A Condensed Goal-Independent Bottom-Up Fixpoint Semantics Modeling the Behavior of *tccp*

MARCO COMINI and LAURA TITOLO ALICIA VILLANUEVA

Abstract

In this paper, we present a new compositional bottom-up semantics for the *Timed Concurrent Constraint Language* (*tccp* in short) which is defined for the full language. In particular, is able to deal with the non-monotonic characteristic of *tccp*, which constitutes a substantial additional technical difficulty w.r.t. other compositional denotational semantics present in literature (which do not tackle the full language).

The semantics is proved to be (correct and) fully abstract w.r.t. the full behavior of *tccp*, including infinite computations. This is particularly important since *tccp* has been defined to model reactive systems and they may not terminate with a purpose.

The overall of these features makes our proposal particularly suitable as the basis for the definition of semantic-based program manipulation tools (like analyzers, debuggers or verifiers), especially in the context of reactive systems.

Key Words: Concurrent Constraint Programming, Denotational Semantics

Paper submitted for publication.

This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, ref. TIN 2010-21062-C02-02, and by Generalitat Valenciana, ref. PROMETEO2011/052.

Author's addresses: M. Comini and L. Titolo, DIMI, Università degli Studi di Udine, Via delle Scienze, 206 33100 Udine, Italy; A. Villanueva, DSIC, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain.

1 Introduction

The concurrent constraint paradigm (*ccp* in short; [25]) is a simple but powerful model for concurrent systems. It is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. In this way, the languages from this paradigm can easily deal with partial information: an underlying constraint system handles constraints on system variables.

Within the *ccp* family, [14] introduced the *Timed Concurrent Constraint Language* (*tccp* in short) by adding to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, it is possible to specify—in a very natural way—behaviors typical of reactive systems

such as *timeouts* or *preemption* actions. For instance, we can have more compact and realistic models since we do not need to explicitly model *error signals*: the absence of information from a given (sub)system can be enough to react to an erroneous situation. Thanks to all its features, *tccp* is certainly the most expressive dialect of the *ccp* family.

In the literature, much effort has been devoted to the development of *appropriate* denotational semantics for languages in the *ccp* paradigm (e.g. [15, 12, 16]). Compositionality and full abstraction are two highly desirable properties for a semantics, since they are needed for many purposes. A fully abstract model can be considered *the* semantics of a language [15].

In [12], the difficulties for handling nondeterminism and infinite behavior in the *ccp* paradigm was investigated. The authors showed that the presence of nondeterminism, local variables and synchronization require relatively complex structures for the denotational model of (non timed) *ccp* languages. In most *ccp* languages, nondeterminism is defined in terms of a global choice, which poses even more difficulties than a local-choice model [16].

Successively, [22] showed that for timed concurrent constraint languages, the presence of timing constructs which handle negative information in addition to non-determinism and local variables significantly complicates the definition of compositional and fully abstract semantics. Moreover, infinite behaviors (which become natural in the timed extensions) are an additional nightmare [12].

Presumably because of all these mentioned difficulties, for all languages of the *ccp* family, the proposals of compositional semantics in the literature have been given by introducing (quite) severe restrictions on the languages. Essentially, they all limit the use of negative information and non-determinism, that are the distinguishing features that enhance the expressiveness of the paradigm w.r.t. other traditional ones. For us, this is contradictory and certainly unsatisfactory. Thus, we strived to develop a semantics which is fully abstract for the full *tccp* language. This is particularly important when one is interested in applying the semantics to develop (semantics-based) *fully automatic* program manipulation tools (like debuggers, verifiers and analyzers).

We have a long experience [9, 6, 8, 7, 1, 2, 4] in the development of semantics-based program manipulation tools for declarative languages via the Abstract Interpretation approach. Abstract Interpretation [11] is a theory of approximation of discrete systems that allows one to formally specify provably correct approximation processes for the behavior of any computing system.

It is important to observe that both our direct experience and the not very satisfactory results in [1, 18] showed that, besides the necessary requirement of full abstraction and the straightforward usefulness of compositionality, there are some other properties of the (concrete) semantics that are *particularly* relevant for having an effective and efficient implementation which computes a precise-as-possible (abstract) approximate semantics. We are confident that this statement applies also to other semantic-based debugging or verification techniques not based on abstract interpretation.

Let us point out these properties and discuss about their positive effects.

Goal-independent A semantics has a *goal-independent* definition when the denotation of any compound (nested) expression is defined in terms of the denotations of most general calls. For instance, the semantics of an expression like $e := f(g(v_1, v_2), v_3)$ is obtained by suitable semantics op-

erators which, considering values v_1, v_2, v_3 and the semantics of $f(x, y)$ and $g(w, z)$, can reconstruct the proper semantics of e. Operational (top-down) semantics are rarely defined in this way since it is more natural (and easy) to give a (compositional) goal-dependent definition which produces the effects of the current expression that has to be evaluated. When one is interested in the results of the evaluation of a *specific* expression e, it would make little sense to define a more complicated goal-independent semantics formalization that first evaluates *all* most general expressions and then tailors such evaluations on e to mimic the effects of a top-down goal-dependent resolution mechanism. In the tailoring process, many parts of the computed denotations will not be used and thus much computation effort would be wasted.

However, when we are no longer focused on determining the actual evolution of a specific expression but we are interested in determining the properties of a program for all possible executions, things change radically. In this case, we necessarily have to determine the semantic information regarding all possible expressions, and then it is more economical to have a goal-independent definition and compute just the semantics of most general calls (and, when is needed, reconstruct from these the semantics of specific instancies).

Condensed A semantics is *condensed* when denotations contain only the minimal necessary number of semantic elements that are needed to characterize the classes of semantically equivalent syntactic objects (or, in other words, the minimal information needed to distinguish a syntactic object x from the other syntactic objects that are not semantically equivalent to x).

This may not seem a useful property for a concrete semantics, which—in general—would nevertheless contain infinite elements even when is condensed. However, this reduction could anyway frequently change some infinite denotations into finite ones and—most important—*all* the abstractions of a condensed concrete semantics will inherit this property *by construction*. Hence, by having minimal (abstract) denotations, one obtains *by definition* algorithms that compute just the minimal number of (abstract) semantic elements. This is definitely a stunning advantage over non condensed approaches which rarely can regain this efficiency in some other way. One could argue that it would be possible to live with a simpler non-condensed concrete semantics and then, for each abstraction of interest, work on the specific case to find out its condensed representation. We find more economical (especially in the long run) to do the effort once for the concrete semantics and then obtain, by construction, that all abstractions are condensed (with no additional effort).

Bottom-up A *bottom-up* definition (in addition to the previous properties), has also an immediate direct benefit for abstract computations. With a bottom-up definition, at each iteration we have to collect the contributions of all rules. For each rule we will use the join operation of the abstract domain in parallel onto all components of the body of the rule. With a top-down definition instead, we have to expand one component of the goal expression at a time, necessarily using several subsequent applications of

the join operation (of the abstract domain) over all components, rather than a unique simultaneous join of all the semantics of components. The reduced use of the join of a bottom-up formulation has a twofold benefit. On one side, it speeds up convergence of the abstract fixpoint computation. On the other side, it considerably improves precision.

Thus—with the application to program manipulation tools in mind—we have developed a new (small-step) compositional, bottom-up, goal-independent and condensed semantics which is (correct and) *fully abstract* w.r.t. the small-step behavior of *full tccp*. To obtain this semantics the idea is to enrich the classical behavioral timed traces with information about the *essential* conditions that the store must (or must not) satisfy in order to make the program proceed with one or another execution branch. Thus, we associate conditions to the store of each computation step and then we collect just the most general hypothetical computations. Since conditions are constructed by using only the information in the guards of a program, we obtain a condensed semantics which also deals with non-monotonicity, because into denotations we have the *minimal* information needed to exploit computations arising from absence of information.

Since *tccp* was originally defined to model reactive systems, which many times include systems that do not terminate *with a purpose*, we have developed our semantics to distinguish among *terminating*, *suspending* and *non-terminating* computations. This improves the original semantics for *tccp* defined in [14] which identifies suspending and non-terminating computations. In particular, terminating computations are those that reach a point in which no agents are pending to be executed. Suspending computations are those that reach a point in which there are some agents pending to be executed, but there is not enough information in the store to entail the conditions that would make them evolve. We think it is essential to distinguish these two kinds of computations since, conceptually, a suspended computation has not completely finished its execution, and, in some cases, it could be a symptom of a system error.

To complete our proposal, we also define a big-step semantics (by abstraction of our small-step semantics) which tackles also outputs of infinite computations. We prove that its fragment for finite computations is (essentially) isomorphic to the traditional big-step semantics of [14]. Moreover, we also formally prove that it is not possible to have a correct input-output semantics which is defined *solely* on the information provided by the input/output pairs.

Organization of the paper The rest of the paper is organized as follows. Section 2 recalls the foundations of the *tccp* language. Section 3 introduces our small-step denotational semantics for *tccp*. It also includes illustrative examples for the main concepts. Then, Section 4 introduces our big-step semantics and formally relates it to the (original) one of [14]. Section 5 discusses about applications. Finally, Sections 6 and 7 present related work and conclude.

To improve readability of the paper, the most technical results and the proofs (of all results) can be found in Appendix A.

2 Preliminaries

The languages defined within the *ccp* paradigm (as extensions of the original model of Saraswat in [28]) are parametric w.r.t. a cylindric constraint system.

The constraint system handles the data information of the program in terms of constraints.

2.1 Cylindric constraint systems

An elegant formalization of constraint systems, the *cylindric constraint systems*, was introduced in [28], where a *hiding* operator is defined in terms of a general notion of existential quantifier (to handle local variables). However in this work, since we are dealing with *tccp*, we use the formalization of [14].

A *cylindric constraint system* is an algebraic structure $\mathbf{C} = \langle \mathcal{C}, \preceq, \otimes, \oplus, \mathit{ff}, \mathit{tt}, \mathit{Var}, \exists \rangle$, where Var is a denumerable set of variables, such that:

1. $(\mathcal{C}, \preceq, \otimes, \oplus, \mathit{ff}, \mathit{tt})$ is a complete algebraic lattice where $\otimes$ is the *lub* operator, $\oplus$ is the *glb* operator, ff and tt are respectively the greatest and least element of $\mathcal{C}$.
2. For each $x \in \mathit{Var}$ there exist a *cylindric operator* $\exists_x : \mathcal{C} \to \mathcal{C}$ such that, for any $x, y \in \mathit{Var}$ and $c, d \in \mathcal{C}$,

$$c \vdash \exists_x c \qquad\qquad c \vdash d \Rightarrow \exists_x c \vdash \exists_x d$$
$$\exists_x(\exists_y c) = \exists_y(\exists_x c) \qquad\qquad \exists_x(c \otimes \exists_x d) = \exists_x c \otimes \exists_x d$$

   where $\vdash$, called *entailment*, is the inverse relation of $\preceq$.

In the sequel, we abuse of notation and, given $C \subseteq \mathcal{C}$, write $\exists_x C$ for $\{\exists_x c \mid c \in C\}$. We can find in the literature several examples of cylindric constraint systems that are useful when modeling data structures, logic programs or other specific domains [29, 12, 13, 3].

In the illustrative examples throughout the paper we will use, for the sake of simplicity, the following classical cylindric constraint system $\mathbf{L}$ of linear disequalities. The domain of constraints $\mathcal{L}$ is formed by taking equivalence classes, modulo logical equivalence $\Leftrightarrow$, of finite conjunctions of either linear disequalities (strict and not) or equalities over $\mathbb{Z}$ and $\mathit{Var} = \{x, y, \ldots\}$ (e.g. $x > 4$, $y \geq 10 \wedge w < -3$, $\ldots$). The entailment relation is implication $\Rightarrow$ (thus, the order of the lattice is $\Leftarrow$). The *lub* is conjunction $\wedge$ and $\exists_x$ is the operation which removes (after information has been propagated within a constraint) all conjuncts referring to variable x (e.g. $\exists_x(x = y \wedge x > 3) = y > 3$). It can be easily verified that $\mathbf{L} := \langle \mathcal{L}, \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ is a cylindric constraint system.

2.2 Timed Concurrent Constraint Programming

The *tccp* language, introduced in [14], is particularly suitable to specify concurrent reactive systems. In *tccp*, the computation progresses as the concurrent and asynchronous activity of several agents that can (monotonically) accumulate information in a *store* or also query information from it. The notion of time is introduced by defining a discrete and global clock[1] and progresses depending on agents that are executed as defined in the following.

Given a cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \preceq, \otimes, \oplus, \mathit{ff}, \mathit{tt}, \mathit{Var}, \exists \rangle$ and a set of process symbols Π, the syntax of agents is given by the following grammar:

$$A ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid A \parallel A \mid \exists x\, A \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid p(x_1, \ldots, x_m)$$

[1] Differently from other languages where time is explicitly introduced by defining new *timing* agents.

$$\frac{}{\langle \mathsf{tell}(c),\, d\rangle \to \langle \mathsf{skip},\, c \otimes d\rangle}\ d \neq \mathit{ff} \qquad (\mathbf{R1})$$

$$\frac{}{\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i,\, d\rangle \to \langle A_j,\, d\rangle}\ j \in [1,n],\ d \vdash c_j,\ d \neq \mathit{ff} \qquad (\mathbf{R2})$$

$$\frac{\langle A,\, d\rangle \to \langle A',\, d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \to \langle A',\, d'\rangle}\ d \vdash c \qquad (\mathbf{R3})$$

$$\frac{\langle A,\, d\rangle \not\to}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \to \langle A,\, d\rangle}\ d \vdash c,\ d \neq \mathit{ff} \qquad (\mathbf{R4})$$

$$\frac{\langle B,\, d\rangle \to \langle B',\, d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \to \langle B',\, d'\rangle}\ d \nvdash c \qquad (\mathbf{R5})$$

$$\frac{\langle B,\, d\rangle \not\to}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \to \langle B,\, d\rangle}\ d \nvdash c \qquad (\mathbf{R6})$$

$$\frac{\langle A,\, d\rangle \to \langle A',\, d'\rangle \quad \langle B,\, d\rangle \to \langle B',\, c'\rangle}{\langle A \parallel B,\, d\rangle \to \langle A' \parallel B',\, d' \otimes c'\rangle} \qquad (\mathbf{R7})$$

$$\frac{\langle A,\, d\rangle \to \langle A',\, d'\rangle \quad \langle B,\, d\rangle \not\to}{\langle A \parallel B,\, d\rangle \to \langle A' \parallel B,\, d'\rangle} \qquad (\mathbf{R8})$$

$$\frac{\langle A,\, l \otimes \exists_x d\rangle \to \langle B,\, l'\rangle}{\langle \exists^{l} x\, A,\, d\rangle \to \langle \exists^{l'} x\, B,\, d \otimes \exists_x l'\rangle} \qquad (\mathbf{R9})$$

$$\frac{}{\langle p(\vec{x}),\, d\rangle \to \langle A,\, d\rangle}\ p(\vec{x}) \coloneq A \in D,\ d \neq \mathit{ff} \qquad (\mathbf{R10})$$

Figure 1: The transition system for *tccp*.

where c, $c_1, \ldots, c_n$ are finite constraints in $\mathcal{C}$; p is a symbol in Π of *arity* m (denoted as $p_{/m} \in \Pi$) and $x, x_1, \ldots, x_m \in \mathit{Var}$.

A *tccp* program P is an object of the form $D\,.\,A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) \coloneq A$ (for some agent A), where $\vec{x}$ denotes a generic tuple of variables.

The following definition introduces the operational semantics of the language. It is slightly different from the original one in [14]. In particular, we have introduced conditions in specific rules (namely Rules **R2**, **R4** and **R10**) in order to detect when the store becomes ff. This modification is made to follow the original philosophy of *ccp* computations defined in [29], where computations that reach an inconsistent store are considered failure computations. In [14], this check is not explicitly done. In our context, we are interested in detecting when a computation reaches ff; however, once ff is reached, no action can modify the store (ff is the greatest element in the domain) and—after that moment—all guards in the program agents are always entailed, thus the computation from that instant has little interest. In particular, we do not want to distinguish computations which end in ff from those which loop on store ff, contrarily to what [14] does.

Definition 2.1 (Operational semantics of *tccp*) *The* operational semantics *of tccp is formally described by a transition system* $T = (\mathit{Conf}, \to)$. *Configura-*

tions in Conf are pairs $\langle A, c\rangle$ *representing the agent* A *to be executed in the current global store* c*. As usually done, we assume that the tccp syntax is closed under the usual structural equivalence relation where the parallelism operator is associative and commutative, and agents* $A \parallel \mathsf{skip}$ *and* A *are equivalent.*

The transition relation $\rightarrow \subseteq Conf \times Conf$ *is the least relation satisfying the rules of Figure 1. Each transition step takes exactly one time-unit.*

In the sequel $\rightarrow^*$ *denotes the reflexive and transitive closure of the relation* $\rightarrow$*.*

As can be seen from the rules, the skip agent represents the successful termination of the computation. The $\mathsf{tell}(c)$ agent adds the constraint c to the store and then stops. It takes one time-unit, thus the constraint c is visible to other agents from the following time instant. The store is updated by means of the $\otimes$ operator of the constraint system.

The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ consults the store and non-deterministically executes (at the following time instant) one of the agents A_i whose corresponding guard c_i is entailed by the current store; otherwise, if no guard is entailed by the store, the agent suspends.

The conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves in the current time instant like A (respectively B) if c is (respectively is not) entailed by the store. Note that, because of the ability of *tccp* to handle partial information, $d \nvdash c$ is not equivalent to $d \vdash \neg c$. Thus, the else branch is taken not only when the condition is falsified, but also when there is not enough information to entail the condition. This characteristic is known in the literature as the ability to process "negative information" [26, 27].

$A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism (in contrast to the interleaving approach of *ccp*), i.e., all the enabled agents of A and B are executed at the same time.

The agent $\exists x\, A$ makes variable x local to A. To this end, it uses the $\exists$ operator of the constraint system. More specifically, it behaves like A with x considered local, i.e., the information on x provided by the external environment is hided to A, and the information on x produced by A is hided to the external world. In [14], an auxiliary construct $\exists^d x$ is used to explicitly show the store local to A. In particular, in Rule **R9**, the store d in the agent $\exists^d x\, A$ represents the store local to A. This auxiliary operator is linked to the hiding construct by setting the initial local store to tt, thus $\exists x\, A := \exists^{tt} x\, A$.

Finally, the agent $p(\vec{x})$ takes from D a declaration of the form $p(\vec{x}) :- A$ and then executes A at the following time instant. For the sake of simplicity, we assume that sets of declarations D are closed w.r.t. renaming of parameter names, i.e., if $p(\vec{x}) :- A \in D$ then, for any $\vec{y} \in Var$, also $p(\vec{y}) :- A\{\vec{x}/\vec{y}\} \in D$.

3 Modeling the small-step operational behavior of *tccp*

In this section, we introduce a new condensed, compositional, bottom-up denotational semantics which is (correct and) fully abstract w.r.t. the small-step (operational) behavior of *tccp*. Note that (as mentioned in the introduction), having a semantics which enjoys simultaneously all these (just mentioned) properties, besides the theoretical interest, it is also a matter of pragmatical relevance

since they are the key factors for having an effective and efficient implementation of semantics-based program manipulation tools.

In order to introduce such semantics, we need first to define some (technical) notions. In the sequel, all definitions are parametric w.r.t. a cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \oplus, ff, tt, Var, \exists \rangle$. We denote by $\mathbb{A}^{\Pi}_{\mathbf{C}}$ the set of agents and $\mathbb{D}^{\Pi}_{\mathbf{C}}$ the set of sets of process declarations built on signature Π and constraint system $\mathbf{C}$. By ϵ we denote the empty sequence; by $s_1 \cdot s_2$ the concatenation of two sequences s_1, s_2. We also abuse notation and, given a set of sequences S, by $s_1 \cdot S$ we denote $\{s_1 \cdot s_2 \,|\, s_2 \in S\}$.

Let us formalize first the notion of behavior of a set D of process declarations in terms of the transition system described in Figure 1. It collects all the small-step computations associated to D as the set of (all the prefixes of) the sequences of computation steps (in terms of sequences of stores), for all possible initial agents and stores.

Definition 3.1 *Let $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$. Then the* small-step (observable) behavior *of D is defined as:*

$$\mathcal{B}^{ss}[\![D]\!] := \bigcup_{\forall c \in \mathbf{C}, \forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}}} \mathcal{B}^{ss}[\![D \,.\, A]\!]_c \qquad \textit{where}$$

$$\mathcal{B}^{ss}[\![D \,.\, A]\!]_{c_0} := \left\{ c_0 \cdot c_1 \cdot \ldots \cdot c_n \,\middle|\, \langle A, c_0 \rangle \rightarrow \langle A_1, c_1 \rangle \rightarrow \ldots \rightarrow \langle A_n, c_n \rangle \right\} \cup \{\epsilon\}$$

(where $\rightarrow$ is the transition relation given in Figure 1).

We call the sequences in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_c$ behavioral timed traces *or simply* traces *(when clear from the context).*

We denote by $\approx_{ss}$ the equivalence relation between process declarations induced by $\mathcal{B}^{ss}$, namely for all $D_1, D_2 \in \mathbb{D}^{\Pi}_{\mathbf{C}}$, $D_1 \approx_{ss} D_2 \iff \mathcal{B}^{ss}[\![D_1]\!] = \mathcal{B}^{ss}[\![D_2]\!]$.

With this definition, we can formally state the requirement of full abstraction for semantics $\mathcal{S}$ as $\mathcal{S}[\![D_1]\!] = \mathcal{S}[\![D_2]\!] \iff D_1 \approx_{ss} D_2$.

To achieve a goal-independent semantics, a typical solution is to define denotations by using only the most general traces (in our case those for the weakest store) *plus* define a suitable semantic operator which can reconstruct the semantics of any expression (in our case agent) from such most general denotations. The wanted result can be achieved in this way only if the set of all traces for each expression is itself *condensing* (borrowing the terminology program analysis [20, 21]), which in our case means that the set of all traces for an agent A with initial store c can be reconstructed from the set of all traces of A with initial store *true*. The problem in following this approach in the *tccp* case is that $\mathcal{B}^{ss}$ is not condensing, since not all behavioral timed traces can be retrieved from the most general ones. This is due to the ask, now and hiding constructs. For instance, consider the agent $A := \mathsf{now}\ x = 3\ \mathsf{then}\ \mathsf{tell}(z = 0)\ \mathsf{else}\ \mathsf{tell}(z = 1)$. Given the initial store *true*, we obtain the trace $true \cdot z = 1$, while for the stronger initial store $x = 3$ we obtain the trace $x = 3 \cdot (x = 3 \wedge z = 0)$, which is not comparable to the former (since $z = 0 \not\Rightarrow z = 1$ and $z = 1 \not\Rightarrow z = 0$). Hence, the latter trace cannot be obtained from the former trace, which has been generated for the *most general* store. Indeed—in general—in *tccp*, given $S := \mathcal{B}^{ss}[\![D \,.\, A]\!]_c$ (the set of traces for an agent A with initial store c), if we compute $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$ with a stronger initial store d ($d \vdash c$), then some traces of S may disappear and, what is more critical, new traces, *which are not instances* of the ones in S, can

appear. In the community of the *ccp* paradigm [12, 27], this characteristic is known as "non-monotonicity of the language".

Because of *tccp*'s non-monotonicity, $\mathcal{B}^{ss}$ is also not compositional. For instance, consider the agents $A_1 := \mathsf{tell}(x = 1)$ and

$$A_2 := \mathsf{ask}(\mathit{true}) \rightarrow \mathsf{now}\ (x = 1)\ \mathsf{then}\ \mathsf{tell}(y = 0)\ \mathsf{else}\ \mathsf{tell}(y = 1)$$

For each c, $\mathcal{B}^{ss}[\![\varnothing\,.\,A_1]\!]_c = \{c \cdot (x = 1 \wedge c)\}$. Moreover, for each c that implies[2] $x = 1$, $\mathcal{B}^{ss}[\![\varnothing\,.\,A_2]\!]_c = \{c \cdot c \cdot (y = 0 \wedge c)\}$ while, when $c \not\Rightarrow (x = 1)$, $\mathcal{B}^{ss}[\![\varnothing\,.\,A_2]\!]_c = \{c \cdot c \cdot (y = 1 \wedge c)\}$. Now, for the parallel composition of these agents $A_1 \parallel A_2$, $\mathcal{B}^{ss}[\![\varnothing\,.\,A_1 \parallel A_2]\!]_{\mathit{true}} = \{\mathit{true} \cdot (x = 1) \cdot (x = 1 \wedge y = 0)\}$ which cannot be computed by *merging* the traces of A_1 and A_2.

Thus, it does not come as a surprise that for all non-monotonic languages of the *ccp* paradigm, the compositional semantics that have been written [29, 26, 12, 13, 16, 23, 18, 24, 17] are not defined for the full language, either because they avoid the constructs that cause non-monotonicity or because they restrict their use. Hence, the ability to handle non-monotonicity (and thus the full language without any limitation) is certainly one of the strengths of our proposal.

The example above shows why, due to the non-monotonicity of *tccp*, in order to obtain a compositional (and goal-independent) semantics *for the full language* it is not possible to follow the traditional strategy and collect in the semantics the traces associated to the weakest initial store. Actually, we have found the solution to the problem of compositionality by trying to solve another (related) problem. Since in a top-down (goal-dependent) approach the (initial) current store is propagated, then the decisions regarding a conditional or choice agent (where the computation evolves depending on the entailment of the guards in the current store) can be taken immediately. However, if we want to define a fixpoint semantics which builds the denotations bottom-up we have the problem that, while we are building the fixpoint, we do not know the current store yet. Thus, it is impossible to know *which* execution branch *has to be* taken in correspondence of a program's guard.

To solve both problems our proposal is to enrich behavioral timed traces with information about the essential conditions that the store must (or must not) satisfy in order to make the program proceed with one or another execution branch. Thus, we associate conditions to the store of each computation step and then we collect (only) the most general hypothetical computations. These conditions are constructed by using the information in the guards of the ask and now constructs of a program.

We will formally show that this indeed solves both the problem of constructing bottom-up the semantics and of having a compositional and condensed semantics coping with non-monotonicity.

3.1 The semantic domain

Let us start by introducing the notion of condition, that is the base to build our denotations. Intuitively, we need "positive conditions" for branches related to the entailment of guards and "negative conditions" for non-entailment, i.e., for the branches where the current store does not entail the associated condition.

[2] We recall that in this exemplification cylindric constraint system, the entailment is logical implication.

Definition 3.2 (Conditions) *A* condition η*, over Cylindric Constraint System* $\mathbf{C}$*, is a pair* $\eta = (\eta^+, \eta^-)$ *where*

- $\eta^+ \in \mathbf{C}$ *is called* positive condition*, and*
- $\eta^- \in \wp(\mathbf{C})$ *is called* negative condition*.*

A condition is valid *when* $\eta^+ \neq ff$*,* $tt \notin \eta^-$ *and* $\forall c \in \eta^-. \eta^+ \nvdash c$*. We denote* $\Lambda_{\mathbf{C}}$ *the set of all conditions and* $\Delta_{\mathbf{C}}$ *the subset of valid ones.*

The conjunction of two conditions $\eta_1 = (\eta_1^+, \eta_1^-)$ *and* $\eta_2 = (\eta_2^+, \eta_2^-)$ *is defined (by abuse of notation) as* $\eta_1 \otimes \eta_2 := (\eta_1^+ \otimes \eta_2^+, \eta_1^- \cup \eta_2^-)$*. Two conditions are called* incompatible *if their conjunction is not valid.*

A store $c \in \mathbf{C}$ *is* consistent *with* η*, written* $c \gg \eta$*, if* $\eta^+ \otimes c \neq ff$ *and* $\forall h \in \eta^-. c \nvdash h$*. Moreover, we say that* c satisfies η*, written* $c \models \eta$*, when* $c \vdash \eta^+$ *and* $\forall h \in \eta^-. c \nvdash h$*.*

We extend the $\exists_x$ *operator to conditions as* $\exists_x(\eta^+, \eta^-) := (\exists_x \eta^+, \exists_x \eta^-)$*.*

Due to the partial nature of the constraint system, for negative conditions we cannot use the *glb* (disjunction) $\bigoplus_{i=1}^n c_i$ instead of set $\{c_1, \ldots, c_n\}$ since we can have a store c such that $c \vdash \bigoplus_{i=1}^n c_i$ while $\forall i. c \nvdash c_i$. For instance, we can have two guards $x > 2$ and $x \leq 2$ and it may happen that the current store does not satisfy any of them, but their *glb* $x > 2 \oplus x \leq 2$ (which is *true*) is entailed by any store.

