt{a!}\{g,R\}.p' \oplus p''$ and $\nu \in [\![g]\!]$. Hence, by rule [M-$\oplus$] we have:
$$s_2 \xrightarrow{\mathsf{A:a!}} (p', [\mathtt{a!}], \nu[R]) \parallel (q,[],\eta) = s_2'$$
Then, $(s_1', s_2') \in \mathcal{R}[2] \subseteq \mathcal{R}$.
- [S-τ]. This case does not apply.
- [S-Del]. We have:
$$\frac{(p,\nu) \xrightarrow{\delta} (p', \nu+\delta) \qquad (q,\eta) \xrightarrow{\delta} (q', \eta+\delta)}{s_1 \xrightarrow{\delta} (p', \nu+\delta) \mid (q', \eta+\delta) = s_1'}$$
The only rule which can be used in the premises of the above is [Del], which implies that $p = p'$, $q = q'$, $\nu + \delta \in \text{rdy}(p)$ and $\eta + \delta \in \text{rdy}(q)$. Then, the thesis follows by rule [M-Del].

(b) To prove item ((b)) of Definition 13.1.6, we consider the possible moves of s_2:

– [M-⊕]. We have $p = \mathtt{a!}\{g, R\}.p' \oplus p''$, $\nu \in [\![g]\!]$, and:

$$s_2 \xrightarrow{\mathsf{A:a!}} (p', [\mathtt{a!}], \nu[R]) \parallel (q, [], \eta) = s_2'$$

So, by rules [⊕] and [S-⊕] we have:

$$\frac{p \xrightarrow{\tau} ([\mathtt{a!}\{g, R\}]\, p', \nu)}{s_1 \xrightarrow{\tau} ([\mathtt{a!}\{g, R\}]\, p', \nu) \mid (q, \eta)} = s_1'$$

and we conclude that $(s_1', s_2') \in \mathcal{R}[2] \subseteq \mathcal{R}$.

– [M-+]. This case does not apply, since both buffers are empty.

– [M-Del]. We have $\nu + \delta \in \mathrm{rdy}(p)$, $\eta + \delta \in \mathrm{rdy}(q)$, and:

$$s_2 \xrightarrow{\delta} (p, [], \nu + \delta) \parallel (q, [], \eta + \delta) = s_2'$$

Then, rule [Del] yields $(p, \nu) \xrightarrow{\delta} (p, \nu + \delta)$ and $(q, \eta) \xrightarrow{\delta} (p, \eta + \delta)$. Hence, by rule [S-Del] we conclude that:

$$s_1 \xrightarrow{\delta} (p, \nu + \delta) \mid (q, \eta + \delta) = s_1'$$

and the thesis follows because $(s_1', s_2') \in \mathcal{R}[1] \subseteq \mathcal{R}$.

(c) To prove item ((c)) of Definition 13.1.6, assume that $s_2 \in S_2$. By definition of S_2, s_2 has the form $(\mathbf{1}, [], \nu) \parallel (q, [], \eta)$. Then, $s_1 = (\mathbf{1}, \nu) \mid (q, \eta) \in S_1$.

Case 2: Let $s_1 = ([\mathtt{a!}\{g, R\}]\, p, \nu) \mid (q, \eta)$ and $s_2 = (p, [\mathtt{a!}], R[\nu]) \parallel (q, [], \eta)$ with $\nu \in [\![g]\!]$.

(a) To prove item ((a)) of Definition 13.1.6, we consider the possible moves of s_1:

– [S-⊕]. We have:

$$\frac{(q, \eta) \xrightarrow{\tau} ([\mathtt{b!}\{f, S\}]\, q', S[\eta])}{s_1 \xrightarrow{\tau} ([\mathtt{a!}\{g, R\}]\, p, \nu) \mid ([\mathtt{b!}\{f, S\}]\, q', S[\eta]) = s_1'}$$

where the premise requires $q = \mathtt{a!}\{f, S\}.q' \oplus q''$ and $\eta \in [\![f]\!]$. Hence, by rule [M-⊕] we have:

$$s_2 \xrightarrow{\mathsf{B:b!}} (p', b, \nu) \parallel (q', [\mathtt{b!}], \eta[S]) = s_2'$$

Then, $(s_1', s_2') \in \mathcal{R}[2] \subseteq \mathcal{R}$.

