An Operational Semantics for Trust Policies

Karl Krukow

BRICS Report Series RS-05-30

ISSN 0909-0878 September 2005

An Operational Semantics for Trust Policies

Karl Krukow[*][†]

September 2005

Abstract

In the trust-structure model of trust management, principals specify their trusting relationships with other principals in terms of trust policies. In their paper on trust structures, Carbone et al. present a language for such policies, and provide a suitable denotational semantics. The semantics ensures that for any collection of trust policies, there is always a unique global trust-state, compatible with all the policies, specifying everyone's degree of trust in everyone else. However, as the authors themselves point out, the language lacks an operational model: the global trust-state is a well-defined mathematical object, but it is not clear how principals can actually compute it. This becomes even more apparent when one considers the intended application environment: vast numbers of autonomous principals, distributed and possibly mobile. We provide a compositional operational semantics for a language of trust policies. The operational semantics is given in terms of a composition of I/O automata. We prove that this semantics is faithful to its corresponding denotational semantics, in the sense that any run of the I/O automaton "converges to" the denotational semantics of the policies. Furthermore, as I/O automata are a natural model of asynchronous distributed computation, the semantics leads to an algorithm for distributedly computing the trust-state, which is suitable in the application environment.

[*]Supported by SECURE: Secure Environments for Collaboration among Ubiquitous Roaming Entities, EU FET-GC IST-2001-32486.

[†]BRICS, University of Aarhus.
BRICS: Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

Contents

1 Introduction

The trust-structure framework was introduced by Carbone, Nielsen and Sassone as a formal model for trust management in global computing environments [2]. In the framework, principals use "trust" as a means for decision-making about other principals. Trust is defined formally in terms of a *trust structure* T, of which a sub-component is a set D of so-called trust values. These trust values, specify the set of possible degrees of trust (or dis-trust) that a principal may have in another. As a simple example $D = \{\mathtt{high}, \mathtt{mid}, \mathtt{low}, \mathtt{unknown}\}$ could be a set of trust values, but certainly trust values may have a much richer structure. A principal's trust in other principals is given by its *trust policy.* In a very simple setting, a trust policy could be a function of type $\mathcal{P} \to D$ where $\mathcal{P}$ is the set of principal identities, i.e., mapping each principal identity to a trust value. However, in the intended global scenario, principals will often want to specify their trust contingent on the knowledge of some third-party (often having more detailed information about the subject). This feature is known as delegation in traditional trust management systems, and in the trust structure framework, it is called *policy referencing*. The idea is simple: principals may specify trust policies that refer to other principal's trust policies. Semantically, this means that trust policies are now functions mapping global trust-states $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to D$ to local trust-states $\mathsf{lts} : \mathcal{P} \to D$.

Trust policies are used for making decisions regarding interaction with other principals. At a high level, the intended mechanism is the following. When principal p needs to make a decision about whether and how to interact with another principal q, principal p will make this decision based on its trust value for q. Hence p must somehow obtain its trust value for q, e.g., by computing this value, or by looking it up in a precomputed store. Note that, because of policy references, principals generally need trust information from other principals to perform such a computation. Since principals are distributed, a computation of trust values becomes a distributed problem. The contribution of this paper is a solution to the problem of distributed trust-value computation. At first sight, this might seem trivial: when p needs to know about q's value for some principal, this value is simply sent. However, q's value may itself depend on other principal's trust policies, including p, which potentially gives cyclic dependencies. Semantically, this problem is elegantly solved by using domain theory, known from programming language semantics [11]. Essentially, the theory ensures that mutually recursive trust policies have a unique "least" solution. However, the theory gives no clue as to how principals can actually compute the trust values.

In the following, we present in more detail the trust-structure framework, explaining how the problem of cyclic trust policies is solved. Before presenting our actual contribution, we motivate further why computing the trust values is a non-trivial problem, especially in a global computing environment.

1.1 The trust-structure framework

In the framework of trust structures [2], trust is something which exists between *pairs of principals*; it is *quantified* and *asymmetric* in that we care of "how *much*" or "to what *degree*" principal p trusts principal q, which may not be to the same degree that q trusts p. Each application instance of the framework defines a so-called *trust structure*, $T = (D, \preceq, \sqsubseteq)$, which consists of a set D of *trust values*, together with two partial orderings of D, the trust ordering ($\preceq$) and the information ordering ($\sqsubseteq$). The elements $c, d \in D$ express the levels of trust that are relevant for the particular instance, and $c \preceq d$ means that d denotes at-least as high a trust degree as c. The information ordering introduces a notion of precision or refinement: $c \sqsubseteq d$ is intended to mean that c may be refined into d (given more information). As a simple example of a trust structure, consider the so-called "MN" trust-structure T_{MN} [5]. In this structure, trust values are pairs (m, n) of natural numbers, representing $m + n$ interactions with a principal; each interaction classified as either "good" or "bad". In a trust value (m, n), the first component, m, denotes the number of "good" interactions, and the second, the number of "bad" ones. The information-ordering is given by: $(m, n) \sqsubseteq (m', n')$ only if one can refine (m, n) into (m', n') by adding zero-or-more good interactions, and, zero-or-more bad interactions, i.e., iff $m \leq m'$ and $n \leq n'$. In contrast, the trust ordering is given by: $(m, n) \preceq (m', n')$ only if $m \leq m'$ and $n \geq n'$. For more examples of trust structures, see Carbone et al. [2], and Nielsen and Krukow [5, 9].

Global trust-states. Given a fixed trust structure $T = (D, \preceq, \sqsubseteq)$, and a set $\mathcal{P}$ of principal identities; a *global trust-state* of the system is a function $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to D$. The interpretation is that gts represents the trust state where p's trust in q (formalized as an element of D) is given by $\mathsf{gts}(p)(q)$. A good way of thinking about gts is to consider it a large matrix, indexed by pairs of principal identities, in which the row indexed by principal p (denoted $\mathsf{gts}(p)$) contains principal p's trust in any other principal. For example, in the row $\mathsf{gts}(p)$, column q represents p's trust in q, given as an element in the set D; this entry is denoted $\mathsf{gts}(p)(q)$ ("row vectors" like $\mathsf{gts}(p)$ are also called *local* trust-states). Thus, the matrix gts gives a complete (system global) description of how everyone trusts everyone else. We shall write $\mathtt{GTS}$ for the set of global trust-states $\mathcal{P} \to \mathcal{P} \to D$. Similarly we write $\mathtt{LTS}$ for the set $\mathcal{P} \to D$ of *local* trust-states (corresponding to rows of gts matrices).

Trust policies. The goal of the trust-structure framework is to define, at any time, a global trust state $\overline{\mathsf{gts}}$, thus giving a precise meaning to "p's trust in q" as the trust value $\overline{\mathsf{gts}}(p)(q)$. In order to uniquely define the global trust state $\overline{\mathsf{gts}}$, an approach similar to that of Weeks [10] is adopted. Each principal $p \in \mathcal{P}$ defines a *trust policy* which is a function π_p of type $\mathtt{GTS} \to \mathtt{LTS}$, i.e. taking a global matrix as input, and providing a local row-*vector* as output. This function then determines p's trust-row within the unique global trust-matrix, i.e. determines row $\overline{\mathsf{gts}}(p)$, as follows. In the simplest case, π_p could be a constant function, ignoring its first argument $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to D$. As an example, $\pi_p(\mathsf{gts}) = \lambda q.t_0$ (for some $t_0 \in D$) defines p's trust in any $q \in \mathcal{P}$ as the constant t_0. In general we allow a form of *delegation* called *policy reference*: policy π_p may refer to other policies (π_z, $z \in \mathcal{P}$), e.g., p might trust q to download if A or B trusts q to download. The general interpretation of π_p is the following. *Given that all principals assign trust-values as specified in the global trust-state* gts, *then* p *assigns trust values as specified in vector* $\pi_p(\mathsf{gts}) : \mathcal{P} \to D$. For example, function $\pi_p(\mathsf{gts})(q) = (\mathsf{gts}(A)(q) \vee_{\preceq} \mathsf{gts}(B)(q)) \wedge_{\preceq} \mathtt{download}$, represents a policy saying "for any $q \in \mathcal{P}$, the trust in q is the least upper-bound of what A and B say, but no more than the constant $\mathtt{download}$."[1]

Unique trust-state. The collection of all trust policies, $\Pi = (\pi_p | p \in \mathcal{P})$, thus "spins a global web-of-trust" in which the trust policies mutually refer to each other. Since trust policies Π may give rise to cyclic policy-references, it is not *a priori* clear how to define the unique global trust-state $\overline{\mathsf{gts}}$ for a given collection of trust policies Π. One may consider the unique function $\Pi_\lambda = \langle \pi_p | p \in \mathcal{P} \rangle$, of type $\mathtt{GTS} \to \mathtt{GTS}$ with the property that $\mathtt{Proj}_p \circ \Pi_\lambda = \pi_p$ for all $p \in \mathcal{P}$, where $\mathtt{Proj}_p$ is the p'th projection.[2] Intuitively, the function Π_λ is easy to understand: each π_p maps a matrix $\mathsf{gts} \in \mathtt{GTS}$ to a "row-vector" $\pi_p(\mathsf{gts})$ in $\mathtt{LTS}$; on input gts, function Π_λ builds the output matrix from all these rows by taking the p'th row of the output matrix to be $\pi_p(\mathsf{gts})$. We can now state a minimal requirement that the unique trust state, $\overline{\mathsf{gts}}$, should satisfy: $\overline{\mathsf{gts}}$ *should be consistent with all policies* π_p. This amounts to requiring that it should satisfy the following fixed-point equation: $\mathsf{gts}(p) = \pi_p(\mathsf{gts})$ for all $p \in \mathcal{P}$; or equivalently:

$$\Pi_\lambda(\mathsf{gts}) \quad = \quad \mathsf{gts}$$

Any matrix $\mathsf{gts} : \mathtt{GTS}$ satisfying this equation is *consistent* with the policies $(\pi_p | p \in \mathcal{P})$, i.e. row p of gts is consistent with π_p in that, if all principals trust as specified in gts, then p trusts as specified in $\pi_p(\mathsf{gts})$

[1] Assuming some appropriate trust structure with $\mathtt{download} \in D$, and where least upper-bounds and greatest-lower bounds exist with respect to $\preceq$.

[2] $\mathtt{Proj}_p$ is given by: for all $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to D$. $\mathtt{Proj}_p(\mathsf{gts}) = \mathsf{gts}(p)$.

which (by the fixed-point equation) can be read-off as the pth row of gts. This means that *any* fixed point of Π_λ is consistent with all policies π_p. But arbitrary functions Π_λ, may have multiple or even no fixed points.

Here we appeal to the power of the mathematical theory of complete partial orders and continuous functions, used e.g. in formal programming language semantics [11]. A crucial requirement in the trust-structure framework is that the information ordering $\sqsubseteq$ makes $(D, \sqsubseteq)$ a complete partial order (cpo) with a least element (this element is denoted $\perp_\sqsubseteq$, and can be thought of as a value representing "unknown"). We require also that all policies $\pi_p : \mathtt{GTS} \longrightarrow \mathtt{LTS}$ are *information continuous*, i.e. continuous with respect to $\sqsubseteq$.[3] Since this implies that Π_λ is also information-continuous, and since $(\mathtt{GTS}, \sqsubseteq)$ is a cpo with bottom, standard theory [11] tells us that Π_λ has a (unique) least fixed-point which we denote $\mathsf{lfp}_\sqsubseteq \Pi_\lambda$ (or simply $\mathsf{lfp}\, \Pi_\lambda$):

$$\mathsf{lfp}_\sqsubseteq \Pi_\lambda = \bigsqcup\nolimits_\sqsubseteq \{\Pi_\lambda^i(\lambda p.\lambda q.\perp_\sqsubseteq) \mid i \in \mathbb{N}\}$$

This global trust-state has the property that it is a fixed-point (i.e., $\Pi_\lambda(\mathsf{lfp}_\sqsubseteq \Pi_\lambda) = \mathsf{lfp}_\sqsubseteq \Pi_\lambda$) and that is is the (information-) least among fixed-points (i.e., for any other fixed point gts, $\mathsf{lfp}_\sqsubseteq \Pi_\lambda \sqsubseteq \mathsf{gts}$). Hence, for any collection Π of trust policies, we can define the *global trust-state induced by that collection*, as $\overline{\mathsf{gts}} = \mathsf{lfp}\, \Pi_\lambda$, which is well-defined by uniqueness.

Consider now two mutually referring functions π_p and π_q, given by $\pi_p(\mathsf{gts}) = \mathtt{Proj}_q(\mathsf{gts})$, and $\pi_q(\mathsf{gts}) = \mathtt{Proj}_p(\mathsf{gts})$. Intuitively, there is no information present in these functions; p delegates all trust-questions to q, and similarly q delegates to p. In this case, we would like the global trust-state $\overline{\mathsf{gts}}$ induced by the functions to take the value $\perp_\sqsubseteq$ on any entry $z \in \mathcal{P}$ for both p and q, i.e., for both $x = p$ and $x = q$ and for all $z \in \mathcal{P}$ we should have $\overline{\mathsf{gts}}(x)(z) = \perp_\sqsubseteq$. This is exactly what is obtained by choosing the information-*least* fixed-point of Π_λ.