Clearly, if a store—different from *ff*—satisfies a condition, then it is also consistent with that condition. If two conditions are incompatible, then there exists no constraint $c \in \mathbf{C} \setminus \{ff\}$ that entails simultaneously both conditions.

Now we are ready to enrich with conditions the notion of trace.

Definition 3.3 (Conditional state) *A* conditional state*, over Cylindric Constraint System* $\mathbf{C}$*, is one of the following constructs.*

Conditional store *A pair* $\eta \rightarrowtail c$*, for each* $\eta \in \Lambda_{\mathbf{C}}$ *and* $c \in \mathbf{C}$*.*

Stuttering *The construct* $stutt(C)$*, for each finite* $C \subseteq \mathbf{C} \setminus \{tt\}$*.*

End-of-process *The construct* $\boxtimes$*.*

In a conditional store $t = \eta \rightarrowtail c$*, the constraint* c *is the* store *of* t*.*

We say that $\eta \rightarrowtail c$ *is* valid *if* η *is valid.*

We extend $\exists_x$ *to conditional states as* $\exists_x \big((\eta^+, \eta^-) \rightarrowtail c\big) := \exists_x(\eta^+, \eta^-) \rightarrowtail \exists_x c$*,* $\exists_x\, stutt(C) := stutt(\exists_x C)$ *and* $\exists_x \boxtimes := \boxtimes$*.*

The conditional store $\eta \rightarrowtail c$ is used to represent a hypothetical computation step where η is the condition that the current store must satisfy in order to make the computation proceed. Moreover, c represents the information that is added by an agent to the global store up to the current time instant.

The stuttering $stutt(C)$ is needed to model the suspension of the computation due to an ask construct, i.e., it represents the fact that there is no guard in C (the guards of a choice agent) entailed by the current store.

Definition 3.4 (Conditional trace) *A* conditional trace *(over Cylindric Constraint System* $\mathbf{C}$*) is a (possibly infinite) sequence* $t_1 \cdots t_n \cdots$ *of valid conditional states (over* $\mathbf{C}$*)—where* $\boxtimes$ *can be used only as a terminator—that respects the following properties:*

Monotonicity *For each* $t_i = \eta_i \rightarrowtail c_i$ *and* $t_j = \eta_j \rightarrowtail c_j$ *such that* $j \geq i$, $c_j \vdash c_i$.
Consistency *For each* $t_i = \eta_i \rightarrowtail c_i$ *and* t_{i+1} *either of the form* $(\eta^+_{i+1}, \eta^-_{i+1}) \rightarrowtail c_{i+1}$ *or* $stutt(\eta^-_{i+1})$, *we have that* $\forall c^- \in \eta^-_{i+1}. c_i \nvdash c^-$.

We denote by $\mathbf{CT_C}$ *the set of all conditional traces, or simply* **CT** *when clear from the context.*

The limit store *of a (finite or infinite) trace* s *is the lub of the stores (of the conditional states) of* s.

A finite conditional trace that is ended with $\boxtimes$ *as well as an infinite conditional trace is said, respectively,* failed *or* (finitely) successful *depending on whether its limit store* c *is ff or not. Such* c *is called* computed result.

Each conditional trace models a hypothetical *tccp* computation where, for each time instant, we have a conditional state where each condition represents the information that the global store has to satisfy in order to proceed to the next time instant.

The Monotonicity property is needed since in *tccp*, as well as in *ccp* but not in all its extensions, each store in a computation entails the previous ones. Note that because of this, for any *finite* conditional trace $t_1, \ldots, t_n$ whose sequence of stores (of the conditional stores) is $c_1, \ldots, c_m$ $(m \leq n)$, the limit store $\otimes_{i=1}^{m} c_i$ is just the last store c_m.

The Consistency property affirms that the store of a given conditional store cannot be in contradiction with the condition associated to the successive conditional state.

Example 3.5
It is easy to verify that the sequence $r_1 \coloneqq (true, \varnothing) \rightarrowtail y = 0 \cdot (x > 2, \varnothing) \rightarrowtail y = 0 \wedge z = 3 \cdot \boxtimes$ is a conditional trace. The first component of the trace states that in the first time instant the store $y = 0$ is computed in any case (the condition $(true, \varnothing)$ is always satisfied). The second component requires the constraint $x > 2$ to be satisfied by the (global) store in order to proceed by adding to the next state the information $z = 3$.

Instead, the sequence $r_2 \coloneqq (true, \varnothing) \rightarrowtail x = 0 \cdot (x = 0, \varnothing) \rightarrowtail tt \cdot \boxtimes$ is not a conditional trace since the Monotonicity property does not hold because $tt \nvdash x = 0$. Also $r_3 \coloneqq (true, \varnothing) \rightarrowtail x = 0 \cdot stutt(\{x \geq 0\}) \cdot \boxtimes$ is not a conditional trace: it does not satisfy the Consistency property since $x = 0$ implies the (only) negative condition in the successive conditional state $(x \geq 0)$.

Note that finite conditional traces not ending in $\boxtimes$ are partial traces that can still evolve and thus they are always a prefix of a longer conditional trace.

Definition 3.6 (Semantic domain) *A set* $R \subseteq \mathbf{CT}$ *is* closed by prefix *if for each* $r \in R$, *all the prefixes* p *of* r *(denoted as* $p \leq_{pref} r$*) are also in* R.

We denote the domain of non-empty *sets of conditional traces that are closed by prefix as* $\mathbb{P}$ *(i.e.,* $\mathbb{P} \coloneqq \{R \subseteq \mathbf{CT} \mid R \neq \varnothing, r \in R \Rightarrow \forall p \leq_{pref} r. p \in R\}$*).*

We order elements in $\mathbb{P}$ *by set inclusion* $\subseteq$.

It is worth noting that $(\mathbb{P}, \subseteq, \bigcup, \bigcap, \mathbf{CT}, \{\epsilon\})$ is a complete lattice.

This conceptual representation is pretty simple, especially to understand the lattice structure, considered the fact that we admit infinite traces. However, each prefix-closed set contains a lot of redundant traces, which are quite

inconvenient for technical definitions. Thus, we will use an equivalent representation obtained by considering the crown of prefix-closed sets. Namely, given $P \in \mathbb{P}$, we remove all the prefixes of a trace in the set with the function $\mathit{maximal}(P) := \{r \in P \mid \nexists p \in P \setminus \{r\}.\, r \leq_{\mathit{pref}} p\}$. Let $\mathbf{M} := \mathit{maximal}(\mathbf{CT})$, $\mathbb{M} := \{\mathit{maximal}(P) \,|\, P \in \mathbb{P}\}$ and call *maximal conditional trace sets* the elements of $\mathbb{M}$. The inverse of map *maximal* is, for each $M \in \mathbb{M}$,

$$\mathit{prefix}(M) := \left\{p \in \mathbf{CT} \,\middle|\, p \leq_{\mathit{pref}} r, r \in M\right\} \tag{3.1}$$

The order of $\mathbb{M}$ is induced from the one in $\mathbb{P}$ as $M_1 \sqsubseteq M_2 \iff \mathit{prefix}(M_1) \subseteq \mathit{prefix}(M_2)$ which is equivalent to say that $M_1 \sqsubseteq M_2 \iff \forall r_1 \in M_1\, \exists r_2 \in M_2.\, r_1 \leq_{\mathit{pref}} r_2$. We define the *lub* $\bigsqcup$ and the *glb* $\bigsqcap$ of $\mathbb{M}$ analogously. It is straightforward to prove that $(\mathbb{P}, \subseteq) \underset{\mathit{maximal}}{\overset{\mathit{prefix}}{\leftrightarrows}} (\mathbb{M}, \sqsubseteq)$ is an *order-preserving isomorphism*, so $(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\})$ is also a complete lattice.

Although this second representation is very convenient for technical definitions, it is not very suited for examples. For instance, different maximal traces have frequently (significant) common prefixes; hence, some parts have to be written many times and, more important, it can be difficult to visualize the repetition (obfuscating the comprehension). Thus, in our examples we will use another equivalent representation in terms of prefix trees. Namely, we will use trees with (non root) nodes labeled with conditional states. Given $P \in \mathbb{P}$, $\mathit{tree}(P)$ builds the prefix tree of P, obtained by combining all the sequences that have a prefix in common in the same path. Let $\mathbb{T} := \{\mathit{tree}(P) \,|\, P \in \mathbb{P}\}$. The inverse of *tree* is the function $\mathit{path}\colon \mathbb{T} \to \mathbb{P}$ which returns the set of all possible paths starting from the root. Let $\trianglelefteq$ be the order on $\mathbb{T}$ induced by the order on $\mathbb{P}$, i.e., $T_1 \trianglelefteq T_2 \iff \mathit{path}(T_1) \subseteq \mathit{path}(T_2)$. We define the *lub* and *glb* of $\mathbb{T}$ in a similar way. It is straightforward to prove that $(\mathbb{P}, \subseteq) \underset{\mathit{tree}}{\overset{\mathit{path}}{\leftrightarrows}} (\mathbb{T}, \trianglelefteq)$ is an order-preserving isomorphism, so also $(\mathbb{T}, \trianglelefteq)$ is a complete lattice. In the sequel we will use the representation which is most convenient in each case.

3.2 Fixpoint Denotations of Programs

The technical core of our semantics definition is the agent semantics evaluation function (Definition 3.15, page 16) which, given an agent A and an interpretation $\mathcal{I}$ (for the process symbols of A), builds the maximal conditional traces associated to A. To define it, we need first to introduce some auxiliary semantic functions.

Definition 3.7 (Propagation Operator) *Let $r \in \mathbf{M}$ and $c \in \mathbf{C}$. We define the* propagation *of c in r, written $r{\downarrow_c}$, by structural induction as $\boxtimes{\downarrow_c} = \boxtimes$, $\epsilon{\downarrow_c} = \epsilon$ and*

$$((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c} = \begin{cases} (\eta^+ \otimes c, \eta^-) \rightarrowtail d \otimes c \cdot (r'{\downarrow_c}) & \text{if } c \gg (\eta^+, \eta^-),\ d \otimes c \neq \mathit{ff} \\ (\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{ff} \cdot \boxtimes & \text{if } c \gg (\eta^+, \eta^-),\ d \otimes c = \mathit{ff} \end{cases}$$

$$(\mathit{stutt}(\eta^-) \cdot r'){\downarrow_c} = \mathit{stutt}(\eta^-) \cdot (r'{\downarrow_c}) \quad \text{if } \forall c^- \in \eta^-.\, c \nvdash c^-$$

We abuse notation and denote by $R{\downarrow_c}$ the point-wise extension of ${\downarrow_c}$ to sets of conditional traces: $R{\downarrow_c} := \{r{\downarrow_c} \,|\, r \in R \text{ and } r{\downarrow_c} \text{ is defined}\}$.

This operator is used in the definition of the semantics of constructs that add new information to traces. By definition, the *propagation operator* $\downarrow$ is a partial function $\mathbf{M}\times\mathbf{C} \to \mathbf{M}$ that instantiates a conditional trace with a given constraint and checks the consistency of the new information with the conditional states in the trace. This information needs to be propagated also to the successive (i.e., future) conditional states in order to maintain the monotonicity of the store.

Example 3.8

Given the conditional trace $r := (true, \varnothing) \rightarrowtail x > 10 \cdot (true, \varnothing) \rightarrowtail x > 20 \cdot \boxtimes$, the propagation of $y > 2$ in r $(r{\downarrow_{y>2}})$ is $(y > 2, \varnothing) \rightarrowtail x > 10 \wedge y > 2 \cdot (y > 2, \varnothing) \rightarrowtail x > 20 \wedge y > 2 \cdot \boxtimes$.

For $r' := (true, \{y > 0\}) \rightarrowtail true \cdot \boxtimes$ the propagation $r'{\downarrow_{y>2}}$ is not defined since $y > 2 \not\gg (true, \{y > 0\})$.

Finally, given the conditional trace $r'' := (true, \varnothing) \rightarrowtail y < 0 \cdot \boxtimes$, the propagation $r''{\downarrow_{y>2}}$ produces the conditional trace $(y > 2, \varnothing) \rightarrowtail false \cdot \boxtimes$ since $y > 2 \gg (true, \varnothing)$ and $y < 0 \wedge y > 2 = false$.

Note that the consecutive propagation of two constraints $(r{\downarrow_c}){\downarrow_{c'}}$ is equivalent to $r{\downarrow_{(c\otimes c')}}$ (as stated formally in Lemma A.2).

The following parallel composition auxiliary operator is used in the definition of the semantics of the parallel construct. Intuitively, this operator combines (with maximal parallelism) the information coming from two conditional traces. It checks the satisfiability of the conditions and the consistency of the resulting stores.

Definition 3.9 (Parallel composition) *The* parallel composition *partial operator* $\overline{\|}\colon \mathbf{M}\times\mathbf{M} \to \mathbf{M}$ *is the commutative closure of the following partial operation defined by structural induction as:* $r \mathbin{\overline{\|}} \epsilon := r$, $r \mathbin{\overline{\|}} \boxtimes := r$ *and*

$$(stutt(\eta_1^-) \cdot r_1') \mathbin{\overline{\|}} (stutt(\eta_2^-) \cdot r_2') := stutt(\eta_1^- \cup \eta_2^-) \cdot (r_1' \mathbin{\overline{\|}} r_2')$$

Moreover, if $\eta_1 \otimes \eta_2$ *is valid, then*

$$(\eta_1 \rightarrowtail c_1 \cdot r_1') \mathbin{\overline{\|}} (\eta_2 \rightarrowtail c_2 \cdot r_2') := \begin{cases} \eta_1 \otimes \eta_2 \rightarrowtail c_1 \otimes c_2 \cdot ((r_1'{\downarrow_{c_2}}) \mathbin{\overline{\|}} (r_2'{\downarrow_{c_1}})) & \text{if } c_1 \otimes c_2 \neq ff \\ \eta_1 \otimes \eta_2 \rightarrowtail ff \cdot \boxtimes & \text{if } c_1 \otimes c_2 = ff, \end{cases}$$

Finally, if $\forall c^- \in \eta_2^-.\, \eta_1^+ \nvdash c^-$, *then*

$$((\eta_1^+, \eta_1^-) \rightarrowtail c_1 \cdot r_1') \mathbin{\overline{\|}} (stutt(\eta_2^-) \cdot r_2') := (\eta_1^+, \eta_1^- \cup \eta_2^-) \rightarrowtail c_1 \cdot (r_1' \mathbin{\overline{\|}} (r_2'{\downarrow_{c_1}}))$$

Clearly, by definition, $\overline{\|}$ is commutative. Moreover, because of $\otimes$ associativity, $\overline{\|}$ is also associative. It is worth noting that, if the propagated constraint is in contradiction with a condition into a trace r then the parallel composition is not defined on that r.

Example 3.10

Consider $r_1 := (true, \varnothing) \rightarrowtail y > 2 \cdot (y > 2, \varnothing) \rightarrowtail y > 2 \cdot \boxtimes$ and $r_2 := (z = 1, \varnothing) \rightarrowtail z = 1 \cdot \boxtimes$. Since r_1 and r_2 do not share variables, the compatibility checks always succeed and then $r_1 \mathbin{\overline{\|}} r_2 = (z = 1, \varnothing) \rightarrowtail y > 2 \wedge z = 1 \cdot (y > 2 \wedge z = 1, \varnothing) \rightarrowtail y > 2 \wedge z = 1 \cdot \boxtimes$.

Consider now $r_3 := stutt(\{y > 0\}) \cdot (y > 0, \varnothing) \rightarrowtail y > 0 \wedge z = 3 \cdot \boxtimes$. Traces r_1 and r_3 share the variable y and it can be seen that the information regarding

y in the two traces is consistent, thus $r_1 \mathbin{\bar{\|}} r_3 = (\mathit{true}, \{y > 0\}) \rightarrowtail y > 2 \cdot (y > 2, \varnothing) \rightarrowtail y > 2 \wedge z = 3 \cdot \boxtimes$.

Finally, consider $r_4 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \{y > 0\}) \rightarrowtail \mathit{true} \cdot \boxtimes$. This trace, in the second time instant, requires that the constraint $y > 0$ cannot be entailed by the current store. However, the trace r_1 states, at the same time instant, that $y > 2$. This is the reason because $r_1 \mathbin{\bar{\|}} r_4$ is not defined.

Note that $\downarrow$ distributes over $\bar{\|}$, in the sense that $(r_1 \mathbin{\bar{\|}} r_2)\downarrow_c = (r_1\downarrow_c) \mathbin{\bar{\|}} (r_2\downarrow_c)$ (as stated formally in Lemma A.3).

The last auxiliary operator that we need is the hiding operator $\bar{\exists}\colon \mathit{Var} \times \mathbf{M} \to \mathbf{M}$ which, intuitively, hides the information regarding a given variable in a conditional trace.

Definition 3.11 (Hiding operator) *Given $r \in \mathbf{M}$ and $x \in \mathcal{V}$, we define the* hiding of x in r, *written $\bar{\exists}_x r$, by structural induction as $\bar{\exists}_x \epsilon \coloneqq \epsilon$, $\bar{\exists}_x \boxtimes \coloneqq \boxtimes$,*

$$\bar{\exists}_x \left((\eta^+, \eta^-) \rightarrowtail c \cdot r'\right) \coloneqq \exists_x \left((\eta^+, \eta^-) \rightarrowtail c\right) \cdot \bar{\exists}_x r'$$
$$\bar{\exists}_x \left(\mathit{stutt}(\eta^-) \cdot r'\right) \coloneqq \exists_x \mathit{stutt}(\eta^-) \cdot \bar{\exists}_x r'$$

We distinguish two special classes of conditional traces.

Definition 3.12 (Self-sufficient and x-self-sufficient conditional trace) *A maximal trace $r \in \mathbf{M}$ is said to be* self-sufficient *if the first condition is $(tt, \varnothing)$ and, for each $t_i = \eta_i \rightarrowtail c_i$ and $t_{i+1} = \eta_{i+1} \rightarrowtail c_{i+1}$, $c_i \models \eta_{i+1}$ (each store satisfies the successive condition).*

Moreover, r is self-sufficient w.r.t. $x \in \mathcal{V}$ *(x-self-sufficient) if $\bar{\exists}_{\mathit{Var} \setminus \{x\}} r$ is self-sufficient.*

Definition 3.12 is stronger than Definition 3.4 since the latter does not require satisfiability but just consistency of the store w.r.t. conditions. Informally, this new definition demands that for self-sufficient conditional traces, no additional information (from other agents) is needed in order to *complete* the computation. In an x-self-sufficient conditional trace the same happens but only considering information about variable x.

Example 3.13

The conditional trace r_1 of Example 3.5 is not self-sufficient since $y = 0 \not\models x > 2$.

Now consider a variation where we add the information $x = 4$ to the stores, namely $r_2 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail y = 0 \wedge x = 4 \cdot (x > 2, \varnothing) \rightarrowtail y = 0 \wedge z = 3 \wedge x = 4 \cdot \boxtimes$. It is easy to see that r_2 is a self-sufficient conditional trace, essentially because we add enough information in the first store to satisfy the second condition, i.e., $y = 0 \wedge x = 4 \models (x > 2, \varnothing)$.

Moreover, r_2 is also x-self-sufficient since $\bar{\exists}_{\mathit{Var} \setminus \{x\}} r_2 = (\mathit{true}, \varnothing) \rightarrowtail x = 4 \cdot (x > 2, \varnothing) \rightarrowtail x = 4 \cdot \boxtimes$, which is a self-sufficient trace.

3.2.1 Interpretations

Now we introduce the notion of interpretation, which is used to give meaning to process calls by associating to each process symbol a set of (maximal) conditional traces "modulo variance".

Definition 3.14 (Interpretations) *Let* $\mathbb{PC}_\Pi \coloneqq \{p(\vec{x}) \mid p \in \Pi,\ \vec{x}$ *are distinct variables*$\}$ *(or simply* $\mathbb{PC}$ *when clear from the context).*

Two functions $I, J \colon \mathbb{PC} \to \mathbb{M}$ *are* variants*, denoted by* $I \cong J$*, if for each* $\pi \in \mathbb{PC}$ *there exists a variable renaming* ρ *such that* $(I(\pi))\rho = J(\pi\rho)$*.*

An interpretation *is a function* $\mathcal{I} \colon \mathbb{PC} \to \mathbb{M}$ *modulo variance*[3]*.*

The semantic domain $\mathbb{I}_\Pi$ *(or simply* $\mathbb{I}$ *when clear from the context) is the set of all interpretations ordered by the pointwise extension of* $\sqsubseteq$ *(which by an abuse of notation we also denote by* $\sqsubseteq$*).*

The partial order on $\mathbb{I}$ formalizes the evolution of the computation process. $(\mathbb{I}, \sqsubseteq)$ is a complete lattice and its least upper bound and greatest lower bound are the pointwise extension of $\bigsqcup$ and $\bigsqcap$, respectively. In the sequel we abuse the notations of $\mathbb{M}$ for $\mathbb{I}$ as well. The bottom element is $\bot_{\mathbb{I}} \coloneqq \lambda\pi.\ \{\epsilon\}$.

Essentially, we define the semantics of each predicate in Π over formal parameters whose names are actually irrelevant. It is important to note that $\mathbb{PC}_\Pi$ (modulo variance) has the same cardinality of Π (and is thus finite) and therefore each interpretation is a finite collection (of possibly infinite elements). Hence, in the sequel, we explicitly write interpretations by cases, like

$$\mathcal{I} \coloneqq \begin{cases} \pi_1 \mapsto T_1 \\ \vdots \\ \pi_n \mapsto T_n \end{cases} \quad \text{representing} \quad \begin{array}{c} \mathcal{I}(\pi_1) \coloneqq T_1 \\ \vdots \\ \mathcal{I}(\pi_n) \coloneqq T_n \end{array}$$

In the following, any $\mathcal{I} \in \mathbb{I}$ is implicitly considered as an arbitrary function $\mathbb{PC} \to \mathbb{M}$ obtained by choosing an arbitrary representative of the elements of $\mathcal{I}$ generated by $\cong$. Actually, all the operators that we use on $\mathbb{I}_\Pi$ are also independent of the choice of the representative. Therefore, we can define any operator on $\mathbb{I}$ in terms of its counterpart defined on functions $\mathbb{PC} \to \mathbb{M}$.

Moreover, we also implicitly assume that the application of an interpretation $\mathcal{I}$ to a process call π, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative I of $\mathcal{I}$ which is defined exactly on π. For example, if $\mathcal{I} = (\lambda p(x,y).\ \{(true, \varnothing) \rightarrowtail x = y\})\big/_{\cong}$ then $\mathcal{I}(p(u,v)) = \{(true, \varnothing) \rightarrowtail u = v\}$.

3.2.2 Semantics Evaluation Function of Agents

We are finally ready to define the evaluation function of an agent A w.r.t. an interpretation $\mathcal{I}$, which computes the set of (maximal) conditional traces associated to the agent A. It is important to note that the computation does not depend on an initial store. Instead, the weakest (most general) condition for each agent is (computed and) accumulated in the conditional traces.

Definition 3.15 (Semantics Evaluation Function for Agents) *Given* $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$ *and* $\mathcal{I} \in \mathbb{I}_\Pi$*, we define the* semantics evaluation $\mathcal{A}[\![A]\!]_{\mathcal{I}} \in \mathbb{M}$ *by structural induction as follows.*

$$\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{I}} \coloneqq \{\boxtimes\} \tag{3.2}$$

$$\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}} \coloneqq \{(tt, \varnothing) \rightarrowtail c \cdot \boxtimes\} \tag{3.3}$$

$$\mathcal{A}[\![A \parallel B]\!]_{\mathcal{I}} \coloneqq \bigsqcup\big\{r_A \mathbin{\bar{\|}} r_B \mid r_A \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ r_B \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\big\} \tag{3.4}$$

$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} \coloneqq \bigsqcup\big\{\,\bar{\exists}_x\, r \mid r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ r \textit{ is } x\textit{-self-sufficient}\big\} \tag{3.5}$$

[3] i.e., a family of elements of $\mathbb{M}$ indexed by $\mathbb{PC}$ modulo variance.

$$\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}} \coloneqq (tt, \varnothing) \rightarrowtail tt \cdot \mathcal{I}(p(\vec{x}))^{4} \tag{3.6}$$

$$\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}} \coloneqq lfp_{\mathbb{M}}\, \lambda R.\Big(stutt(\{c_1, \ldots, c_n\}) \cdot R \sqcup \bigsqcup\big\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \,\big|\, 1 \leq i \leq n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}}\big\}\Big) \tag{3.7}$$

$\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}} \coloneqq$

$$\{(c, \varnothing) \rightarrowtail c \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\} \sqcup \tag{3.8a}$$

$$\bigsqcup\{(\eta^{+} \otimes c, \eta^{-}) \rightarrowtail d \otimes c \cdot (r{\downarrow_c}) \mid (\eta^{+}, \eta^{-}) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ d \otimes c \neq ff,\ \forall c^{-} \in \eta^{-}.\, \eta^{+} \otimes c \nvdash c^{-}\} \sqcup \tag{3.8b}$$

$$\bigsqcup\{(\eta^{+} \otimes c, \eta^{-}) \rightarrowtail ff \cdot \boxtimes \mid (\eta^{+}, \eta^{-}) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ d \otimes c = ff,\ \forall c^{-} \in \eta^{-}.\, \eta^{+} \otimes c \nvdash c^{-}\} \sqcup \tag{3.8c}$$

$$\bigsqcup\{(c, \eta^{-}) \rightarrowtail c \cdot (r{\downarrow_c}) \mid stutt(\eta^{-}) \cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ \forall c^{-} \in \eta^{-}.\, c \nvdash c^{-}\} \sqcup \tag{3.8d}$$

$$\bigsqcup\{(tt, \{c\}) \rightarrowtail tt \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \sqcup \tag{3.8e}$$

$$\bigsqcup\{(\eta^{+}, \eta^{-} \cup \{c\}) \rightarrowtail d \cdot r \mid (\eta^{+}, \eta^{-}) \rightarrowtail d \cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}},\ \eta^{+} \nvdash c\} \sqcup \tag{3.8f}$$

$$\bigsqcup\{(tt, \eta^{-} \cup \{c\}) \rightarrowtail tt \cdot r \mid stutt(\eta^{-}) \cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \tag{3.8g}$$

By $lfp(F)$ we denote the least fixed point of any monotonic function $F\colon \mathcal{L} \to \mathcal{L}$, over some lattice $\mathcal{L}$.

We now explain in detail each case of the definition.

(3.2) The semantics of the skip agent contains just the trace composed of the end-of-process construct that marks the end of the computation.

(3.3) For the $\mathsf{tell}(c)$ agent we have condition $(tt, \varnothing)$ since c must be added to the store in any case (in the next time instant). Next, the computation terminates (with the end-of-process symbol $\boxtimes$).

(3.4) The semantics for the parallel composition of two agents is defined in terms of the auxiliary operator $\overline{\|}$, explained in Definition 3.9.

(3.5) The hiding construct must hide the information about x from all traces that cannot be altered by the presence of external information about x, thus the hiding operation is applied just to x-self-sufficient conditional traces (Definition 3.12), that are those for which no additional information about variable x is needed (from other agents) in order to complete the computation.

(3.6) The semantics of process call $p(\vec{x})$ simply delays by one time instant the traces for $p(\vec{x})$ in interpretation $\mathcal{I}$ by prefixing them with $(tt, \varnothing) \rightarrowtail tt$.

(3.7) The semantics for the non-deterministic choice collects, for each guard c_i, a conditional trace of the form $(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}})$. This trace requires that c_i has to be satisfied by the current store (positive part of the condition in the first state). Then, the constraint c_i is propagated to the trace r (the continuation of the computation, which belongs to the semantics of A_i).

[4] Recall that by $s_1 \cdot S$ we denote $\{s_1 \cdot s_2 \mid s_2 \in S\}$.

