

# An Operational Semantics for Trust Policies<sup>\*</sup>

Karl Krukow<sup>\*\*</sup>

BRICS<sup>\*\*\*</sup>  
University of Aarhus,  
Denmark,  
[krukow@brics.dk](mailto:krukow@brics.dk)

**Abstract.** In the trust-structure framework for trust management, principals specify their trusting relationships 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 policies, there is always a unique global trust-state, consistent 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, I/O automata are a natural model of asynchronous distributed computations, and thus the semantics provides an asynchronous algorithm for distributedly computing the trust-state, suitable in the application environment.

## 1 Introduction

The trust-structure framework was introduced by Carbone, Nielsen and Sassone as a formal model for trust management in global computing environments [1]. In the framework, principals use ‘trust’ as basis for making decisions about interaction with other principals. Trust is defined formally in terms of a *trust structure*, 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

---

<sup>\*</sup> Extended Abstract, BRICS Technical Report, RS-05-30, <http://www.brics.dk/RS/05/30>.

<sup>\*\*</sup> Supported by SECURE: Secure Environments for Collaboration among Ubiquitous Roaming Entities, EU FET-GC IST-2001-32486.

<sup>\*\*\*</sup> BRICS: Basic Research in Computer Science ([www.brics.dk](http://www.brics.dk)), funded by the Danish National Research Foundation.

principal may have in another, for a particular application. As a simple example  $D = \{\text{high}, \text{mid}, \text{low}, \text{unknown}\}$  could be a set of trust values, but certainly trust values may have a much richer structure [1, 2]. 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  $\mathcal{P} \rightarrow D$  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  $\text{gts} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$  to local trust-states  $\text{lts} : \mathcal{P} \rightarrow D$ .

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. 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 applying domain theory, known from programming language semantics [1, 3]. Essentially, the theory ensures that the mutually recursive equations of the trust policies have a unique “least” solution. However, the theory gives no clue as to how principals can actually compute the trust values; the formal denotational semantics of policies needs a corresponding formal operational semantics.

In the following, we present in more detail the trust-structure framework, explaining how the problem of cyclic trust policies is solved (denotationally). Before presenting our actual contribution, we motivate further why computing the trust values is a non-trivial problem, especially in a global computing environment. Note, all proofs are omitted due to space constraints. Proofs are given in detail in the associated technical report [4].

## 1.1 The Trust-Structure Framework

We briefly explain the ideas behind the trust structure framework. For more detailed descriptions, we refer to the bibliography [1, 5].

The trust structure framework considers a set  $\mathcal{P}$  of principal *identities*. Principals are the entities of an application, and may encompass people, programs, public keys, etc. Principals assign certain degrees of trust to other principals.

These degrees are drawn from a set  $D$  of possible trust values, which is a parameter of the framework, with different instantiations in different applications. A *trust structure* is a triple  $T = (D, \preceq, \sqsubseteq)$  consisting of a set  $D$  of such trust values, ordered by two partial orderings: the trust ordering ( $\preceq$ ) and the information ordering ( $\sqsubseteq$ ). Intuitively,  $c \preceq d$  means that  $d$  denotes at-least as high a trust degree as  $c$ . Instead, the information ordering introduces a notion of refinement;  $c \sqsubseteq d$  is intended to mean that  $c$  can be refined into  $d$ . For  $D = \{\mathbf{high}, \mathbf{mid}, \mathbf{low}, \mathbf{unknown}\}$ , one could have,  $\mathbf{low} \preceq \mathbf{mid} \preceq \mathbf{high}$ , perhaps  $\mathbf{low} \preceq \mathbf{unknown} \preceq \mathbf{high}$ , and  $\mathbf{unknown} \sqsubseteq \mathbf{low}, \mathbf{mid}, \mathbf{high}$ ; whereas  $\mathbf{high}$ ,  $\mathbf{low}$  and  $\mathbf{mid}$  would be unrelated by  $\sqsubseteq$ .