– [S-τ]. We have:

$$\frac{([\mathtt{a!}\{g, R\}]\, p, \nu) \xrightarrow{\mathtt{a!}} (p, \nu[R]) \quad (q, \eta) \xrightarrow{\mathtt{a?}} (q', \eta[S])}{(p, \nu) \mid (q, \eta) \xrightarrow{\tau} (p', \nu') \mid (q', \eta') = s_1'}$$

where the premise requires $q = \mathtt{a?}\{f, S\}.q' + q''$, with $\nu \in [\![g]\!]$ and $\eta \in [\![f]\!]$. Since $\eta \in [\![f]\!]$, by rule [M-+] we have:

$$(p, [\mathtt{a!}], \nu) \parallel (\mathtt{a?}\{g, R\}.\, q' + q'', [], \eta) \xrightarrow{\mathsf{B:a?}} (p, [], \nu) \parallel (q', [], \eta[R]) = s_2'$$

Then, $(s_1', s_2') \in \mathcal{R}[2] \subseteq \mathcal{R}$.

– [S-Del]. This case does not apply, since one buffers is not empty.

(b) To prove item ((b)) of Definition 13.1.6, we consider the possible moves of s_2:

– [M-⊕]. This case does not apply, given the form of s_2.

– [M-+]. We have:

$$s_2 \xrightarrow{\mathsf{B:a?}} (p, [], \nu[R]) \parallel (q', [], \eta[S]) = s_2'$$

which requires $q = \mathtt{a?}\{f, S\}.q' + q''$, with $\eta \in [\![f]\!]$. Since by hypothesis $\nu \in [\![g]\!]$, by rule [S-⊕] we have:

$$\frac{\overline{([\mathtt{a!}\{g, R\}]\, p, \nu) \xrightarrow{\mathtt{a!}} (p, \nu[R])}\ [\oplus] \qquad \overline{(\mathtt{a?}\{f, S\}.q' + q'', \eta) \xrightarrow{\mathtt{a?}} (q', \eta[S])}\ [+]}{s_1 \xrightarrow{\tau} (p, \nu[R]) \mid (q', \eta[S]) = s_1'}$$

and the thesis follows because $(s_1', s_2') \in \mathcal{R}[1] \subseteq \mathcal{R}$.

– [M-Del]. This case does not apply, given the form of s_2.

(c) To prove item ((c)) of Definition 13.1.6, assume that $s_2 \in S_2$. By definition of S_2, s_2 has the form $(\mathbf{1}, [], \nu) \parallel (q, [], \eta)$. Then, this case does not apply.

Case 3: Let $s_1 = ([\mathtt{a!}\{g, R\}]\, p \mid [\mathtt{b!}\{f, S\}]\, q, \nu)$, and let $s_2 = (p, [\mathtt{a!}], \nu[R]) \parallel (\mathtt{b!}\{f, S\}.q \oplus q', [], \eta)$. The thesis follows trivially, since both s_1 and s_2 are stuck, and neither of them is a success state.

Part B: s_1 turn-simulates s_2 via $\mathcal{R}$.
Case 1: Let $s_1 = (p, \nu) \mid (q, \eta)$ and $s_2 = (p, [], \nu) \parallel (q, [], \eta)$.