Furthermore, we collect the stuttering traces, which correspond to the case when the computation suspends. Traces representing this situation are of the form $stutt(\{c_1, \ldots, c_n\}) \cdot r$ where r is, recursively, an element of the semantics of the choice agent.

(3.8) The definition for the conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ is in principle similar to the previous case. However, since the now construct must be instantaneous, in order to correctly model the timing of the agent we have seven cases depending on the possible forms of the first conditional state of the semantics of A (respectively B), on the value of the resulting store (ff or not) and on the fact that the guard c is satisfied or not in the current time instant.

(3.8a)–(3.8d) represent the case in which the guard c is satisfied by the current store. In this case, the agent now must behave instantaneously as A. For this reason, we distinguish four different cases corresponding to the possible form of conditional traces associated to A. In particular, (3.8a) corresponds to the case when the computation of A ends, thus also the computation of the conditional agent must end. In (3.8b), the information added (in one step) by A is compatible with the condition and with the rest of the computation and, moreover, does not produce ff when merged—by using $\otimes$—with the current store d. (3.8c) stops the conditional trace since the information produced by A added to the current store produces the inconsistent store ff. Finally, (3.8d) corresponds to the case when A suspends.

(3.8e)–(3.8g) consider the cases when c is not entailed by the current store. In this situation, the agent now must behave instantaneously as B, and the definition follows the same reasoning as for (3.8a), (3.8b) and (3.8d). The main difference is that, instead of adding c to the positive condition in the first conditional state, we add $\{c\}$ to the negative condition.

In the sequel, we use a standard notation for the iterates of the computation of the least fixpoint of a monotonic function $F{:}\mathcal{L} \to \mathcal{L}$, over lattice $\mathcal{L}$ whose bottom is $\bot$ and *lub* is $\bigsqcup$. Namely, $F{\uparrow}k$ denotes, for each $k \in \mathbb{N}$, $F^k(\bot)$ and $F{\uparrow}\omega$ denotes $\bigsqcup\{F^k(\bot) \mid k \in \mathbb{N}\}$. Recall that, for a continuos F, $lfp(F) = F{\uparrow}\omega$.

Example 3.16

Let us evaluate the semantics for the *tccp* agent $A_1 := A_2 \parallel A_3$ where

$A_2 := \mathsf{tell}(y = 2) \parallel \mathsf{tell}(x = y)$

$A_3 := \mathsf{ask}(true) \to \mathsf{now}\ (x = 0)\ \mathsf{then}\ \mathsf{tell}(z > 0)\ \mathsf{else}\ A_4$

$A_4 := \mathsf{ask}(y \geq 0) \to \mathsf{tell}(z \leq 0)$

Since there are no process calls, the interpretation $\mathcal{I}$ is irrelevant for the result. We start by computing the semantics for A_4, i.e., $\mathcal{A}[\![A_4]\!]_{\mathcal{I}} = lfp_{\mathbb{M}}(F)$ where

$$F(R) := \{r\} \sqcup stutt(\{y \geq 0\}) \cdot R \quad \text{and}$$
$$r := (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \cdot (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0 \cdot \boxtimes$$

The iterates of F are:

$$F{\uparrow}1 = F(\{\epsilon\}) = \{r,\ stutt(\{y \geq 0\})\}$$

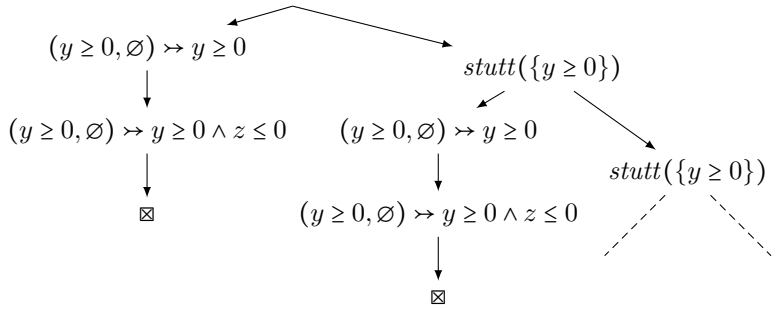

Figure 2: Tree representation of $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$ of Example 3.16.

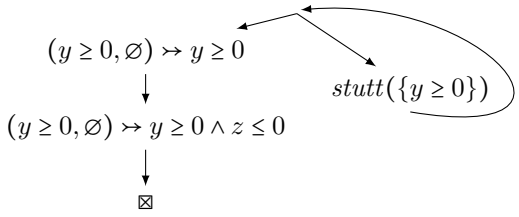

Figure 3: Graph representation of $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$ of Example 3.16.

$$F{\uparrow}2 = F(F{\uparrow}1) = \{r,\ stutt(\{y \geq 0\}) \cdot r,\ stutt(\{y \geq 0\}) \cdot stutt(\{y \geq 0\})\}$$
$$\vdots$$
$$lfp_{\mathbb{M}}(F) = \left\{ \left(stutt(\{y \geq 0\}) \right)^n \cdot r \,\middle|\, n \in \mathbb{N} \right\} \sqcup \{stutt(\{y \geq 0\}) \cdots stutt(\{y \geq 0\}) \cdots\}$$

Figure 2 graphically represents $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$, which consists of a trace for the case in which the guard is satisfied, and a set of traces for the case in which it suspends. As it can be observed, the tree in Figure 2 consists of an infinite replication of the same pattern. We can depict such infinite trees as finite graphs, as in Figure 3. The back-loop arc is just a graphical shortcut which represents the (infinite) tree that is obtained by unrolling the loop. It is important to note that nodes reached by a path of length 2 (via the back-loop arc) have to be considered as a single arc, thus corresponding just to a one time instant delay.

With the semantics of A_4, we compute $\mathcal{A}[\![A_3]\!]_{\mathcal{I}} = \{r_1, r_2\} \cup R$ where

$$r_1 := (true, \varnothing) \rightarrowtail true \cdot (x = 0, \varnothing) \rightarrowtail x = 0 \wedge z > 0 \cdot \boxtimes$$
$$r_2 := (true, \varnothing) \rightarrowtail true \cdot (y \geq 0, \{x = 0\}) \rightarrowtail y \geq 0 \cdot (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0 \cdot \boxtimes$$
$$R := (true, \varnothing) \rightarrowtail true \cdot (true, \{y \geq 0,\ x = 0\}) \rightarrowtail true \cdot \mathcal{A}[\![A_4]\!]_{\mathcal{I}}$$

All the traces of $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ start with the conditional store $(true, \varnothing) \rightarrowtail true$ corresponding to the ask agent with guard $true$. The trace r_1 corresponds to the case when (in the current time instant) the guard $x = 0$ is satisfied; the trace r_2 corresponds to $x = 0$ not satisfied and $y \geq 0$ satisfied; while we have R when none is satisfied and A_4 is executed.

Now we can compute the semantics for A_1 by parallel composition of $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ with $\mathcal{A}[\![A_2]\!]_{\mathcal{I}} = \{(true, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot \boxtimes\}$. The combination of the trace r_1 in $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ with the trace in $\mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ does not produce contributes since

the constraint $y = 2$, when propagated to the second component of r_1, is in contradiction with the positive part of the condition $(y = 2 \wedge x = y \wedge x = 0 \equiv \mathit{false})$. Indeed, $(\mathit{true}, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot ((x = 0, \varnothing) \rightarrowtail x = 0 \wedge z > 0 \cdot \boxtimes)\downarrow_{(y=2 \wedge x=y)} = (\mathit{true}, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot (\mathit{false}, \varnothing) \rightarrowtail \mathit{false} \cdot \boxtimes$ is not a trace since $(\mathit{false}, \varnothing)$ is not a valid condition.

The combination of the set of traces R (corresponding to the suspension of the agent A_4) and the $\mathsf{tell}(y = 2)$ agent also produces no trace. Definition 3.15 prescribes to compute $(\mathit{true}, \varnothing) \rightarrowtail y = 2 \wedge x = y \cdot \bigsqcup\{((\mathit{true}, \{y \geq 0, x = 0\}) \rightarrowtail \mathit{true} \cdot r')\downarrow_{(y=2 \wedge x=y)} \mid r' \in \mathcal{A}[\![A_4]\!]_{\mathcal{I}}\}$, which is empty, since $y = 2 \wedge x = y \not\gg (\mathit{true}, \{y \geq 0, x = 0\})$ because $y = 2 \wedge x = y \Rightarrow y \geq 0$. These traces would correspond to the suspension of the agent A_4, and this can happen only when $y \geq 0$ is not satisfied, but the first component of the parallel agent tells $y = 2$ (thus $y \geq 0$ is satisfied). Therefore, only the combination of the trace r_2 in $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ and the trace of $\mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ produces a trace. Namely

$$\mathcal{A}[\![A_1]\!]_{\mathcal{I}} = \{(\mathit{true}, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot (y = 2 \wedge x = y, \{x = 0\}) \rightarrowtail (y = 2 \wedge x = y) \cdot (y = 2 \wedge x = y, \varnothing) \rightarrowtail (y = 2 \wedge x = y \wedge z \leq 0) \cdot \boxtimes\}$$

Due to the partial nature of the constraint system, the combination of the hiding operator with non-determinism can make the language behavior non-monotonic. As already mentioned, this is the reason because for all non-monotonic languages of the *ccp* paradigm, the compositional semantics that have been written either avoid non-monotonic constructs or restrict their use. Let us show now that we are able to handle the following example, which is an adaptation to *tccp* of the one used in [13, 23] to illustrate the non-monotonicity problem.

Example 3.17

Consider the non-monotonic agent

$$A \coloneqq \mathsf{ask}(x = 1) \rightarrow \mathsf{tell}(\mathit{true}) + \mathsf{ask}(\mathit{true}) \rightarrow \mathsf{tell}(y = 2).$$

It is easy to see that for the initial store *true* just the second branch can be taken, whereas for the (greater) initial store $x = 1$, the two branches can be executed.

Since there are no process calls, for any interpretation $\mathcal{I}$, $\mathcal{A}[\![A]\!]_{\mathcal{I}} = \{r_1, r_2\}$, where

$$r_1 \coloneqq (x = 1, \varnothing) \rightarrowtail x = 1 \cdot (x = 1, \varnothing) \rightarrowtail x = 1 \cdot \boxtimes$$
$$r_2 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \varnothing) \rightarrowtail y = 2 \cdot \boxtimes$$

We have two possible traces depending on whether the initial store is strong enough to entail $x = 1$ or not.

[13, 23] show that within their semantics they do not collect all possible evaluations for agent $A' \coloneqq \mathsf{tell}(x = 1) \parallel \exists x\, A$. On the contrary, in our case, since

$$\bar{\exists}_{Var \setminus \{x\}}\, r_1 = (x = 1, \varnothing) \rightarrowtail x = 1 \cdot (x = 1, \varnothing) \rightarrowtail x = 1 \cdot \boxtimes$$
$$\bar{\exists}_{Var \setminus \{x\}}\, r_2 = (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot \boxtimes$$

only r_2 is x-self-sufficient and, by Definition 3.15,

$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} = \{(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \varnothing) \rightarrowtail y = 2 \cdot \boxtimes\}.$$

By composing we have

$$\mathcal{A}[\![A']\!]_{\mathcal{I}} = \{(true, \varnothing) \rightarrowtail x = 1 \cdot (x = 1, \varnothing) \rightarrowtail y = 2 \wedge x = 1 \cdot \boxtimes\}.$$

It is easy to see that the information on the variable x added by the tell agent does not affect the *internal* execution of the agent A, as expected.

There are some technical decisions that ensure the correctness of the defined semantics. One can note that in the definition of the propagation operator (Definition 3.7), the propagated information is added not only to the store of the state, but also to the (positive part of the) condition. This means that the positive part of the conditions in a trace contains not only the information that had to be satisfied up to that computation step, but also the constraints that have been added during computation in the previous time instants. From the computations in the examples above, it may seem that the propagation of the accumulated information in the conditions of the states could be redundant. However, it is necessary in order to have full abstraction w.r.t. the behavior, otherwise we would distinguish agents whose behavior is actually the same, as shown in the following example.

Example 3.18

Consider the following two (very similar) agents:

$$A_1 \coloneqq \mathsf{ask}(x > 2) \rightarrow \mathsf{tell}(y = 1) \qquad A_2 \coloneqq \mathsf{ask}(x > 4) \rightarrow \mathsf{tell}(y = 1)$$

We have similar but different semantics. Namely,

$$\begin{aligned}
\mathcal{A}[\![A_1]\!]_{\mathcal{I}} &= \{\big(\, stutt(\{x > 2\})\big)^n \cdot r_1 \,|\, n \in \mathbb{N}\} \sqcup \{stutt(\{x > 2\}) \cdots stutt(\{x > 2\}) \cdots\} \\
r_1 &= (x > 2, \varnothing) \rightarrowtail true \cdot (x > 2, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes \\
\mathcal{A}[\![A_2]\!]_{\mathcal{I}} &= \{\big(\, stutt(\{x > 4\})\big)^n \cdot r_2 \,|\, n \in \mathbb{N}\} \sqcup \{stutt(\{x > 4\}) \cdots stutt(\{x > 4\}) \cdots\} \\
r_2 &= (x > 4, \varnothing) \rightarrowtail true \cdot (x > 4, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes
\end{aligned}$$

However, consider now the following two agents, which embed A_1 and A_2 in the same context:

$$A_1' \coloneqq \mathsf{tell}(x = 7) \parallel \mathsf{ask}(true) \rightarrow A_1 \qquad A_2' \coloneqq \mathsf{tell}(x = 7) \parallel \mathsf{ask}(true) \rightarrow A_2$$

Then, the two traces corresponding to the satisfaction of the guards are, respectively:

$$r_3 = (true, \varnothing) \rightarrowtail x = 7 \cdot r_1{\downarrow_{(x=7)}} \qquad r_4 = (true, \varnothing) \rightarrowtail x = 7 \cdot r_2{\downarrow_{(x=7)}}$$

Since the propagated constraint is stronger than the guards in both the agents, the resulting compositions are the same. In fact, thanks to the accumulation of the store in the condition, we do not distinguish them:

$$r_1{\downarrow_{(x=7)}} = r_2{\downarrow_{(x=7)}} = (true, \varnothing) \rightarrowtail x = 7 \cdot (x = 7, \varnothing) \rightarrowtail x = 7 \cdot (x = 7, \varnothing) \rightarrowtail x = 7 \wedge y = 1 \cdot \boxtimes$$

If the constraint $x = 7$ were not added to the condition, but only to the store of the state, then we would have two different conditional traces for these two agents.

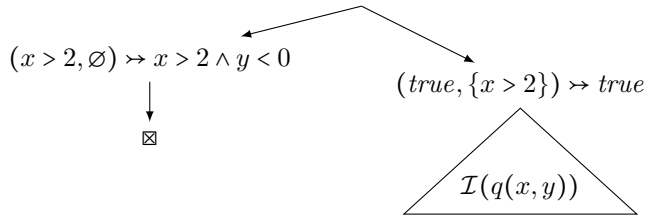

Figure 4: Tree representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ of Example 3.20.

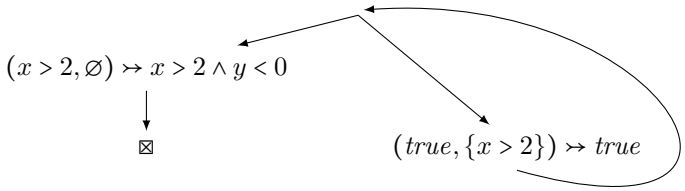

Figure 5: Graph representation of the fixpoint $\mathcal{F}[\![D]\!](q(x, y))$ of Example 3.20.

3.2.3 Fixpoint Denotations of Process Declarations

Now we can finally define the semantics for a set of process declarations D.

Definition 3.19 (Fixpoint semantics) *Given $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$, we define $\mathcal{D}[\![D]\!] : \mathbb{I} \to \mathbb{I}$, for each $p \in \Pi$, as*

$$\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) := \bigsqcup \{ \mathcal{A}[\![A]\!]_{\mathcal{I}} \mid p(\vec{x}) :- A \in D \}.$$

The fixpoint denotation *of D is $\mathcal{F}[\![D]\!] := lfp(\mathcal{D}[\![D]\!]) = \mathcal{D}[\![D]\!]\uparrow\omega$.*

We denote with $\approx_{\mathcal{F}}$ the equivalence relation on $\mathbb{D}^{\Pi}_{\mathbf{C}}$ induced by $\mathcal{F}$. Namely, $D_1 \approx_{\mathcal{F}} D_2 \iff \mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$.

The semantics of a tccp program $D \,.\, A$ is $\mathcal{P}[\![D \,.\, A]\!] := \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$.

$\mathcal{F}[\![D]\!]$ is well defined since $\mathcal{D}[\![D]\!]$ is continuous (as stated formally in Lemma A.5).

Let us show how the semantics for a set of process declarations is computed by means of some examples.

Example 3.20

Let $D := \{q(x, y) :- A\}$ where

$$A := \mathsf{now}\ (x > 2)\ \mathsf{then}\ \mathsf{tell}(y < 0)\ \mathsf{else}\ q(x, y).$$

First we need to compute, for each $\mathcal{I} \in \mathbb{I}$, the evaluation of the body of the process declaration. Namely,

$$\mathcal{A}[\![A]\!]_{\mathcal{I}} = \{\bar{r}\} \sqcup \{(true, \{x > 2\}) \rightarrowtail true \cdot s \mid s \in \mathcal{I}(q(x, y))\}$$

where $\bar{r} := (x > 2, \varnothing) \rightarrowtail x > 2 \wedge y < 0 \cdot \boxtimes$. Intuitively, the trace $\bar{r}$ corresponds to the then branch of the conditional agent, whereas the else branch is represented by a set of traces, one for each trace in the interpretation of the process call. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ is graphically represented in Figure 4.

The iterates of $\mathcal{D}[\![D]\!]$ are

$$\mathcal{D}[\![D]\!]\uparrow 1 = \left\{ q(x, y) \mapsto \{\bar{r},\ (true, \{x > 2\}) \rightarrowtail true \right\}$$

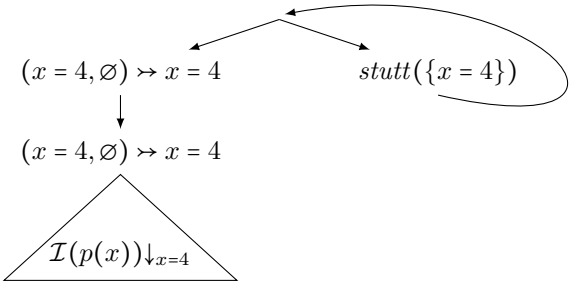

Figure 6: Graph representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ of Example 3.21.

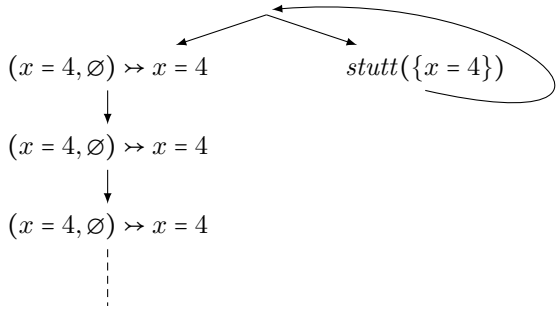

Figure 7: Graph representation of the fixpoint $\mathcal{F}[\![D]\!](p(x))$ for the Example 3.21.

$$\mathcal{D}[\![D]\!]\uparrow 2 = \begin{cases} q(x,y) \mapsto & \{\bar{r},\ (true, \{x > 2\}) \rightarrowtail true \cdot \bar{r}, \\ & (true, \{x > 2\}) \rightarrowtail true \cdot (true, \{x > 2\}) \rightarrowtail true\} \end{cases}$$

$$\vdots$$

$$\mathcal{D}[\![D]\!]\uparrow \omega = \begin{cases} q(x,y) \mapsto & \left\{\left((true, \{x > 2\}) \rightarrowtail true\right)^n \cdot \bar{r} \,\middle|\, n \in \mathbb{N}\right\} \\ & \sqcup\{(true, \{x > 2\}) \rightarrowtail true \cdots (true, \{x > 2\}) \rightarrowtail true \cdots\} \end{cases}$$

The limit $\mathcal{F}[\![D]\!](q(x,y)) = (\mathcal{D}[\![D]\!]\uparrow\omega)(q(x,y))$ is graphically represented in Figure 5.

Example 3.21

Let $D \coloneqq \{p(x) \coloneq A\}$ where $A \coloneqq \mathsf{ask}(x = 4) \rightarrow p(x)$. First we need to compute, for each $\mathcal{I} \in \mathbb{I}$, the evaluation of the body of the process declaration. Namely,

$$\mathcal{A}[\![A]\!]_{\mathcal{I}} = \left\{\left(\, stutt(\{x = 4\})\right)^n \cdot \bar{r} \cdot s \,\middle|\, n \in \mathbb{N},\ s \in \mathcal{I}(p(x))\right\} \sqcup \{stutt(\{x = 4\}) \cdots stutt(\{x = 4\}) \cdots\}$$

where $\bar{r} \coloneqq (x = 4, \varnothing) \rightarrowtail x = 4 \cdot (x = 4, \varnothing) \rightarrowtail x = 4$. It is worth noticing that the second conditional state of $\bar{r}$ corresponds to the delay that is introduced each time that a process call is run. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ is graphically represented in Figure 6.

The iterates of $\mathcal{D}[\![D]\!]$ are

$$\mathcal{D}[\![D]\!]\uparrow 1 = \begin{cases} p(x) \mapsto & \{(stutt(\{x = 4\}))^n \cdot \bar{r} \,|\, n \in \mathbb{N}\} \sqcup \\ & \{stutt(\{x = 4\}) \cdots stutt(\{x = 4\}) \cdots\} \end{cases}$$

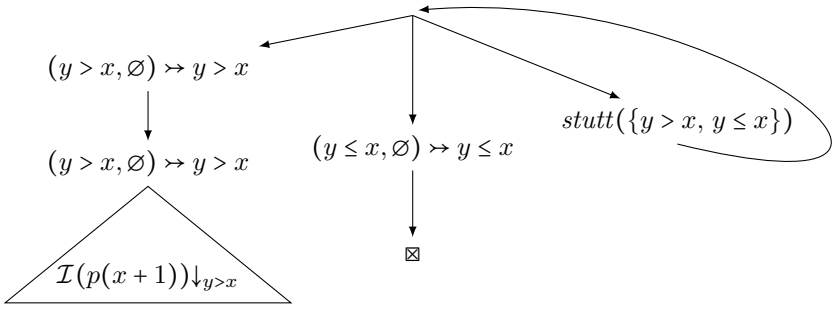

Figure 8: Graph representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ in Example 3.22.

$$\mathcal{D}[\![D]\!]\uparrow 2 = \begin{cases} p(x) \mapsto & \{(stutt(\{x = 4\}))^n \cdot \bar{r} \cdot \bar{r} \mid n \in \mathbb{N}\} \sqcup \\ & \{stutt(\{x = 4\}) \cdots stutt(\{x = 4\}) \cdots\} \end{cases}$$

$$\vdots$$

$$\mathcal{F}[\![D]\!] = \begin{cases} p(x) \mapsto & \{(stutt(\{x = 4\}))^n \cdot \bar{r} \cdots \bar{r} \cdots \mid n \in \mathbb{N}\} \sqcup \\ & \{stutt(\{x = 4\}) \cdots stutt(\{x = 4\}) \cdots\} \end{cases}$$

$\mathcal{F}[\![D]\!](p(x))$ is graphically represented in Figure 7. Note that the application of the propagation operator to the previous iterates removes all the stuttering sequences, and this is the reason because just the first stuttering sequence remains.

Example 3.22

Let $D := \{p(x, y) :- A\}$ where

$$A := \mathsf{ask}(y > x) \rightarrow p(x + 1, y) + \mathsf{ask}(y \leq x) \rightarrow \mathsf{skip}$$

As usually done in the *tccp* community, we assume that we can use expressions of the form $x + 1$ directly in the arguments of a process call. We can simulate this behavior by writing $\exists x' \, (\mathsf{tell}(x' = x + 1) \parallel p(x', y))$ instead of $p(x+1, y)$ (but introducing a delay of one time unit). We have

$$\begin{aligned} \mathcal{A}[\![A]\!]_{\mathcal{I}} = & \{(y > x, \varnothing) \rightarrowtail y > x \cdot (y > x, \varnothing) \rightarrowtail y > x \cdot r\downarrow_{y>x} \mid r \in \mathcal{I}(p(x + 1, y))\} \sqcup \\ & \{(y \leq x, \varnothing) \rightarrowtail y \leq x \cdot \boxtimes\} \sqcup \\ & \{(stutt(\{y > x,\, y \leq x\}))^n \cdot (y > x, \varnothing) \rightarrowtail y > x \cdot \\ & \quad (y > x, \varnothing) \rightarrowtail y > x \cdot r\downarrow_{y>x} \mid n \in \mathbb{N},\ r \in \mathcal{I}(p(x + 1, y))\} \sqcup \\ & \{(stutt(\{y > x,\, y \leq x\}))^n \cdot (y \leq x, \varnothing) \rightarrowtail y \leq x \cdot \boxtimes \mid n \in \mathbb{N}\} \sqcup \\ & \{stutt(\{y > x,\, y \leq x\}) \cdots stutt(\{y > x,\, y \leq x\}) \cdots\} \end{aligned}$$

which is graphically shown in Figure 8. For this agent, we have three branches, one for each condition of the choice and one corresponding to the stuttering possibility.

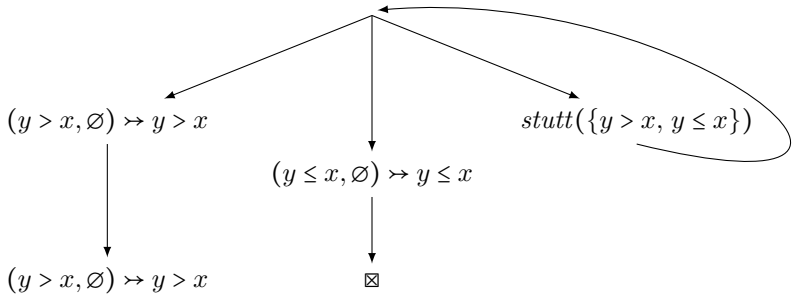

Figure 9: Graph representation of $\mathcal{D}[\![D]\!]\uparrow 1(p(x,y))$ in Example 3.22.

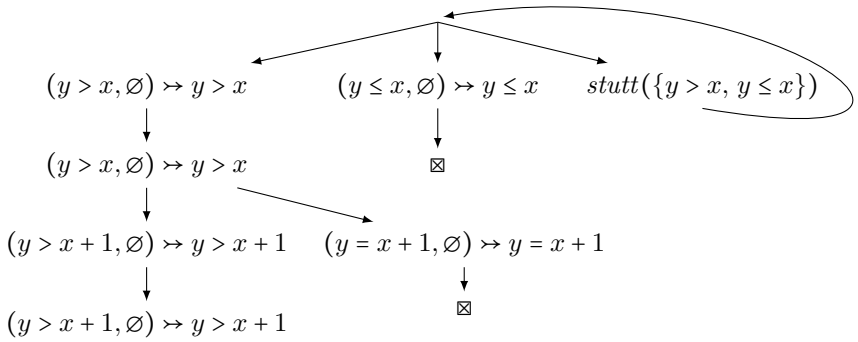

Figure 10: Graph representation of $\mathcal{D}[\![D]\!]\uparrow 2(p(x,y))$ in Example 3.22.