The goal of the framework is to define, given  $\mathcal{P}$  and  $T$ , a unique *global trust-state*, to represent every principal's trust in every other principal. Mathematically, this amounts to specifying a function  $\overline{\mathbf{gts}} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D = \mathbf{GTS}$ . Principals control how this function is defined by specifying their *trust policies*. Formally, trust policies are functions  $\pi_p$  of type  $\mathbf{GTS} \rightarrow (\mathcal{P} \rightarrow D)$  (the set  $\mathcal{P} \rightarrow D$  is written  $\mathbf{LTS}$ ). The interpretation of  $\pi_p$  is the following. *Given that all principals assign trust-values as specified in the global trust-state  $\overline{\mathbf{gts}}$ , then  $p$  assigns trust values as specified in vector  $\pi_p(\overline{\mathbf{gts}}) : \mathcal{P} \rightarrow D$ .* Note, as discussed in the introduction, that trust policies  $\pi_p$  map *global* trust states to *local* trust states, and hence  $p$ 's policy may depend on other principal's policies.

Since the collection of all trust policies  $\Pi = (\pi_p \mid p \in \mathcal{P})$  may contain cyclic policy-references, it is not obvious how to define the unique global trust-state  $\overline{\mathbf{gts}}$ . Clearly,  $\overline{\mathbf{gts}}$  should be consistent with all policies  $\pi_p$ . This amounts to requiring that it should satisfy the following fixed-point equation:  $\overline{\mathbf{gts}}(p) = \pi_p(\overline{\mathbf{gts}})$  for all  $p \in \mathcal{P}$ ; or equivalently:

$$\Pi_\lambda(\overline{\mathbf{gts}}) = \overline{\mathbf{gts}}$$

where  $\Pi_\lambda$  is the product function  $\Pi_\lambda = \langle \pi_p \mid p \in \mathcal{P} \rangle$ . Any  $\mathbf{gts} : \mathbf{GTS}$  satisfying this equation is *consistent* with the policies  $(\pi_p \mid p \in \mathcal{P})$ . This means that *any* fixed point of  $\Pi_\lambda$  is consistent with all policies  $\pi_p$ . But arbitrary  $\Pi_\lambda$ , may have multiple or even no fixed points.

The trust structure framework solves this problem by turning to simple domain theory. 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”). The framework then requires that all policies  $\pi_p : \mathbf{GTS} \rightarrow \mathbf{LTS}$  be *information continuous*, i.e. continuous with respect to  $\sqsubseteq$ . Since this implies that  $\Pi_\lambda$  is also information-continuous, and since  $(\mathbf{GTS}, \sqsubseteq)$  is a cpo with bottom, standard theory [3] tells us that  $\Pi_\lambda$  has a (unique) least fixed-point which we denote  $\text{lfp}_\sqsubseteq \Pi_\lambda$  (or simply  $\text{lfp} \Pi_\lambda$ ):

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

This global trust-state has the property that it is a fixed-point (i.e.,  $\Pi_\lambda(\text{lfp}_\sqsubseteq \Pi_\lambda) = \text{lfp}_\sqsubseteq \Pi_\lambda$ ) and that it is the (information-) least among fixed-points (i.e., for any

other fixed point  $\text{gts}$ ,  $\text{lfp}_{\sqsubseteq} II_{\lambda} \sqsubseteq \text{gts}$ ). Hence, for any collection  $II$  of trust policies, we can define the *global trust-state induced by that collection*, as  $\overline{\text{gts}} = \text{lfp } II_{\lambda}$ , which is well-defined by uniqueness.

## 1.2 The Operational Problem

As we have seen, the trust structure framework guarantees the existence of a unique global trust-state in *any* trust structure whenever all policies are information-continuous. However, in practice, mere existence is not sufficient; principals must be able to compute (or at least approximate) this global trust state. In particular, assuming that principal  $p$  needs to make a decision regarding interaction with another principal  $q$ , clearly,  $p$  needs information about  $\overline{\text{gts}}(p)(q)$ ,  $p$ 's trust in  $q$ . Note that since policies are distributed (each principal stores its own policy), one cannot simply use the standard iterative algorithm for computing the least fixed point.

In many applications, it is often sufficient to merely *approximate* the fixed-point value, e.g., computing  $\overline{\text{gts}}$  may not be necessary when only  $\overline{\text{gts}}(p)(q)$  is needed. Krukow and Twigg present a collection of techniques for approximating the idealized fixed-point  $\text{lfp } II_{\lambda}$  [5]. 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 paper 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 [6, 7], providing a formal operational foundation for trust policies, and formalizing the asynchronous distributed algorithm of Krukow and Twigg [5] in a semantic model. Our main theorem (Theorem 2) proves in this *formal* model, that even in infinite height over 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 simple automata for describing the fixed-point algorithm. Secondly, the model is operational and reasonably 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. [1], 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  $\text{op}_n^i$  for each  $n > 0$ . These are meant to denote functions  $\llbracket \text{op}_n^i \rrbracket^{\text{den}} : D^n \rightarrow D$ , continuous with respect to  $\sqsubseteq$ .