1.2 The operational problem

Many interesting systems are instances of the trust-structure framework [2, 5, 9], but one could argue against its usefulness as a basis for the actual construction of trust-management systems. In order to make security decisions, each principal p will need to reason about its trust in

[3]We overload $\sqsubseteq$ (respectively $\preceq$) to denote also the pointwise extension of $\sqsubseteq$ ($\preceq$) to the function space $\mathtt{LTS} = \mathcal{P} \to D$ as well as to $\mathtt{GTS} = \mathcal{P} \to \mathcal{P} \to D$. Saying that a policy is information-continuous means that the function is continuous w.r.t. $\sqsubseteq$.

others, that is, the values of $\overline{\mathsf{gts}}(p)$. While the framework does ensure the existence of a unique (theoretically well-founded) global trust-state, it is not "operational" in the sense of providing a way for principals to actually *compute* the trust values. Furthermore, as we shall argue in the following, the standard way of computing least fixed-points is inadequate in our scenario.

When the cpo $(D, \sqsubseteq)$ is of finite height h, the cpo $(\mathcal{P} \to \mathcal{P} \to D, \sqsubseteq)$ has height $|\mathcal{P}|^2 \cdot h$ (the height of a cpo is the size of its longest chain). In this case, the least fixed-point of Π_λ can, *in principle*, be computed by finding the first identity in the chain of approximants $(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda^2(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \cdots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p.\lambda q.\bot_\sqsubseteq)$ [11]. However, in the environment envisioned, such a computation is infeasible. The functions $(\pi_p : p \in \mathcal{P})$ defining Π_λ are distributed throughout the network, and, more importantly, even if the height h is finite, the number of principals $|\mathcal{P}|$, though finite, will be *very* large. Furthermore, even if resources were available to make this computation, we can not assume that any central authority is present to perform it. Finally, since each principal p defines its trust policy π_p autonomously, an inherent problem with trying to compute the fixed point is the fact that p might decide to change its policy π_p to π'_p at any time. Such a policy update would be likely to invalidate data obtained from a fixed-point computation done with global function Π_λ, i.e., one might not have time to compute $\mathsf{lfp}\,\Pi_\lambda$ before the policies have changed to Π'.

While the above discussion indicates that exact computation of the fixed point is infeasible (and hence that the framework is not suitable as an operational model), in many applications, it is often sufficient to merely *approximate* the fixed-point value. Krukow and Twigg present a collection of techniques for approximating the idealized fixed-point $\mathsf{lfp}\,\Pi_\lambda$ [6]. Among these techniques is an asynchronous algorithm which distributedly computes the least fixed-point of a collection of policies, assuming that these policies remain fixed throughout the computation. While Krukow and Twigg argue that the algorithm is correct at an abstract level, there is a logical gap between the algorithm-description and the abstract model of reasoning; and the algorithm itself is described with an informal "pseudo-notation" which doesn't have a formal semantics.

Contribution and Structure. The purpose of this report is to make precise the mentioned distributed algorithm, and to "fill the logical gap." More precisely, we describe a general language for specifying trust policies, and provide it with a compositional operational semantics. The semantics of a collection of policies will be defined by translation into an I/O automaton [7, 8], formalizing the asynchronous distributed algorithm of Krukow and Twigg [6] in a semantic model. Our main theorem (Theorem 5.1) proves in this *formal* model, that even in infinite height cpos the I/O automata will converge towards the least fixed-point, as intended.

As mentioned, the semantics of policies is given in terms of I/O automata, and there are two main reasons for this. First, I/O automata are a natural model of asynchronous distributed algorithms which results in relatively simple automata for describing the fixed-point algorithm. Secondly, the model is operational and relatively low-level, which means that there is a short distance between the semantic model and an actual implementation that can run in real distributed systems. However, the relatively complex reasoning about the algorithm is best done at a more abstract level, and hence we introduce the more abstract model of Bertsekas Abstract Asynchronous Systems (BAASs), together with a "simulation-like" relation from the concrete I/O automata to the BAAS. Although the main theorem does not mention the abstract model, its proof uses this model together with the "simulation," to prove its statement about the actual operational semantics.

2 A Basic Language for Trust Policies

In this section we present a simple language for writing trust policies. The language is similar to that of Carbone et al. [2], but simplified slightly. We provide a denotational semantics for the language which is similar to the denotational semantics of Carbone et al. Throughout this paper we let $\mathcal{P}$ be a finite set of principal identities, and $(D, \sqsubseteq, \preceq)$ be a trust structure.

2.1 Syntax

We assume a countable collection of n'ary function symbols op_n^i for each $n > 0$. These are meant to denote functions $[\![\mathsf{op}_n^i]\!]^{\mathrm{den}} : D^n \to D$, continuous with respect to $\sqsubseteq$. The syntax of our simple language is given in Figure 1. A policy π is essentially a list of pairs $p : \tau$, where p is a principal identity, and τ is an expression defining the policy's trust-specification for p. Since we cannot assume that the writer of the policy

$$
\begin{array}{lll}
\pi ::= & \star : \tau & \text{(default policy, } \star \notin \mathcal{P}\text{)} \\
 & \mid p : \tau, \pi & \text{(specific policies, } p \in \mathcal{P}\text{)} \\
 & & \\
\tau ::= & d & \text{(constant, } d \in D\text{)} \\
 & \mid p?q & \text{(policy reference, } p, q \in \mathcal{P} \cup \{\star\}\text{)} \\
 & \mid \mathtt{op}_n^i(\tau_1, \tau_2, \ldots, \tau_n) & \text{(}n\text{'ary continuous operator)}
\end{array}
$$

Figure 1: A basic policy language.

knows all principals, we include a generic construct $\star : \tau$, which intuitively means "for everyone not mentioned explicitly in this policy, the trust specification is τ." Note this could easily be extended to more practical constructs, say $G : \tau$ meaning that τ is the trust-specification for any member of the group G.

The syntactic category τ represents trust specifications. In this language, the category is very general and simple. We have constants $d \in D$, which are meant to be interpreted as themselves, e.g., $p : d$ means "the trust in p is d." Construct $p?q$ is the policy reference; it is meant to refer to "principal p's trust in principal q", e.g., $r : p?q$ says that "the trust in r is what-ever p's trust in q is." Finally $\mathtt{op}_n^i(\tau_1, \ldots, \tau_n)$ is the application of operator $[\![\mathtt{op}_n^i]\!]^{\mathrm{den}}$ to the trust specifications $(\tau_1, \ldots, \tau_n)$. For example, if $(D, \preceq)$ is a lattice, this could be the n'ary least upper bound (provided this is continuous with respect to $\sqsubseteq$).

We say that a policy is well-formed if there are no double occurrences of a principal identity, say, $p : \tau$ and $p : \tau'$. We assume that all policies are well-formed throughout this paper.

2.2 Denotational semantics

The denotational semantics of the basic policy language is given in figures 2, 3 and 4. We assume that for each of the function symbols $\mathtt{op}_n^i$, $[\![\mathtt{op}_n^i]\!]^{\mathrm{den}}$ is a $\sqsubseteq$-continuous function of type $D^n \to D$. The semantics follows the ideas of Carbone et al., presented in the introduction. For a collection $\Pi = (\pi_p \mid p \in \mathcal{P})$, the semantics of each π_p is an information-continuous function $[\![\pi_p]\!]^{\mathrm{den}}$ of type $\mathtt{GTS} \to \mathtt{LTS}$. As expected, the denota-

$$
\begin{array}{rcl}
[\![\star : \tau]\!]^{\mathrm{den}}\ \mathsf{gts}\ q & = & [\![\tau]\!]^{\mathrm{den}}\ ([\star \mapsto q]/id_{\mathcal{P}})\ \mathsf{gts} \\
[\![p : \tau, \pi]\!]^{\mathrm{den}}\ \mathsf{gts}\ q & = & \texttt{if}\ (q = p)\ \texttt{then}\ [\![\tau]\!]^{\mathrm{den}}\ ([\star \mapsto p]/id_{\mathcal{P}})\ \mathsf{gts} \\
 & & \qquad\qquad \texttt{else}\ [\![\pi]\!]^{\mathrm{den}}\ \mathsf{gts}\ q
\end{array}
$$

Figure 2: Denotational semantics of the policy language. For syntactic category π, $[\![\pi]\!]^{\mathrm{den}}$ is a continuous function of type $(\mathcal{P} \to \mathcal{P} \to D) \to \mathcal{P} \to D$. The term $id_{\mathcal{P}}$ denotes the identity function on $\mathcal{P}$.

$$
\begin{array}{rcl}
[\![d]\!]^{\mathrm{den}}\ env & = & \lambda \mathsf{gts}.\ d \\
[\![p?q]\!]^{\mathrm{den}}\ env & = & \lambda \mathsf{gts}.\ \mathsf{gts}\ (env\ p)\ (env\ q) \\
[\![\mathsf{op}^i_n(\tau_1, \ldots, \tau_n)]\!]^{\mathrm{den}}\ env & = & [\![\mathsf{op}^i_n]\!]^{\mathrm{den}} \circ \left\langle [\![\tau_1]\!]^{\mathrm{den}}\ env, \ldots, [\![\tau_n]\!]^{\mathrm{den}}\ env \right\rangle
\end{array}
$$

Figure 3: Denotational semantics of the policy language. For syntactic category τ, when env is an environment, i.e., a function of type $\mathcal{P} \cup \{\star\} \to \mathcal{P}$ then $[\![\tau]\!]^{\mathrm{den}}\ env$ is a continuous function of type $(\mathcal{P} \to \mathcal{P} \to D) \to D$.

tional semantics of the collection Π is the least fixed-point of the function $\Pi_\lambda = \left\langle [\![\pi_p]\!]^{\mathrm{den}} \mid p \in \mathcal{P} \right\rangle$.

3 Two Models of Distributed Computation

In this section, we describe two models of distributed computation. The models will be used in the next sections, where we provide an operational semantics for the basic policy language of Section 2. The operational semantics is given by two translations into the respective structures of each of these models. More specifically, in the operational semantics, a principal-indexed collection of policies Π is translated into an I/O Automaton, denoted $[\![\Pi]\!]^{\mathrm{op}}$. I/O Automata are a form of labeled transition-system, suitable for modelling and reasoning about distributed discrete

$$
[\![(\pi_p \mid p \in \mathcal{P})]\!]^{\mathrm{den}} \quad = \quad \mathsf{lfp}_{\sqsubseteq} \left\langle [\![\pi_p]\!]^{\mathrm{den}} \mid p \in \mathcal{P} \right\rangle
$$

Figure 4: Denotational semantics of the policy language.

event systems [7,8]. We shall define also another translation $[\![\Pi]\!]^{\text{op-abs}}$ into what we call a Bertsekas Abstract Asynchronous System (BAAS). There will be a tight correspondence between the "abstract" operational semantics $[\![\Pi]\!]^{\text{op-abs}}$ and the actual operational semantics $[\![\Pi]\!]^{\text{op}}$. The reason for introducing $[\![\cdot]\!]^{\text{op-abs}}$ is to make reasoning about the actual operational semantics easier. More specifically, we will make use of a general convergence result of Bertsekas for BAAS's. By virtue of the connection between the semantics, this result translates into a result about the runs of the concrete I/O automaton $[\![\Pi]\!]^{\text{op}}$.

We now present the I/O automaton model, and the Bertsekas abstract systems.

3.1 The I/O automata model

We review the basic definitions of I/O automata. For a more in-depth treatment, we refer to Lynch's book [7]. An I/O automaton is a (possibly infinite) state automaton, where transitions are labeled with so-called actions. There are three types of actions: input, output and internal. An important feature of I/O automata is that input-actions are always enabled. This property means that while the automaton can put restrictions on when output and internal actions are performed, it cannot control when input actions are performed. Instead, this is controlled by the environment.

An *action signature* S is given by a set $acts(S)$ of actions, and a partition of this set into three sets $in(S)$, $out(S)$ and $int(S)$ of input, output and internal actions, respectively. We denote by $local(S) = out(S) \cup int(S)$ the set of locally controlled actions.

Definition 3.1 (I/O Automaton). An *input/output automaton* A, consists of five components:

$$A = (sig(A), states(A), start(A), steps(A), part(A))$$

The components are: an action signature $sig(A)$, a set of states $states(A)$, a non-empty set of start states $start(A) \subseteq states(A)$, a transition relation $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, satisfying that for every $s \in states(A)$ and every input action $a \in in(sig(A))$ there exists $s' \in states(A)$ so that $(s, a, s') \in steps(A)$. Finally, $part(A)$ is an equivalence relation, partitioning the set $local(sig(A))$ into at most countably many classes.

A *run* r of an I/O automaton A is a sequence $r = s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n$ or an infinite sequence $r = s_0 a_1 s_1 a_2 s_2 \cdots$, so that $s_0 \in \mathit{start}(A)$ and for all i, $(s_i, a_{i+1}, s_{i+1}) \in \mathit{steps}(A)$. For a finite run $r = s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n$, the length of r, is the number of state occurrences, i.e., $|r| = n + 1$. For infinite runs r, we write $|r| = \infty$.

A finite run r of A is *fair* if for every class C of $\mathit{part}(A)$, we have that no action of C is enabled in the final state of r. An infinite run r is *fair* if for every class C of $\mathit{part}(A)$ then either r contains infinitely many events from C, or r contains infinitely many occurrences of states in which no action of C is enabled.