The first iteration of $\mathcal{D}[\![D]\!]$ is

$$\mathcal{D}[\![D]\!]\uparrow 1 = \begin{cases} p(x,y) \mapsto & \{(y > x, \varnothing) \rightarrowtail y > x \cdot (y > x, \varnothing) \rightarrowtail y > x\} \sqcup \\ & \{(y \leq x, \varnothing) \rightarrowtail y \leq x \cdot \boxtimes\} \sqcup \\ & \{(stutt(\{y > x,\, y \leq x\}))^n \cdot (y > x, \varnothing) \rightarrowtail y > x \cdot \\ & \quad (y > x, \varnothing) \rightarrowtail y > x \;\mid\; n \in \mathbb{N}\} \sqcup \\ & \{(stutt(\{y > x,\, y \leq x\}))^n \cdot (y \leq x, \varnothing) \rightarrowtail y \leq x \cdot \boxtimes \mid n \in \mathbb{N}\} \sqcup \\ & \{stutt(\{y > x,\, y \leq x\}) \cdots stutt(\{y > x,\, y \leq x\}) \cdots\} \end{cases}$$

which is graphically represented in Figure 9. Figure 10 represents the second iteration $\mathcal{D}[\![D]\!]\uparrow 2(p(x,y))$, whereas Figure 11 is the graphical representation of $\mathcal{F}[\![D]\!](p(x,y))$. By looking at the semantics, it can be observed that the process stops in one time instant when $y \leq x$ and in $1 + 2(y - x)$ time instants otherwise.

Example 3.23

Consider the following process declaration, presented in [19], which models a subsystem of a microwave controller. The underlying constraint system is the

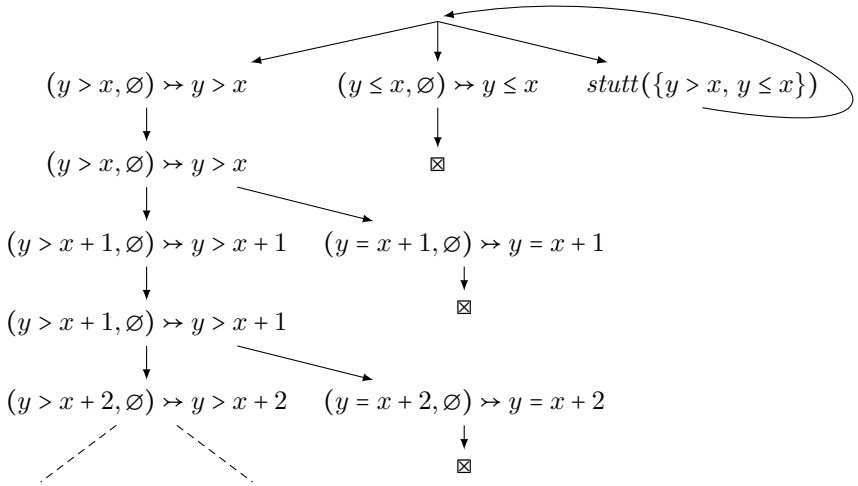

Figure 11: Graph representation of $\mathcal{F}[\![D]\!](p(x,y))$ in Example 3.22.

(well-known) Herbrand constraint system [12].

$$
\begin{array}{l}
\mathit{microwave}(\mathit{Door},\ \mathit{Button},\ \mathit{Error}) \text{ :- } \exists D\, \exists B\, \exists E \\
\quad \big(\mathsf{tell}(\mathit{Error} = [\_ \mid E]) \parallel \mathsf{tell}(\mathit{Door} = [\_ \mid D]) \parallel \mathsf{tell}(\mathit{Button} = [\_ \mid B]) \\
\qquad \parallel \mathsf{now}(\mathit{Door} = [\mathit{open} \mid D] \wedge \mathit{Button} = [\mathit{on} \mid B]) \\
\qquad\quad \mathsf{then}\ \big(\exists E1\, \mathsf{tell}(E = [1 \mid E1]) \parallel \exists B1\, \mathsf{tell}(B = [\mathit{off} \mid B1]) \big) \\
\qquad\quad \mathsf{else}\ \exists E1\, \mathsf{tell}(E = [0 \mid E1]) \\
\qquad \parallel \mathit{microwave}(D, B, E) \big)
\end{array}
$$

This process declaration detects if the door is open while the microwave is turned on. In that case, it forces that in the next time instant the microwave is turned-off and it emits an error signal (value 1); otherwise, the agent emits a signal of no error (value 0). Due to the monotonicity of the store, streams are used to model *imperative-style* variables [14]. In the example, the streams *Error*, *Door* and *Button* store the values that the simulated *modifiable variables* get along the computation. The first three tell agents link the future values of the streams with the *future streams* E, D and B. Then, when it is detected a possible *risk* (characterized by the guard of the now agent), the microwave is turned off and an *error* signal is emitted (by the then branch of the conditional agent). The final recursive call restarts the same control at the next time instant.

The fixpoint semantics $\mathcal{F}(\mathit{microwave}(D, B, E))$ is graphically represented in Figure 12, where:

$$
\mathit{risk}_k := \exists_D\, \exists_B (\mathit{Door} = [\underbrace{\mathit{open} \mid \ldots}_{k \text{ times}} \mid D] \wedge \mathit{Button} = [\mathit{on} \mid \underbrace{\mathit{off} \mid \mathit{on} \mid \ldots}_{k-1 \text{ times}} \mid B])
$$

$$
\begin{array}{l}
\mathit{state}_{b_1 \ldots b_n} := \exists_{E1}\, \exists_D\, \exists_{B1} (\mathit{Error} = [\_ \mid b_1 \mid \ldots \mid b_n \mid E1] \wedge \mathit{Door} = [\_ \mid D] \wedge \\
\qquad\qquad \mathit{Button} = [\_ \mid \underbrace{\mathit{on} \mid \mathit{off} \mid \ldots}_{\Sigma_{i=1}^{n} b_i \text{ times}} \mid B1])
\end{array}
$$

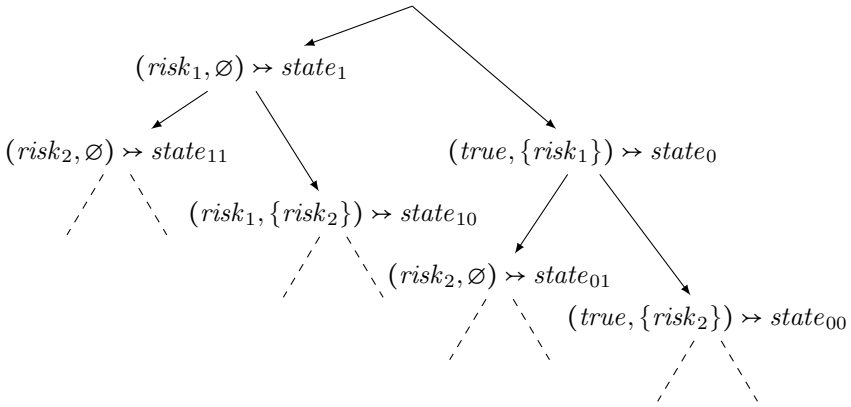

Figure 12: Tree representation of $\mathcal{F}[\![D]\!](microwave(D, B, E))$ in Example 3.23.

We have coded the indices of stores in the conditional states with a binary number in order to make the figure more readable. It is worth noticing that the stores labeled with $state_b$ where the last digit of b is 1 correspond to states where an error is emitted.

All the conditional sequences in the semantics of this process are infinite sequences. This is consistent with the fact that we are modeling a process that is intended to be active forever (checking whether the risky situation holds). It is worth noticing that this kind of processes can be handled only if the semantics is able to capture infinite computations, which is one of the main features of our proposal.

3.2.4 Full abstraction of $\mathcal{F}$ semantics

This subsection is dedicated to formally prove that our semantics $\mathcal{F}$ is (correct and) fully abstract w.r.t. the small-step operational behavior. To formally link hypothetical computations with real ones, we first need to define an auxiliary operator which, taken an initial store c, instantiates the hypothetical states of a conditional trace r producing the corresponding (real) behavioral timed trace. Intuitively, this operator works by consistently adding to each conditional state the information given by the initial store c, discarding those sequences which falsify conditions.

Definition 3.24 (Instantiation operator) *The* instantiation *operator* $\Downarrow : \mathbf{M} \times \mathbf{C} \to \mathbf{C}^*$ *is a partial function defined by structural induction as:* $\epsilon\Downarrow_c := \epsilon$*; otherwise* $r\Downarrow_{ff} := ff$*; otherwise* $\boxtimes\Downarrow_c := c$*, otherwise*

$$
\begin{array}{ll}
(stutt(\eta^-) \cdot r')\Downarrow_c := c & \text{if } \forall c^- \in \eta^- . \, c \nvDash c^- \\
\quad (\eta \rightarrowtail d \cdot r')\Downarrow_c := c \cdot (r'\Downarrow_{c \otimes d}) & \text{if } c \vDash \eta \text{ and } (c \otimes d) \neq ff
\end{array}
$$

We abuse notation by denoting with $R\Downarrow_c$ *the extension of* $\Downarrow_c$ *to* $\mathbb{M}$*:* $R\Downarrow_c := \{r\Downarrow_c \mid r \in R \text{ and } r\Downarrow_c \text{ is defined}\}$.

The instantiation operator is consistent w.r.t. the propagation operator (Definition 3.7), in the sense that, for any c' that entails c, $r\Downarrow_c = (r\downarrow_{c'})\Downarrow_c$ (as stated formally in Lemma A.6). Moreover, the instantiation operator $\Downarrow$ "distributes"

over the parallel composition operator $\bar{\|}$ (Definition 3.9) (as stated formally in Lemma A.8).

The key result to prove correctness of $\mathcal{F}$ w.r.t. $\approx_{ss}$ is the following theorem which shows that the small-step behavior of a program P can be determined by instantiation of the semantics $\mathcal{P}[\![P]\!]$.

Theorem 3.25 *For each program P and each $c \in \mathbf{C}$, $prefix(\mathcal{P}[\![P]\!]\Downarrow_c) = \mathcal{B}^{ss}[\![P]\!]_c$.*

The following theorem is the key result to prove full abstraction of $\mathcal{F}$ w.r.t. $\approx_{ss}$.

Theorem 3.26 *Let P_1, P_2 be two programs. Then $\mathcal{P}[\![P_1]\!] = \mathcal{P}[\![P_2]\!]$ if and only if $\mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!]$.*

Proposition 3.27 *Let D_1, $D_2 \in \mathbb{D}^{\Pi}_{\mathbf{C}}$. Then $D_1 \approx_{\mathcal{F}} D_2$ if and only if $\forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}}. \mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{P}[\![D_2 \,.\, A]\!]$.*

Correctness and full abstraction is a direct consequence of Theorems 3.25 and 3.26 and Proposition 3.27.

Corollary 3.28 (Correctness and full abstraction of $\mathcal{F}$) *Let D_1, $D_2 \in \mathbb{D}^{\Pi}_{\mathbf{C}}$. Then $D_1 \approx_{ss} D_2$ if and only if $D_1 \approx_{\mathcal{F}} D_2$.*

4 Big-step semantics

A small-step behavior contains all the details of the computation. However typically only some parts of the execution are considered relevant. So frequently is better to reason only about a specific abstraction of the small-step behavior, instead of dealing with all execution details. In the literature, many authors (like [14]) call *observables* all the abstractions of the small-step behavior of a specific program[5] (including the small-step behavior itself as the degenerate identity abstraction). Moreover, they typically use this same name for the collection of all observables of a set of declarations.

Many other authors use the term *observable property* (or simply observable) for an abstraction function ϕ which, when applied to the set of traces of a program, delivers the observations of interest. Then the *observation*, or *observable behavior*, of program P is just the application of ϕ to the traces of P.

We prefer to use the latter nomenclature and, in the sequel, we call *observable behavior of a program Q w.r.t. observable ϕ* (or simply *ϕ-observable behavior of Q*) the image $\phi(\mathcal{B}^{ss}[\![Q]\!])$ and we denote it by $\mathcal{B}^{\phi}[\![Q]\!]$.

The observable property which is usually considered in papers dealing with semantics of *ccp* languages (e.g. see [15]) is the one that collects the input/output pairs of terminating computations, including deadlocked ones. Indeed, using the (original version of the) transition system of Definition 2.1, [14] defines the notion of *input-output observable behavior* as $\mathcal{O}^{io}(A) := \{\langle c_0, c_n\rangle \,|\, \langle A_0, c_0\rangle \rightarrow^* \langle A_n, c_n\rangle \not\rightarrow\}$. In this definition, there is an implicit reference to a set of declarations D. Since in the sequel we need to state some formal results for two (different) sets of declarations simultaneously, we use the explicit notation $\mathcal{O}^{io}[\![D \,.\, A]\!]$ instead of $\mathcal{O}^{io}(A)$.

[5] Notice that a *tccp* program is the syntactic correspondent of a (program's) expression of a generic language, while a *tccp* set of declarations is the syntactic correspondent of a program.

As we already mentioned, in *tccp* also infinite computations *must* be considered, for example when we are modeling reactive systems. However, we nevertheless want to be able to distinguish if an input-output pair refers to a finite or infinite computation. Thus, we use *input-output pairs with associated termination mode* of the form $\langle c_0, mode(c_n)\rangle$, where $c_0 \in \mathbf{C}$ is the input store of the computation, $c_n \in \mathbf{C}$ is the output store (which is the *lub* of the stores of the computation) and *mode* is either *fin* or *inf* for finite or infinite computations, respectively.

Definition 4.1 *Given $c, c' \in \mathbf{C}$ such that $c' \vdash c$, an* input-output pair with termination mode *is either $\langle c, fin(c')\rangle$ or $\langle c, inf(c')\rangle$.*

We denote by **IO** *the set of input-output pairs with termination mode and by $\mathbb{IO}$ the domain $\wp(\mathbf{IO})$, ordered by set inclusion.*

Clearly, $(\mathbb{IO}, \subseteq, \bigcup, \bigcap, \mathbf{IO}, \varnothing)$ is a complete lattice.

Definition 4.2 (Input-output behavior of programs) *The* input-output observable *is defined as*

$$io(T) := \big\{\langle c_0, fin(c_n)\rangle \,\big|\, c_0 \cdots c_n \in T\big\} \cup \big\{\langle c_0, inf(\otimes_{i\geq 0} c_i)\rangle \,\big|\, c_0 \cdots c_n \cdots \in T\big\}.$$

For each $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$, the induced input-output behavior $\mathcal{B}^{io}[\![D\,.\,A]\!]$ *is defined as $io(\mathcal{B}^{ss}[\![D\,.\,A]\!])$. We denote by $\approx_{io}$ the equivalence relation between process declarations induced by $\mathcal{B}^{io}$, namely $D_1 \approx_{io} D_2 \iff \forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}}.\ \mathcal{B}^{io}[\![D_1\,.\,A]\!] = \mathcal{B}^{io}[\![D_2\,.\,A]\!]$.*

We denote by π_F the projection which selects just the pairs whose mode is fin and by $\mathbb{IO}_F$ we denote $\pi_F(\mathbb{IO})$. Moreover, we denote by $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$ the finite fragment of $\mathcal{B}^{io}[\![D\,.\,A]\!]$ i.e., $\pi_F(\mathcal{B}^{io}[\![D\,.\,A]\!])$.

Note that, by Definitions 3.1 and 4.2,

$$\begin{aligned}\mathcal{B}^{io}[\![D\,.\,A_0]\!] = &\big\{\langle c_0, fin(c_n)\rangle \,\big|\, c_0 \in \mathbf{C},\ \langle A_0, c_0\rangle \to^* \langle A_n, c_n\rangle \not\to\big\} \cup \\ &\big\{\langle c_0, inf(\otimes_{i\geq 0} c_i)\rangle \,\big|\, c_0 \in \mathbf{C},\ \langle A_0, c_0\rangle \to \cdots \to \langle A_i, c_i\rangle \to \cdots\big\}\end{aligned}$$

In the sequel, we define an abstract interpretation ([11]) of the small-step semantics $\mathcal{P}[\![D\,.\,A]\!]$ (Definition 3.19) which gives $\mathcal{B}^{io}[\![D\,.\,A]\!]$. Then we prove that the finite fragment of this abstraction (i.e., $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$) is essentially isomorphic to $\mathcal{O}^{io}[\![D\,.\,A]\!]$. Actually, there is a negligible difference between $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$ and $\mathcal{O}^{io}[\![D\,.\,A]\!]$ due to the change we made in the definition of the small-step operational semantics. We will state the formal result in Subsection 4.2.

To define the semantics modeling the input-output observable as suggested by the abstract interpretation approach, we proceed as described in the following. We assume familiarity with basic results of abstract interpretation. However, this familiarity is needed only to be confident about the soundness of the results that we obtain by standard results of abstract interpretation theory. To understand the definitions it is sufficient to know that, given two complete lattices $(\mathbb{C}, \sqsubseteq)$ and $(\mathbb{A}, \leq)$, a Galois Insertion of $(\mathbb{A}, \leq)$ into $(\mathbb{C}, \sqsubseteq)$—denoted by $(\mathbb{C}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\rightarrow}} (\mathbb{A}, \leq)$—is a pair of maps $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ that satisfy certain properties (see [11]). $(\mathbb{C}, \sqsubseteq)$ and $(\mathbb{A}, \leq)$ are the *concrete* and *abstract* domains, while α and γ are the *abstraction* and *concretization* maps.

First, we formalize program properties of interest (in this particular case the input-output behavior) as a Galois Insertion $(\mathbb{M}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\rightarrow}} (\mathbb{IO}, \subseteq)$ and then

we lift it over interpretations $\mathbb{I} \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} [\mathbb{PC} \to \mathbb{IO}]$ by function composition as $\dot{\alpha}(f) = \alpha \circ f$. The best correct (optimal) abstract version of the semantics $\mathcal{D}[\![D]\!]$ is simply obtained as $\mathcal{D}^{\alpha}[\![D]\!] := \dot{\alpha} \circ \mathcal{D}[\![D]\!] \circ \dot{\gamma}$. Abstract interpretation theory assures that $\mathcal{F}^{\alpha}[\![D]\!] := \mathit{lfp}(\mathcal{D}^{\alpha}[\![D]\!])$ is the best correct approximation of $\mathcal{F}[\![D]\!]$. Correct because $\alpha(\mathcal{F}[\![D]\!]) \subseteq \mathcal{F}^{\alpha}[\![D]\!]$ and best because it is the minimum (w.r.t. $\subseteq$) of all correct approximations.

4.1 Input-output semantics with infinite outcomes

Now we formally define the Galois Insertion which abstracts conditional traces to input-output pairs with termination mode. In the sequel, we denote by $\mathit{last}(s)$ the partial function that, for a non-empty finite sequence s, gives its last element and is otherwise undefined.

Definition 4.3 (Input-Output abstraction) *Given any $M \in \mathbb{M}$, we define*

$$\begin{aligned} \alpha_{io}(M) := {} & \{\langle c_0, \mathit{fin}(c_n)\rangle \mid c_0 \in \mathbf{C},\, r \in M,\, \mathit{last}(r{\Downarrow_{c_0}}) = c_n\} \cup \\ & \{\langle c_0, \mathit{inf}(\otimes_{i\geq 0} c_i)\rangle \mid c_0 \in \mathbf{C},\, r \in M,\, r{\Downarrow_{c_0}} = c_0 \dots c_i \dots\} \end{aligned} \tag{4.1}$$

$$\begin{aligned} \gamma_{io}(P) := {} & \bigsqcup\{r \in \mathbf{M} \mid \langle c_0, \mathit{fin}(c_n)\rangle \in P,\, \mathit{last}(r{\Downarrow_{c_0}}) = c_n\} \sqcup \\ & \bigsqcup\{r \in \mathbf{M} \mid \langle c_0, \mathit{inf}(c)\rangle \in P,\, r{\Downarrow_{c_0}} = c_0 \dots c_i \dots,\, c = \otimes_{i\geq 0} c_i\} \end{aligned} \tag{4.2}$$

We abuse notation and denote with the same symbols the lifting to interpretations, i.e., $\alpha_{io}(\mathcal{I}) := \alpha_{io} \circ \mathcal{I}$, $\gamma_{io}(\mathcal{I}^{\alpha}) := \gamma_{io} \circ \mathcal{I}^{\alpha}$.

$(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \underset{\alpha_{io}}{\overset{\gamma_{io}}{\leftrightarrows}} (\mathbb{IO}, \subseteq, \bigcup, \bigcap, \mathbf{IO}, \varnothing)$ is a Galois insertion (as stated formally in Lemma A.9).

The input-output behavior of a program is indeed obtainable by abstraction of its (concrete) semantics.

Proposition 4.4 *Let $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$. Then, $\alpha_{io}(\mathcal{P}[\![D \,.\, A]\!]) = \mathcal{B}^{io}[\![D \,.\, A]\!]$.*

Now (as anticipated), following the (classical) abstract interpretation approach, we define the optimal abstract version of $\mathcal{D}$ as $\mathcal{D}^{io} := \alpha_{io} \circ \mathcal{D} \circ \gamma_{io}$,[6] and thus the best (possible) correct approximation w.r.t. α_{io} of the semantic function $\mathcal{F}$ is the least fixpoint of $\mathcal{D}^{io}$, i.e., $\mathcal{F}^{io}[\![D]\!] := \mathit{lfp}(\mathcal{D}^{io}[\![D]\!])$. Unfortunately, $\mathcal{F}^{io}[\![D]\!]$ turns out to be very imprecise, mainly because the information contained in the input-output pairs is not enough to keep the synchronization between parallel processes. Indeed, the declarations equivalence induced by $\mathcal{F}^{io}$ is *not correct* w.r.t. $\approx_{io}$ (Definition 4.2), since we can have two programs with the same $\mathcal{F}^{io}$ that have different $\mathcal{B}^{io}$, as shown by the following example.

Example 4.5

Consider the two sets of declarations $D_1 := \{d_1, d_2\}$ and $D_2 := \{d_1, d_3\}$ where

$d_1 := p(x, y) :- q(x) \parallel \mathsf{ask}(\mathit{true}) \to \mathsf{now}\ x = 2\ \mathsf{then}\ \mathsf{tell}(y = 0)\ \mathsf{else}\ \mathsf{tell}(y = 1)$

$d_2 := q(x) :- \mathsf{tell}(x = 2)$

$d_3 := q(x) :- \mathsf{ask}(\mathit{true}) \to \mathsf{tell}(x = 2)$

[6] Although possible, a direct (expanded) definition of $\mathcal{D}^{io}$ is not relevant for our present purposes.

Clearly, D_2 differs from D_1 just because of the delay in adding the constraint $x = 2$ to the store. This difference shows up in the input-output behavior of $p(x, y)$. Indeed,

$$\begin{aligned}\alpha_{io}(\mathcal{P}[\![D_1 \,.\, p(x,y)]\!]) &= \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 0)\rangle \mid c \in \mathbf{L}\}\\ \alpha_{io}(\mathcal{P}[\![D_2 \,.\, p(x,y)]\!]) &= \{\langle c, \mathit{fin}(c \wedge y = 0)\rangle \mid c \in \mathbf{L},\, c \Rightarrow x = 2\} \cup\\ &\quad\ \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 1)\rangle \mid c \in \mathbf{L},\, c \not\Rightarrow x = 2\}\end{aligned}$$

and then (by Proposition 4.4) $D_1 \not\simeq_{io} D_2$. However, the abstract fixpoint semantics $\mathcal{F}^{io}$ does not distinguish D_1 from D_2. Indeed,

$$\mathcal{F}^{io}[\![D_1]\!] = \mathcal{F}^{io}[\![D_2]\!] = \begin{cases} q(x) \mapsto \{\langle c, \mathit{fin}(c \wedge x = 2)\rangle \mid c \in \mathbf{L}\} \\ p(x,y) \mapsto \{\langle c, \mathit{fin}(c \wedge y = 0)\rangle \mid c \in \mathbf{L},\, c \Rightarrow x = 2\}\} \cup \\ \qquad\qquad \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 1)\rangle \mid c \in \mathbf{L},\, c \not\Rightarrow x = 2\} \end{cases}$$

Given that $\mathcal{F}^{io}$ is the best possible approximation, this also formally proves that it is not possible to have a correct input-output semantics defined solely on the information provided by the input/output pairs (some more information in denotations is necessarily needed to be correct).

This also formally justifies (a posteriori) why [14] defined $\mathcal{O}^{io}(A)$ as a filter of a more concrete semantics instead of using a direct definition.

4.2 Modeling the input-output semantics of [14]

In this section, we formally show that the original input-output semantics of *tccp* $\mathcal{O}^{io}[\![D \,.\, A]\!]$ (defined in [14]) is essentially isomorphic to $\mathcal{B}^{io}_F[\![D \,.\, A]\!]$ (the finite fragment of the semantics introduced in the previous section).

Theorem 4.6 *Let P_1 and P_2 be two tccp programs such that no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is a failed conditional trace. Then, $\mathcal{O}^{io}[\![P_1]\!] = \mathcal{O}^{io}[\![P_2]\!]$ if and only if $\mathcal{B}^{io}_F[\![P_1]\!] = \mathcal{B}^{io}_F[\![P_2]\!]$.*

This theorem does not hold for *any* pair of *tccp* programs. When none of the programs reaches store *ff* (along some execution path), we actually have the same input-output pairs (except for the tag *fin*). However, when the store *ff* is reached during a computation, this is no longer necessarily true, as shown by the following example. This explains why we qualify as "essentially isomorphic" the relation between $\mathcal{O}^{io}[\![D \,.\, A]\!]$ and $\mathcal{B}^{io}_F[\![D \,.\, A]\!]$.

Example 4.7
Let $P_1 := D \,.\, \mathit{loop}$ and $P_2 := D \,.\, \mathsf{tell}(\mathit{false})$, where $D := \{\mathit{loop} :\!- \mathsf{tell}(\mathit{false}) \parallel \mathit{loop}\}$. We have that $\mathcal{B}^{io}_F[\![P_1]\!] = \mathcal{B}^{io}_F[\![P_2]\!] = \{\langle c, \mathit{fin}(\mathit{false})\rangle\}$ while $\mathcal{O}^{io}[\![P_1]\!] = \varnothing \neq \mathcal{O}^{io}[\![P_2]\!] = \{\langle c, \mathit{false}\rangle\}$.

The difference is due to the change we made in the definition of the small-step operational semantics. More specifically, in the operational semantics that we use (Definition 2.1), when the store *ff* is reached, we cannot have further transitions. We devised $\rightarrow$ in this way to be conform with the original rationale of the *ccp* paradigm. As a consequence, when a sequence computes *ff*, it is considered as a failed computation with output *ff*. In contrast, in the operational

semantics of [14], the transition relation $\rightarrow$ does not consider the *ff* store as a special case and then it is possible to execute an agent on the *ff* store.

Note that, if one is interested, it is straightforward to modify Definition 4.3 to compute exactly $\mathcal{O}^{io}[\![P]\!]$.

To conclude, it is interesting to note that $\mathcal{B}^{io}_F[\![P]\!]$ can be equivalently obtained by *first* appropriately filtering the conditional traces and *then* applying the abstraction α_{io}. Formally, given $M \in \mathbb{M}$, let $\pi^{\mathbb{M}}_F(M) \coloneqq \{r \in M \mid r$ ends with $\boxtimes$ or it contains a stuttering$\}$ and let $\mathbb{M}_F \coloneqq \pi^{\mathbb{M}}_F(\mathbb{M})$. Note that this domain contains only traces such that the application of the $\Downarrow$ operator produces only finite sequences of stores. It is straightforward to prove that the following diagram commutes

$$\begin{array}{ccc} (\mathbb{M}, \sqsubseteq) & \xrightarrow{\pi^{\mathbb{M}}_F} & (\mathbb{M}_F, \sqsubseteq) \\ \downarrow{\scriptstyle \alpha_{io}} & & \downarrow{\scriptstyle \alpha_{io}} \\ (\mathbb{IO}, \subseteq) & \xrightarrow{\pi_F} & (\mathbb{IO}_F, \subseteq) \end{array}$$

5 Application of our semantics

In [10] we have used (a preliminary version of) our semantics to develop an Abstract Diagnosis framework for *tccp* (following the approach for Logic Programming of [9, 6]). Since we could use a condensed semantics, we were able to formulate an efficacious parametric debugging methodology for *tccp* based on approximating the $\mathcal{D}[\![D]\!]$ operator by means of an abstract $\mathcal{D}^{\alpha}[\![D]\!]$ operator obtained by abstract interpretation.

We showed that, given the intended abstract specification $\mathcal{S}^{\alpha}$ of the semantics of a set of declarations D, we can determine all the rules which are wrong w.r.t. $\mathcal{S}^{\alpha}$ by a single application of $\mathcal{D}^{\alpha}[\![D]\!]$ to $\mathcal{S}^{\alpha}$. Thus, for suitable abstract domains, we obtain an effective *static check* which is able to identify the exact sources of errors.