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

A policy  $\pi$  is essentially a list of pairs  $p : \tau$ , where  $p$  is a principal identity, and  $\tau$  is an expression defining the policy's trust-specification for  $p$ . Since we cannot assume that the writer of the policy 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  $\tau$ .” Note this could easily be extended to more practical constructs, say  $G : \tau$  meaning that  $\tau$  is the trust-specification for any member of the group (or role)  $G$ .

The syntactic category  $\tau$  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  $\text{op}_n^i(\tau_1, \dots, \tau_n)$  is the application of an  $n$ 'ary operator to the trust specifications  $(\tau_1, \dots, \tau_n)$ . 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 the following. We assume that for each of the function symbols  $\text{op}_n^i$ ,  $\llbracket \text{op}_n^i \rrbracket^{\text{den}}$  is a  $\sqsubseteq$ -continuous function of type  $D^n \rightarrow D$ . 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  $\pi_p$  is an information-continuous function  $\llbracket \pi_p \rrbracket^{\text{den}}$  of type  $\text{GTS} \rightarrow \text{LTS}$ .

$$\begin{aligned} \llbracket \star : \tau \rrbracket^{\text{den}} \text{ gts } q &= \llbracket \tau \rrbracket^{\text{den}} (\star \mapsto q) / \text{id}_{\mathcal{P}} \text{ gts} \\ \llbracket p : \tau, \pi \rrbracket^{\text{den}} \text{ gts } q &= \text{if } (q = p) \text{ then } \llbracket \tau \rrbracket^{\text{den}} (\star \mapsto p) / \text{id}_{\mathcal{P}} \text{ gts} \\ &\quad \text{else } \llbracket \pi \rrbracket^{\text{den}} \text{ gts } q \end{aligned}$$

$$\begin{aligned} \llbracket d \rrbracket^{\text{den}} \text{ env} &= \lambda \text{gts. } d \\ \llbracket p? q \rrbracket^{\text{den}} \text{ env} &= \lambda \text{gts. gts } (\text{env } p) (\text{env } q) \\ \llbracket \text{op}_n^i(\tau_1, \dots, \tau_n) \rrbracket^{\text{den}} \text{ env} &= \llbracket \text{op}_n^i \rrbracket^{\text{den}} \circ \langle \llbracket \tau_1 \rrbracket^{\text{den}} \text{ env}, \dots, \llbracket \tau_n \rrbracket^{\text{den}} \text{ env} \rangle \end{aligned}$$

As expected, the denotational semantics of the collection  $\Pi$  is the least fixed-point of the function  $\Pi_\lambda = \langle \llbracket \pi_p \rrbracket^{\text{den}} \mid p \in \mathcal{P} \rangle$ .

$$\llbracket (\pi_p \mid p \in \mathcal{P}) \rrbracket^{\text{den}} = \text{lfp}_{\sqsubseteq} \langle \llbracket \pi_p \rrbracket^{\text{den}} \mid p \in \mathcal{P} \rangle$$

## 3 Two Models of Distributed Computation

In the following sections, we will be using two different models of distributed computation to provide an operational semantics for the basic policy language. 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  $\Pi$  is translated into an I/O Automaton, denoted  $\llbracket \Pi \rrbracket^{\text{op}}$ . I/O Automata are a form of labeled transition-systems, suitable for modelling and reasoning about distributed discrete event systems [6, 7]. We shall define also another translation  $\llbracket \Pi \rrbracket^{\text{op-abs}}$  into what we call a Bertsekas Abstract Asynchronous System (BAAS). There will be a tight correspondence between the “abstract” operational semantics  $\llbracket \Pi \rrbracket^{\text{op-abs}}$  and the actual operational semantics  $\llbracket \Pi \rrbracket^{\text{op}}$ . The reason for introducing  $\llbracket \cdot \rrbracket^{\text{op-abs}}$  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  $\llbracket \Pi \rrbracket^{\text{op}}$ .