Composition. Compatible I/O automata can be composed to form larger I/O automata. Composition of I/O automata is defined for countable sets of automata, so let $C = (S_i)_{i \in I}$ be a countable collection of action signatures. Say that C is compatible if for all $i, j \in I$ with $i \neq j$ we have

1. $\mathit{out}(S_i) \cap \mathit{out}(S_j) = \emptyset$, and
2. $\mathit{int}(S_i) \cap \mathit{acts}(S_j) = \emptyset$

Let $(A_i \mid i \in I)$ be a countable collection of I/O automata with $(\mathit{sig}(A_i) \mid i \in I)$ being compatible. Writing S_i for $\mathit{sig}(A_i)$, the *composition signature* $S = \prod_{i \in I} S_i$ is the action signature with

1. $\mathit{in}(S) = \left(\bigcup_{i \in I} \mathit{in}(S_i)\right) \setminus \bigcup_{i \in I} \mathit{out}(S_i)$
2. $\mathit{out}(S) = \bigcup_{i \in I} \mathit{out}(S_i)$
3. $\mathit{int}(S) = \bigcup_{i \in I} \mathit{int}(S_i)$

For a countable collection $(A_i \mid i \in I)$ of automata with compatible signatures $S_i = \mathit{sig}(A_i)$, their composition, A, is denoted $A = \prod_{i \in I} A_i$. The composition is the I/O automaton defined as follows.

1. $\mathit{sig}(A) = \prod_{i \in I} \mathit{sig}(A_i)$, i.e., $\mathit{sig}(A)$ is the composition signature of $(S_i \mid i \in I)$.
2. $\mathit{states}(A) = \prod_{i \in I} \mathit{states}(A_i)$. We use $\bar{s}$ to denotes elements of the Cartesian product. If $\bar{s} \in \mathit{states}(A)$ then $\bar{s}_i$ refers to the ith component of $\bar{s}$.
3. $\mathit{start}(A) = \prod_{i \in I} \mathit{start}(A_i)$

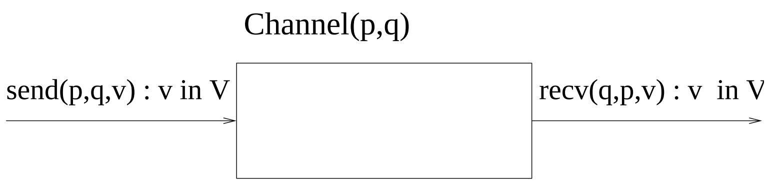

Figure 5: The $\texttt{Channel}(p, q)$ I/O automaton interface.

4. $\mathit{steps}(A)$ is the set of triples $(\bar{s}, a, \bar{s}')$ so that, for all $i \in I$, if $a \in \mathit{acts}(S_i)$ then $(\bar{s}_i, a, \bar{s}'_i) \in \mathit{steps}(A_i)$, and if $a \notin \mathit{acts}(S_i)$ then $\bar{s}_i = \bar{s}'_i$.

5. $\mathit{part}(A) = \bigcup_{i \in I} \mathit{part}(A_i)$

If A and B are compatible automata, we use also $A \times B$ to denote their composition.

We give a brief example of I/O automata and composition. The following `Channel` automaton is a simplified version of an automaton that we shall use in the actual semantics.

Example 3.1 (Channel). The `Channel` automaton is meant to model a reliable asynchronous communication channel in a network. Suppose $\mathcal{P}$ is a set of principal identities, and V is a countable set of values. The channel is a one-way communication channel between two identities, transmitting values from V. The automaton is *parametric* in two principal identities, meaning that for any $p, q \in \mathcal{P}$, $\texttt{Channel}(p, q)$ is an I/O automaton (intend to model a FIFO communication channel that can p can use to send V-values to q). Fix any two $p, q \in \mathcal{P}$, and consider the following data.

- The action signature $\mathit{sig}(\texttt{Channel}(p, q)) = S$ is given by the following. We have $\mathit{int}(S) = \emptyset$, and $\mathit{acts}(S) = \mathit{in}(S) \cup \mathit{out}(S)$. The input actions are $\mathit{in}(S) = \{\texttt{send}(p, q, v) \mid v \in V\}$ and the output actions are $\mathit{out}(S) = \{\texttt{recv}(q, p, v) \mid v \in V\}$. The signature is illustrated graphically in Figure 5.

- $\mathit{states}(\texttt{Channel}(p, q)) = V^*$, the set of finite sequences of elements from V. A state $s = v_1 \cdot v_2 \cdots v_n$ represents n messages in transit from p to q (sent in that particular order).

- $\mathit{start}(\texttt{Channel}(p, q)) = \epsilon$ (the empty sequence).

- *steps*(`Channel`(p, q)) is given by the following. For any state $s \in V^*$ and any $v \in V$, we have $(s, \texttt{send}(p, q, v), s{\cdot}v) \in \mathit{steps}(\texttt{Channel}(p, q))$. For any $v_0 \in V$ and any non-empty sequence $s = v_0 \cdot s' \in V^+$, we have $(s, \texttt{recv}(q, p, v_0), s') \in \mathit{steps}(\texttt{Channel}(p, q))$.

- *part*(`Channel`(p, q)) is the trivial partition where all `recv`(q, p, v) actions are in the same equivalence class.

We will often use a pseudo-language for specifying I/O automata. The language is similar to IOA [3, 4], and its semantics should be clear. In the language, an automaton is given by specifying its signature, state, actions, transitions and partition. The state is given in terms of a collection of variables, for example, `buffer` : *Seq*[V] := {} declares a variable "`buffer`" of type "sequences of values from the set V," and initializes this variable to the empty sequence. The transitions are given in a precondition/effect-style, where the precondition represents the set of states in which the action is enabled. The effect is an imperative program, executed atomically, manipulating the state variables.

The syntactic representation of the `Channel`(p, q) automaton is the following.

```
automaton Channel(p, q : P)
 signature
    input     send(const p, const q, v : V)
    output    recv(const q, const p, v : V)
 state
    buffer: Seq[V] := {}
 transitions
    input send(p, q, v)
      eff buffer := buffer |- v
    output recv(q, p, v)
      pre buffer != {} /\ v = head(buffer)
      eff buffer := tail(buffer)
 partition {recv(q, p, v) where v : D}
```

Example 3.2 (Composition). Continuing from the previous example, consider now an automaton A which will represent principal p. Suppose A has the following signature $\mathit{sig}(A)$: $\mathit{acts}(A) = \{\texttt{send}(p, q, v) \mid v \in V\} \cup I$, $\mathit{in}(A) = \emptyset$, $\mathit{int}(A) = I$, and $\mathit{out}(A) = \{\texttt{send}(p, q, v) \mid v \in V\}$, where I is some set of internal actions (disjoint from any other set of actions in this example). Then A and `Channel`(p, q) are compatible automata, and their

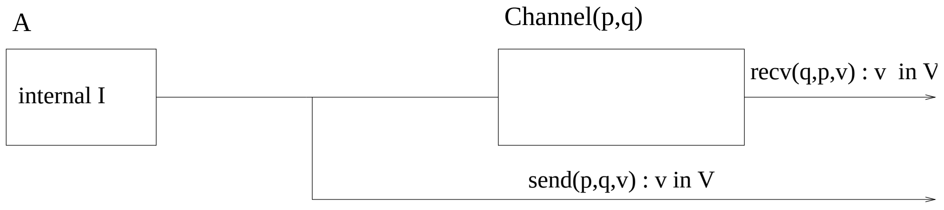

Figure 6: The interface of $A \times \texttt{Channel}(p, q)$.

composition $A \times \texttt{Channel}(p, q)$ is illustrated in Figure 6. Notice that the composition has no input actions, but output actions $\{\texttt{send}(p, q, v) \mid v \in V\} \cup \{\texttt{recv}(q, p, v) \mid v \in V\}$.

3.2 Bertsekas abstract asynchronous systems

A Bertsekas abstract asynchronous system (BAAS) is a general model of distributed asynchronous fixed-point algorithms. Many algorithms in concrete systems like message-passing or shared-memory systems are instances of the general model. Bertsekas has a convergence theorem that supplies sufficient conditions for a BAAS to compute certain fixed points. We describe the model and the theorem in this section.

BAAS. A *Bertsekas Abstract Asynchronous System* (BAAS) is a pair $B = ((X_i)_{i=1}^n, (f_i)_{i=1}^n)$ consisting of n sets $X_1, X_2, \ldots, X_n$, and n functions $f_1, f_2, \ldots, f_n$, where for each i, $f_i : \prod_{j=1}^n X_j \to X_i$. Let $X = \prod_{i=1}^n X_i$. We assume that there is a (partial) notion of convergence on X, so that some sequences $(x^i)_{i=1}^\infty, x^i \in X$ have a unique limit point, $\lim_i x_i \in X$. We let f denote the product function $f = \langle f_1, f_2, \ldots, f_n \rangle : X \to X$. The objective of a BAAS is to find a fixed point x^* of f.

We can think each $i \in [n]$ as a node in a network, and function f_i is then associated with that node. Each node i has a current best value x_i (which is supposed to be an approximation of x_i^*), and an estimate $x^i = (x_1^i, x_2^i, \ldots, x_n^i)$ for the current best values of all other nodes. Occasionally node i recomputes its current best value, using the current best estimates, by executing the assignment

$$x_i := f_i(x^i)$$

Once a node has updated its current value, this value is transmitted (by some means) to the other nodes, that (upon reception) update their estimates (e.g., x_i^j is updated at node j when receiving an update from node i).

Examples of BAAS's include many distributed optimization-, numerical- and dynamic programming algorithms [1].

BAAS runs. Let $B = ((X_i)_{i=1}^n, (f_i)_{i=1}^n)$ be a BAAS, and let $\hat{x} \in X = \prod_{i=1}^n X_i$. A *run of B, with initial solution estimate $\hat{x}$*, is given by the following.

1. A collection of (update-time) sets $(T^i)_{i \in [n]}$. For each i, the set T^i is a subset of $\mathbb{N}$, and represents the set of times where node i updates its current value.

2. A collection of (value) functions $(x_i)_{i \in [n]}$, each of type $x_i : \mathbb{N} \to X_i$. For $t \in \mathbb{N}$, $x_i(t)$ represents the value of node i, at time t. Function x_i satisfies $x_i(0) = \hat{x}_i$, and we use $x(t)$ to denote the vector $(x_1(t), x_2(t), \ldots, x_n(t))$.

3. For each $i \in [n]$, a collection of (estimate) functions $(\tau_j^i)_{j \in [n]}$, each of type $\tau_j^i : \mathbb{N} \to \mathbb{N}$, and each satisfying: for all $t \in \mathbb{N}$,

$$0 \leq \tau_j^i(t) \leq t$$

We let $x^i(t)$ denote i's estimate (of the values of all nodes) at time t. The estimates $x^i(t)$ are given by the estimate and value functions, as follows.

$$x^i(t) = (x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \ldots, x_n(\tau_n^i(t)))$$

Hence $t - \tau_j^i(t)$ can be seen as a form of transmission delay, as the current value of j at time t is $x_j(t)$, but node i only knows the older value $x^i(t)_j = x_j(\tau_j^i(t))$.

4. The value functions must satisfy the following requirements. If $t \in T^i$ then at time t, node i updates its value by applying f_i to its current estimates. That is,

$$\text{if } t \in T^i \text{ then } x_i(t+1) = f_i(x^i(t))$$

If $t \notin T^i$ then no updates are performed (on x_i). That is,

$$\text{if } t \notin T^i \text{ then } x_i(t+1) = x_i(t)$$

Note that the property of the τ-functions implies that, at time 0, all nodes agree on their estimates, $x^i(0) = x^j(0) = \hat{x}$ for all $i, j \in [n]$.

Definition 3.2 (Fairness). We say that a run is *finite* if all the sets T^i are finite. If a run is not finite, it is *infinite*. An infinite run r of a BAAS is *fair* if for each $i \in [n]$:

- the set T^i is infinite; and
- whenever $\{t^k\}_{k=0}^{\infty}$ is a sequence of elements all in T^i, tending to infinity, then also $\lim_{k\to\infty} \tau^i_j(t_k) = \infty$ for every $j \in [n]$.

A finite run r of a BAAS is *fair* if the following holds. Let $t^*_i = \max\ T^i$, and let $t^* = max_{i\in[n]}\ t^*_i + 1$. Then r satisfies:

- $x^i(t^*_i)_j = x_j(t^*)$ for all $i, j \in [n]$.

When an infinite run is fair, each node is guaranteed to recompute infinitely often. Moreover, all old estimate values are always eventually updated. For finite runs, the fairness assumption means that for each i, at the last update of i, its estimate for each node j is equal to the final value computed by j.