There are some good features of this application that show up because of the properties of the concrete semantics we have proposed in this paper. Namely,

- it can be used with partial specifications,
- it can be used with partial sets of declarations.

Obviously, one cannot detect errors in rules involving processes which have not been specified; but for the rules that involve only processes that have a specification, the check can be made, even if the whole set of declarations has not been written yet. To the best of our knowledge, this is the only method for debugging that can work with incomplete programs.

Comparing our approach to other abstract diagnosis approaches for the *ccp* paradigm, thanks to our semantics, we have some advantages too. [18] proposes a first approach to the declarative debugging of a language of the *ccp* family (*utcc*), which (like in the case of [10]) is also based on the abstract diagnosis approach of [9]. However, that work does not cover the particular extra difficulty

of non-monotonicity, common to (most of) the other timed concurrent constraint languages. We have already justified through the paper why we consider essential to model also that aspect of concurrent constraint programs. We recall that this ability is crucial in order to model specific behaviors of reactive systems, such as timeouts or preemption actions. This is also the main reason why our abstract (and concrete) semantics are significantly different from [18] and from formalizations for other declarative languages.

Moreover, the abstract diagnosis instance on the $depth(k)$ domain presented in [18] is not effective, unless one considers a finite constraint system (situation that limits enormously the applicability of the proposal). The reason is due to the fact that the underlying semantics is not condensed and then, in general, even if abstract denotations contain only traces that are bounded in length, they need to maintain an infinite number of traces, one for each possible initial constraint. By applying the same abstraction technique on our condensed semantics, one obtains instead finite denotations.

One would argue that it is theoretically possible to find a suitable transformation of the $depth(k)$ proposal of [18] to make it effective, but in the end this would boil down to regain the condensation at the abstract semantics level. However, given the equality between $depth(k)$ traces and the initial part of concrete traces, essentially this would be the same of giving a condensed version of the concrete semantics.

Recently, we have also developed another abstraction of our semantics over a domain of formulas expressed in an extension with constraints of a linear temporal logic (csLTL). In that work, we provide an alternative automatic decision method to check whether a given property specified in csLTL is *valid* w.r.t. a *tccp* program. Most of the classical automatic program verification approaches are based on browsing the structure of some form of model (which represents the behavior of the program) to check if a given specification is valid. This implies that a subset of the model has to be built, and sometimes the needed fragment is quite huge. This is known as the state explosion problem. Our proposal, unlike other automatic program verification techniques does not require to build a model at all. The compact definition of the semantics evaluation functions over csLTL that we have obtained is due to the formulation of our concrete semantics definition.

6 Related Work

As we have said in the introduction, for timed concurrent constraint languages, the presence of

- non-determinism,
- local variables and
- timing constructs which are able to handle negative information

significantly complicates the definition of a fully abstract compositional semantics. In this section, we briefly show the impact of these difficulties when defining appropriate denotational semantics for other (timed) concurrent constraint languages. Most of the defined semantics are inspired in that of *ccp*, and characterize the finite input-output or strongest postcondition observable behaviors. The *strongest postcondition* observable collects the pairs of input-output stores

such that the program does not produce additional information, i.e., the input coincides with the output.

Soon in [12], the difficulties for handling nondeterminism and infinite behavior in the *ccp* paradigm was investigated. The authors showed that the presence of nondeterminism and synchronization requires relatively complex structures for the denotational model of (non timed) *ccp* languages. Moreover, infinite behaviors (which become natural in the timed extensions) are an additional nightmare. Traditionally, solutions to these difficulties have been based on the introduction of restrictions on the language. In [29] are given the basic ideas for the definition of appropriate semantics for *ccp* languages and—more specifically—is given a model based on observing the resting points of (finite) *ccp* processes. The defined semantics is fully abstract for the determinate fragment of *ccp* (i.e., choice agents have always a single branch). For (finite) nondeterminate processes that are monotonic in nature, a fully abstract semantics is given basing on the observation of ask/tell interactions. In [16], a simple denotational semantics fully abstract w.r.t. the upward-closed observable behavior is defined for *confluent ccp*, which is the subclass of *ccp* programs whose observable behavior does not depend on the chosen (non-deterministic) branch. They also define a correct semantics characterizing the input-output relation of (finite) processes for the *restricted-choice ccp*, which is a confluent sublanguage of *ccp* (syntactically restricted to choice agents where either all the branches have the same guard or the guards are all mutually exclusive). As the basis for a method to prove (partial) correctness of *ccp* programs, in [13] a denotational semantics which characterizes the strongest postcondition is given. The semantics is fully abstract for confluent *ccp*. It is also shown that the strongest postcondition semantics is not compositional w.r.t. the hiding agent.

The introduction of time in the *ccp* paradigm raises even more difficulties, in all different timed languages that have been proposed. Based on the deterministic fragment of *ccp*, in [26] the authors defined the *tcc* language and its semantics which is fully abstract just for *hiding free processes*. This restriction allows one to avoid the problem of non-monotonic behaviors. As we have shown in Example 3.17, due to the partial nature of the constraint system, the combination of the hiding operator with non-determinism can make the language behavior non-monotonic [12, 23].

The *ntcc* language extends *tcc* with non-determinism [22] and, inspired by the elegant model for *ccp* based on closure operators of [29], a denotational semantics for the strongest postcondition is defined. The semantics is fully abstract for *locally-independent* processes, i.e., processes in which the non-monotonic agents do not contain bounded variables (i.e., local variables via the hiding construct). More recently, [18] proposed a denotational semantics of the fragment of *ntcc* that excludes the non-monotonic construct *unless*.

The *Default tcc* language [27] is an extension of *tcc* that makes use of *default values* in order to model *strong preemption*. It adds to *tcc* language a limited form of negative information handling, with a construct that has to be used under so called *stable assumptions* for the negative information in order to avoid chaotic behaviors (notion borrowed from reactive languages like ESTEREL [5]). This aids to overcome the problem of the non-monotonic behavior since, in some sense, defaults *force* to have the Monotonicity property of Definition 3.4. The proposed compositional semantics is fully abstract for agents which satisfy stable assumptions. Their denotational model associates a condition to the

computation which plays a similar role to the first positive condition of our conditional traces (but we can allot more behaviors thanks to the others positive and negative conditions along the trace). The *Default tcc* language however has a limited expressive power compared w.r.t. *tccp* since it is deterministic and does not have process calls (and is thus not Turing complete).

The most recent dialect of timed *ccp* we know, the *utcc* language, was introduced in [24] as an extension of *tcc* for modeling mobility (communication of private names, typically used in security protocols or mobile systems). In [17], a denotational model for *utcc* processes based on a simple domain is defined for data-flow analysis. This semantics is fully abstract only for the monotonic fragment of the language. For the same language, [24] defines a denotational semantics characterizing the input-output behavior of processes. This semantics is fully abstract for the monotonic fragment of *utcc* and is based on temporal formulas.

Thus—to conclude—to our knowledge ours is the only proposal which defines a fully abstract semantics for a *full* dialect of timed *ccp* with "negative" constructs (having so a non-monotonic behavior), except the one of default *tcc*. However, *tccp* is non-deterministic and turing complete whereas default *tcc* is deterministic and not turing complete.

7 Conclusions

In this work, we have presented a small-step semantics that is fully abstract w.r.t. the *tccp* language behavior and that is suitable to be used as the basis of semantics-based program manipulation techniques such as abstract diagnosis. The task of defining a compositional fully-abstract semantics for the language has shown to be difficult due to the non-monotonic nature of the language, which is a characteristic shared with other concurrent languages of the *ccp* family.

To our knowledge, this is the first fully abstract compositional denotational semantics for a non-deterministic language in the *ccp* family that covers the whole language (including the non-monotonic behavior).

We have also defined a big-step semantics for *tccp* as an abstraction of the small-step one. This semantics collects the *limit* stores of (finite and infinite) computations. We have proven that its fragment for finite computations is precise enough to recover the original input-output semantics of the language [14]. Moreover, we also have proven that it is not possible to have a correct input-output semantics which is defined *solely* on the information provided by the input/output pairs.

As future work, we plan to investigate further on applications of our semantics to obtain novel analysis and verification methods. Moreover, we plan to adapt the ideas presented here to define appropriate fully-abstract semantics for other concurrent languages of the *ccp* family, such as *ntcc*, *utcc* and *tcc*. Thanks to this we will be able to straightforwardly adapt the abstract diagnosis methodology to such languages. These adaptations of the semantics are not immediate, since these languages have significant differences w.r.t. *tccp*, but (given the richness of *tccp* w.r.t. the other languages) we are confident that the required effort will be reasonable.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.

[2] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science*, 346(1):58–95, 2005.

[3] A. Aristizábal, F. Bonchi, C. Palamidessi, L. F. Pino, and F. D. Valencia. Deriving Labels and Bisimilarity for Concurrent Constraint Programming. In *14th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2011)*, volume 6604 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.

[4] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, 2011. Springer-Verlag.

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[6] M. Comini. *An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs.* PhD thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy, 1998.

[7] M. Comini, R. Gori, and G. Levi. Assertion based Inductive Verification Methods for Logic Programs. *Electronic Notes in Theoretical Computer Science*, 40:52–69, 2001.

[8] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Science of Computer Programming*, 49(1-3):89–123, 2003.

[9] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[10] M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011.

[11] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press.

[12] F. S. de Boer, A. di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151:37–78, 1995.

[13] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *ACM Trans. Program. Lang. Syst.*, 19(5):685–725, 1997.

[14] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[15] F. S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319, Berlin, 1991. Springer-Verlag.

[16] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183:281–315, 1997.

[17] M. Falaschi, C. Olarte, and C. Palamidessi. A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 207–218, New York, NY, USA, 2009. Acm.

[18] M. Falaschi, C. Olarte, C. Palamidessi, and F. D. Valencia. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 271–285. Springer-Verlag, 2007.

[19] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006.

[20] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[21] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[22] M. Nielsen, C. Palamidessi, and F. D. Valencia. On the Expressive Power of Temporal Concurrent Constraint Programming Languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 156–167, New York, NY, USA, 2002. ACM Press.

[23] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.

[24] C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In R. Wainwright and H. Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC08)*, pages 145–150. ACM, 2008.

[25] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass., 1993.

[26] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994.

[27] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal on Symbolic Computation*, 11:1–42, 1999.

[28] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM.

[29] V. A. Saraswat, M. Rinard, and P. Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM.

A Appendix

In the sequel, to avoid a proliferation of parenthesis, we assume that $\downarrow_c$ and $\Downarrow_c$ have priority over $\cdot$ and $\|$.

A.1 Proofs of Section 3

By construction, we can see that the conditional traces computed by $\mathcal{A}$ always satisfy that the store in a given time instant entails the positive condition. Formally,

Property A.1 *Let $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$, $\mathcal{I} \in \mathbb{I}_{\Pi}$ and $r \in \mathcal{A}[\![A]\!]_{\mathcal{I}}$. For each conditional tuple $(\eta^+, \eta^-) \rightarrowtail a$ occurring in r, $a \vdash \eta^+$.*

Proof.
This property is directly verified by (3.7) and (3.8) of Definition 3.15: when a guard is added to the positive condition, it is also added to the correspondent store, and propagated to the subsequent trace.

There exists a relation between the propagation operator $\downarrow$ and the *lub* $\otimes$ of the constraint system: the consecutive propagation of two constraints $(r\downarrow_c)\downarrow_{c'}$ is equivalent to $r\downarrow_{(c\otimes c')}$.

Lemma A.2 *Let $c, c' \in \mathbf{C}$ and $r \in \mathbf{M}$ such that $(r\downarrow_{c'})\downarrow_c$ is defined. Then $r\downarrow_{(c\otimes c')}$ is defined and $(r\downarrow_{c'})\downarrow_c = r\downarrow_{(c\otimes c')}$.*

Proof.
We proceed by structural induction on r.

$r = \epsilon$ and $r = \boxtimes$. Straightforward.

$r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'$ By hypothesis, $(r{\downarrow_{c'}}){\downarrow_c}$ is defined, thus, $c \gg (\eta^+ \otimes c', \eta^-)$ and $(r'{\downarrow_{c'}}){\downarrow_c}$ is defined. It follows directly that $c \otimes c' \gg (\eta^+, \eta^-)$ and, by inductive hypothesis, $(r'{\downarrow_{c\otimes c'}})$ is defined. Thus, $(r{\downarrow_{c\otimes c'}})$ is defined too.

$$
\begin{aligned}
(r{\downarrow_{c'}}){\downarrow_c} =& (((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Definition 3.7}] \\
=& ((c' \otimes \eta^+, \eta^-) \rightarrowtail c' \otimes d \cdot r'{\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Definition 3.7}] \\
=& (c \otimes c' \otimes \eta^+, \eta^-) \rightarrowtail c \otimes c' \otimes d \cdot (r'{\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Inductive Hypothesis}] \\
=& (c \otimes c' \otimes \eta^+, \eta^-) \rightarrowtail c \otimes c' \otimes d \cdot r'{\downarrow_{c\otimes c'}} \\
& \quad [\text{by Definition 3.7}] \\
=& r{\downarrow_{c\otimes c'}}
\end{aligned}
$$

$r = stutt(\eta^-) \cdot r'$ By hypothesis, $(r{\downarrow_{c'}}){\downarrow_c}$ is defined, thus, for all $c^- \in \eta^-$, $c \not\bowtie c^-$ and $c' \not\bowtie c^-$. Furthermore, $(r'{\downarrow_{c'}}){\downarrow_c}$ is defined as well. It follows directly that for all $c^- \in \eta^-$, $c \otimes c' \not\bowtie c^-$ and, by inductive hypothesis, $(r'{\downarrow_{c\otimes c'}})$ is defined. Thus, $(r{\downarrow_{c\otimes c'}})$ is defined too.

$$
\begin{aligned}
(r{\downarrow_{c'}}){\downarrow_c} =& ((stutt(\eta^-) \cdot r'){\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Definition 3.7}] \\
=& (stutt(\eta^-) \cdot r'{\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Definition 3.7}] \\
=& stutt(\eta^-) \cdot (r'{\downarrow_{c'}}){\downarrow_c} \\
& \quad [\text{by Inductive Hypothesis}] \\
=& stutt(\eta^-) \cdot r'{\downarrow_{c\otimes c'}} \\
& \quad [\text{by Definition 3.7}] \\
=& r{\downarrow_{c\otimes c'}}
\end{aligned}
$$

There exists a relation between the parallel composition and the operator of propagation as stated by the following lemma.

Lemma A.3 *Let $r_1, r_2 \in \mathbf{M}$ and $c \in \mathbf{C}$ such that $r_1{\downarrow_c} \mathbin{\overline{\|}} r_2{\downarrow_c}$ is defined. Then $(r_1 \mathbin{\overline{\|}} r_2){\downarrow_c}$ is defined and $r_1{\downarrow_c} \mathbin{\overline{\|}} r_2{\downarrow_c} = (r_1 \mathbin{\overline{\|}} r_2){\downarrow_c}$.*

Proof.

We proceed by structural induction on r_1. Note that, since $r_1{\downarrow_c} \mathbin{\overline{\|}} r_2{\downarrow_c}$ is defined, it follows that $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined as well.

$r_1 = \epsilon$ (or $r_1 = \boxtimes$) and any r_2 The statement follows directly from Definitions 3.7 and 3.9.

$r_1 = (\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'$ and $r_2 = (\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2'$ Since $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined, it follows that c is consistent with both $\eta_1 = (\eta_1^+, \eta_1^-)$ and $\eta_2 = (\eta_2^+, \eta_2^-)$, and thus, with $\eta_1 \otimes \eta_2$. We have to distinguish two cases.

$c \otimes d_1 \neq \mathit{ff}$ and $c \otimes d_2 \neq \mathit{ff}$ By inductive hypothesis, $(r_1' \parallel r_2')\downarrow_c$ is defined and, since $c \gg \eta_1 \otimes \eta_2$, $(r_1 \parallel r_2)\downarrow_c$ is defined as well.

$$\begin{aligned}
r_1\downarrow_c \parallel r_2\downarrow_c =& ((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1')\downarrow_c \parallel ((\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7}] \\
=& ((\eta_1^+ \otimes c, \eta_1^-) \rightarrowtail d_1 \otimes c \cdot r_1'\downarrow_c) \parallel ((\eta_2^+ \otimes c, \eta_2^-) \rightarrowtail d_2 \otimes c \cdot r_2'\downarrow_c) \\
& \quad [\text{by Definition 3.9}] \\
=& (\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes d_2 \otimes c \cdot (r_1'\downarrow_c \parallel r_2'\downarrow_c) \\
& \quad [\text{by Inductive Hypothesis}] \\
=& (\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes d_2 \otimes c \cdot (r_1' \parallel r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7}] \\
=& (r_1 \parallel r_2)\downarrow_c
\end{aligned}$$

$c \otimes d_1 = \mathit{ff}$ or $c \otimes d_2 = \mathit{ff}$ In this case, $r_1\downarrow_c \parallel r_2\downarrow_c$ reaches the store ff in one step, as also occurs when we compute $(r_1 \parallel r_2)\downarrow_c$:

$$\begin{aligned}
r_1\downarrow_c \parallel r_2\downarrow_c =& ((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1')\downarrow_c \parallel ((\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7 and Definition 3.9}] \\
=& (\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail \mathit{ff} \cdot \boxtimes \\
& \quad [\text{by Definition 3.7 and Definition 3.9}] \\
=& (r_1 \parallel r_2)\downarrow_c
\end{aligned}$$

$r_1 = (\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'$ and $r_2 = \mathit{stutt}(\eta_2^-) \cdot r_2'$ Since $r_1\downarrow_c$ and $r_2\downarrow_c$ are defined, we have that $c \gg \eta_1$ and $c \nvdash c^-$ for all $c^- \in \eta_2^-$. Therefore, $c \gg (\eta_1^+, \eta_1^- \cup \eta_2^-)$. By inductive hypothesis, $(r_1' \parallel r_2')\downarrow_c$ is defined, thus also $(r_1 \parallel r_2)\downarrow_c$ is defined.

$$\begin{aligned}
r_1\downarrow_c \parallel r_2\downarrow_c =& ((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1')\downarrow_c \parallel (\mathit{stutt}(\eta_2^-) \cdot r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7}] \\
=& ((\eta_1^+ \otimes c, \eta_1^-) \rightarrowtail d_1 \otimes c \cdot r_1'\downarrow_c) \parallel (\mathit{stutt}(\eta_2^-) \cdot r_2'\downarrow_c) \\
& \quad [\text{by Definition 3.9}] \\
=& (\eta_1^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes c \cdot (r_1'\downarrow_c \parallel r_2'\downarrow_c) \\
& \quad [\text{by Inductive Hypothesis}] \\
=& (\eta_1^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes c \cdot (r_1' \parallel r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7}] \\
=& (r_1 \parallel r_2)\downarrow_c
\end{aligned}$$

$r_1 = \mathit{stutt}(\eta_1^-) \cdot r_1'$ and $r_2 = \mathit{stutt}(\eta_2^-) \cdot r_2'$ Since $r_1\downarrow_c$ and $r_2\downarrow_c$ are defined, we have that c does not entail any constraint in $\eta_1^- \cup \eta_2^-$. By inductive hypothesis, $(r_1' \parallel r_2')\downarrow_c$ is defined, thus, we can conclude that also $(r_1 \parallel r_2)\downarrow_c$ is defined.

$$\begin{aligned}
r_1\downarrow_c \parallel r_2\downarrow_c =& (\mathit{stutt}(\eta_1^-) \cdot r_1')\downarrow_c \parallel (\mathit{stutt}(\eta_2^-) \cdot r_2')\downarrow_c \\
& \quad [\text{by Definition 3.7}]
\end{aligned}$$

$$
\begin{aligned}
&=(stutt(\eta_1^-)\cdot r_1'{\downarrow_c})\ \overline{\|}\ (stutt(\eta_2^-)\cdot r_2'{\downarrow_c})\\
&\qquad[\text{by Definition 3.9}]\\
&=stutt(\eta_1^- \cup \eta_2^-)\cdot(r_1'{\downarrow_c}\ \overline{\|}\ r_2'{\downarrow_c})\\
&\qquad[\text{by Inductive Hypothesis}]\\
&=stutt(\eta_1^- \cup \eta_2^-)\cdot(r_1'\ \overline{\|}\ r_2'){\downarrow_c}\\
&\qquad[\text{by Definition 3.7}]\\
&=(r_1\ \overline{\|}\ r_2){\downarrow_c}
\end{aligned}
$$

An important technical result states that the evaluation function for agents $\mathcal{A}$ is closed under context embedding. A context $C[\,]$ consists in a *tccp* agent with a *hole*, which means that $C[A]$ represents the result of replacing the hole in $C[\,]$ with the agent A.

Lemma A.4 *Let $A_1, A_2 \in \mathbb{A}_{\mathbf{C}}^{\Pi}$ and $\mathcal{I} \in \mathbb{I}$. Then $\mathcal{A}[\![A_1]\!]_{\mathcal{I}} = \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ if and only if, for all context $C[\,]$, $\mathcal{A}[\![C[A_1]]\!]_{\mathcal{I}} = \mathcal{A}[\![C[A_2]]\!]_{\mathcal{I}}$.*

Proof.

⇐ Directly holds.

⇒ This implication follows from Definition 3.15. The evaluation function $\mathcal{A}$ is defined by composition of the semantics of its subagents. In particular, the semantics of both, $C[A_1]$ and $C[A_2]$, is computed from the semantics of A_1 and A_2, respectively. Since A_1 and A_2 are equivalent, then also the semantics of $C[A_1]$ and $C[A_2]$ coincide.

Lemma A.5 *For each $A \in \mathbb{A}_{\mathbf{C}}^{\Pi}$ and each $D \in \mathbb{D}_{\mathbf{C}}^{\Pi}$, $\mathcal{A}[\![A]\!]$ and $\mathcal{D}[\![D]\!]$ are continuos.*

Proof.

Consider $A \in \mathbb{A}_{\mathbf{C}}^{\Pi}$ and $D \in \mathbb{D}_{\mathbf{C}}^{\Pi}$. To prove the continuity of $\mathcal{A}[\![A]\!]$, we have to verify two properties: monotonicity and finitarity. The continuity of $\mathcal{D}[\![D]\!]$ follows directly from the continuity of $\mathcal{A}[\![A]\!]$ and from Definition 3.19.

Monotonicity. It is sufficient to show that for each $\mathcal{I}_1, \mathcal{I}_2 \in \mathbb{I}$ and and for each $A \in \mathbb{A}_{\mathbf{C}}^{\Pi}$, $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \Rightarrow \mathcal{A}[\![A]\!]_{\mathcal{I}_1} \sqsubseteq \mathcal{A}[\![A]\!]_{\mathcal{I}_2}$. Observe that the only case in which $\mathcal{A}$ depends on the interpretation is the case of the process call.

By definition of $\sqsubseteq$, $\mathcal{I}_1(p(\vec{x})) \sqsubseteq \mathcal{I}_2(p(\vec{x}))$, thus:

$$
\begin{aligned}
\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}_1} &= \bigsqcup\{(tt,\varnothing) \rightarrowtail tt\cdot r \mid r \in \mathcal{I}_1(p(\vec{x}))\}\\
&\sqsubseteq \bigsqcup\{(tt,\varnothing) \rightarrowtail tt\cdot r \mid r \in \mathcal{I}_2(p(\vec{x}))\} = \mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}_2}
\end{aligned}
$$

Finitarity. Again, it is sufficient to consider the evaluation function $\mathcal{A}$ for the case of the process call. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ depends on a finitary subset of $\mathcal{I}$, in particular on the subset regarding $p(\vec{x})$ which is a finitary set of conditional traces closed by prefix.

Lemma A.6 *Let $r \in \mathbf{M}$ and $c, c' \in \mathbf{C}$ such that $c \vdash c'$ and $r{\Downarrow_c}$ is defined. Then $(r{\downarrow_{c'}}){\Downarrow_c}$ is defined and $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.*

Proof.

By hypothesis, $r{\Downarrow_c}$ is defined, thus c is compatible with all the conditions occurring in r. Since $c \vdash c'$, it is easy to notice that also c' is compatible with all the conditions occurring in r, thus $r{\downarrow_{c'}}$ is defined. Then, $(r{\downarrow_{c'}}){\Downarrow_c}$ is defined as well. If $c = \mathit{ff}$, by Definition 3.24, $r{\Downarrow_{\mathit{ff}}} = \mathit{ff} = (r{\downarrow_{c'}}){\Downarrow_{\mathit{ff}}}$. Otherwise, if $c \neq \mathit{ff}$, we proceed by induction on the structure of r.

$r = \epsilon$ and $r = \boxtimes$ The statement follows directly from Definitions 3.7 and 3.24.

$r = (\eta^+, \eta^-) \twoheadrightarrow d \cdot r'$ We distinguish three sub-cases.

$d \otimes c \neq \mathit{ff}$ Since $c \vdash c'$, it follows that $d \otimes c' \neq \mathit{ff}$, thus:

$$
\begin{aligned}
(r{\downarrow_{c'}}){\Downarrow_c} &= (((\eta^+, \eta^-) \twoheadrightarrow d \cdot r'){\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.7}] \\
&= ((\eta^+ \otimes c', \eta^-) \twoheadrightarrow d \otimes c' \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.24}] \\
&= c \cdot (r'{\downarrow_{c'}}){\Downarrow_{c \otimes d \otimes c'}} \\
&\quad [\text{by Inductive Hypothesis}] \\
&= c \cdot r'{\Downarrow_{c \otimes d \otimes c'}} \\
&\quad [\text{since } c \vdash c'] \\
&= c \cdot r'{\Downarrow_{c \otimes d}}
\end{aligned}
$$

By Definition 3.24, $r{\Downarrow_c} = ((\eta^+, \eta^-) \twoheadrightarrow d \cdot r'){\Downarrow_c} = c \cdot r'{\Downarrow_{c \otimes d}}$, thus $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$d \otimes c = \mathit{ff}$ and $d \otimes c' \neq \mathit{ff}$ We have that:

$$
\begin{aligned}
(r{\downarrow_{c'}}){\Downarrow_c} &= ((\eta^+, \eta^-) \twoheadrightarrow d \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.7}] \\
&= ((\eta^+ \otimes c', \eta^-) \twoheadrightarrow d \otimes c' \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.24}] \\
&= c \cdot \mathit{ff}
\end{aligned}
$$

By Definition 3.24, $r{\Downarrow_c} = ((\eta^+, \eta^-) \twoheadrightarrow d \cdot r'){\Downarrow_c} = c \cdot \mathit{ff}$, thus $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$d \otimes c' = \mathit{ff}$ Since $c \vdash c'$, it follows that $d \otimes c = \mathit{ff}$, thus:

$$
\begin{aligned}
(r{\downarrow_{c'}}){\Downarrow_c} &= (((\eta^+, \eta^-) \twoheadrightarrow d \cdot r'){\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.7}] \\
&= ((\eta^+ \otimes c', \eta^-) \twoheadrightarrow \mathit{ff} \cdot \boxtimes){\Downarrow_c} \\
&\quad [\text{by Definition 3.24}] \\
&= c \cdot \mathit{ff}
\end{aligned}
$$

By Definition 3.24, it follows that $r{\Downarrow_c} = c \cdot \mathit{ff} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$r = \mathit{stutt}(\eta^-) \cdot r'$ By Definition 3.24, it follows that:

$$
\begin{aligned}
(r{\downarrow_{c'}}){\Downarrow_c} &= ((\mathit{stutt}(\eta^-) \cdot r'){\downarrow_{c'}}){\Downarrow_c} \\
&\quad [\text{by Definition 3.7}]
\end{aligned}
$$

$$= (stutt(\eta^-) \cdot r'{\downarrow_{c'}})\Downarrow_c$$
[by Definition 3.24]
$$= c$$

By Definition 3.24, $r\Downarrow_c = (stutt(\eta^-) \cdot r')\Downarrow_c = c$, thus $r\Downarrow_c = (r{\downarrow_{c'}})\Downarrow_c$.

In order to formulate the following Lemma A.8, we need to introduce the counterpart of $\between$ on behavioral timed traces.