(a) To prove item ((a)) of Definition 13.1.6, we consider the possible moves of s_2:

– [M-⊕]. We have:

$$s_2 \xrightarrow{\mathsf{A:a!}} (p', [\mathtt{a!}], \nu[R]) \parallel (q, [], \eta) = s_2'$$

where the premise requires $p = \mathtt{a!}\{g, R\}.p' \oplus p''$ and $\nu \in [\![g]\!]$. Hence, by rule [S-⊕] we have:

$$\frac{(p, \nu) \xrightarrow{\tau} ([\mathtt{a!}\{g, R\}]\, p', \nu)}{s_1 \xrightarrow{\tau} ([\mathtt{a!}\{g, R\}]\, p', \nu) \mid (q, \eta) = s_1'}$$

Then, $(s_1', s_2') \in \mathcal{R}[2] \subseteq \mathcal{R}$.

– [M-+]. This case does not apply.

– [M-Del]. We have:

$$s_2 \xrightarrow{\delta} (p, [], \nu + \delta \parallel q, [], \eta + \delta)$$

where the premise requires $\nu + \delta \in \mathrm{rdy}(p)$ and $\nu + \delta \in \mathrm{rdy}(q)$. Hence, by [Del] we have:

$$\frac{(p, \nu) \xrightarrow{\delta} (p, \nu + \delta) \qquad (q, \eta) \xrightarrow{\delta} (q, \eta + \delta)}{s_1 \xrightarrow{\delta} (p, \nu + \delta) \mid (q, \eta + \delta) = s_1'}$$

Then, $(s_1', s_2') \in \mathcal{R}$.

(b) To prove item ((b)) of Definition 13.1.6, we consider the possible moves of s_1:

- [S-$\oplus$]. We have:
$$s_1 \xrightarrow{\oplus} ([\mathtt{a!}\{g, R\}]\, p', \nu[R]) \mid (q, \eta)$$
whose premise requires $p = \mathtt{a!}\{g, R\}.p' \oplus p''$, with $\nu \in [\![g]\!]$. Hence $s_2 \xrightarrow{\mathsf{A:a!}}\!\!\!\!\!\rightarrow$.
- [S-τ]. This case does not apply.
- [S-Del]. We have:
$$s_1 \xrightarrow{\delta} (p, \nu + \delta) \mid (q, \eta + \delta)$$
where the premise requires $\nu + \delta \in \mathrm{rdy}(p)$ and $\nu + \delta \in \mathrm{rdy}(q)$. Hence, by [M-Del] we have $s_2 \xrightarrow{\delta}\!\!\!\!\!\rightarrow$.

(c) To prove item ((b)) of Definition 13.1.6, assume that $s_2 \in S_2$. By definition of $S - 2$, s_2 has the form $(\mathbf{1} \mid \mathbf{1}, [], \nu)$. Then $s_1 = (\mathbf{1}, \nu) \mid (\mathbf{1}, \eta) \in S_1$.

Case 2: Let $s_1 = ([\mathtt{a!}\{g, R\}]\, p, \nu) \mid (q, \eta)$, $s_2 = (p, [\mathtt{a!}], \nu[R]) \parallel (q, [], \eta)$, and $\nu \in [\![g]\!]$.

(a) To prove item ((a)) of Definition 13.1.6, we consider the possible moves of s_2:

- [M-$\oplus$]. This case does not apply.
- [M-+]. We have:
$$s_2 \xrightarrow{\mathsf{B:a?}}\!\!\!\!\!\rightarrow (p, [], \nu[R]),\ \parallel (q', [], \eta[S]) = s_2'$$
whose premise requires $q = \mathtt{a?}\{f, S\}.q' + q''$ with $\eta \in [\![f]\!]$. Since by hypothesis $\nu \in [\![g]\!]$, by [S-τ] we have:
$$\frac{([\mathtt{a!}\{g, R\}]\, p, \nu) \xrightarrow{\mathtt{a!}} (p, \nu[R]) \quad (q, \eta) \xrightarrow{\mathtt{a?}} (q', \eta[S])}{(p, \nu) \mid (q, \eta) \xrightarrow{\tau} (p', \nu') \mid (q', \eta') = s_1'}$$
Then, $(s_1', s_2') \in \mathcal{R}$.
- [M-Del]. This case does not apply.