Lemma 3.1. *If r is a finite fair run of a BAAS B, then $x(t^*)$ is a fixed point of the product function of B.*

Proof. Let $i \in [n]$ be arbitrary but fixed. We show that $f_i(x(t^*)) = x(t^*)_i$. Since r is finite fair, we have: $x_j(\tau^i_j(t^*_i)) = x_j(t^*)$ for all $j \in [n]$. Hence $f_i(x^i(t^*_i)) = f_i(x(t^*))$. Since $t^*_i \in T^i$ we get $x_i(t^*_i + 1) = f_i(x^i(t^*_i))$. Now $t^* \geq t^*_i + 1$ and by the definition of t^*_i, for every t' with $t^*_i + 1 \leq t' \leq t^*$ we have $t' \notin T^i$. Hence $x_i(t^*) = x_i(t^*_i + 1)$. Putting it all together, we get

$$f_i(x(t^*)) = f_i(x^i(t^*_i)) = x_i(t^*_i + 1) = x_i(t^*) = x(t^*)_i$$

□

3.2.1 The asynchronous convergence theorem

The Bertsekas abstract asynchronous systems are a model of asynchronous distributed algorithms. The Asynchronous Convergence Theorem (ACT) (Proposition 6.2.1 of Bertsekas' book [1]) is a general theorem which gives sufficient conditions for BAAS runs to converge to a fixed point of the product function f. The ACT applies in any scenario in which the so-called "Synchronous Convergence Condition" and the "Box Condition" are satisfied. Intuitively, the synchronous convergence condition states that if the algorithm is executed synchronously, then one obtains the desired result. In our case, this amounts to requiring that the "synchronous" sequence $\perp_{\sqsubseteq}^{n} \sqsubseteq f(\perp_{\sqsubseteq}^{n}) \sqsubseteq \cdots$ converges to the least fixed-point, which is true for continuous f. Intuitively, the box condition requires that one can split the set of possible values appearing during synchronous computation into a product ("box") of sets of values that appear locally at each node in the asynchronous computation.

We now recall the definition of the Synchronous Convergence Condition (SCC) and the Box Condition (BC) (Section 6.2 [1]). Consider a BAAS with $X = \prod_{i=1}^{n} X_i$, and $f : X \to X$ any function with $f = \langle f_1, f_2, \ldots, f_n \rangle$.

Definition 3.3 (SCC and BC). Let $\{X(k)\}_{k=0}^{\infty}$ be a sequence of subsets $X(k) \subseteq X$ satisfying $X(k+1) \subseteq X(k)$ for all $k \geq 0$.

SCC The sequence $\{X(k)\}_{k=0}^{\infty}$ satisfies the *Synchronous Convergence Condition* if for all $k \geq 0$ we have

$$x \in X(k) \Rightarrow f(x) \in X(k+1)$$

and furthermore, if $\{y^k\}_{k\in\mathbb{N}}$ is a sequence which has a limit point $\lim_k y^k$, and which satisfies $y^k \in X(k)$ for all k, then $\lim_k y^k$ is a fixed-point of f.

BC The sequence $\{X(k)\}_{k=0}^{\infty}$ satisfies the *Box Condition* if for every $k \geq 0$, there exist sets $X_i(k) \subseteq X_i$ such that

$$X(k) = \prod_{i=1}^{n} X_i(k)$$

The following Asynchronous Convergence Theorem gives sufficient conditions for a BAAS run to converge to the fixed point of its product function.

Theorem 3.1 (ACT, Bertsekas). *Let $B = ((X_i)_{i=1}^{n}, (f_i)_{i=1}^{n})$ be a BAAS, $X = \prod_{i=1}^{n} X_i$, and $f = \langle f_i : i \in [n] \rangle$. Let $\{X(k)\}_{k=0}^{\infty}$ be a sequence of sets with $X(k) \subseteq X$ and $X(k+1) \subseteq X(k)$ for all $k \geq 0$. Assume that $\{X(k)\}_{k=0}^{\infty}$ satisfies the SCC and the BC. Let r be any infinite fair run of B, with initial solution estimate $x(0) \in X(0)$. Then, if $\{x(t)\}_{t\in\mathbb{N}}$ has a limit point, this limit point is a fixed point of f.*

4 An Operational Semantics

In this section we present the operational semantics of the basic policy language. This will be given by a semantic function $[\![\cdot]\!]^{\text{op}}$ mapping a collection Π of policies of the basic language to an I/O automaton $[\![\Pi]\!]^{\text{op}}$. We introduce also an "abstract" operational semantics, which is given by another semantic function $[\![\cdot]\!]^{\text{ob-abs}}$ mapping Π to a BAAS. The systems $[\![\Pi]\!]^{\text{op}}$ and $[\![\Pi]\!]^{\text{op-abs}}$ will correspond in a "simulation-like" manner: runs of $[\![\Pi]\!]^{\text{op}}$ can be faithfully matched by corresponding runs of $[\![\Pi]\!]^{\text{op-abs}}$ (in a formal sense, described later in this section).

4.1 $[\![\cdot]\!]^{\text{op}}$ translation, an operational semantics

We first provide the concrete operational semantics. Function $[\![\cdot]\!]^{\text{op}}$ maps a collection of trust policies from the basic language to an I/O automaton. The semantics uses two parameterized I/O automata: $\texttt{Channel}(p, q, r)$ and $\texttt{IOTemplate}(p, q, f)$, where $p, q, r \in \mathcal{P}$ and $f : (\mathcal{P} \to \mathcal{P} \to D) \to D$ is a continuous function. The semantic function $[\![\cdot]\!]^{\text{op}}$ is given in Figure 7 and Figure 8. The parameterized automata are described syntactically in Figure 9 and Figure 10.

A principal p is represented as the automaton $[\![\pi_p]\!]^{\text{op}}_{p,\emptyset}$ which is the composition of a collection of automata $\texttt{IOTemplate}(p, q, f_{pq})$ for $q \in \mathcal{P}$ and where $f_{pq}(\texttt{gts}) = [\![\pi_p]\!]^{\text{den}}\ \texttt{gts}\ q$, i.e, policy π_p's entry for q. The component $\texttt{IOTemplate}(p, q, f_{pq})$, which we denote simply as "pq", is responsible for computing (or approximating) principal p's trust value for principal q, i.e., the value $\overline{\texttt{gts}}(p)(q)$.

The I/O automaton $[\![\Pi]\!]^{\text{op}}$ is a composition, $A \times B$, of two automata where $A = \prod_{p \in \mathcal{P}} [\![\pi_p]\!]^{\text{op}}_{p,\emptyset}$ represents the composition-automaton of each of the principals policies, and $B = \prod_{p,r,q \in \mathcal{P}} \texttt{Channel}(p, r, q)$ is a composition of channel automata. For $p, r, q \in \mathcal{P}$, the channel automaton $\texttt{Channel}(p, r, q)$ represent a reliable FIFO communication channel, and will be used by the automaton $pq = \texttt{IOTemplate}(p, q, f_{pq})$, to communicate trust-values of p about principal q to principal r.

The $\texttt{IOTemplate}$-automata from Figure 10 are designed to implement the following algorithm, described previously by Krukow and Twigg [6].

An asynchronous algorithm. The asynchronous algorithm is executed in a network of nodes pq for $p, q \in \mathcal{P}$. Each node pq allocates variables $pq.t_{cur}$ and $pq.t_{old}$ of type D, which will later record the "current" value and the last computed value. Each node pq has also a matrix,

$$[\![(\pi_p \mid p \in \mathcal{P})]\!]^{\mathrm{op}} \quad = \quad (\prod_{p\in\mathcal{P}} [\![\pi_p]\!]^{\mathrm{op}}_{p,\emptyset}) \times \prod_{p,r,q\in\mathcal{P}} \mathtt{Channel}(p,r,q)$$

Figure 7: Operational semantics of the policy language. For a principal-indexed collection of policies $\Pi = (\pi_p \mid p \in \mathcal{P})$, the semantics $[\![\Pi]\!]^{\mathrm{op}}$ is an I/O automaton.

$$\begin{aligned}
[\![q : \tau, \pi]\!]^{\mathrm{op}}_{p,F} &= [\![\tau]\!]^{\mathrm{op}}_{p,q} \times [\![\pi]\!]^{\mathrm{op}}_{p,F\cup\{q\}} \\
[\![\star : \tau]\!]^{\mathrm{op}}_{p,F} &= \prod_{q\in\mathcal{P}\setminus F} [\![\tau]\!]^{\mathrm{op}}_{p,q} \\
[\![\tau]\!]^{\mathrm{op}}_{p,q} &= \mathtt{IOTemplate}(p, q, [\![\tau]\!]^{\mathrm{den}}([\star \mapsto q]/id_{\mathcal{P}}))
\end{aligned}$$

Figure 8: Operational semantics of the policy language. Function $[\![\cdot]\!]^{\mathrm{op}}_{F,p}$ maps a policy to an I/O automaton when $F \subseteq \mathcal{P}$ and $p \in \mathcal{P}$. `IOTemplate` is a parameterized I/O automaton, taking three arguments, $p, q \in \mathcal{P}$ and $f_{pq} : (\mathcal{P} \to \mathcal{P} \to D) \to D$.

```
automaton Channel(p, r, q : P)
 signature
    input     send(const p, const r, const q, d : D)
    output    recv(const r, const p, const q, d : D)
 state
    buffer: Seq[D] := {}
 transitions
    input send(p, r, q, d)
      eff buffer := buffer |- d
    output recv(r, p, q, d)
      pre buffer != {} /\ d = head(buffer)
      eff buffer := tail(buffer)
 partition {recv(r, p, q, d) where d : D}
```

Figure 9: The $\mathtt{Channel}(p : \mathcal{P}, r : \mathcal{P}, q : \mathcal{P})$ parameterized automaton.

```
automaton IOTemplate (p : P, q : P, f_pq : (P -> P -> D) -> D)
   signature
     input  recv(const p, r : P, s : P, d : D)
     output send(const p, r : P, const q, d : D)
     internal eval(const p, const q)
  state
    gts : P -> P -> D,
    t_old : D := bot,
    t_cur : D := bot,
    wake : Bool := true,
    send : P -> Bool
    initially
           \forall r,s : P (gts(r)(s) = bot)
           \forall r : P (send(r) = false)
  transitions
    input    recv(p, r, s, d)
                eff wake := true;
                    if ((r,s) != (p,q)) then gts(r)(s) := d fi

    output   send(p, r, q, d)
                pre send(r) = true   /\   d = t_cur
                eff send(r) := false

    internal eval(p,q)
                pre wake   /\   \forall r : P (send(r) = false) %all scheduled
                                                                %messages sent.
                eff
                    t_old := t_cur;
                    t_cur := f_pq(gts); % evaluate policy on gts
                    if (t_old != t_cur)
                    then
                        gts(p)(q) := t_cur;
                        for each r : P do send(r) := true od
                    else
                        wake := false
                    fi

    partition {eval(p,q)};
          {send(p, r, q, d) where d : D} for each r : P
```

Figure 10: The IOTemplate$(p : \mathcal{P}, q : \mathcal{P}, f_{pq} : (P \rightarrow P \rightarrow D) \rightarrow D)$ parameterized automaton.

denoted by $pq.gts$, of type $\mathcal{P} \to \mathcal{P} \to D$. Initially, $pq.t_{cur} = pq.t_{old} = \bot_{\sqsubseteq}$, and the matrix is also initialized with $\bot_{\sqsubseteq}$. For any nodes pq and rs, when rs receives a message from pq (which is always a value in D), it stores this message in $rs.gts(p)(q)$ (except the special case where $rs = pq$ where this is unnecessary).

Any node is always in one of two states: *sleep* or *wake*. All nodes start in the *wake* state, and if a node is in the *sleep* state, the reception of a message triggers a transition to the *wake* state. In the *wake* state any node pq repeats the following: it starts by assigning to variable $pq.t_{cur}$ the result of applying its function f_{pq} to the values in $pq.gts$, i.e., node pq executes assignment $pq.t_{cur} := f_{pq}(pq.gts)$. If there is no change in the resulting value (compared to the last value computed, which is stored in $pq.t_{old}$), it will go to the *sleep* state (unless a new message was received since $f_{pq}(pq.gts)$ was computed). Otherwise, if a new value resulted from the computation (i.e., if $pq.t_{old} \neq f_{pq}(pq.gts)$), this value is sent to all nodes.

In the I/O automata version of this algorithm, the sending of a message d from node pq to another node, say rs, is represented by the action $\texttt{send}(p, r, q, d)$ (note this is independent of s). The message d i stored in the buffer of $\texttt{Channel}(p, r, q)$, and eventually retrieved by node rs, performing input-action $\texttt{recv}(r, p, q, d)$ (note all nodes rs' for $s' \in \mathcal{P}$ perform this action simultaneously, reflecting that principal r is modelled by the entire collection $(\texttt{IOTemplate}(r, s', f_{rs'}) \mid s' \in \mathcal{P})$). Action $\texttt{eval}(p, q)$ represents the node pq recomputing its current value. The fairness partition of $\texttt{IOTemplate}(p, q, f_{pq})$ ensures that the $\texttt{eval}(p, q)$ action is always eventually executed once it is enabled. Similarly, $\texttt{send}(p, r, q, pq.t_{cur})$ is always eventually executed when variable $pq.send(r)$ is $\texttt{true}$.

Lemma 4.1 (Composability). *If all policies of* Π *are well-formed, then all the automata occurring in the definition of* $[\![\Pi]\!]^{op}$ *have compatible signatures, and, hence, are composable.*

Proof. Simple inspection shows disjointness of all (output and internal) actions of the involved automata. $\square$

4.2 Cause and effect

In the following, we establish some structure on runs of the operational-semantics automaton. For a run r_c of $[\![\Pi]\!]^{\mathrm{op}}$, we define a “causality” function, $cause_{r_c}$, mapping each index $k > 0$ to a smaller index k'. If $cause_r(k) = k' > 0$ we say that action $a_{k'}$ causes action a_k.

For a (finite or infinite) run $r_c = s_0 a_1 s_1 a_2 s_2 \cdots$ of $[\![\Pi]\!]^{\mathrm{op}}$, we write $\mathit{ActIndex}(r_c)$ for the set $\{j \in \mathbb{N} \mid 0 < j < |r_c|\}$ of action indexes of r_c. Define the function $\mathit{cause}_{r_c} : \mathit{ActIndex}(r_c) \to \mathbb{N}$ inductively.

$$\mathit{cause}_{r_c}(1) = 0$$

For any $k \in \mathbb{N}$, define $\mathit{cause}_{r_c}(k+1)$ by cases.