Definition A.7 *Let $s, s_1, s_2 \in \mathbf{C}^*$. $\breve{\between}: \mathbf{C}^* \times \mathbf{C}^* \to \mathbf{C}^*$ is defined by structural induction as:*

$$s \breve{\between} \epsilon := s \qquad \epsilon \breve{\between} s := s \tag{A.1a}$$

$$(c_1 \cdot s_1) \breve{\between} (c_2 \cdot s_2) := \begin{cases} (c_1 \otimes c_2) \cdot (c_2 \otimes s_1 \breve{\between} c_1 \otimes s_2) & \text{if } c_1 \otimes c_2 \neq \mathit{ff} \\ \mathit{ff} & \text{if } c_1 \otimes c_2 = \mathit{ff} \end{cases} \tag{A.1b}$$

where, by abusing notation, $c \otimes (c_1 \cdots c_n)$ denotes $(c \otimes c_1) \cdots (c \otimes c_n)$.

We extend this operator to sets of behavioral timed traces as $S_1 \breve{\between} S_2 = \{s_1 \breve{\between} s_2 \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$.

Lemma A.8 *Let $c \in \mathbf{C}$; $A_1, A_2 \in \mathbb{A}^{\Pi}_{\mathbf{C}}$; $\mathcal{I} \in \mathbb{I}$; $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{I}}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ such that $r_1 \between r_2$, $r_1\Downarrow_c$ and $r_2\Downarrow_c$ are defined. Then, $(r_1 \between r_2)\Downarrow_c$ is defined and $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = (r_1 \between r_2)\Downarrow_c$.*

Proof.

Since both $r_1\Downarrow_c$ and $r_2\Downarrow_c$ are defined, c satisfies all the conditions in r_1 and r_2. It is easy to notice from Definition 3.9 that c satisfies also the conditions of $r_1 \between r_2$, thus, $(r_1 \between r_2)\Downarrow_c$ is defined as well.

We proceed to prove that $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = (r_1 \between r_2)\Downarrow_c$ by induction on the structure of r_1.

$r_1 = \epsilon$ and any r_2 By Definition 3.9, $(r_1 \between r_2)\Downarrow_c = (\epsilon \between r_2)\Downarrow_c = r_2\Downarrow_c$. By Definition 3.24 and by Equation (A.1a), we obtain: $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = \epsilon \breve{\between} r_2\Downarrow_c = r_2\Downarrow_c$. Thus, $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = (r_1 \between r_2)\Downarrow_c$.

$r_1 = \boxtimes$ and any $r_2 \neq \epsilon$ By Definition 3.9, $(r_1 \between r_2)\Downarrow_c = (\boxtimes \between r_2)\Downarrow_c = r_2\Downarrow_c$. By Definition 3.24 and by Equation (A.1b), $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = c \breve{\between} r_2\Downarrow_c = r_2\Downarrow_c$, since $r_2 \neq \epsilon$. Thus, $r_1\Downarrow_c \breve{\between} r_2\Downarrow_c = (r_1 \between r_2)\Downarrow_c$.

$r_1 = \eta_1 \nrightarrow d_1 \cdot r'_1$ and $r_2 = \eta_2 \nrightarrow d_2 \cdot r'_2$

$d_1 \otimes d_2 \neq \mathit{ff}$

$$(r_1 \between r_2)\Downarrow_c = ((\eta_1 \nrightarrow d_1 \cdot r'_1) \between (\eta_2 \nrightarrow d_2 \cdot r'_2))\Downarrow_c$$
[by Definition 3.9]
$$= (\eta_1 \otimes \eta_2 \nrightarrow d_1 \otimes d_2 \cdot (r'_1{\downarrow_{d_2}} \between r'_2{\downarrow_{d_1}}))\Downarrow_c$$
[by Definition 3.24]
$$= c \cdot (r'_1{\downarrow_{d_2}} \between r'_2{\downarrow_{d_1}})\Downarrow_{c \otimes d_1 \otimes d_2}$$
[by Inductive Hypothesis]

$$
\begin{aligned}
&= c \cdot ((r_1'{\downarrow_{d_2}}){\Downarrow_{c\otimes d_1\otimes d_2}} \mathbin{\breve{\|}} (r_2'{\downarrow_{d_1}}){\Downarrow_{c\otimes d_1\otimes d_2}}) \\
&\qquad [\text{by Lemma A.6}] \\
&= c \cdot (r_1'{\Downarrow_{c\otimes d_1\otimes d_2}} \mathbin{\breve{\|}} r_2'{\Downarrow_{c\otimes d_1\otimes d_2}}) \\
&\qquad [d_1 \text{ (resp. } d_2) \text{ is entailed by the stores in } r_1' \text{ (resp. } r_2')] \\
&= c \cdot (r_1'{\Downarrow_{c\otimes d_1}} \mathbin{\breve{\|}} r_2'{\Downarrow_{c\otimes d_2}}) \\
&\qquad [\text{by Equation (A.1b)}] \\
&= (c \cdot r_1'{\Downarrow_{c\otimes d_1}}) \mathbin{\breve{\|}} (c \cdot r_2'{\Downarrow_{c\otimes d_2}})
\end{aligned}
$$

By Definition 3.24, $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (c \cdot r_1'{\Downarrow_{c\otimes d_1}}) \mathbin{\breve{\|}} (c \cdot r_2'{\Downarrow_{c\otimes d_2}})$; therefore, we conclude $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c}$.

$\boldsymbol{d_1 \otimes d_2 = \mathit{ff}}$

$$
\begin{aligned}
(r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c} &= ((\eta_1 \rightarrowtail d_1 \cdot r_1') \mathbin{\bar{\|}} (\eta_2 \rightarrowtail d_2 \cdot r_2')){\Downarrow_c} \\
&\qquad [\text{by Definition 3.9}] \\
&= (\eta_1 \otimes \eta_2 \rightarrowtail \mathit{ff} \cdot \boxtimes){\Downarrow_c} \\
&\qquad [\text{by Definition 3.24}] \\
&= c \cdot \mathit{ff}
\end{aligned}
$$

By Definition 3.24 and by Equation (A.1b), $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = c \cdot \mathit{ff}$, thus $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c}$.

$\boldsymbol{r_1 = \eta_1 \rightarrowtail d_1 \cdot r_1'}$ and $\boldsymbol{r_2 = \mathit{stutt}(\eta_2^-) \cdot r_2'}$

$$
\begin{aligned}
(r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c} &= ((\eta_1 \rightarrowtail d_1 \cdot r_1') \mathbin{\bar{\|}} (\mathit{stutt}(\eta_2^-) \cdot r_2')){\Downarrow_c} \\
&\qquad [\text{by Definition 3.9}] \\
&= ((\eta_1^+, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \cdot (r_1' \mathbin{\bar{\|}} r_2'{\downarrow_{d_1}})){\Downarrow_c} \\
&\qquad [\text{by Definition 3.24}] \\
&= c \cdot (r_1' \mathbin{\bar{\|}} r_2'{\downarrow_{d_1}}){\Downarrow_{c\otimes d_1}} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= c \cdot (r_1'{\Downarrow_{c\otimes d_1}} \mathbin{\breve{\|}} (r_2'{\downarrow_{d_1}}){\Downarrow_{c\otimes d_1}}) \\
&\qquad [\text{by Lemma A.6}] \\
&= c \cdot (r_1'{\Downarrow_{c\otimes d_1}} \mathbin{\breve{\|}} r_2'{\Downarrow_{c\otimes d_1}}) \\
&\qquad [\text{by Equation (A.1b)}] \\
&= (c \cdot r_1'{\Downarrow_{c\otimes d_1}}) \mathbin{\breve{\|}} (c \cdot r_2'{\Downarrow_c})
\end{aligned}
$$

By Definition 3.24, $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (c \cdot r_1'{\Downarrow_{c\otimes d_1}}) \mathbin{\breve{\|}} (c \cdot r_2'{\Downarrow_c})$, thus $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c}$.

$\boldsymbol{r_1 = \mathit{stutt}(\eta_1^-) \cdot r_1'}$ and $\boldsymbol{r_2 = \mathit{stutt}(\eta_2^-) \cdot r_2'}$

$$
\begin{aligned}
(r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c} &= ((\mathit{stutt}(\eta_1^-) \cdot r_1') \mathbin{\bar{\|}} (\mathit{stutt}(\eta_2^-) \cdot r_2')){\Downarrow_c} \\
&\qquad [\text{by Definition 3.9}] \\
&= (\mathit{stutt}(\eta_1^- \cup \eta_2^-) \cdot (r_1' \mathbin{\bar{\|}} r_2')){\Downarrow_c} \\
&\qquad [\text{by Definition 3.24}]
\end{aligned}
$$

$$
\begin{aligned}
&= c \cdot (r_1' \mathbin{\bar{\|}} r_2')\Downarrow_c \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= c \cdot (r_1'\Downarrow_c \mathbin{\breve{\|}} r_2'\Downarrow_c) \\
&\qquad [\text{by Equation (A.1b)}] \\
&= (c \cdot r_1'\Downarrow_c) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)
\end{aligned}
$$

By Definition 3.24, $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (c \cdot r_1'\Downarrow_c) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)$, thus $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (r_1 \mathbin{\bar{\|}} r_2)\Downarrow_c$.

Proof of Theorem 3.25.

Let $d \in \mathbf{C}$ and $P = D.A$ with $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$, we proceed by structural induction on A.

skip The proof in this case is straightforward.

$$
\mathit{prefix}(\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_d = \mathit{prefix}(\{\boxtimes\})\Downarrow_d = \{\epsilon,\ d\} = \mathcal{B}^{ss}[\![D\,.\,\mathsf{skip}]\!]_d
$$

tell(c)

$$
\begin{aligned}
\mathit{prefix}((\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) &= \mathit{prefix}((tt, \varnothing) \rightarrowtail c \cdot \boxtimes)\Downarrow_d) \\
&= \mathit{prefix}(d \cdot (d \otimes c)) \\
&= \mathcal{B}^{ss}[\![D\,.\,\mathsf{tell}(c)]\!]_d
\end{aligned}
$$

$A = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ We prove the two directions separately.

$\subseteq$ We show that, given a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$, it holds that $\forall d \in \mathbf{C}.\ \mathit{prefix}(r\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. We have to distinguish two cases.

$r = (c_j, \varnothing) \rightarrowtail c_j \cdot r_j\downarrow_{c_j}$ with $1 \le j \le n$ By (3.7) it follows that $r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}$. In case $r\Downarrow_d$ is not defined (i.e., $d \nvdash c_j$), $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Otherwise, if $r\Downarrow_d$ is defined, we have that $d \vdash c_j$ and $(r_j\downarrow_{c_j})\Downarrow_{d \otimes c_j}$ is defined too. We distinguish two sub-cases.

$d \neq ff$ In this case we have:

$$
\begin{aligned}
&\mathit{prefix}(r\Downarrow_d) \\
&= \mathit{prefix}(\{((c_j, \varnothing) \rightarrowtail c_j \cdot r_j\downarrow_{c_j})\Downarrow_d \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.24}] \\
&= \mathit{prefix}(\{d \cdot (r_j\downarrow_{c_j})\Downarrow_{d \otimes c_j} \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Lemma A.6 and since } d \vdash c_j] \\
&= \mathit{prefix}(\{d \cdot r_j\Downarrow_d \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Equation (3.1)}] \\
&= \{\epsilon,\ d\} \cup \{d \cdot s \mid 1 \le j \le n,\ s \in \mathit{prefix}(\mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \{\epsilon,\ d\} \cup \{d \cdot s \mid 1 \le j \le n,\ s \in \mathcal{B}^{ss}[\![D\,.\,A_j]\!]_d\}
\end{aligned}
$$

The element ϵ directly belongs to $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Since $d \vdash c_j$, also d belongs to $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$ (at least one step is performed

in the computation). Finally, the set $\{d \cdot s \,|\, 1 \leq j \leq n,\ s \in \mathcal{B}^{ss}[\![D \,.\, A_j]\!]_d\}$ is also contained in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. In particular, following Rule **R2**, the agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ (executed with a store d that entails one of the guards, e.g. c_j) behaves, in the next time instant, as the corresponding agent A_j over the store (which is not modified in that step).

$d = ff$ By definition of $\Downarrow$ (3.24), we have that $prefix(r\Downarrow_{ff}) = \{\epsilon, ff\}$ which corresponds to the set $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{ff}$ since the transition relation $\rightarrow$ is not defined for the configuration $\langle A, ff\rangle$.

$r = stutt(\{c_1, \ldots, c_n\}) \cdot r'$ By (3.7), we have that $r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ and for all $1 \leq j \leq n$, $c_j \neq tt$. In case $r\Downarrow_d$ is not defined (i.e., it exists $1 \leq j \leq n$ such that $d \vdash c_j$), $prefix(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. Otherwise, if $r\Downarrow_d$ is defined then $prefix(r\Downarrow_d) = \{\epsilon,\ d\} \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$\supseteq$ For each $d \in \mathbf{C}$, it exists a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $prefix(r\Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. There are three cases to be considered.

d does not satisfy any guard This means that for all $1 \leq j \leq n$, $d \nvdash c_j$; then, the small-step behavior is $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d = \{\epsilon,\ d\}$. Thus, it exists a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $r = stutt(\{c_1, \ldots, c_n\}) \cdot r'$ with $r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Moreover, by Definition 3.24 and Definition 3.1, it follows that $prefix(r\Downarrow_d) = \{\epsilon,\ d\} \supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

there exists c_j such that $d \vdash c_j$ and $d \neq ff$ In this case, one of the conditional traces computed by the semantics evaluation function $\mathcal{A}$ is $r = (c_j, \varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}$ with $r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}$. Then, we have:

$$
\begin{aligned}
prefix(r\Downarrow_d) &= prefix(((c_j, \varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}})\Downarrow_d) \\
&\quad [\text{by Definition 3.24}] \\
&= prefix(\{d \cdot (r_j{\downarrow_{c_j}})\Downarrow_{d \otimes c_j} \,|\, r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\quad [\text{by Lemma A.6 and since } d \vdash c_j] \\
&= prefix(\{d \cdot r_j\Downarrow_d \,|\, r_j\Downarrow_d \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\}) \\
&\quad [\text{by Equation (3.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s \,|\, s \in prefix(\mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\} \\
&\quad [\text{by Inductive Hypothesis}] \\
&\supseteq \{\epsilon, d\} \cup \{d \cdot s \,|\, s \in \mathcal{B}^{ss}[\![D \,.\, A_j]\!]_d\} \\
&\quad [\text{by Rule } \mathbf{R2}] \\
&\supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}
$$

$d = ff$ In this case we have:

$$
\begin{aligned}
prefix(r\Downarrow_{ff}) &= prefix(((c_j, \varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}})\Downarrow_{ff}) \\
&\quad [\text{by Definition 3.24}] \\
&= \{\epsilon, ff\} \\
&\quad [\text{by Definition 3.1}]
\end{aligned}
$$

$$\supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_{ff}$$

Therefore, we can conclude that $\mathit{prefix}(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) = \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

now c then A_1 else A_2 We prove the two directions independently. We abbreviate the conditional agent and call it A ($A :=$ now c then A_1 else A_2).

$\subseteq$ We show that $\forall d \in \mathbf{C}.\ \mathit{prefix}(\mathcal{A}[\![\text{now } c \text{ then } A_1 \text{ else } A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D \,.\, \text{now } c \text{ then } A_1 \text{ else } A_2]\!]_d$. There are seven possible cases, one for each type of trace r in (3.8).

$r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes$ By (3.8) we have that $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, which means, by Definition 3.15, that $A_1 =$ skip. We consider now the three possible cases:

$d \vdash c$ and $d \neq ff$ It is straightforward that $\mathit{prefix}(r\Downarrow_d) = \mathit{prefix}(d \cdot d) = \{\epsilon,\ d,\ d \cdot d\}$. On the behavioral part, we know from Rule **R4** that the observable of A is the set of all prefixes of $d \cdot d$, so we can conclude $\mathit{prefix}(r\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$d = ff$ The small-step behavior is $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{ff} = \{\epsilon, ff\}$. Since $ff \vdash c$ it is straightforward that $\mathit{prefix}(r\Downarrow_{ff}) = \{\epsilon, ff\} = \mathcal{B}^{ss}[\![D \,.\, A]\!]_{ff}$.

$d \nvdash c$ Then the application of $\Downarrow_d$ to the agent semantics does not compute any behavioral timed trace. Therefore, $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$r = (\eta^+ \otimes c, \eta^-) \rightarrowtail a \otimes c \cdot r'\downarrow_c$ From (3.8) it follows that $(\eta^+, \eta^-) \rightarrowtail a \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, d being compatible with all the conditions occurring in r', $a \otimes c \neq ff$ and $\forall h^- \in \eta^- . \eta^+ \otimes c \nvdash h^-$.
In case $r\Downarrow_d$ is not defined (i.e., $d \not\Vdash (\eta^+ \otimes c, \eta^-)$ or when d is not compatible with some condition occurring in r'), we have that $\mathit{prefix}(r\Downarrow_d) = \varnothing$ which is directly included in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.
Otherwise, if $r\Downarrow_d$ is defined, it follows that $d \Vdash (\eta^+ \otimes c, \eta^-)$. This implies that $d \vdash c$ since c belongs to the positive condition. Under these conditions, we have:

$$
\begin{aligned}
&\mathit{prefix}(r\Downarrow_d) = \\
&= \mathit{prefix}(\{((\eta^+ \otimes c, \eta^-) \rightarrowtail a \otimes c \cdot r'\downarrow_c)\Downarrow_d \,|\, (\eta^+, \eta^-) \rightarrowtail a \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.24}] \\
&= \mathit{prefix}(\{d \cdot (r'\downarrow_c)\Downarrow_{d \otimes a \otimes c} \,|\, d \cdot r'\Downarrow_{d \otimes a} \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\}) \\
&\qquad [\text{by Lemma A.6 since } d \vdash c] \\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d \otimes a} \,|\, d \cdot r'\Downarrow_{d \otimes a} \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\}) \\
&= \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d \\
&\qquad [\text{by Rule } \mathbf{R3}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}
$$

$r = (\eta^+ \otimes c, \eta^-) \rightarrowtail ff \cdot \boxtimes$ We consider two possible cases:

$d \models (\eta^+ \otimes c, \eta^-)$ This implies that $d \vdash c$. Under these conditions, we get:

$$
\begin{aligned}
& \mathit{prefix}(r\Downarrow_d) \\
&= \mathit{prefix}(((\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{ff} \cdot \boxtimes)\Downarrow_d) \\
&\qquad [\text{by Definition 3.24}] \\
&= \{\epsilon, \mathit{ff}\} \\
&\qquad [\text{by Definition 3.1}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}
$$

$d \not\models (\eta^+ \otimes c, \eta^-)$ In this case $\mathit{prefix}(r\Downarrow_d) = \varnothing$ which is directly included in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$r = (c, \eta^-) \rightarrowtail c \cdot r'$ In case $r\Downarrow_d$ is not defined (i.e., $d \not\models (c, \eta^-)$ or also when d is not compatible with some condition occurring in r') we have that $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. Otherwise, by (3.8), $\mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and d is compatible with all the conditions occurring in r'. We have to consider two sub-cases.

$d \neq \mathit{ff}$ In this case we have:

$$
\begin{aligned}
& \mathit{prefix}(r\Downarrow_d) = \\
&= \mathit{prefix}(\{((c, \eta^-) \rightarrowtail c \cdot r')\Downarrow_d \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.24}] \\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d \otimes c} \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{since } d \vdash c] \\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_d \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.15}] \\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_d \mid r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Equation (3.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d\} \\
&\qquad [\text{by Rule } \mathbf{R4}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}
$$

The fourth step follows from the definition of the semantics $\mathcal{A}$ (Definition 3.15). The construct *stutt* is introduced only by an ask agent. Thus, we know that A_1 is an ask agent. The Equation (3.8), states that $\mathit{stutt}(\eta^-)$ is always followed by a conditional trace which belongs to the semantics of the ask, which can be reduced to say that r' belongs to $\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$.

$d = \mathit{ff}$ In this case we have that $\mathit{prefix}(r\Downarrow_{\mathit{ff}}) = \{\epsilon, \mathit{ff}\}$ which corresponds to the behavior $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{\mathit{ff}}$ since the transition relation $\rightarrow$ is not defined for the agent A starting with store ff.

$\underline{r = (tt, \{c\}) \rightarrowtail tt \cdot \boxtimes}$ By (3.8), $\boxtimes \in \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$. By Definition 3.15, it follows that A_2 is a skip agent. We consider two sub-cases.

$\underline{d \nvdash c}$ It is straightforward that $prefix(r\Downarrow_d) = prefix(d \cdot d) = \{\epsilon,\ d,\ d \cdot d\}$. From Rule **R6**, we know that the observable of the agent A consists of the set of all prefixes of $d \cdot d$. Therefore, $prefix(r\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\underline{d \vdash c}$ In this case $r\Downarrow_d$ does not compute any trace because d does not satisfy the condition, thus $prefix(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\underline{r = (\eta^+, \eta^- \cup \{c\}) \rightarrowtail c' \cdot r'}$ In case $r\Downarrow_d$ is not defined (i.e., $d \not\Vdash (\eta^+, \eta^- \cup \{c\})$), $prefix(r\Downarrow_d) = \varnothing$, which is directly contained in $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Otherwise, if $r\Downarrow_d$ it follows that $(\eta^+, \eta^-) \rightarrowtail c' \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $c' \nvdash c$. If $d \Vdash (\eta^+, \eta^- \cup \{c\})$, we know also that $d \nvdash c$. Under these conditions, we have:

$$
\begin{aligned}
&prefix(\{r\Downarrow_d) \\
&= prefix(\{((\eta^+, \eta^- \cup \{c\}) \rightarrowtail c' \cdot r')\Downarrow_d \,|\, (\eta^+, \eta^-) \rightarrowtail c' \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.24}] \\
&= prefix(\{d \cdot r'\Downarrow_{d \otimes c'} \,|\, d \cdot r'\Downarrow_{d \otimes c'} \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\}) \\
&= prefix(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A_2]\!]_d \\
&\qquad [\text{by Rule } \mathbf{R5}] \\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d
\end{aligned}
$$

$\underline{r = (tt, \eta^- \cup \{c\}) \rightarrowtail tt \cdot r'}$ By (3.8), we have that $stutt(\eta^-) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$.
In case $r\Downarrow_d$ is not defined (i.e., $d \not\Vdash (tt, \eta^- \cup \{c\})$), $prefix(r\Downarrow_d) = \varnothing$, which is directly contained in $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.
Otherwise, if $r\Downarrow_d$ is defined it follows that $d \Vdash (tt, \eta^- \cup \{c\})$. Then, we have:

$$
\begin{aligned}
&prefix(r\Downarrow_d) \\
&= prefix(\{((tt, \eta^- \cup \{c\}) \rightarrowtail tt \cdot r')\Downarrow_d \,|\, stutt(\eta^-) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.24}] \\
&= prefix(\{d \cdot r'\Downarrow_d \,|\, stutt(\eta^-) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.15}] \\
&= prefix(\{d \cdot r'\Downarrow_d \,|\, r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Equation (3.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s \,|\, s \in prefix(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D\,.\,A_2]\!]_d\} \\
&\qquad [\text{by Rule } \mathbf{R6}] \\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d
\end{aligned}
$$

The third step can be done since each construct $stutt(\eta^-)$ is introduced by a choice agent, and Equation (3.7) states that it is always followed by a conditional trace r' belonging recursively to the semantics of A_2.

$\supseteq$ We have four cases, one for each rule defining the operational semantics for the conditional agent in Figure 1.

Rule R3 Let us recall the conditions to apply Rule **R3**: it must occur $\langle A_1, d\rangle \to \langle A_1', d'\rangle$ and $d \vdash c$. In this case, we have that $\mathcal{B}^{ss}[\![D.A]\!]_d = \mathcal{B}^{ss}[\![D.A_1]\!]_d$. By inductive hypothesis, we know that $prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D.A_1]\!]_d$, thus also $prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Next, we prove the inclusion $prefix(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_d$. We proceed by induction on the structure of a generic $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ in order to find $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $prefix(r_1\Downarrow_d) \subseteq prefix(r\Downarrow_d)$.

$r_1 = \boxtimes$ By (3.8), $r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We know that $\boxtimes\Downarrow_d = d$ and $r\Downarrow_d = d \cdot (d \otimes c) = d \cdot d$, since $d \vdash c$. It is easy to see that the prefixes of d are all included in the prefixes of $d \cdot d$.

$r_1 = (\eta^+, \eta^-) \rightarrowtail c' \cdot r'$ By definition, $r = (\eta^+ \otimes c, \eta^-) \rightarrowtail c' \otimes c \cdot r'\downarrow_c \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. If $d \Vdash (\eta^+, \eta^-)$, then $r_1\Downarrow_d = d \cdot r'\Downarrow_{d\otimes c'}$ and, since $d \vdash c$ by the initial assumptions, $r\Downarrow_d = d \cdot r'\Downarrow_{d\otimes c'\otimes c} = d \cdot r'\Downarrow_{d\otimes c'} = r_1\Downarrow_d$, thus the inclusion of the prefixes directly holds. Otherwise, if $d \nVdash (\eta^+, \eta^-)$, then the operator $\Downarrow_d$ is undefined in both cases.

$r_1 = stutt(\eta^-) \cdot r'$ By definition, $r = (c, \eta^-) \rightarrowtail c \cdot r'\downarrow_c \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. If for all $h^- \in \eta^-$, $d \nvdash h^-$, then $r_1\Downarrow_d = d$ and it holds that its prefixes are all included in the prefixes of $r\Downarrow_d = d \cdot r\Downarrow_{d\otimes c}$. Otherwise, if it exists $h^- \in \eta^-$ such that $d \vdash h^-$, then the $\Downarrow_d$ operator is undefined in both cases.

Rule R4 The conditions to apply this rule are $\langle A_1, d\rangle \not\to$, $d \vdash c$ and $d \neq ff$, in which case the small-step behavior is defined as $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d = prefix(d \cdot d)$. There are two cases in which it may happen that $\langle A_1, d\rangle \not\to$:

$A_1 = \mathsf{skip}$ By (3.2), $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We now have that $r\Downarrow_d = d \cdot (d \otimes c) = d \cdot d$, whose prefixes coincide with $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$A_1 = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i$ and $\forall 1 \le i \le n \nvdash c_i$ By (3.7), $stutt(\{c_1, \ldots, c_n\}) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and, as a consequence, $r = (c, \{c_1, \ldots, c_n\}) \rightarrowtail c \cdot r'$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Now we compute $r\Downarrow_d$ and we get the trace $d \cdot r'\Downarrow_{d\otimes c} = d \cdot r'\Downarrow_d$. By definition of the evaluation function $\mathcal{A}$, r' is different from the empty conditional trace ϵ (by (3.7) a *stutt* construct is always followed by another conditional state). Therefore, $r'\Downarrow_d = d \cdot d \cdot s$ for some behavioral trace s. As a consequence, the behavior of the agent $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d = d \cdot d$ is included in the set of prefixes of $r\Downarrow_d = d \cdot d \cdot s$.

In case $d = ff$ we are not allowed to apply any rule in Figure 1, so the small-step behavior is $\mathcal{B}^{ss}[\![D\,.\,A]\!]_{ff} = \{\epsilon, ff\}$. In this case,

$A_1 = \mathsf{skip}$ since ff is strong enough to entail any guard of a generic agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i$. As explained above, $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (c, \varnothing) \twoheadrightarrow c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$, thus $r{\Downarrow_{\mathit{ff}}} = \mathit{ff}$, and it is easy to note that $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{\mathit{ff}} \in \mathit{prefix}(\mathit{ff})$.

Rule R5 This case is analogous to the case for Rule **R3** but, instead of executing the then branch (A_1), the else branch of the conditional agent (A_2) is taken, under the condition that $d \nvdash c$. More specifically, the conditions imposed for the application of the rule are $\langle A_2, d\rangle \to \langle A_2', d'\rangle$ and $d \nvdash c$, in which case $\mathcal{B}^{ss}[\![D.A]\!]_d = \mathcal{B}^{ss}[\![D.A_2]\!]_d$. By inductive hypothesis, we know that $\mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) \supseteq \mathcal{B}^{ss}[\![D.A_1]\!]_d$, thus also $\mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) \supseteq \mathcal{B}^{ss}[\![D.A]\!]_d$. In the following, we prove that $\mathit{prefix}(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) \supseteq \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d})$ when $d \nvdash c$. We proceed by induction on the structure of a generic $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ in order to find a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $\mathit{prefix}(r_2{\Downarrow_d}) \subseteq \mathit{prefix}(r{\Downarrow_d})$.