The I/O automata model of Lynch et al. is probably the most well-known of the two models we consider, and therefore (due to space constraints) we refer to Lynch et al. [6, 7] regarding this model (a brief introduction is also provided in the associated technical report [4]).

### 3.1 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  $X_1, X_2, \dots, X_n$ , and  $n$  functions  $f_1, f_2, \dots, f_n$ , where for each  $i$ ,  $f_i : \prod_{j=1}^n X_j \rightarrow X_i$ . 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  $f = \langle f_1, f_2, \dots, f_n \rangle : X \rightarrow X$ . The objective of a BAAS is to find a fixed point  $x^*$  of  $f$ .

We can think of each  $i \in [n] = \{1, 2, \dots, n\}$  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  $x^i = (x_1^i, x_2^i, \dots, x_n^i)$  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 [8].

*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  $x_i : \mathbb{N} \rightarrow X_i$ . 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  $(x_1(t), x_2(t), \dots, x_n(t))$ .
3. For each  $i \in [n]$ , a collection of (estimate) functions  $(\tau_j^i)_{j \in [n]}$ , each of type  $\tau_j^i : \mathbb{N} \rightarrow \mathbb{N}$ , 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.

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

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  $\tau$ -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 1 (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  $\lim_{k \rightarrow \infty} \tau_j^i(t_k) = \infty$  for every  $j \in [n]$ .

We are only concerned with infinite runs in this paper (the technical report shows that our theory is also valid for finite runs [4]). When an infinite run is fair, each node is guaranteed to recompute infinitely often. Moreover, all old estimate values are always eventually updated.

*The Asynchronous Convergence Theorem.* The BAAS is a model of asynchronous distributed algorithms. The Asynchronous Convergence Theorem (ACT) (Proposition 6.2.1 of Bertsekas' book [8]) 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  $\perp_{\underline{\square}}^n \sqsubseteq f(\perp_{\underline{\square}}^n) \sqsubseteq \dots$  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 [8]). Consider a BAAS with  $X = \prod_{i=1}^n X_i$ , and  $f : X \rightarrow X$  any function with  $f = \langle f_1, f_2, \dots, f_n \rangle$ .

**Definition 2 (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  $n$  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 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  $\llbracket \cdot \rrbracket^{\text{op}}$  mapping a collection  $\Pi$  of policies of the basic language to an I/O automaton  $\llbracket \Pi \rrbracket^{\text{op}}$ . We introduce also an “abstract” operational semantics, which is given by another semantic function  $\llbracket \cdot \rrbracket^{\text{ob-abs}}$  mapping  $\Pi$  to a BAAS. The systems  $\llbracket \Pi \rrbracket^{\text{op}}$  and  $\llbracket \Pi \rrbracket^{\text{ob-abs}}$  will correspond in a “simulation-like” manner: runs of  $\llbracket \Pi \rrbracket^{\text{op}}$  can be faithfully matched by corresponding runs of  $\llbracket \Pi \rrbracket^{\text{ob-abs}}$  (in a formal sense, described later in this section).

### 4.1 $\llbracket \cdot \rrbracket^{\text{op}}$ Translation: An Operational Semantics

The concrete operational semantics can be seen as an asynchronous distributed algorithm in which the principals  $\mathcal{P}$  perform a computation of  $\llbracket \Pi \rrbracket^{\text{den}}$ . Each principal  $p \in \mathcal{P}$  will be computing its local values (i.e.,  $\llbracket \Pi \rrbracket^{\text{den}}(p)(q)$  for each  $q \in \mathcal{P}$ ). We present now the algorithm, as described previously by Krukow and Twigg [5]. We then give an I/O automata version of this algorithm, which is used for formal proofs.

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

$p.t_{cur} = p.t_{old} = \perp_{\square}$ , and the matrix is also initialized with  $\perp_{\square}$ . For all nodes  $pq$  and  $rs$ , when  $pq$  receives a message from  $rs$ , it stores this message in  $pq.gts(r)(s)$  (messages are always values in  $D$ , i.e., ‘updates’).

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  $pq.gts$  (function  $f_{pq}$  corresponds to  $p$ ’s policy entry for  $q$ ). If there is no change in the resulting value (compared to  $pq.t_{old}$ ), it will go to the *sleep* state (unless a new message was received since the computation). Otherwise, if a new value resulted from the computation, an update is sent to all nodes.