- Case $a_{k+1} = \mathtt{eval}(p,q)$ for some $p, q \in \mathcal{P}$. As we are not interested in "causes" of `eval` events, we simply define $\mathit{cause}_{r_c}(k+1) = 0$.
- Case $a_{k+1} = \mathtt{send}(p,r,q,d)$ for some $p,q,r \in \mathcal{P}$, $d \in D$. Note that since initially, $pq.\mathit{send}(r) = \mathtt{false}$, there must exist some largest index $j < k$ so that $s_j.pq.\mathit{send}(r) = \mathtt{false}$ and $s_{j+1}.pq.\mathit{send}(r) = \mathtt{true}$ (since this is a pre-condition of a_{k+1}). Then $\mathit{cause}_{r_c}(k+1) = j+1$. Note that a_{j+1} must be an $\mathtt{eval}(p,q)$ event, and that we must have $s_j.pq.t_{cur} \neq s_{j+1}.pq.t_{cur}$, and $s_{j+1}.pq.\mathit{wake} = \mathtt{true}$.
- Case $a_{k+1} = \mathtt{recv}(p,r,s,d)$ for some $p,r,s \in \mathcal{P}$, $d \in D$. Let $R_0 = \{j \mid j < k{+}1, a_j = \mathtt{recv}(p,r,s,d)\}$, and let $S_0 = \{j \mid j < k{+}1, a_j = \mathtt{send}(r,p,s,d)\}$. S_0 is a candidate set of indices for the result. Now let

  $$S \overset{(\mathrm{def})}{=} S_0 \setminus \mathit{cause}_{r_c}(R_0) = S_0 \setminus \{\mathit{cause}_{r_c}(r_0) \mid r_0 \in R_0\}$$

  (cause_{r_c} is defined on $r_0 \in R_0$ since $r_0 < k+1$). Note that $|S_0| > |R_0|$. This follows from the fact that for each $\mathtt{recv}(p,r,s,d)$ action, there must be at least one previous occurrence of a $\mathtt{send}(r,p,s,d)$ action, and $a_{k+1} = \mathtt{recv}(p,r,s,d)$. This, in turn, implies that S is non-empty. Now, define

  $$\mathit{cause}_{r_c}(k+1) = \min\ S$$

  Writing $k' = \mathit{cause}_{r_c}(k+1)$, note that $a_{k'} = \mathtt{send}(r,p,s,d)$. Note also that $s_{k'}.rs.t_{cur} = d$, and $s_{k'}.rs.\mathit{send}(p) = \mathtt{false}$.

We define a "dual" function of cause_{r_c}, called the "effect" function, and denoted effect_{r_c}. Function $\mathit{effect}_{r_c} : \mathit{ActIndex}(r_c) \to 2^{\mathit{ActIndex}(r_c)}$ is defined as follows:

$$\mathit{effect}_{r_c}(k) = \mathit{cause}_{r_c}^{-1}(\{k\}) = \{k' \in \mathit{ActIndex}(r_c) \mid \mathit{cause}_{r_c}(k') = k\}$$

The following lemma establishes some simple properties of the cause_{r_c} function.

Lemma 4.2 (Simple properties of *cause*). *For any run r_c, function $cause_{r_c}$ satisfies the following.*

- *For every $k \in ActIndex(r_c)$, $cause_{r_c}(k) < k$ (which implies that $\forall k' \in effect_{r_c}(k).\ k < k'$).*
- *Each $\texttt{send}(p, r, q, d)$ action in r_c is caused by a unique $\texttt{eval}(p, q)$ action, and each $\texttt{recv}(p, r, s, d)$ action in r_c is caused by a unique $\texttt{send}(r, p, s, d)$ action.*
- *The $cause_{r_c}$ function is injective when restricted to $\texttt{recv}$ actions. That is, for any indices k, k' with $k \neq k'$, if $a_k = \texttt{recv}(\dots)$ and $a_{k'} = \texttt{recv}(\dots)$, then also $cause_{r_c}(k) \neq cause_{r_c}(k')$.*

Proof. The first two items follow immediately from the definition. For the last item, let $k < k'$ with $a_k = \texttt{recv}(p, r, s, d)$ and $a_{k'} = \texttt{recv}(p', r', s', d')$. Let $j = cause_{r_c}(k)$ and $j' = cause_{r_c}(k')$, then by the above, $a_j = \texttt{send}(r, p, s, d)$ and $a_{j'} = \texttt{send}(r', p', s', d')$. Hence if $(p, r, s, d) \neq (p', r', s', d')$ then $j \neq j'$. So assume that $(p, r, s, d) = (p', r', s', d')$. We want to prove that $cause_{r_c}(k) \neq cause_{r_c}(k')$. Let $S_0 = \{j \mid j < k, a_j = \texttt{send}(r, p, s, d)\}$, $S'_0 = \{j \mid j < k', a_j = \texttt{send}(r, p, s, d)\}$, $R_0 = \{j \mid j < k, a_j = \texttt{recv}(p, r, s, d)\}$ and $R'_0 = \{j \mid j < k', a_j = \texttt{recv}(p, r, s, d)\}$. We have $S_0 \subseteq S'_0$ and $R_0 \subsetneq R'_0$, in particular, $k \in R'_0 \setminus R_0$.

$$cause_{r_c}(k) = \min(S_0 \setminus cause_{r_c}(R_0))$$
$$cause_{r_c}(k') = \min(S'_0 \setminus cause_{r_c}(R'_0))$$

Injectivity follows, as $cause_{r_c}(k) \in cause_{r_c}(R'_0)$. $\square$

The following lemma formalizes the fact that the channels are reliable, and act in a FIFO manner.

Lemma 4.3 (FIFO). *Let $r_c = s_0 a_1 s_2 \cdots$ be a finite or infinite fair run of $\texttt{Channel}(p, r, q)$ for $p, r, q \in \mathcal{P}$. Suppose that $a_k = \texttt{send}(p, r, q, d)$ and $a_{k'} = \texttt{send}(p, r, q, d')$ for some $d, d' \in D$. If $k \leq k'$ then there exists unique j, j' with $k < j$ and $k' < j'$, so that $j \leq j'$, $a_j = \texttt{recv}(r, p, q, d)$, $a_{j'} = \texttt{recv}(r, p, q, d')$, $cause_{r_c}(j) = k$ and $cause_{r_c}(j') = k'$.*

Proof. Since $a_k = \texttt{send}(p, r, q, d)$ then we have $s_k.\mathit{buffer} = u \cdot d$, for some $u \in D^*$. Let $N = |u| \geq 0$, then by fairness, there must be $N + 1$ unique indices $(k_i)_{i=1}^{N+1}$, satisfying the following four points.

- $k < k_i < k_{i+1}$, for all $1 \leq i \leq N$,

- $a_{k_i} = \texttt{recv}(r, p, q, u_i)$ for all $1 \leq i \leq N$,
- $a_{k_{N+1}} = \texttt{recv}(r, p, q, d)$, and
- for all $l \in \mathbb{N}$ with $k < l < k_{N+1}$ and $l \neq k_i$ for all $1 \leq i \leq N+1$, action a_l is not a $\texttt{recv}$ action, i.e., $a_l \neq \texttt{recv}(p, r, q, d_0)$ for all $d_0 \in D$.

We prove in the following that $\mathit{cause}_{r_c}(k_{N+1}) = k$. Define $S_0 = \{m \mid m < k_{N+1}, a_m = \texttt{send}(p, r, q, d)\}$ and $R_0 = \{m \mid m < k_{N+1}, a_m = \texttt{recv}(r, p, q, d)\}$. Define also $S_0^k = \{m \mid m < k, a_m = \texttt{send}(p, r, q, d)\}$, and $R_0^k = \{m \mid m < k, a_m = \texttt{recv}(r, p, q, d)\}$, and note that $k \in S_0$ but $k \notin S_0^k$. Since $s_{k-1}.\mathit{buffer} = u$, we have $|S_0^k| = |R_0^k| + N_d$, where $N_d = |\{n \mid 1 \leq n \leq N, u_n = d\}|$. This implies that $|R_0| = |R_0^k| + N_d = |S_0^k|$. Furthermore, for all $r \in R_0$, we have $\mathit{cause}_{r_c}(r) < k$, by the following argument. Let $r \in R_0$, and assume $r \geq k$ (if $r < k$ then $\mathit{cause}_{r_c}(r) < r < k$). Define $S_0^r = \{m \mid m < r, a_m = \texttt{send}(p, r, q, d)\}, R_0^r = \{m \mid m < r, a_m = \texttt{recv}(r, p, q, d)\}$, and note that $S_0^k \subsetneq S_0^r$, $r \in R_0 \setminus R_0^r$, and for all $m \in S_0^r \setminus S_0^k$, $m \geq k$. By definition, $\mathit{cause}_{r_c}(r) = \min(S_0^r \setminus \mathit{cause}_{r_c}(R_0^r))$. Because $k < r < k_{N+1}$, we have $|S_0^r| > |S_0^k| = |R_0^k| + N_d > |R_0^r|$, implying that $S_0^k \setminus \mathit{cause}_{r_c}(R_0^r) \neq \emptyset$. Hence, because $\forall m \in S_0^r \setminus S_0^k . m \geq k$, we obtain, $\mathit{cause}_{r_c}(r) = \min(S_0^r \setminus \mathit{cause}_{r_c}(R_0^r)) = \min(S_0^k \setminus R_0^r) < k$.

Now, we have an injective function cause_{r_c} mapping the set R_0 to the set S_0^k, so $|S_0^k| = |R_0|$ implies that $\mathit{cause}_{r_c}(R_0) = S_0^k$. Hence,

$$S_0 \setminus \mathit{cause}_{r_c}(R_0) = S_0 \setminus S_0'$$

and since $k = \min(S_0 \setminus S_0')$, we have $\mathit{cause}_{r_c}(k_{N+1}) = k$.

Similar reasoning applies to k', so let j, j' be so that $a_j = \texttt{recv}(r, p, q, d)$, $k = \mathit{cause}_{r_c}(j)$, $a_{j'} = \texttt{recv}(r, p, q, d')$ and $k' = \mathit{cause}_{r_c}(j')$. To show that $j \leq j'$, assume first that $k' > j$, then because $j' > k'$, clearly $j < j'$. So assume instead for some $i \geq 0$ we have $k_i < k' < k_{i+1}$ (writing $k = k_0$). Note that then $s_{k'}.\mathit{buffer} = u_{i+1}u_{i+2} \cdots u_N d s' d'$ for some $s' \in D^*$, and hence, $j' = k'_{N'+1} > k_{N+1} = j$. ☐

Notice that by the above lemma, if $a_k = \texttt{send}(p, r, q, d)$ then uniqueness of j with $\mathit{cause}_{r_c}(j) = k$ implies that $\mathit{effect}_{r_c}(k) = \{j\}$. By abuse of notation, we write $\mathit{effect}_{r_c}(k) = j$. Hence, $\mathit{cause}_{r_c}(\mathit{effect}_{r_c}(k)) = k$. This implies also that if $a_m = \texttt{recv}(r, p, q, d)$ then $\mathit{effect}_{r_c}(\mathit{cause}_{r_c}(m)) = m$.

Lemma 4.4 (Cause and Effect). *Let $\Pi = (\pi_p \mid p \in \mathcal{P})$ be a collection of policies, and let $r_c = s_0 a_1 s_1 a_2 \cdots$ be a finite or infinite fair run of $[\![\Pi]\!]^{op}$. The following properties hold of r_c:*

1. *Assume that for some $k \geq 0$, we have $s_k.pq.wake = \texttt{true}$, then there exists a $k' > k$ so that $a_{k'} = \texttt{eval}(p, q)$.*

2. *Assume that $a_{k_0} = \texttt{eval}(p, q)$ and $s_{k_0-1}.pq.t_{cur} \neq s_{k_0}.pq.t_{cur} = d$. Let $k_1 > k$ be least with $a_{k_1} = \texttt{eval}(p, q)$ (note, such an index must exist by the above). Then, for every $r \in \mathcal{P}$ there exists a unique k_r with $k_0 < k_r < k_1$ so that $a_{k_r} = \texttt{send}(p, r, q, \_)$. Furthermore, $a_{k_r} = \texttt{send}(p, r, q, d)$ and $cause_{r_c}(k_r) = k_0$.*

3. *Assume that $a_k = \texttt{send}(p, r, q, d)$ and $a_{k'} = \texttt{send}(p, r, q, d')$. Then, $k < k'$ implies $cause_{r_c}(k) < cause_{r_c}(k')$.*

4. *Assume that $a_k = \texttt{recv}(r, p, q, d)$ and $a_{k'} = \texttt{recv}(r, p, q, d')$. Then, $k < k'$ implies $cause_{r_c}(k) < cause_{r_c}(k')$.*

Proof. Let $r_c = s_0 a_1 s_1 a_2 \cdots$ be a finite or infinite fair run of $[\![\Pi]\!]^{\text{op}}$. We prove each point separately.