- **$r_2 = \boxtimes$** In this case, $r = (tt, \{c\}) \twoheadrightarrow tt \cdot \boxtimes$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We have $\boxtimes{\Downarrow_d} = d$, whose prefixes are included in those of $r{\Downarrow_d} = d \cdot d$.
- **$r_2 = (\eta^+, \eta^-) \twoheadrightarrow c' \cdot r'$** In this case, $r = (\eta^+, \eta^- \cup \{c\}) \twoheadrightarrow c' \cdot r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Let us now assume that $d \models (\eta^+, \eta^-)$; then, $r_2{\Downarrow_d} = d \cdot r'{\Downarrow_{d \otimes c'}}$. In addition, since by the initial assumptions $d \nvdash c$, $r{\Downarrow_d} = d \cdot r'{\Downarrow_{d \otimes c'}}$, the inclusion of the prefixes directly holds. Otherwise, if $d \not\models (\eta^+, \eta^-)$, then the operator ${\Downarrow_d}$ is undefined in both cases.
- **$r_2 = \mathit{stutt}(\eta^-) \cdot r'$** By definition, $r = (tt, \eta^- \cup \{c\}) \twoheadrightarrow tt \cdot r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Assume that for all $h^- \in \eta^-$, $d \nvdash h^-$. Then, $r_2{\Downarrow_d} = d$, and its prefixes are all included in the prefixes of $r{\Downarrow_d} = d \cdot r'{\Downarrow_d}$. Otherwise, if it exists $h^- \in \eta^-$ such that $d \vdash h^-$, then the ${\Downarrow_d}$ operator is undefined in both cases.

Rule R6 This case is analogous to the case for Rule **R4**. Now, the conditions to apply the rule are that $\langle A_2, d\rangle \not\to$ and $d \nvdash c$. In this case, the small-step behavior is $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d = \mathit{prefix}(d \cdot d)$. There are two cases in which it may happen that $\langle A_2, d\rangle \not\to$:

- **$A_2 = \mathsf{skip}$** By (3.2), $\boxtimes \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (tt, \{c\}) \twoheadrightarrow tt \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Then, since $d \nvdash c$, we have that $r{\Downarrow_d} = d \cdot d$, which coincides with $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.
- **$A_2 = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i$ and $\forall 1 \le i \le n \nvdash c_i$** By (3.7), $\mathit{stutt}(\{c_1, \ldots, c_n\}) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and, as a consequence, $r = (c, \{c_1, \ldots, c_n\}) \twoheadrightarrow c \cdot r'$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Now, we compute $r{\Downarrow_d}$ and we get as result the trace $d \cdot r'{\Downarrow_d}$. Since, by definition of the semantics evaluation function $\mathcal{A}$, a *stutt* is always followed by another conditional tuple, then r' is different from the empty trace. Therefore, $r'{\Downarrow_d} = d \cdot s$ for some trace s. As a consequence, the behavior of the agent $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d = d \cdot d$ is included in the set of prefixes of $r{\Downarrow_d} = d \cdot d \cdot s$.

$A_1 \parallel A_2$ We prove the two directions separately.

$\subseteq$ We distinguish five different cases. Let $r := r_1 \mathbin{\overline{\|}} r_2 \in \mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$.

$r = r_1$ By Definition 3.9, r_1 is a generic conditional trace and $r_2 = \boxtimes$ (or $r_2 = \epsilon$). In other words, r_2 is associated to an agent that adds no information. We have:

$$\begin{aligned} \mathit{prefix}((r_1 \mathbin{\overline{\|}} r_2)\Downarrow_d) &= \mathit{prefix}(r_1\Downarrow_d) \\ &= \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d \\ &\qquad [\text{by Inductive Hypothesis}] \\ &\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d \\ &= \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d \end{aligned}$$

Since A_2 does not modify the store, we can conclude that the two behaviors $\mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d$ and $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$ coincide.

$r = \mathit{stutt}(\eta_1^- \cup \eta_2^-) \cdot r'$ In case $d \nvDash h^- \ \forall h^- \in (\eta^- \cup \delta^-)$, we have that $\mathit{prefix}(r\Downarrow_d) = \mathit{prefix}(d) = \{\epsilon,\ d\} \subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$
Otherwise, $r\Downarrow_d$ is not defined, thus, the set $\mathit{prefix}(r\Downarrow_d)$ is empty and the inclusion directly holds.

$r = (\eta \otimes \delta) \rightarrowtail c_1 \otimes c_2 \cdot (r_1'\downarrow_{c_2} \mathbin{\overline{\|}} r_2'\downarrow_{c_1})$ By Definition 3.9, $r_1 = \eta \rightarrowtail c_1 \cdot r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, $r_2 = \eta \rightarrowtail c_2 \cdot r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $(c_1 \otimes c_2) \neq \mathit{ff}$. Let us distinguish three sub-cases.

$d \models (\eta \otimes \delta)$ and $d \neq \mathit{ff}$ Due to the form of r_1 and r_2, we know that there exist two agents A_1' and A_2' such that $\langle A_1, d\rangle \to \langle A_1', d \otimes c_1\rangle$ and $\langle A_2, d\rangle \to \langle A_2', d \otimes c_2\rangle$, respectively. Then,

$$\begin{aligned} &\mathit{prefix}(r\Downarrow_d) = \\ &= \mathit{prefix}(\{d \cdot (r_1'\downarrow_{c_2} \mathbin{\overline{\|}} r_2'\downarrow_{c_1})\Downarrow_{d\otimes c_1\otimes c_2} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [c_1 \text{ and } c_2 \text{ are already in the stores of } r_1 \text{ and } r_2, \text{ respectively}] \\ &= \mathit{prefix}(\{d \cdot (r_1'\downarrow_{c_1\otimes c_2} \mathbin{\overline{\|}} r_2'\downarrow_{c_1\otimes c_2})\Downarrow_{d\otimes c_1\otimes c_2} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Lemma A.3}] \\ &= \mathit{prefix}(\{d \cdot (r_1' \mathbin{\overline{\|}} r_2')\downarrow_{c_1\otimes c_2}\Downarrow_{d\otimes c_1\otimes c_2} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Lemma A.6}] \\ &= \mathit{prefix}(\{d \cdot (r_1' \mathbin{\overline{\|}} r_2')\Downarrow_{d\otimes c_1\otimes c_2} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Lemma A.8}] \\ &= \mathit{prefix}(\{d \cdot (r_1'\Downarrow_{d\otimes c_1\otimes c_2} \breve{\|} r_2'\Downarrow_{d\otimes c_1\otimes c_2}) \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Equation (3.1)}] \\ &= \{\epsilon, d\} \cup \{d \cdot (s_1' \breve{\|} s_2') \mid s_1' \in \mathit{prefix}(\mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d\otimes c_1\otimes c_2}), \\ &\qquad\qquad s_2' \in \mathit{prefix}(\mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d\otimes c_1\otimes c_2})\} \\ &\qquad [\text{by Inductive Hypothesis}] \\ &\subseteq \{\epsilon, d\} \cup \{d \cdot (s_1' \breve{\|} s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d\otimes c_1\otimes c_2},\ s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2']\!]_{d\otimes c_1\otimes c_2}\} \\ &\qquad [\text{by Definition A.7 and by Definition 3.1}] \\ &\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2']\!]_{d\otimes c_1\otimes c_2}\} \\ &\qquad [\text{by Rule } \mathbf{R7}] \end{aligned}$$

$\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$

$d = ff$ We have that $\langle A_1, ff\rangle \not\to$ and $\langle A_2, ff\rangle \not\to$, thus $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{ff} = \{\epsilon, ff\}$. Since $d \models (\eta \otimes \delta)$, we have that $prefix(r\Downarrow_{ff}) = \{\epsilon, ff\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{ff}$.

$d \not\models (\eta \otimes \delta)$ In this case the set $prefix(r\Downarrow_d)$ is empty since $\Downarrow_d$ is not defined under these conditions, thus it is directly included in $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$.

$r = (\eta \otimes \delta) \rightarrowtail ff \cdot \boxtimes$ By Definition 3.9 we have that $r_1 = \eta \rightarrowtail c_1 \cdot r_1'$, $r_2 = \delta \rightarrowtail c_2 \cdot r_2'$ and $c_1 \otimes c_2 = ff$. We have to consider three cases:

$d \models (\eta \otimes \delta)$ and $d \neq ff$

$$\begin{aligned} prefix(r\Downarrow_d) &= prefix(d \cdot c_1 \otimes c_2) \\ &= prefix(d \cdot ff) \\ &= \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2']\!]_{ff}\} \\ &\qquad [\text{by Rule } \mathbf{R7}] \\ &\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d \end{aligned}$$

In fact, also the second component of the behavior is the store ff. This case represents the situation in which the contribution of the two conditional traces results in an inconsistent conditional trace.

$d = ff$ We have that $\langle A_1, ff\rangle \not\to$ and $\langle A_2, ff\rangle \not\to$, thus $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{ff} = \{\epsilon, ff\}$. Since $d \models (\eta \otimes \delta)$, we have that $prefix(r\Downarrow_{ff}) = \{\epsilon, ff\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{ff}$.

$d \not\models (\eta \otimes \delta)$ In this case, $r\Downarrow_d$ is undefined, thus we have that $\varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$.

$r = (\eta^+, \eta^- \cup \delta^-) \rightarrowtail c_1 \cdot (r_1' \,\overline{\|}\, r_2'\downarrow_{c_1})$ By Definition 3.9, $r_1 = \eta \rightarrowtail c_1 \cdot r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, $r_2 = stutt(\delta^-) \cdot r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ with r_2' that recursively belongs to $\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and for all $h^- \in \delta^-$, $\eta^+ \nvdash h^-$. Let us distinguish three sub-cases.

$d \models (\eta^+, \eta^- \cup \delta^-)$. Then,

$$\begin{aligned} &prefix(r\Downarrow_d) \\ &= prefix(\{d \cdot (r_1' \,\overline{\|}\, r_2'\downarrow_{c_1})\Downarrow_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [c_1 \text{ is already contained in the stores of } r_1] \\ &= prefix(\{d \cdot (r_1' \,\overline{\|}\, r_2')\downarrow_{c_1}\Downarrow_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Lemma A.6}] \\ &= prefix(\{d \cdot (r_1' \,\overline{\|}\, r_2')\Downarrow_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Lemma A.8}] \\ &= prefix(\{d \cdot (r_1'\Downarrow_{d\otimes c_1} \,\breve{\|}\, r_2'\Downarrow_{d\otimes c_1}) \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\ &\qquad [\text{by Equation (3.1)}] \end{aligned}$$

$= \{\epsilon, d\} \cup \{d \cdot (s_1' \mathbin{\breve{\|}} s_2') \mid s_1' \in \mathit{prefix}(\mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d\otimes c_1}),\ s_2' \in \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d\otimes c_1})\}$
[by Inductive Hypothesis]
$\subseteq \{\epsilon, d\} \cup \{d \cdot (s_1' \mathbin{\breve{\|}} s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d\otimes c_1},\ s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_{d\otimes c_1}\}$
[by Definition A.7 and by Definition 3.1]
$\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2]\!]_{d\otimes c_1}\}$
[by Rule **R8**]
$\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$

$d = \mathit{ff}$ We have that $\langle A_1, \mathit{ff}\rangle \not\to$ and $\langle A_2, \mathit{ff}\rangle \not\to$, thus $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{\mathit{ff}} = \{\epsilon, \mathit{ff}\}$. Since $d \models (\eta \otimes \delta)$, we have that $\mathit{prefix}(r\Downarrow_{\mathit{ff}}) = \{\epsilon, \mathit{ff}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_{\mathit{ff}}$.

$d \not\models (\eta^+, \eta^- \cup \delta^-)$ In this case, we have that $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$.

$\supseteq$ In the following, we show that if $s \in \mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d$, then $s \in \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)$, i.e., we can find a conditional trace $r \in \mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $s \in \mathit{prefix}(r\Downarrow_d)$. We have four possible cases, depending on the rules defining the operational semantics for the agent.

1. If $\langle A_1,\ d\rangle \to \langle A_1',\ d_1'\rangle$ and $\langle A_2,\ d\rangle \to \langle A_2',\ d_2'\rangle$, the behavior of the parallel composition is $\mathcal{B}^{ss}[\![D.A_1 \parallel A_2]\!]_d = \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D.A_1' \parallel A_2']\!]_{d_1'\otimes d_2'}\}$. Let s be an element of that set. By inductive hypothesis, we know that there exist $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $d\cdot s_1' \in \mathit{prefix}(r_1\Downarrow_d)$ and $d\cdot s_2' \in \mathit{prefix}(r_2\Downarrow_d)$, with $s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_d$ and $s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2']\!]_d$. Now, consider $r = r_1 \mathbin{\bar{\|}} r_2$; this conditional trace belongs to $\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ whenever r_1 and r_2 are compatible via parallel composition (i.e., $r_1 \mathbin{\bar{\|}} r_2$ is a valid conditional trace). We show that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\|}} r_2)\Downarrow_d)$.

$\mathit{prefix}((r_1 \mathbin{\bar{\|}} r_2)\Downarrow_d)$
[by Lemma A.8]
$= \mathit{prefix}(r_1\Downarrow_d \mathbin{\breve{\|}} r_2\Downarrow_d)$
[by Definition 3.24]
$= \{\epsilon, d\} \cup \{(d\cdot s_1') \mathbin{\breve{\|}} (d\cdot s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d_1'}$ and $s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2']\!]_{d_2'}\}$
[by Definition A.7]
$= \{\epsilon, d\} \cup \{d\cdot s_1' \mathbin{\breve{\|}} s_2' \mid s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d_1'}$ and $s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2']\!]_{d_2'}\}$
[by Definition A.7 and by Definition 3.1]
$= \{\epsilon, d\} \cup \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d_1'} \mathbin{\breve{\|}} \mathcal{B}^{ss}[\![D \,.\, A_2']\!]_{d_2'}\}$
[by Rule **R7** and Equation (A.1)]
$= \{\epsilon, d\} \cup \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2']\!]_{d_1'\otimes d_2'}\}$

It follows directly that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\|}} r_2)\Downarrow_d)$.

2. If $\langle A_1,\ d\rangle \to \langle A_1',\ d_1'\rangle$ and $\langle A_2,\ d\rangle \not\to$, then Rule **R8** is applied and we have that $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d = \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2]\!]_{d_1'}\}$.

Let s be an element of that set. By inductive hypothesis, we know that it exists $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $d \cdot s_1' \in \mathit{prefix}(r_1\Downarrow_d)$ with $s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_d$. Moreover, it exists $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_2\Downarrow_d = d$. We distinguish two cases (corresponding to the two agents that can make the agent A_2 not to proceed) in order to prove that $s \in \mathit{prefix}((r_1 \,\bar{\|}\, r_2)\Downarrow_d)$.

A_2 = skip In this case, the behavior of the parallel composition is that of A_1 since A_2 makes no contribution to the computation. Then, $(r_1 \,\bar{\|}\, \boxtimes)\Downarrow_d = d \cdot s'$ with $s' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_d = \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2]\!]_d$, thus $s \in \mathit{prefix}((r_1 \,\bar{\|}\, \boxtimes)\Downarrow_d)$.

$A_2 = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i$ Consider $r_2 = \mathit{stutt}(\{c_1, \ldots, c_n\}) \cdot r_2'$ with $r_2' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$. We can assume that $d \nvdash c_i$ for all c_i, otherwise, the agent A_2 would proceed.

$$
\begin{aligned}
&\mathit{prefix}((r_1 \,\bar{\|}\, \mathit{stutt}(c_1, \ldots, c_n) \cdot r_2')\Downarrow_d) = \\
&= \mathit{prefix}(\{d \cdot (r_1' \,\bar{\|}\, r_2')\Downarrow_{d_1'} \mid r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Lemma A.8}] \\
&= \mathit{prefix}(\{d \cdot (r_1'\Downarrow_{d_1'} \,\breve{\|}\, r_2'\Downarrow_{d_1'}) \mid r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Equation (3.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot (s_1' \,\breve{\|}\, s_2') \mid s_1' \in \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d_1'}) \text{ and } \\
&\qquad\qquad\qquad\qquad s_2' \in \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d_1'})\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= \{\epsilon, d\} \cup \{d \cdot (s_1' \,\breve{\|}\, s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d_1'} \text{ and } s_2' \in \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_{d_1'}\} \\
&\qquad [\text{by Definition A.7 and by Definition 3.1}] \\
&= \{\epsilon, d\} \cup \{d \cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_{d_1'} \,\breve{\|}\, \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_{d_1'}\} \\
&\qquad [\text{by Rule } \mathbf{R7}\text{, Rule } \mathbf{R8} \text{ and Equation (A.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A_1' \parallel A_2]\!]_{d_1'}\}
\end{aligned}
$$

It follows directly that $s \in \mathit{prefix}((r_1 \,\bar{\|}\, r_2)\Downarrow_d)$.

3. If $\langle A_1,\ d\rangle \not\to$ and $\langle A_2,\ d\rangle \to \langle A_2', d_2'\rangle$, then the situation is symmetric to the previous case, so we can conclude that $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d \subseteq \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)$.
4. Finally, if $\langle A_1,\ d\rangle \not\to$ and $\langle A_2,\ d\rangle \not\to$, then we can reason similarly to Point 2, considering, for both A_1 and A_2, the two cases in which they cannot proceed. We can conclude that $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d = \{\epsilon,\ d\} \subseteq \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)$.

$\exists x\, A_1$ We prove the two directions independently.

$\subseteq$ We show that: $\mathit{prefix}(\mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$. Let $r = \bar{\exists}_x r_1$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and r_1 is x-self-sufficient. We show that the prefixes of $(\bar{\exists}_x r_1)\Downarrow_d$ are included in the behavior $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$ by structural induction on r_1:

$r_1 = \epsilon$ The statement directly holds.

$r_1 = \boxtimes$ Then, $\boxtimes\Downarrow_d = d$, which belongs to $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$.

$\boldsymbol{r_1 = \eta \nrightarrow l \cdot r_1'}$ By Definition 3.15, we have that $r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}$ and, by inductive hypothesis, there exists a transition $\langle A_1, d\rangle \to \langle A_1', d'\rangle$.
Since r_1 is x-self-sufficient, also r_1' is x-self-sufficient. Now, we have three cases.

$\boldsymbol{d \models \exists_x \eta}$ **and** $\boldsymbol{d \neq \mathit{ff}}$

$$
\begin{aligned}
&\mathit{prefix}(r\Downarrow_d)\\
&= \mathit{prefix}(\{\bar{\exists}_x(\eta \nrightarrow l \cdot r_1')\Downarrow_d \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_1' \ x\text{-self-sufficient}\})\\
&\qquad [\text{by Definition 3.24}]\\
&= \mathit{prefix}(\{d \cdot (\bar{\exists}_x r_1')\Downarrow_{d\otimes\exists_x l} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_1' \ x\text{-self-sufficient}\})\\
&\qquad [r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_1' \ x\text{-self-sufficient}]\\
&= \mathit{prefix}(\{d \cdot s \mid s \in (\mathcal{A}[\![\exists x\, A_1']\!]_{\mathcal{F}[\![D]\!]})\Downarrow_{d\otimes\exists_x l}\})\\
&\qquad [\text{by Equation (3.1)}]\\
&= \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathit{prefix}(\mathcal{A}[\![\exists x\, A_1']\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{d\otimes\exists_x l})\}\\
&\qquad [\text{by Inductive Hypothesis}]\\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1']\!]_{d\otimes\exists_x l}\}\\
&\qquad [\text{by Rule } \mathbf{R9}]\\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d
\end{aligned}
$$

$\boldsymbol{d = \mathit{ff}}$ We have that $\langle \exists x\, A_1, \mathit{ff}\rangle \not\to$, thus $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_{\mathit{ff}} = \{\epsilon, \mathit{ff}\}$. On the other hand, since $d \models \exists_x \eta$, we have that $\mathit{prefix}(r\Downarrow_{\mathit{ff}}) = \{\epsilon, \mathit{ff}\}$ which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_{\mathit{ff}}$.

$\boldsymbol{d \not\models \exists_x \eta}$ Then, the operator $\Downarrow_d$ is undefined for the conditional trace, thus $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$.

$\boldsymbol{r_1 = \mathit{stutt}(\{c_1, \ldots, c_n\}) \cdot r_1'}$ By Definition 3.15, $r_1' \in \mathcal{A}[\![\sum_{i=1}^{i} \mathsf{ask}(n_i) \to B_i]\!]_{\mathcal{F}[\![D]\!]}$. If it exists no index $1 \leq j \leq n$ such that $d \vdash c_j$, then this implies that $d \vdash \exists_x c_j$. In such case, we have

$$
\begin{aligned}
\mathit{prefix}(r\Downarrow_d) &= \mathit{prefix}(\bar{\exists}_x(\mathit{stutt}(\{c_1, \ldots, c_n\}) \cdot r_1')\Downarrow_d)\\
&= \mathit{prefix}(\mathit{stutt}(\{\exists_x c_1, \ldots, \exists_x c_n\}) \cdot \bar{\exists}_x r_1'\Downarrow_d)\\
&\qquad [\text{by Definition 3.24}]\\
&= d \subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d
\end{aligned}
$$

Otherwise, if it exists an index j such that $d \vdash c_j$, then $r\Downarrow_d$ is undefined, thus $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$.

$\supseteq$ From Rule **R9**, we know that, if $d \neq \mathit{ff}$, then $\mathcal{B}^{ss}[\![D \,.\, A_1]\!]_{l\otimes\exists_x d} = l' \cdot \mathcal{B}^{ss}[\![D \,.\, A_1']\!]_d$, where l and l' are local stores. Moreover, $l = \mathit{tt}$ because it is the initial (local) store for A_1. In the following, we show that $d \cdot \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1']\!]_{d\otimes\exists_x l} \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d$, i.e., it exists a trace $r \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r\Downarrow_d = d \cdot s$ with $s \in \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1']\!]_{d\otimes\exists_x l}$. By inductive hypothesis, $\mathcal{B}^{ss}[\![D \,.\, A_1]\!]_{\exists_x d} \subseteq \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{\exists_x d})$, and by Rule **R9**, it holds that there exists $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_1\Downarrow_{\exists_x d} = \exists_x d \cdot \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1']\!]_{l'}$.

Now, r_1 is x-self-sufficient since the only external information is provided by $\exists_x d$, which in fact does not contain information about x. Moreover, r_1 is of the form $\eta \rightarrowtail l' \cdot r_1'$ with $r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}$. Therefore, it exists $r \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r = \bar{\exists}_x\, r_1$. Then,

$$
\begin{aligned}
r{\Downarrow_d} &= (\bar{\exists}_x\, \eta \rightarrowtail l' \cdot r_1'){\Downarrow_d} \\
&= (\exists_x\, \eta \rightarrowtail \exists_x\, l' \cdot \bar{\exists}_x\, r_1'){\Downarrow_d} \\
&\qquad [\text{by Definition 3.24}] \\
&= d \cdot (\bar{\exists}_x\, r_1'){\Downarrow_{\exists_x\, l' \otimes d}} \\
&\qquad [\text{by Definition 3.24}] \\
&= d \cdot s \qquad \text{with } s \in \mathcal{A}[\![\exists x\, A_1']\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_{\exists_x\, l' \otimes d}} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= d \cdot s \qquad \text{with } s \in \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1']\!]_{\exists_x\, l' \otimes d}
\end{aligned}
$$

If $d = \mathit{ff}$, then we have that $\mathit{prefix}(r{\Downarrow_{\mathit{ff}}}) = \{\epsilon, \mathit{ff}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_{\mathit{ff}}$ since the transition relation $\rightarrow$ is not defined for $\langle \exists x\, A_1, \mathit{ff} \rangle$.

$p(\vec{x})$ We have to distinguish two sub-cases.

$d \neq \mathit{ff}$

$$
\begin{aligned}
\mathit{prefix}(\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) &= \mathit{prefix}(\{(tt, \varnothing) \rightarrowtail tt \cdot r' \mid r' \in \mathcal{F}[\![D]\!](p(\vec{x}))\}{\Downarrow_d}) \\
&\qquad [\text{since } \mathcal{F}[\![D]\!] = \mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}\,] \\
&= \mathit{prefix}(\{(tt, \varnothing) \rightarrowtail tt \cdot r' \mid r' \in \mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}(p(\vec{x}))\}{\Downarrow_d}) \\
&\qquad [\text{by Definition 3.19}] \\
&= \mathit{prefix}(\{(tt, \varnothing) \rightarrowtail tt \cdot r' \mid r' \in \mathcal{A}[\![B]\!]_{\mathcal{F}[\![D]\!]},\ p(\vec{x}) \coloneq B \in D\}{\Downarrow_d}) \\
&= \mathit{prefix}(\{d \cdot s' \mid s' \in (\mathcal{A}[\![B]\!]_{\mathcal{F}[\![D]\!]}){\Downarrow_d},\ p(\vec{x}) \coloneq B \in D\}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= \mathit{prefix}(\{d \cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, B]\!]_d,\ p(\vec{x}) \coloneq B \in D\}) \\
&\qquad [\text{by Rule } \mathbf{R10}] \\
&= \mathcal{B}^{ss}[\![D \,.\, p(\vec{x})]\!]_d
\end{aligned}
$$

Notice that, in the second last equality, the structural induction hypothesis cannot be applied because B can be structurally greater than $p(\vec{x})$. For this reason, we have to introduce a second induction on the number of $p(\vec{x})$ present on B. If B does not contain any process call $p(\vec{x})$, then we can directly apply structural induction. Otherwise, if the agent contains one process call $p(\vec{x})$, it is sufficient to replace the call with the body of the declaration. In this way, B has less process calls $p(\vec{x})$ than A and we can apply the inductive hypothesis.

$d = \mathit{ff}$ In this case, the transition relation $\rightarrow$ is not defined for the configuration $\langle p(\vec{x}), \mathit{ff} \rangle$, hence

$$
\begin{aligned}
\mathit{prefix}(\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_{\mathit{ff}}}) &= \mathit{prefix}(\{((tt, \varnothing) \rightarrowtail tt \cdot r'){\Downarrow_{\mathit{ff}}} \mid r' \in \mathcal{F}[\![D]\!](p(\vec{x}))\}) \\
&= \{\epsilon, \mathit{ff}\} \\
&= \mathcal{B}^{ss}[\![D \,.\, p(\vec{x})]\!]_{\mathit{ff}}
\end{aligned}
$$

Proof of Theorem 3.26.
By Theorem 3.25 it follows that for each program P and each $c \in \mathbf{C}$, $\mathit{prefix}(\mathcal{P}[\![P]\!]\Downarrow_c) = \mathcal{B}^{ss}[\![P]\!]_c$. Thus, we show that $\mathcal{P}[\![P_1]\!] = \mathcal{P}[\![P_2]\!] \iff \forall c \in \mathbf{C}.\, \mathit{prefix}(\mathcal{P}[\![P_1]\!]\Downarrow_c) = \mathit{prefix}(\mathcal{P}[\![P_2]\!]\Downarrow_c)$.

- $\Rightarrow$ Follows directly from Definition 3.24 and by definition of *prefix*.
- $\Leftarrow$ To prove this implication we first need to show that $\mathcal{P}[\![P_1]\!] \neq \mathcal{P}[\![P_2]\!] \Rightarrow \exists \bar{c} \in \mathbf{C}.\, \mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. Without loss of generality, assume that $\mathcal{P}[\![P_1]\!] \supset \mathcal{P}[\![P_2]\!]$, thus, it exists $r_1 \in \mathcal{P}[\![P_1]\!]$ such that $r_1 \notin \mathcal{P}[\![P_2]\!]$. We can distinguish two cases: $\mathcal{P}[\![P_2]\!]$ is empty or $\mathcal{P}[\![P_2]\!]$ contains at least one conditional trace.