(b) To prove item ((b)) of Definition 13.1.6, we consider the possible moves of s_1:

- [S-$\oplus$]. We have:
$$s_1 \xrightarrow{\oplus} ([\mathtt{a!}\{g, R\}]\, p, \nu) \mid ([\mathtt{b!}\{f, S\}]\, q, \eta) = s_1'$$
whose premise requires $q = \mathtt{b!}\{f, S\}.q' \oplus q''$ and $\nu \in [\![f]\!]$. We have that s_2 is stuck but so is s_1'; and $(s_1', s_2) \in \mathcal{R}$.
- [S-τ]. We have:
$$s_1 \xrightarrow{\tau} (p, \nu[R]) \mid (q', \eta[S])$$
whose premise requires $q = \mathtt{b?}\{f, S\}.q' + q''$ and $\eta \in [\![f]\!]$. Hence $s_2 \xrightarrow{\mathsf{B:a?}}\!\!\!\!\!\rightarrow$ $(p \parallel q', [], \nu[R] \sqcup \eta[S]) = s_2'$ and $(s_1', s_2') \in \mathcal{R}$.

– [S-Del]. This case does not apply.

(c) To prove (c), let us assume $s_2 \in S_2$, which implies $s_2 = (\mathbf{1} \mid \mathbf{1}, \nu)$. By hypothesis of *case 2* this is not possible.

Case 3: Let $s_1 = ([\mathtt{a!}\{g, R\}]\, p, \nu) \mid ([\mathtt{b!}\{f, S\}]\, q, \eta)$, $s_2 = (p, [\mathtt{a!}], \nu[R]) \parallel (\mathtt{b!}\{f, S\}.q \oplus q'), [], \eta)$. In this case, both s_1 and s_2 are stuck and neither of them is a success state.

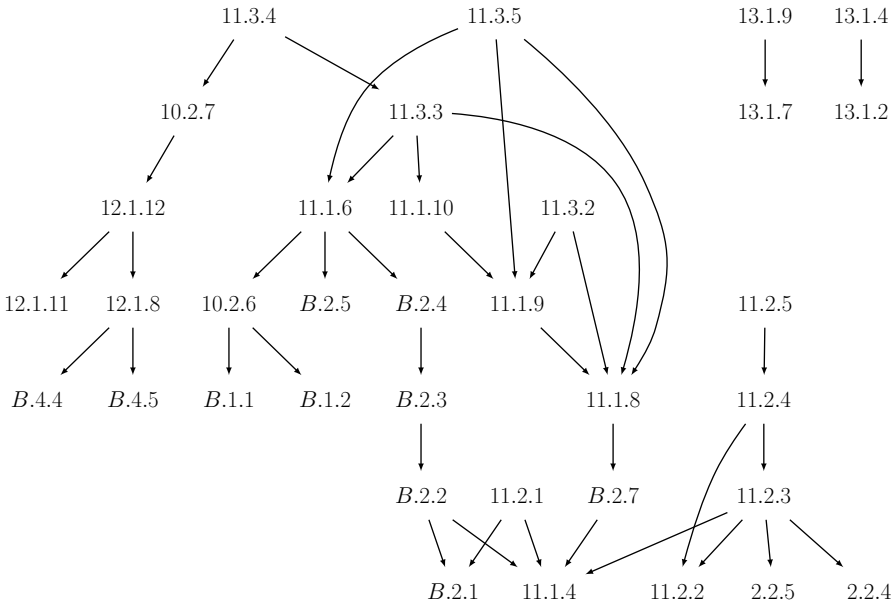

Figure B.2: Dependencies among the proofs.

The diagram in Figure A.4 illustrates the dependencies among the proofs.