1. Assume that $s_k.pq.wake = \texttt{true}$. Assume first that for every $r \in \mathcal{P}$ we have $s_k.pq.send(r) = \texttt{false}$. Then action $\texttt{eval}(p, q)$ is enabled. Notice that this action stays enabled until a $\texttt{eval}(p, q)$ event occurs. Since $\{\texttt{eval}(p, q)\}$ is an equivalence class, fairness of r_c implies that there exists some $k' > k$ so that $a_{k'} = \texttt{eval}(p, q)$. Now, suppose instead that for some $r \in \mathcal{P}$ we have $s_k.pq.send(r) = \texttt{true}$. Then action $\texttt{send}(p, r, q, d)$ is enabled for $d = s_k.pq.t_{cur}$, and notice that this action stays enabled until a $\texttt{send}(p, r, q, d)$ event occurs. Since $\{\texttt{send}(p, r, q, c) \mid c \in D\}$ is an equivalence class, and only $\texttt{send}(p, r, q, d)$ is enabled in the class, fairness of r_c means that for some $k'_0 > k$ we have $a_{k'_0} = \texttt{send}(p, r, q, d)$, and hence $s_{k'_0}.pq.send(r) = \texttt{false}$. Let k_0 be the least such index, and note that for all j with $k \leq j \leq k_0$ we have $s_j.pq.wake = \texttt{true}$ (as no $\texttt{eval}(p, q)$ action can occur while $pq.send(r) = \texttt{true}$). Since this holds for all r, there must exist a $k' > k$ so that $s_{k'}.pq.wake = \texttt{true}$ and for all $r \in \mathcal{P}$ we have $s_{k'}.pq.send(r) = \texttt{false}$, and we are done by the initial comment.

2. Assume that $a_{k_0} = \texttt{eval}(p, q)$ and $s_{k_0-1}.pq.t_{cur} \neq s_{k_0}.pq.t_{cur} = d$. Notice that $s_{k_0}.pq.wake = \texttt{true}$, and let $k_1 > k_0$ be the (index of the) first occurrence of an $\texttt{eval}(p, q)$ event after time k_0. Notice that since no $\texttt{eval}(p, q)$ event occurs in the interval (k_0, k_1) we have $s_l.pq.t_{cur} = d$ for all $l \in [k_0, k_1)$. Let $r \in \mathcal{P}$ be arbitrary. Notice that $\texttt{send}(p, r, q, d)$ is enabled at time k_0, and

stays enabled until a $\texttt{send}(p, r, q, d)$ action occurs. By fairness such an action must occur, so let $k_r > k_0$ be the least index so that $a_{k_r} = \texttt{send}(p, r, q, d)$. Notice that after time k_0, no $\texttt{eval}(p, q)$ action can occur before a $\texttt{send}(p, r, q, d)$ action has occurred, hence we have $k_r < k_1$. Uniqueness of k_r follows from the fact that for k in $[k_0, k_r)$ we have $s_k.pq.send(r) = \texttt{true}$ and for k in $[k_r, k_1)$ we have $s_k.pq.send(r) = \texttt{false}$. Hence, there can only be one occurrence of a $\texttt{send}(p, r, q, d)$ event in the time interval (k_0, k_1). Finally, since for all k with $k_0 < k < k_r$ we have $a_k \neq \texttt{eval}(p, q)$, it follows that $cause_{r_c}(k_r) = k_0$.

3. Assume that $a_k = \texttt{send}(p, r, q, d)$ and $a_{k'} = \texttt{send}(p, r, q, d')$. Assume also that $k < k'$. Notice first that

$$\{j \mid j < k, s_j.pq.send(r) = \texttt{false}, s_{j+1}.pq.send(r) = \texttt{true}\} \subseteq \{j \mid j < k', s_j.pq.send(r) = \texttt{false}, s_{j+1}.pq.send(r) = \texttt{true}\}$$

   Hence, $cause_{r_c}(k) \leq cause_{r_c}(k')$. Assume for the sake of contradiction that, $cause_{r_c}(k) = cause_{r_c}(k') = k_0$. Let k_1 be the first occurrence of $\texttt{eval}(p, q)$ after time k_0. Then $k_1 > k$ because by definition of $cause_{r_c}(k)$, there can be no $\texttt{eval}(p, q)$ occurrences in the interval $(cause_{r_c}(k), k] = (k_0, k]$ (because $pq.send(r) = \texttt{true}$). Since also $cause_{r_c}(k_1) = k_0$, by the same argument we must have $k_1 > k'$. But now the uniqueness property in point (2) of this lemma implies that there can only be one $\texttt{send}(p, r, q, \_)$ occurrence in the interval $[k_0, k_1]$, and hence $k = k'$, which contradicts $k < k'$. So, by contradiction, we must have $cause_{r_c}(k) < cause_{r_c}(k')$.

4. Assume that $a_k = \texttt{recv}(r, p, q, d)$ and $a_{k'} = \texttt{recv}(r, p, q, d')$. Assume also that $k < k'$. Then $cause_{r_c}(k) < cause_{r_c}(k')$ follows because if we would have $cause_{r_c}(k') \leq cause_{r_c}(k)$ then by the FIFO Lemma, $k' = \mathit{effect}_{r_c}(cause_{r_c}(k')) \leq \mathit{effect}_{r_c}(cause_{r_c}(k)) = k$.

$\square$

4.3 $[\![\cdot]\!]^{\text{op-abs}}$ translation, an abstract operational semantics

We also map a collection $\Pi = (\pi_p \mid p \in \mathcal{P})$ to a BAAS, in a similar way. The BAAS $[\![\Pi]\!]^{\text{abs-op}}$ consists of the set D of trust values, a collection of

$n = |\mathcal{P}|^2$ functions $f_{pq} : D^n \to D$ (the functions are indexed by pairs pq where $p, q \in \mathcal{P}$). The functions f_{pq} are given by the policies,

$$f_{pq}(\mathsf{gts}) = [\![\pi_p]\!]^{\text{den}}\ \mathsf{gts}\ q$$

i.e., f_{pq} is the q-projection of policy p. This function represents the I/O automaton $pq = \texttt{IOTemplate}(p, q, f_{pq})$ which is a component of $[\![\Pi]\!]^{\text{op}}$.

In the rest of this paper, we shall not distinguish between $[n] = \{1, 2, \ldots, n\}$ and the set $\mathcal{P} \times \mathcal{P}$, nor shall we distinguish between D^n and $\mathcal{P} \to \mathcal{P} \to D$. Note that $\Pi_\lambda = \langle\langle f_{pq} \mid q \in \mathcal{P}\rangle \mid p \in \mathcal{P}\rangle = f$ (hence a value $\hat{d} \in D^n$ is a fixed point of f if-and-only-if it is a fixed point of Π_λ).

The notion of convergence of sequences in D^n is the following. A sequence $(\bar{d}^k)_{k=0}^{\infty}$ has a limit iff the set $\{\bar{d}^k \mid k \in \mathbb{N}\}$ has a least upper bound in $(D^n, \sqsubseteq)$, and in this case, $\lim_k \bar{d}^k = \bigsqcup_k \bar{d}^k$.

4.4 Correspondence of abstract and concrete operational semantics

The two translations $[\![\cdot]\!]^{\text{op}}$ and $[\![\cdot]\!]^{\text{op-abs}}$ are closely related: the latter can be viewed as an abstract version of the former. In fact, in the following we will map runs of $[\![\Pi]\!]^{\text{op}}$ to "corresponding" runs of $[\![\Pi]\!]^{\text{op-abs}}$.

Correspondence of runs. Let us map a (finite or infinite, fair or not) run $r_c = s_0 a_1 s_1 a_2 s_2 \cdots$ of the concrete I/O-automaton $[\![\Pi]\!]^{\text{op}}$ to a run r_a of the BAAS $[\![\Pi]\!]^{\text{op-abs}}$, called *the corresponding run (of r_c)*, as follows.

1. For any $p, q \in \mathcal{P}$, T^{pq} is defined as $\{k - 1 \mid k \in \mathbb{N}, a_k = \texttt{eval}(p, q)\}$. That is, the update-times of pq are the indexes of pre-states of $\texttt{eval}(p, q)$ actions in r_c. Note that for $(p, q) \neq (r, s)$ we have an empty intersection, $T^{pq} \cap T^{rs} = \emptyset$.

2. For each $p, q \in \mathcal{P}$, the function $\tau_{pq}^{pq} : \mathbb{N} \to \mathbb{N}$ is given by the identity $\tau_{pq}^{pq}(t) = t$. This reflects the fact that node pq always has an exact "estimate" of its own current value. This rule implies that $x^{pq}(t)_{pq} = x_{pq}(\tau_{pq}^{pq}(t)) = x_{pq}(t)$.

3. For each $p, q \in \mathcal{P}$ and each $r, s \in \mathcal{P}$ with $(r, s) \neq (p, q)$, the function $\tau_{rs}^{pq} : \mathbb{N} \to \mathbb{N}$ is given by the following. Let $t \in \mathbb{N}$ be arbitrary but fixed.

(a) Let $k \leq t$ be the *largest*, with the property that $a_k = \texttt{recv}(p, r, s, d)$ for some $d \in D$. If no such index exists, then $\tau^{pq}_{rs}(t)$ is defined as the largest $j \leq t$ with the property that for all j' with $0 \leq j' \leq j$ we have $s_{j'}.rs.t_{cur} = \bot_{\sqsubseteq}$. If such k exists, let $k' = cause_{r_c}(k)$. Note that $a_{k'} = \texttt{send}(r, p, s, d)$.

(b) We then define $k'' = cause_{r_c}(k')$. Note that $a_{k''} = \texttt{eval}(r, s)$, and that we must have $s_{k''}.rs.t_{cur} = d$.

(c) Finally, define $\tau^{pq}_{rs}(t)$ to be the largest index $j \leq t$ with the property that for all j' with $k'' \leq j' \leq j$, also $s_{j'}.rs.t_{cur} = d$. Note, in particular $s_j.rs.t_{cur} = d$.

Note that $0 \leq \tau^{pq}_{rs}(t) \leq t$ is satisfied.

4. The value functions $x_{pq} : \mathbb{N} \to D$ are given inductively. We have $x_{pq}(0) = \bot_{\sqsubseteq}$. For each $t \in \mathbb{N}$, $x_{pq}(t + 1)$ is given by the recursive equation

$$x_{pq}(t+1) = \begin{cases} x_{pq}(t) & \text{if } t \notin T^{pq} \\ f_{pq}(x^{pq}(t)) & \text{if } t \in T^{pq} \end{cases}$$

Note that this definition obviously satisfies the requirement for value functions in the definition of runs of BAAS's.

Lemma 4.5. *For any $p, q, r, s \in \mathcal{P}$, function τ^{pq}_{rs} is monotonically increasing.*

Proof. We must show for all t, u if $t \leq u$ then $\tau^{pq}_{rs}(t) \leq \tau^{pq}_{rs}(u)$. If no $\texttt{recv}(p, r, s, d)$ exists before u, or no $\texttt{recv}(p, r, s, d)$ exists before t but $\texttt{recv}(p, r, s, d)$ exists before u, then it is simple to verify that $\tau^{pq}_{rs}(t) \leq \tau^{pq}_{rs}(u)$. Let $t \leq u$, and let k, l denote the "k's", corresponding to t and u respectively, in the definition of τ^{pq}_{rs}, i.e., $a_k = \texttt{recv}(p, r, s, d)$, $k \leq t$, $a_l = \texttt{recv}(p, r, s, d')$, $l \leq u$. Because $t \leq u$, clearly $k \leq l$. By Lemma 4.4 (4), $k' = cause_{r_c}(k) \leq cause_{r_c}(l) = l'$. Similarly, by Lemma 4.4 (3), $k'' = cause_{r_c}(k') \leq cause_{r_c}(l') = l''$. If $d = d'$ then $k'' \leq l''$ and $t \leq u$ implies that the largest index $j \leq t$ with the property that for all j' with $k'' \leq j' \leq j$, is less than the similar largest index $j \leq u$ with the property that for all j' with $l'' \leq j' \leq j$, hence $\tau^{pq}_{rs}(t) \leq \tau^{pq}_{rs}(u)$. If $d \neq d'$ then for all j' with $k'' \leq j' \leq \tau^{pq}_{rs}(t)$, $s_k.rs.t_{cur} = d$, means that $k'' \leq l''$ implies $\tau^{pq}_{rs}(t) < l'' \leq \tau^{pq}_{rs}(u)$. □

Abstract state. For a run r_a of $[\![\Pi]\!]^{\text{op-abs}}$ and a time $t \in \mathbb{N}$, we let $state_{abs}(r_a, t)$ be the following (estimate-value) pair: $state_{abs}(r_a, t) = (E^{\text{abs}}, V^{\text{abs}})$, where

- E^{abs} is the function of type $\mathcal{P} \to \mathcal{P} \to (\mathcal{P} \to \mathcal{P} \to D)$, given by $E(p)(q) = x^{pq}(t)$.
- V^{abs} is the function of type $\mathcal{P} \to \mathcal{P} \to D$, given by $V(p)(q) = x_{pq}(t)$.

Similarly, for a run $r_c = s_0 a_1 s_1 \cdots$ of $[\![\Pi]\!]^{\text{op}}$, and an index $0 \leq k < |r_c|$, we let $state_{op}(r_c, k)$ be the following pair: $state_{op}(r_c, k) = (E^{\text{con}}, V^{\text{con}})$, where

- E^{con} is the function of type $\mathcal{P} \to \mathcal{P} \to (\mathcal{P} \to \mathcal{P} \to D)$, given by $E^{\text{con}}(p)(q) = s_k.pq.gts$.
- V^{con} is the function of type $\mathcal{P} \to \mathcal{P} \to D$, given by $V(p)(q) = s_k.pq.t_{cur}$.