  If $\mathcal{P}[\![P_2]\!] = \varnothing$, then $\mathcal{P}[\![P_2]\!]\Downarrow_c$ is empty for any possible $c \in \mathbf{C}$. Now, if we choose $\bar{c}$ to be the *lub* ($\otimes$) of all the positive conditions occurring in r_1, then $r_1\Downarrow_{\bar{c}}$ is a valid trace. Therefore, $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \supseteq \{r_1\Downarrow_{\bar{c}}\} \neq \varnothing$.

  If $\mathcal{P}[\![P_2]\!] \neq \varnothing$, by the initial assumptions, it exists a conditional trace $r_2 \in \mathcal{P}[\![P_2]\!]$ such that $r_1 \neq r_2$. Without loss of generality, assume that $\mathit{length}(r_1) \leq \mathit{length}(r_2)$ and that r_1 differs from r_2 at position k, with $k \in [1, \mathit{length}(r_1)]$. The index k is guaranteed to exist.[7] We consider the six possible cases, corresponding to the possible forms of the conditional state at position k, in order to prove that there exists a store $\bar{c}$ such that $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. In the following, the stores $\bar{c}_1$ and $\bar{c}_2$ correspond to the *lub* ($\otimes$) of all the positive conditions occurring in r_1 and r_2, respectively.

  1. Let be $(\eta_1^+, \eta_1^-) \rightarrowtail d_1$ and $(\eta_2^+, \eta_2^-) \rightarrowtail d_2$ the k-th conditional tuple in r_1 and r_2, respectively. There are three possible ways in which these two tuples can differ:

     $\boldsymbol{\eta_1^+ \neq \eta_2^+}$ Let us assume that $\eta_1^+ \vdash \eta_2^+$ and $\eta_2^+ \nvdash \eta_1^+$. Notice that r_1 has to come from the semantics of an ask or a now construct since they are the only *tccp* agents that can add information to the positive condition (see Definition 3.15). Hence, there exists also a conditional trace $\bar{r}_1 \in \mathcal{P}[\![P_1]\!]$ in which η_1^+ occurs in a negative condition (corresponding to the else branch of a now agent) or in a *stutt* construct (corresponding to the suspension of an ask agent) of the sequence. There are two cases in which $\bar{r}_1$ does not exists, but both are in contradiction with the hypothesis: (1) when $\eta_1^+ = tt$, but this contradicts $\eta_2^+ \nvdash \eta_1^+$ or (2) when a constraint d stronger than η_1^+ ($d \vdash \eta_1^+$) is propagated. In this last case, the trace $\bar{r}_1$ does not exists since the condition is in contradiction with the propagated store. However, since $\eta_1^+ \vdash \eta_2^+$, it follows that d entails also η_2^+ ($d \otimes \eta_1^+ = d \otimes \eta_2^+ = d$). Therefore, the propagation of d makes r_1 and r_2 equal. Since they were supposed to be different only at this point, this is a contradiction with the hypothesis $r_1 \neq r_2$. Therefore, $\bar{r}_1$ exists and belongs to $\mathcal{P}[\![P_1]\!]$. Furthermore, $\bar{r}_1$ differs from any trace in $\mathcal{P}[\![P_2]\!]$ for at least the negative part of a condition or the body of a *stutt*,

[7] There are two cases in which k does not exist, but both are in contradiction with the initial hypothesis: (1) $r_1 = r_2$ or (2) one of the traces is a prefix of the other.

otherwise, reasoning in a similar way as above, r_1 would also belong to $\mathcal{P}[\![P_2]\!]$, and this is not possible.
If $\eta_1^+ \nvdash \eta_2^+$ and $\eta_2^+ \vdash \eta_1^+$, we can reason in a symmetric way, thus concluding that it exists $\bar{r}_2 \in \mathcal{P}[\![P_2]\!]$ that differs from any trace in $\mathcal{P}[\![P_1]\!]$ for at least the negative part of a condition or the body of a *stutt*.
Finally, if $\eta_1^+ \nvdash \eta_2^+$ and $\eta_2^+ \nvdash \eta_1^+$, we can reason as before and deduce that there exist two traces $\bar{r}_1 \in \mathcal{P}[\![P_1]\!]$ and $\bar{r}_2 \in \mathcal{P}[\![P_2]\!]$, which contains respectively η_1^+ and η_2^+ in the negative part of the condition, and such that $\bar{r}_1 \notin \mathcal{P}[\![P_2]\!]$ and $\bar{r}_2 \notin \mathcal{P}[\![P_1]\!]$.
In case $\bar{r}_1$ (respectively $\bar{r}_2$) comes from an ask agent we remand to the following Points 2, 3 and 4 of the proof, where we deal with the conditional traces containing *stutt* constructs. Otherwise, if r_1 comes from a now agent we can reduce to the following case where we deal with the negative part of the conditions ($\eta_1^- \neq \eta_2^-$).

$\eta_1^- \neq \eta_2^-$ Let us first assume that $\eta_1^- \subset \eta_2^-$. This means that the store at position k in r_2 has to satisfy a stronger condition than the one in r_1. Let $\bar{c} := \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \setminus \eta_1^-$. Under these conditions, $r_1 \Downarrow_{\bar{c}}$ computes a behavioral timed trace whereas $r_2 \Downarrow_{\bar{c}}$ computes no trace since, at position k, $\bar{c}$ entails one of the stores in the negative condition.
For the case in which $\eta_2^- \subset \eta_1^-$ we choose $c = \bar{c}_2 \otimes h_1^-$, with $h_1^- \in \eta_1^- \setminus \eta_2^-$ and reason in an symmetric way.
Finally, if $\eta_1^- \not\subseteq \eta_2^-$ and $\eta_2^- \not\subseteq \eta_1^-$, we can choose indifferently $\bar{c} = \bar{c}_1 \otimes h_2^-$ or $\bar{c} = \bar{c}_2 \otimes h_1^-$ and conclude that $r_1 \Downarrow_{\bar{c}}$ computes a behavioral timed trace but $r_2 \Downarrow_{\bar{c}}$ is not defined, or vice-versa.
Thus, we can conclude that $\mathcal{P}[\![P_1]\!] \Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!] \Downarrow_{\bar{c}}$.

$d_1 \neq d_2$ Consider $\bar{c} = \bar{c}_1 = \bar{c}_2$. There are two possible cases. Assume first that $\bar{c} \nvdash d_1$ and $\bar{c} \nvdash d_2$. Both r_1 and r_2 must be *compatible* with their own conditions, thus, being the store monotonic, it happens that $r_1 \Downarrow_{\bar{c}}$ and $r_2 \Downarrow_{\bar{c}}$ are both defined. Moreover, we know that $\eta_1^+ = \eta_2^+$ and from Property A.1 $d_1 \vdash \eta_1^+$ and $d_2 \vdash \eta_1^+$. Since $\bar{c} \nvdash d_1$ and $\bar{c} \nvdash d_2$, we can conclude that in $r_1 \Downarrow_{\bar{c}}$ at position k we have the store d_1, whereas in $r_2 \Downarrow_{\bar{c}}$ at the same position we find the store d_2 that is different from d_1 by the initial assumptions. Thus $r_1 \Downarrow_{\bar{c}} \neq r_2 \Downarrow_{\bar{c}}$. Assume now that $\bar{c}$ contains more information than the store d_1 (respectively d_2). Then, we know that, at certain point in r_1 (respectively r_2), the positive condition is stronger than d_1 (respectively d_2). Therefore, we can reason as in the previous case when $\eta_1^+ \neq \eta_2^+$ and r_1 (respectively r_2) are produced by the semantics of an ask or a now agent.

2. Let $stutt(\eta_1^-)$ (respectively $stutt(\eta_2^-)$) be the k-th conditional state in r_1 (respectively r_2). It is sufficient to proceed as in Point 1 of this proof (case $\eta_1^- \neq \eta_2^-$) to show that there exists a store $\bar{c}$ such that $r_1 \Downarrow_{\bar{c}}$ is well defined while $r_2 \Downarrow_{\bar{c}}$ is not. For instance, if $\eta_1^- \subset \eta_2^-$ we set $\bar{c} = \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \setminus \eta_1^-$. It is easy to notice that $r_1 \Downarrow_{\bar{c}}$ computes a behavioral timed trace but $r_2 \Downarrow_{\bar{c}}$ recovers no trace since at position k the constraint h_2^- belongs to the negative part of the condition. Therefore, $\mathcal{P}[\![P_1]\!] \Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!] \Downarrow_{\bar{c}}$.

3. Let $\eta_1 \rightarrowtail d_1$ be the k-th conditional tuple in r_1 and $stutt(\eta_2^-)$ the k-th element in r_2. Consider $\bar{c} = \bar{c}_1$. Up to instant k, $r_1\Downarrow_{\bar{c}}$ and $r_2\Downarrow_{\bar{c}}$ coincide and, as r_1 and r_2 differ only at position k, $\bar{c}$ satisfies all the conditions in r_1 and in r_2 till up that position. The behavioral timed trace $r_2\Downarrow_{\bar{c}}$ ends at position k since a *stutt* has been encountered (see Definition 3.24). However, since r_1 is maximal, $r_1\Downarrow_{\bar{c}}$ does not end at position k but continues with at least another state, otherwise we would have found an ending symbol $\boxtimes$. In conclusion, $r_2\Downarrow_{\bar{c}}$ is at least one store longer than $r_1\Downarrow_{\bar{c}}$, thus, $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$.
4. If $\eta_2 \rightarrowtail d_2$ is the k-th element in r_2 and $stutt(\eta_1^-)$ that in r_1, then the proof is symmetric to previous Point 3.
5. Let $\boxtimes$ and $\eta_2 \rightarrowtail d_2$ be the k-th states of r_1 and r_2, respectively. We can reason similarly to Point 3 above in this proof, by choosing $\bar{c} = \bar{c}_2$. By hypothesis, r_1 and r_2 differ only at position k, thus, $r_1\Downarrow_{\bar{c}}$ and $r_2\Downarrow_{\bar{c}}$ compute the same behavioral timed trace up to position k-th. However, while $r_1\Downarrow_{\bar{c}}$ stops at instant k (an ending symbol $\boxtimes$ is found), $r_2\Downarrow_{\bar{c}}$ is at least one store longer. Thus, $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$.
6. Let $\boxtimes$ be the k-th element of r_1 and $stutt(\eta_2)$ the conditional state occurring in r_2 at the same position. We set $\bar{c} = \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \setminus \eta_1^-$. In this way, $r_1\Downarrow_{\bar{c}}$ is defined but $r_2\Downarrow_{\bar{c}}$ computes no trace since, at position k, the constraint h_2^- is required not to be entailed by the current store. Thus, $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$.

In conclusion, we can always choose an adequate $\bar{c}$ which differentiates $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}$ from $\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. From Definition 3.15 and Definition 3.19, it can be noticed that the traces contained in $\mathcal{P}[\![P_1]\!]$ and $\mathcal{P}[\![P_2]\!]$ either end in $\boxtimes$ or are infinite. From this observation, it follows directly that, if $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$, then $prefix(\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}) \neq prefix(\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}})$. Otherwise, there would exists a trace in $\mathcal{P}[\![P_1]\!]$ that is prefix of a trace in $\mathcal{P}[\![P_2]\!]$ (or viceversa), which is not possible since $\boxtimes$ is a termination symbol and an infinite trace cannot prefix another infinite trace. Thus, we can conclude that if $\mathcal{P}[\![P_1]\!] \neq \mathcal{P}[\![P_2]\!]$, then there exists $\bar{c} \in \mathbf{C}$ such that $prefix(\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}) \neq prefix(\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}})$, and this concludes the proof.

Proof of Proposition 3.27.

$\Rightarrow$ Straightforward.

$\Leftarrow$ By Definition 3.19, $\mathcal{P}[\![D_1\,.\,A]\!] = \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D_1]\!]}$ and $\mathcal{P}[\![D_2\,.\,A]\!] = \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D_2]\!]}$. We have to check that $\mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$. The only case depending on the interpretation is when $A = p(\vec{x})$. By hypothesis,

$$\begin{aligned}\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D_1]\!]} &= \bigsqcup\{(true, \varnothing) \rightarrowtail true \cdot r \mid r \in \mathcal{F}[\![D_1]\!](p(\vec{x}))\} \\ &= \bigsqcup\{(true, \varnothing) \rightarrowtail true \cdot r \mid r \in \mathcal{F}[\![D_2]\!](p(\vec{x}))\} = \mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D_2]\!]}\end{aligned}$$

We have to check that $\mathcal{F}[\![D_1]\!](p(\vec{x}))$ and $\mathcal{F}[\![D_2]\!](p(\vec{x}))$ coincide for each $p(\vec{x}) \in \mathbb{PC}$. Since $\mathcal{F}[\![D_1]\!]$ (respectively $\mathcal{F}[\![D_2]\!]$) is the least fixpoint of $\mathcal{D}[\![D_1]\!]_\perp$ (respectively $\mathcal{D}[\![D_2]\!]_\perp$), we know that it contains only information regarding the procedure calls in D_1 (respectively D_2). So we can conclude that $\mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$.

Proof of Corollary 3.28.
Consider $D_1, D_2 \in \mathbb{D}^{\Pi}_{\mathbf{C}}$:

$$
\begin{aligned}
D_1 \approx_{\mathcal{F}} D_2 &\Leftrightarrow \mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!] \\
&\quad [\text{by Proposition 3.27}] \\
&\Leftrightarrow \forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}}.\, \mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{P}[\![D_2 \,.\, A]\!] \\
&\quad [\text{by Theorem 3.26}] \\
&\Leftrightarrow \forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}} \forall c \in \mathbf{C}.\, \mathit{prefix}(\mathcal{P}[\![D_1 \,.\, A]\!]\Downarrow_c) = \mathit{prefix}(\mathcal{P}[\![D_2 \,.\, A]\!]\Downarrow_c) \\
&\quad [\text{by Theorem 3.25}] \\
&\Leftrightarrow \forall A \in \mathbb{A}^{\Pi}_{\mathbf{C}} \forall c \in \mathbf{C}.\, \mathcal{B}^{ss}[\![D_1 \,.\, A]\!]_c = \mathcal{B}^{ss}[\![D_2 \,.\, A]\!]_c \\
&\Leftrightarrow D_1 \approx_{ss} D_2
\end{aligned}
$$

A.2 Proofs of Section 4

Lemma A.9 $(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \underset{\alpha_{io}}{\overset{\gamma_{io}}{\leftrightarrows}} (\mathbb{IO}, \subseteq, \bigcup, \bigcap, \mathbf{IO}, \varnothing)$

Proof.
α_{io} is monotonic. Let $R_1, R_2 \in \mathbb{M}$ such that $R_1 \sqsubseteq R_2$, thus, $\alpha_{io}(R_1) \subseteq \alpha_{io}(R_2)$. Otherwise, if there exists an input-output pair belonging to $\alpha_{io}(R_1)$ but not to $\alpha_{io}(R_2)$, this means that the associated trace belongs to R_1 but not to R_2, and this contradicts the hypothesis.

γ_{io} is monotonic. Let $P_1, P_2 \in \mathbb{IO}$ such that $P_1 \subseteq P_2$. Suppose that $\gamma_{io}(P_1) \nsubseteq \gamma_{io}(P_2)$, in this case, there exists $r_1 \in \gamma_{io}(P_1)$ but not $r_2 \in \gamma_{io}(P_1)$ that extends r_1 (r_1 is a prefix of r_2). It is easy to see that this situation is impossible since, by the definition of γ_{io}, r_1 has to belong also to $\gamma_{io}(P_2)$ (since $P_1 \subseteq P_2$) and r_1 trivially extends itself.

$(\gamma_{io} \circ \alpha_{io})$ is extensive. This means that for all $R \in \mathbb{M}$, $R \sqsubseteq \gamma_{io}(\alpha_{io}(R))$. We show that $r \in R \Rightarrow r \in \gamma_{io}(\alpha_{io}(R))$; we distinguish three cases:

- **$r = \eta_1 \twoheadrightarrow c_1 \cdot \cdots \cdot \eta_n \twoheadrightarrow c_n \cdot \boxtimes$.** We have that $\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{fin}(c)\rangle \mid c_0 \in \mathbf{C} \text{ and } \mathit{last}(r\Downarrow_{c_0}) = c\}$. Thus, by (4.2), it follows that $r \in \gamma_{io}(\alpha_{io}(r))$.
- **$r = \eta_1 \twoheadrightarrow c_1 \cdot \cdots \cdot \mathit{stutt}(\eta_n^-) \cdot \ldots$.** We have that $\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{fin}(c)\rangle \mid c_0 \in \mathbf{C} \text{ and } \mathit{last}(r\Downarrow_{c_0}) = c\}$. From (4.2), it follows that $r \in \gamma_{io}(\alpha_{io}(r))$.
- **$r = \eta_1 \twoheadrightarrow c_1 \ldots \eta_n \twoheadrightarrow c_n \ldots$** (an infinite sequence that does not contain any *stutt*). We have that $\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{inf}(c)\rangle \mid c_0 \in \mathbf{C},\ r\Downarrow_{c_0} = c_0' \ldots c_i' \ldots, \text{ and } \otimes_{i\geq 0} c_i' = c\}$. By (4.2), we have that $r \in \gamma_{io}(\alpha_{io}(r))$.

$(\alpha_{io} \circ \gamma_{io})$ is the identity for $\mathbb{IO}$. This means that for all $P \in \mathbb{IO}$, $P = \alpha_{io}(\gamma_{io}(P))$. We show the two inclusions separately.

- $\subseteq$ We first show that $p \in P \Rightarrow p \in \alpha_{io}(\gamma_{io}(P))$ by distinguishing two sub-cases.
  - **$p = \langle c_0, \mathit{fin}(c_n)\rangle$.** In this case, $\gamma_{io}(P)$ contains all the conditional traces r such that $\mathit{last}(r\Downarrow_{c_0}) = c_n$. By (4.1), $p \in \alpha_{io}(\gamma_{io}(P))$.

$\boldsymbol{p = \langle c_0,\, inf(c)\rangle}$ We have that $\gamma_{io}(P)$ contains all the conditional state sequences r such that $r{\Downarrow_{c_0}} = c_0 \dots c_i \dots$ and $\otimes_{i\geq 0} = c$. By (4.1), $p \in \alpha_{io}(\gamma_{io}(P))$.

$\supseteq$ Now we show the other inclusion i.e., $p \in \alpha_{io}(\gamma_{io}(P)) \Rightarrow p \in P$. We have to consider two sub-cases.

$\boldsymbol{p = \langle c_0,\, fin(c_n)\rangle}$ In this case, it exists $r \in \gamma_{io}(P)$ such that $last(r{\Downarrow_{c_0}}) = c_n$. Obviously, $p \in P$, otherwise r would not belong to $\gamma_{io}(P)$.

$\boldsymbol{p = \langle c_0,\, inf(c)\rangle}$ In this case, it exists $r \in \gamma_{io}(P)$ such that $r{\Downarrow_{c_0}} = c_0 \dots c_i \dots$ and $\otimes_{i\geq 0} = c$. It is easy to notice that $p \in P$, otherwise, by using γ_{io}, we would not obtain r.

Proof of Proposition 4.4.
Consider $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$, then $\alpha_{io}(\mathcal{P}[\![D \,.\, A]\!]) = \mathcal{B}^{io}[\![D \,.\, A]\!]$. We show the two inclusions independently.

$\subseteq$ Let $r \in \mathcal{P}[\![D \,.\, A]\!]$ and $c_0 \in \mathbf{C}$ such that $r{\Downarrow_{c_0}}$ is defined. In order to show that $\alpha_{io}(\{r\}) \subseteq \mathcal{B}^{io}[\![D \,.\, A]\!]$, we distinguish two cases.

1. In case $r{\Downarrow_{c_0}}$ is finite, by (4.1), $\alpha_{io}(\{r\}) = \langle c_0, fin(c_n)\rangle \in \alpha_{io}(\mathcal{P}[\![D.A]\!])$, where $c_n \coloneqq last(r{\Downarrow_{c_0}})$. Moreover, by Definitions 3.19, 3.15 and 3.24, it is easy to notice that r must be of one of the following forms:
   (a) r ends with $\boxtimes$,
   (b) r contains a *stutt* or
   (c) r contains a conditional store $\eta \nrightarrow d$ such that there is no *stutt* before it and $c_0 \otimes d = ff$.

   Now, let us show that on the behavioral part, when A, with initial store c_0, behaves as $\langle A, c_0\rangle \rightarrow^* \langle A_n, c_n\rangle \not\rightarrow$ (the sequence is finite), r takes also one of those forms. Looking at the agent semantics $\mathcal{A}$ (Definition 3.15) we observe that:
   (a) we obtain a sequence that ends with $\boxtimes$ if a subagent of A is equal to skip or tell, this means that, starting from an initial store c_0 such that $last(r{\Downarrow_{c_0}})$ is well defined, the operational semantics cannot perform any step from the reached configuration $\langle \mathsf{skip}, c_n\rangle \not\rightarrow$;
   (b) when A contains an agent $\sum_{i=1}^{n} \mathsf{ask}(g_i) \rightarrow A_i$ and $\forall i \in [1, n].\, g_i \neq ff$, then a $stutt(\cup_{i=1}^{n})$ is introduced. Since we assume that $r{\Downarrow_{c_0}}$ is well defined, it holds that the guards are not entailed by c_0 (merged with the store produced by the sequence up to that position), thus the operational semantics cannot perform any step from the reached configuration $\langle \sum_{i=1}^{n} \mathsf{ask}(g_i) \rightarrow A_i, c_n\rangle \not\rightarrow$;
   (c) when r contains a conditional state $\eta \nrightarrow d$ (that occurs before any *stutt*) such that $c_0 \otimes d = ff$, we can deduce that, starting from $\langle A, c_0\rangle$, we reach in a finite number of operational steps the state $\langle A_n, ff\rangle \not\rightarrow$, from which no further derivation is possible since an inconsistent store has been produced.

   Thus, by Definition 4.2, $\langle c_0, fin(c_n)\rangle \in \mathcal{B}^{io}[\![D \,.\, A]\!]$.

2. In case $r{\Downarrow_{c_0}} = c_0 \cdots c_i \cdots$ is infinite, let us define $c \coloneqq \otimes_{i \geq 0} c_i$. By (4.1), $\alpha_{io}(\{r\}) = \langle c_0, \mathit{inf}(c) \rangle \in \alpha_{io}(\mathcal{P}[\![D \,.\, A]\!])$. By Theorem 3.25, it is easy to notice that $r{\Downarrow_{c_0}} \in \mathcal{B}^{ss}[\![D \,.\, A]\!]_{c_0}$, in fact, agent A with initial store c_0 behaves in the following way: $\langle A, c_0 \rangle \to \ldots \to \langle A_i, c_i \rangle \to \ldots$. By Definition 4.2, it follows that $\langle c_0, \mathit{inf}(c) \rangle \in \mathcal{B}^{io}[\![D \,.\, A]\!]$.

$\supseteq$ Let $p \in \mathbf{IO}$, we show that $p \in \mathcal{B}^{io}[\![D \,.\, A]\!] \Rightarrow p \in \alpha_{io}(\mathcal{P}[\![D \,.\, A]\!])$. Let us distinguish two cases.

$\underline{\boldsymbol{p = \langle c_0, \mathit{fin}(c_n) \rangle}}$ By Definition 4.2, it follows that $\langle A, c_0 \rangle \to \ldots \to \langle A_n, c_0 \rangle \not\to$, and by Definition 3.1, $c_0 \cdots c_n \in \mathcal{B}^{ss}[\![D \,.\, A]\!]_{c_0}$. By Theorem 3.25, it exists $r \in \mathcal{P}[\![D \,.\, A]\!]$ such that $r{\Downarrow_{c_0}} = c_0 \cdots c_n$, and by (4.1) it follows that $\langle c_0, \mathit{fin}(c_n) \rangle \in \alpha_{io}(\mathcal{P}[\![D \,.\, A]\!])$.

$\underline{\boldsymbol{p = \langle c_0, \mathit{inf}(c) \rangle}}$ By Definition 4.2, it follows that $\langle A, c_0 \rangle \to \ldots \to \langle A_i, c_i \rangle \to$, and by Definition 3.1, $c_0 \cdots c_i \cdots \in \mathcal{B}^{ss}[\![D \,.\, A]\!]_{c_0}$. By Theorem 3.25, it exists $r \in \mathcal{P}[\![D \,.\, A]\!]$ such that $r{\Downarrow_{c_0}} = c_0 \cdots c_i \cdots$, and by (4.1) it follows that $\langle c_0, \mathit{inf}(c) \rangle \in \alpha_{io}(\mathcal{P}[\![D \,.\, A]\!])$.

Proof of Theorem 4.6.

From Proposition 4.4 and by definition of π_F (Definition 4.2), for each *tccp* program P, $\pi_F(\alpha_{io}(\mathcal{P}[\![P]\!])) = \mathcal{B}^{io}_F[\![P]\!]$. Thus, it is sufficient to show that $\mathcal{O}^{io}[\![P_1]\!] = \mathcal{O}^{io}[\![P_2]\!] \iff \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) = \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$ for P_1 and P_2 *tccp* programs such that no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is a failed conditional trace. We prove the two directions separately.

$\Rightarrow$ We prove the equivalent implication: $\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!])) \Rightarrow \mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!]$. Let us assume, without loss of generality, that $\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \subset \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$, which means that there exist $r_2 \in \mathcal{P}[\![P_2]\!]$ and $c_0 \in \mathbf{C}$ such that $r_2{\Downarrow_{c_0}} = c_0 \cdots c_n$, but it does not exist $r_1 \in \mathcal{P}[\![P_1]\!]$ such that $r_1{\Downarrow_{c_0}} = c_0 \cdots c_n$. Furthermore, $c_n \neq \mathit{ff}$ since, by hypothesis, r_2 is not a failed conditional trace. By Theorem 3.25, $c_0 \cdots c_n \in \mathcal{B}^{ss}[\![P_2]\!]$ and, by Definition 4.2, $\langle c_0, c_n \rangle \in \mathcal{B}^{io}_F[\![P_2]\!]$. Since $\mathcal{B}^{io}_F$ and $\mathcal{O}^{io}$ differ only on sequences terminating in ff and $c_n \neq \mathit{ff}$, it follows that $\langle c_0, c_n \rangle \in \mathcal{O}^{io}[\![P_2]\!]$. On the other hand, we have that $c_0 \cdots c_n \notin \mathcal{B}^{ss}[\![P_1]\!]$, thus $\langle c_0, c_n \rangle \notin \mathcal{B}^{io}_F[\![P_1]\!]$. It is easy to see that, given a *tccp* program P, $\mathcal{O}^{io}[\![P]\!] \subseteq \mathcal{B}^{io}_F[\![P]\!]$, thus it holds that $\langle c_0, c_n \rangle \notin \mathcal{O}^{io}[\![P_1]\!]$. This means that $\langle c_0, c_n \rangle \in \mathcal{O}^{io}[\![P_2]\!] \setminus \mathcal{O}^{io}[\![P_1]\!]$ and we can conclude that $\mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!]$.

$\Leftarrow$ We prove the equivalent implication: $\mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!] \Rightarrow \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$. Without loss of generality, assume that $\mathcal{O}^{io}[\![P_1]\!] \subset \mathcal{O}^{io}[\![P_2]\!]$, thus, there exists $\langle c_0, c_n \rangle \in \mathcal{O}^{io}[\![P_2]\!]$ such that $\langle c_0, c_n \rangle \notin \mathcal{O}^{io}[\![P_1]\!]$. Since no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is failed, we can assume that $c_n \neq \mathit{ff}$. This means that, by using the transition relation defined in [14], we have a derivation of the form $\langle A_2, c_0 \rangle \to \ldots \langle A'_2, c_n \rangle \not\to$, with $A_2, A'_2 \in \mathbb{A}^{\Pi}_{\mathbf{C}}$, $D_2 \in \mathbb{D}^{\Pi}_{\mathbf{C}}$ and $P_2 = D_2 \,.\, A_2$; On the other hand, it can be noticed that, by using the transition relation of Figure 1, for P_1 there is no derivation starting with c_0 and ending in c_n. Thus, we have that $\langle c_0, c_n \rangle \in \mathcal{B}^{io}_F[\![P_2]\!]$ and $\langle c_0, c_n \rangle \notin \mathcal{B}^{io}_F[\![P_1]\!]$. From Proposition 4.4, it follows that $\langle c_0, c_n \rangle \in \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!])) \setminus \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!]))$ and we can conclude that $\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$.