Let us call $state_{op}$ and $state_{abs}$ the "abstract state." The following lemma relates concrete and abstract runs via the abstract state.

Lemma 4.6 (Corresponding runs). *Let $\Pi = (\pi_p \mid p \in \mathcal{P})$ be a collection of policies. Let r_c be a run of $[\![\Pi]\!]^{op}$, and let r_a be the corresponding run. Then,*

$$\forall k. 0 \leq k < |r_c| \Rightarrow state_{op}(r_c, k) = state_{abs}(r_a, k)$$

Proof. By induction in k. The base case $k = 0$ is immediate.

Inductive step. Assume that for all $k' \leq k$, $state_{op}(r_c, k') = state_{abs}(r_a, k')$, where $k + 1 < |r_c|$. Show that $state_{op}(r_c, k + 1) = state_{abs}(r_a, k + 1)$.

- Case $a_{k+1} = \texttt{eval}(p, q)$ for some $p, q \in \mathcal{P}$. Since $state_{op}(r_c, k) = state_{abs}(r_a, k)$, we get $x^{pq}(k) = s_k.pq.gts$. Hence, since $k \in T^{pq}$, we get $x_{pq}(k + 1) = f_{pq}(x^{pq}(k)) = f_{pq}(s_k.pq.gts) = s_{k+1}.pq.t_{cur}$. Further, we have $x^{pq}(k+1)_{pq} = x_{pq}(k+1) = s_{k+1}.pq.t_{cur} = s_{k+1}.pq.gts(p)(q)$. For all $rs \neq pq$, $x^{pq}(k + 1)_{rs} = x^{pq}(k)_{rs} = s_k.pq.gts(r)(s) = s_{k+1}.pq.gts(r)(s)$.
- Case $a_{k+1} = \texttt{send}(p, q, s, v)$. Notice that send-actions don't affect the abstract state: $state_{op}(r, k+1) = state_{op}(r, k) = state_{abs}(r, k) = state_{abs}(r, k + 1)$.

- Case $a_{k+1} = \texttt{recv}(p, r, s, v)$. Note first that for all $u, v \in \mathcal{P}$, we have $x_{uv}(k+1) = x_{uv}(k) = s_k.uv.t_{cur} = s_{k+1}.uv.t_{cur}$. For the estimate-part, let $q \in \mathcal{P}$ be arbitrary, and notice first that for all $u, v, w \in \mathcal{P}$ if $u \neq p$ or $(v, w) \neq (r, s)$,

  $$s_{k+1}.uq.gts(v)(w) = s_k.uq.gts(v)(w) = x^{uq}(k)_{vw} = x^{uq}(k+1)_{vw}$$

  Furthermore, if $(p, q) = (r, s)$ then the abstract state is not affected, and we are done. So assume that $(p, q) \neq (r, s)$. We have

  $$s_{k+1}.pq.gts(r)(s) = v$$

  We must show that $x^{pq}(k+1)_{rs} = v$. We have $x^{pq}(k+1)_{rs} = x_{rs}(\tau^{pq}_{rs}(k+1))$, and we simply recall that for any m, if $m = \texttt{recv}(p, r, s, d)$, then, as noted in the definition of τ^{pq}_{rs}, $s_{\tau^{pq}_{rs}(m)}.pq.t_{cur} = d$. Now, since $\tau^{pq}_{rs}(k+1) \leq k+1$, there are two cases. If $\tau^{pq}_{rs}(k+1) \leq k$ then the i.h. implies

  $$x^{pq}(k+1)_{rs} = x_{rs}(\tau^{pq}_{rs}(k+1)) \overset{\text{(i.h.)}}{=} s_{\tau^{pq}_{rs}(k+1)}.rs.t_{cur} = v$$

  If $\tau^{pq}_{rs}(k+1) = k+1$ then simply note that since $a_{k+1} = \texttt{recv}(p, r, s, v)$ clearly $a_{k+1} \neq \texttt{eval}(r, s)$, which implies $k \notin T^{rs}$. Hence, we get $x_{rs}(\tau^{pq}_{rs}(k+1)) = x_{rs}(k+1) = x_{rs}(k)$, and we have

  $$x_{rs}(k) \overset{\text{(i.h.)}}{=} s_k.rs.t_{cur} = s_{k+1}.rs.t_{cur} = s_{\tau^{pq}_{rs}(k+1)}.rs.t_{cur} = v$$

□

Lemma 4.7. *For any fair run r_c of $[\![\Pi]\!]^{op}$, let r_a denote its corresponding run. Then,*

- *If r_c is infinite, then r_a is an infinite fair run of $[\![\Pi]\!]^{op\text{-}abs}$.*
- *If r_c is finite, then r_a is a finite fair run of $[\![\Pi]\!]^{op\text{-}abs}$.*

Proof. We prove each point separately. In the following "The Lemma" refers to Lemma 4.4.

- Let $r_c = s_0 a_1 s_1 \cdots$ be an infinite fair run of $[\![\Pi]\!]^{\mathrm{op}}$. We let $p, q \in \mathcal{P}$ be arbitrary, and prove that there are infinitely many $\texttt{eval}(p, q)$ events in r_c, which implies that T^{pq} is infinite. Note that it suffices to prove that there are infinitely many $\texttt{send}$-events in r_c, by the following. Assume there are infinitely many $\texttt{send}$-events in r_c, and note that since $\mathcal{P}$ is finite, there must exist $r, s, t \in \mathcal{P}$ so that $\texttt{send}(r, t, s, \_)$ events occur infinitely often (i.o.) in r_c. By The Lemma (2,3), cause_{r_c} maps the indexes of these $\texttt{send}(r, t, s, \_)$ events, injectively, to indexes k with $a_k = \texttt{eval}(r, s)$ *and* $s_{k-1}.rs.t_{cur} \neq s_k.rs.t_{cur}$, hence, such indexes occur i.o. in r_c. By The Lemma (2), also, $\texttt{send}(r, p, s, \_)$ events occur i.o. in r_c, hence, by the FIFO Lemma, $\texttt{recv}(p, \ldots)$ events occur i.o. in r_c. But this implies that $pq.\mathit{wake} = \texttt{true}$ infinitely often, and hence by The Lemma (1), we have $\texttt{eval}(p, q)$ infinitely often.

So let us prove that there are infinitely many $\texttt{send}$-events. Assume this is not the case, and let k be an arbitrary index so that there are no $\texttt{send}$ events after k. Note that by The Lemma (2) it suffices to prove that there is some $k' > k$ so that $a_{k'} = \texttt{eval}(u, v)$ and $s_{k'-1}.uv.t_{cur} \neq s_{k'}.uv.t_{cur}$, for some $u, v \in \mathcal{P}$. Now, because all message buffers are finite at time k, and no $\texttt{send}$ events occur later than k, then there can be only finitely many $\texttt{recv}$-events after k. So let $K \geq k$ be arbitrary so that there are no $\texttt{recv}$-events after K. By construction there can only be $\texttt{eval}$-events after K, but since r_c is infinite there must also be some $\texttt{eval}$ event with a change in the t_{cur} variable (otherwise all *wake* variables eventually become $\texttt{false}$).

Now, let $p, q, r, s \in \mathcal{P}$, and let $(t^j)_{j=0}^{\infty}$ be a sequence tending towards infinity, and let $K \in \mathbb{N}$ be arbitrary but fixed. We show that there exists j so that $\tau_{rs}^{pq}(t^j) \geq K$. If $(r, s) = (p, q)$ this is trivial as τ_{pq}^{pq} is the identity function. So assume this is not the case. We know that there are infinitely many $\texttt{eval}(r, s)$ events in r_c. There are three cases.

If there are no k with $a_k = \texttt{eval}(r, s)$ *and* $s_k.rs.t_{cur} \neq s_{k+1}.rs.t_{cur}$. Then for all $k \geq 0$ we have $a_k \neq \texttt{recv}(p, r, s, \_)$ and $s_k.rs.t_{cur} = \bot_{\sqsubseteq}$. Hence τ_{rs}^{pq} is the identity function, and we are done.

If there are some but only finitely many k with $a_k = \texttt{eval}(r, s)$ *and* $s_{k-1}.rs.t_{cur} \neq s_k.rs.t_{cur}$, let k_0 be the largest such, and let $v = s_{k_0}.rs.t_{cur}$. Note that for all $k' \geq k_0$ we have $s_{k'}.rs.t_{cur} = v$. Then by The Lemma (2) and the FIFO Lemma, let l be so that $a_l = \texttt{recv}(p, r, s, v)$ and $\mathit{cause}_{r_c}(\mathit{cause}_{r_c}(l)) = k_0$. Now we get,

$$\tau_{rs}^{pq}(t) = t \qquad \text{for all } t \geq l$$

In the final case, there are infinitely many k with $a_k = \texttt{eval}(r, s)$ *and* $s_k.rs.t_{cur} \neq s_{k+1}.rs.t_{cur}$. Hence by The Lemma (2), there are infinitely many $\texttt{send}(r, p, s, \_)$ events. By The Lemma (3), the injectivity of $cause_{r_c}$ on these events implies that for some $v \in D$, there exist a $\texttt{send}(r, p, s, v)$ event with index $k' > K$ so that also $k = cause_{r_c}(k') \geq K$. Hence by the FIFO Lemma, there exists a $\texttt{recv}(p, r, s, v)$ event with index $k'' > k'$ so that $cause(k'') = k'$. Now let $t^{j_0} > k''$, then by monotonicity of τ^{pq}_{rs}, we obtain $\tau^{pq}_{rs}(t^{j_0}) \geq \tau^{pq}_{rs}(k'') \geq k > K$.

- Now assume that r_c is finite fair. Clearly r_a is finite. We must show it is also fair. So let $p, q, r, s \in \mathcal{P}$ and consider $x_{rs}(\tau^{pq}_{rs}(t^*_{pq}))$. If $(p, q) = (r, s)$ then τ^{pq}_{rs} is the identity, and hence, by definition of t^*_{pq}, $x_{pq}(t^*) = x_{pq}(t^*_{pq}) = x_{pq}(\tau^{pq}_{rs}(t^*_{pq}))$. So assume that $(p, q) \neq (r, s)$. Assume, for the sake of contradiction, that $x_{rs}(\tau^{pq}_{rs}(t^*_{pq})) \neq x_{rs}(t^*)$. By Lemma 4.6, this implies that there must exist an index k with $a_k = \texttt{eval}(r, s)$ and $s_{k-1}.rs.t_{cur} \neq s_k.rs.t_{cur}$. Let K be the greatest index with this property (such a greatest index must exist since r_c is finite), and note that for all j with $K \leq j \leq t^*$, $s_j.rs.t_{cur} = x_{rs}(t^*) \overset{\text{(def)}}{=} d$. By Lemma 4.4 (2) and the FIFO Lemma, there exists $j_p > k_p > K$ so that $a_{j_p} = \texttt{recv}(p, r, s, d)$, $a_{k_p} = \texttt{send}(r, p, s, d)$, $cause_{r_c}(k_p) = K$ and $cause_{r_c}(j_p) = k_p$.

  Note, there cannot be a later index $j' > k_p$ with $a_{j'} = \texttt{recv}(p, r, s, \_)$, because then $cause_{r_c}(cause_{r_c}(j')) > cause_{r_c}(cause_{r_c}(k_p) = K$, which contradicts maximality of K. Hence, if $k_p \leq t^*_{pq}$ then $s_{t^*_{pq}}.pq.t_{cur} = d$, and by Lemma 4.6, $x_{pq}(\tau^{pq}_{rs}(t^*_{pq})) = d = x_{rs}(t^*)$: a contradiction. But, if $k_p > t^*_{pq}$ then $s_{k_p}.pq.wake = \texttt{true}$ and by Lemma 4.4 there must be a later $\texttt{eval}(p, q)$ event, which contradicts maximality of t^*_{pq}.

□

5 Correspondence between the Denotational and Operational Semantics

In this section, we present the main theorem of this paper: the operational semantics $[\![\cdot]\!]^{\text{op}}$ and the denotational semantics $[\![\cdot]\!]^{\text{den}}$ correspond, in the sense that the I/O automaton $[\![\Pi]\!]^{\text{op}}$ distributedly computes $[\![\Pi]\!]^{\text{den}}$ for any collection of policies Π.

Because of the correspondence between the abstract operational semantics and the concrete operational semantics, we can prove the main theorem by first proving that the abstract operational semantics "computes" the least fixed-point of the product function. To prove this, we first establish the following invariance property of the abstract system.

Proposition 5.1 (Invariance property of $[\![\cdot]\!]^{\text{op-abs}}$). *Let $\Pi = (\pi_p \mid p \in P)$ be a collection of policies. Let r be any run of $[\![\Pi]\!]^{op\text{-}abs}$. Then, for every time $t \in \mathbb{N}$ and for every $p, q \in \mathcal{P}$, we have*

- *approximation: $x^{pq}(t) \sqsubseteq [\![\Pi]\!]^{den}$,*
- *increasing: $x_{pq}(t) \sqsubseteq f_{pq}(x^{pq}(t))$, and*
- *monotonic: $\forall t' \leq t. x^{pq}(t') \sqsubseteq x^{pq}(t)$*

Proof. By induction in t.

Base. Since $x^{pq}(0) = (\bot_\sqsubseteq, \bot_\sqsubseteq, \ldots, \bot_\sqsubseteq)$, all three properties follow trivially.

Inductive step.

1. Show that $x^{pq}(t+1) \sqsubseteq [\![\Pi]\!]^{\text{den}} = \mathsf{lfp}\ \Pi_\lambda$. Let $r, s \in \mathcal{P}$ be arbitrary but fixed. By definition, $x^{pq}(t+1)_{rs} = x_{rs}(\tau^{pq}_{rs}(t+1))$. Now there are two cases.
   (a) $\exists t' \leq t. x_{rs}(\tau^{pq}_{rs}(t+1)) = f_{rs}(x^{rs}(t'))$. Since $t' \leq t$ the induction hypothesis (i.h.) implies that $x^{rs}(t') \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$, hence $f_{rs}(x^{rs}(t')) \sqsubseteq f_{rs}(\mathsf{lfp}\ \Pi_\lambda) = (\mathsf{lfp}\ \Pi_\lambda)_{rs}$.
   (b) No such t' exists. Then $x_{rs}(\tau^{pq}_{rs}(t+1)) = \bot_\sqsubseteq$.

2. Show that $x_{pq}(t+1) \sqsubseteq f_{pq}(x^{pq}(t+1))$. Again there are two cases. In both cases we will assume that $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$, which we prove later (note that this is essentially the 'monotonic'-property).
   (a) If $t \notin T^{pq}$ then $x_{pq}(t+1) = x_{pq}(t)$. Now the i.h. implies that $x_{pq}(t) \sqsubseteq f_{pq}(x^{pq}(t))$. Now, monotonicity of f_{pq} together with $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$ implies $f_{pq}(x^{pq}(t)) \sqsubseteq f_{pq}(x^{pq}(t+1))$.
   (b) If $t \in T^{pq}$ then $x_{pq}(t+1) = f_{pq}(x^{pq}(t))$. Now since we assume $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$, monotonicity of f_{pq} implies $f_{pq}(x^{pq}(t)) \sqsubseteq f_{pq}(x^{pq}(t+1))$.

3. Show that $\forall t' \leq t+1.\ x^{pq}(t') \sqsubseteq x^{pq}(t+1)$. Note that by the i.h., it suffices proving $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$. So let $r, s \in \mathcal{P}$ be arbitrary but fixed. Show that $x^{pq}(t)_{rs} \sqsubseteq x^{pq}(t+1)_{rs}$. We have

$$x^{pq}(t)_{rs} = x_{rs}(\tau^{pq}_{rs}(t)) \quad \text{and} \quad x^{pq}(t+1)_{rs} = x_{rs}(\tau^{pq}_{rs}(t+1))$$

Note that since τ^{pq}_{rs} is monotonically increasing, we have $\tau^{pq}_{rs}(t) \leq \tau^{pq}_{rs}(t+1)$. Note also, that if $\tau^{pq}_{rs}(t+1) \leq t$ we can just refer to the i.h., and we are done. So assume, finally, that $\tau^{pq}_{rs}(t+1) = t+1$. Again, there are two cases.

   (a) If $t \notin T^{rs}$ then $x_{rs}(t+1) = x_{rs}(t)$, and we can simply refer to the induction hypothesis.

   (b) If $t \in T^{rs}$ then $x_{rs}(t+1) = f_{rs}(x^{rs}(t))$. By the i.h., $x_{rs}(\tau^{pq}_{rs}(t)) = x^{rs}(\tau^{pq}_{rs}(t))_{rs} \sqsubseteq x^{rs}(t)_{rs} = x_{rs}(t)$ (the i.h. applies since $\tau^{pq}_{rs}(t) \leq t$). Now we are done, because we have already proved that $x_{rs}(t) \sqsubseteq f_{rs}(x^{rs}(t)) = x_{rs}(t+1)$.

□

We are now able to prove that the abstract operational semantics of Π converges to $\mathsf{lfp}\ \Pi_\lambda$. However, we prove instead a slightly more general result. We use the following definition of an information approximation.

Definition 5.1 (Information Approximation). *Let $(X, \sqsubseteq)$ be a CPO with bottom $\bot_\sqsubseteq$. Let $f : X \to X$ be any continuous function. An element $x \in X$ is an* information approximation *for f if*

$$x \sqsubseteq \mathsf{lfp}\ f \qquad \textit{and} \qquad x \sqsubseteq f(x)$$

Lemma 5.1. *Let $(X, \sqsubseteq)$ be a CPO with bottom $\bot_\sqsubseteq$. Let $f : X \to X$ be any continuous function and $\hat{x} \in X$ an information approximation for f. Then, $\{f^k(\hat{x}) \mid k \in \mathbb{N}\}$ is a chain and*

$$\bigsqcup_k f^k(\hat{x}) = \mathsf{lfp}\ f$$

Proof. A simple induction proof shows that for all k we have $f^k(\hat{x}) \sqsubseteq f^{k+1}(\hat{x})$ and $f^k(\hat{x}) \sqsubseteq \mathsf{lfp}\ f$. Hence $\{f^k(\hat{x}) \mid k \in \mathbb{N}\}$ is a chain, and

$$\bigsqcup_k f^k(\hat{x}) \sqsubseteq \mathsf{lfp}\ f$$

To see that $\bigsqcup_k f^k(\hat{x})$ is a fixed point for f, note that by continuity,

$$f(\bigsqcup_k f^k(\hat{x})) = \bigsqcup_k f^{k+1}(\hat{x}) = \bigsqcup_k f^k(\hat{x})$$

□

Proposition 5.2 (Convergence of $[\![\cdot]\!]^{\text{op-abs}}$). *Let* $\Pi = (\pi_p \mid p \in P)$ *be a collection of policies. Let* $\hat{d} : \mathcal{P} \to \mathcal{P} \to D$ *be an information approximation for* Π_λ*. Let* r *be any fair run of* $[\![\Pi]\!]^{op\text{-}abs}$ *with initial solution estimate* $x(0) = \hat{d}$*. Then the sequence* $\{x(t)\}_{t\in\mathbb{N}}$ *has a limit point, and* $\lim_t x(t) = \mathsf{lfp}\ \Pi_\lambda$*.*

Proof. First we must show that the sequence $\{x(t)\}_t$ actually has a limit. We show that $x(0) \sqsubseteq x(1) \sqsubseteq \cdots \sqsubseteq x(t) \sqsubseteq \cdots$, i.e., $\{x(t)\}_t$ is an increasing omega chain. This follows from Proposition 5.1 since $x(t) = (\ldots, x_{pq}(t), \ldots) = (\ldots, x^{pq}(t)_{pq}, \ldots)$ and we have $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$ for all t. Now to show that $\lim_t x(t)$ (which is actually $\bigsqcup_t x(t)$) is a fixed point of Π_λ, we shall invoke the Asynchronous Convergence Theorem of Bertsekas. Define a sequence of subsets of X^n, $X(0) \supseteq X(1) \supseteq \cdots \supseteq X(k) \supseteq X(k+1) \supseteq \cdots$ by

$$X(k) = \{y \in D^n \mid \Pi_\lambda^k(\hat{d}) \sqsubseteq y \sqsubseteq \mathsf{lfp}\ \Pi_\lambda\}$$

Note that $X(k+1) \subseteq X(k)$ follows from the fact that $\Pi_\lambda^k(\hat{d}\,) \sqsubseteq \Pi_\lambda^{k+1}(\hat{d}\,)$ for any $k \in \mathbb{N}$, which, in turn, holds since $\hat{d}$ is an information approximation. For the synchronous convergence condition, assume that $y \in X(k)$ for some $k \in \mathbb{N}$. Since $\Pi_\lambda^k(\hat{d}) \sqsubseteq y \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$, we get by monotonicity $\Pi_\lambda^{k+1}(\hat{d}\,) \sqsubseteq \Pi_\lambda(y) \sqsubseteq \Pi_\lambda(\mathsf{lfp}\ \Pi_\lambda) = \mathsf{lfp}\ \Pi_\lambda$.

Now, let $(y^k)_{k\in\mathbb{N}}$ be a converging sequence so that $y^k \in X(k)$ for every k. Then, for all k we have $\Pi_\lambda^k(\hat{d}) \sqsubseteq y^k \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$. This implies that $\mathsf{lfp}\ \Pi_\lambda = \bigsqcup_k \Pi_\lambda^k(\hat{d}) \sqsubseteq \bigsqcup_k y^k \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$, and hence $\bigsqcup_k y^k = \mathsf{lfp}\ \Pi_\lambda$. This means that $\lim_k y^k$ is a fixed point of Π_λ.

The box condition is easy:

$$X(k) = \prod_{i=1}^{n} \{y(i) \mid y \in D^n \text{ and } \Pi_\lambda^k(\hat{d}\,) \sqsubseteq y \sqsubseteq \mathsf{lfp}\,\Pi_\lambda\ \}$$

However, this only proves that the system converges to some fixed point $x^* = \lim_t x(t)$ of Π_λ. But note that the invariance property (Proposition 5.1) implies that $x^{pq}(t) \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$ for all t. Hence, x^* is a fixed point of Π_λ, and $x^* \sqsubseteq \mathsf{lfp}\ \Pi_\lambda$. So we must have $x^* = \mathsf{lfp}\ \Pi_\lambda$. □

We are now able to prove the main theorem of this paper: the operational semantics is correct in the sense that the I/O automaton $[\![\Pi]\!]^{\text{op}}$ "computes" the least fixed-point of the function Π_λ, and, hence, the operational and denotational semantics agree.

Theorem 5.1 (Correspondence of semantics). *Let* Π *be any collection of policies, indexed by a finite set* $\mathcal{P}$ *of principal identities. Let* $r = s_0\pi_1 s_1 \pi_2 s_2 \cdots$ *be any fair run of the operational semantics of* Π, $[\![\Pi]\!]^{op}$. *Let* $state_{op}(r,k) = (E^k, V^k)$, *then we have*

- $\{V^k \mid k \in \mathbb{N}\}$ *is a chain in* $(\mathcal{P} \to \mathcal{P} \to X, \sqsubseteq)$.
- $\bigsqcup_{k\in\mathbb{N}} V^k = [\![\Pi]\!]^{den}$.

Proof. First, map r_c to its corresponding run r_a. This is a fair run of $[\![\Pi]\!]^{\text{op-abs}}$ by Lemma 4.7. By Lemma 4.6, $\{x(t)\}_{t\in\mathbb{N}} = \{V^k\}_{k\in\mathbb{N}}$, and by the Proposition 5.2 $\{x(t)\}_{t\in\mathbb{N}}$ has a limit which is $\mathsf{lfp}\ \Pi_\lambda = [\![\Pi]\!]^{\text{den}}$. $\square$

Corollary 5.1. *Let* Π *be any collection of policies over trust structure* $(D, \preceq, \sqsubseteq)$, *indexed by a finite set* $\mathcal{P}$ *of principal identities. If the CPO* $(D, \sqsubseteq)$ *is of finite height, then any fair run* r *of* $[\![\Pi]\!]^{op}$ *is finite, and if* N *is the length of* r, *and* $state_{op}(r, N-1) = (E, V)$, *then* $V = [\![\Pi]\!]^{den}$.

Proof. Let r' denote the corresponding run of $r = s_0 a_1 s_1 \cdots$. Proposition 5.1 implies that $x(t)$ is an increasing chain. When $(D, \sqsubseteq)$ has finite height, there exists some t_0 so that for all $t \geq t_0$, we have $x(t) = x(t_0)$. But by Lemma 4.6 then for all t with $t_0 \leq t < |r_c|$ we have $s_t.pq.t_{cur} = s_{t+1}.pq.t_{cur}$ for all $p, q \in \mathcal{P}$. Hence there can only be finitely many `send` actions after t_0, and hence only finitely many `recv` actions after t_0. But then there can only be finitely many `eval` actions in r, and, hence r must be finite. Since $\bigsqcup x(t) = x(t_0)$ the correspondence theorem implies that $V = [\![\Pi]\!]^{\text{den}}$. $\square$

References

[1] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall International Editions. Prentice-Hall, Inc., 1989.

[2] Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A formal model for trust in dynamic networks. In *Proceedings from Software Engineering and Formal Methods (SEFM'03).* IEEE Computer Society Press, 2003.

[3] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, NY, 2000.

[4] Stephen J. Garland, A. Lynch Nancy, Joshua Tauber, and Mandana Vaziri. IOA user guide and reference manual. Technical Report MIT-LCS-TR-961, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), Cambridge, MA, December 2004.

[5] Karl Krukow. On foundations for dynamic trust management. Unpublished PhD Progress Report, available online: `http://www.brics.dk/~krukow`, 2004.

[6] Karl Krukow and Andrew Twigg. Distributed approximation of fixed-points in trust structures. In *Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 805–814. IEEE, 2005.

[7] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[8] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–151. ACM Press, 1987.

[9] Mogens Nielsen and Karl Krukow. On the formal modelling of trust in reputation-based systems. In J. Karhumäki, H. Maurer, G. Paun, and G. Rozenberg, editors, *Theory Is Forever: Essays Dedicated to Arto Salomaa*, volume 3113 of *Lecture Notes in Computer Science*, pages 192–204. Springer Verlag, 2004.

[10] Stephen Weeks. Understanding trust management systems. In *Proceedings from the 2001 IEEE Symposium on Security and Privacy*, pages 94–106. IEEE Computer Society Press, 2001.

[11] Glynn Winskel. *Formal Semantics of Programming Languages : an introduction.* Foundations of computing. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.

Recent BRICS Report Series Publications