In the I/O automaton version of this algorithm, each principal  $p$  is modelled as a collection of nodes ( $pq \mid q \in \mathcal{P}$ ), where component  $pq$  of  $p$  is responsible for computing  $p$ ’s value for  $q$ . The sending of a message  $d$  from component  $pq$  to, say  $rs$ , is represented by the I/O-automaton action  $\mathbf{send}(p, r, q, d)$  (note this is independent of  $s$  because all components  $rs$  will receive this update simultaneously). The semantic function  $\llbracket \cdot \rrbracket^{\text{op}}$  maps a collection of trust policies from the basic language to an I/O automaton. The semantics uses two parameterized I/O automata:  $\mathbf{Channel}(p, r, q)$  (modelling a communication medium for sending values from  $pq$  to  $r$ ), and  $\mathbf{IOTemplate}(p, q, f_{pq})$ , where  $p, q, r \in \mathcal{P}$  and  $f : (\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow D$  is a continuous function. The latter  $\mathbf{IOTemplate}(p, q, f_{pq})$  is the component we denoted “ $pq$ ” when  $f_{pq}$  is principal  $p$ ’s policy for principal  $q$ , i.e, the entry for  $q$  in  $\pi_p$ .

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

$$\llbracket (\pi_p \mid p \in \mathcal{P}) \rrbracket^{\text{op}} = \left( \prod_{p \in \mathcal{P}} \llbracket \pi_p \rrbracket_{p, \emptyset}^{\text{op}} \right) \times \prod_{p, r, q \in \mathcal{P}} \mathbf{Channel}(p, r, q)$$

The Channel automaton is described syntactically (in a ‘IOA’-like language [9]) below.

```

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}

```

A principal  $p$  is represented as the automaton  $\llbracket \pi_p \rrbracket_{p,\emptyset}^{\text{op}}$  which is the composition of the collection of automata  $\text{IOTemplate}(p, q, f_{pq})$  for  $q \in \mathcal{P}$  and where  $f_{pq}(\text{gts}) = \llbracket \pi_p \rrbracket^{\text{den}} \text{gts } q$ . As mentioned, the component  $\text{IOTemplate}(p, q, f_{pq})$ , which we denote simply as “ $pq$ ”, is responsible for computing principal  $p$ 's trust value for principal  $q$ , i.e., the value  $\overline{\text{gts}}(p)(q)$ .

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

The most important automata are the  $\text{IOTemplate}$  automata. The parameterized automaton is described syntactically below. Note the close correspondence between the high-level algorithm description in the beginning of this section, and the actions of the following automaton. Most importantly, action  $\text{eval}(p, q)$  represents the node  $pq$  recomputing its current value. The fairness partition of  $\text{IOTemplate}(p, q, f_{pq})$  ensures that the  $\text{eval}(p, q)$  action is always eventually executed once it is enabled. Similarly,  $\text{send}(p, r, q, pq.t_{\text{cur}})$  is always eventually executed when variable  $pq.\text{send}(r)$  is **true**.

```

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)
      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

```

**Lemma 1 (Composability).** *If all policies of  $\Pi$  are well-formed, then all the automata occurring in the definition of  $\llbracket \Pi \rrbracket^{\text{op}}$  have compatible signatures, and, hence, are composable.*

#### 4.2 Reasoning about the semantics

For a run  $r_c$  of  $\llbracket \Pi \rrbracket^{\text{op}}$ , it is possible to define a “causality” function,  $\text{cause}_{r_c}$ , mapping each index  $k > 0$  to a smaller index  $k'$ . If  $\text{cause}_{r_c}(k) = k' > 0$  we say that action  $a_{k'}$  causes action  $a_k$ . The causality function gives a link between action with index  $k$ , say a **recv**-action, and a previous action of the automaton, e.g., a **send**-action. The definition of the  $\text{cause}_{r_c}$  is of some complexity, and we choose to leave it out here as it provides no deep insight into the understanding of the semantics (for the exact definition we refer to the technical report).

**Lemma 2 (Cause).** For a run  $r_c = s_0 a_1 s_1 a_2 s_2 \dots$  of  $\llbracket \Pi \rrbracket^{\text{op}}$  one can define a function  $\text{cause}_{r_c} : \mathbb{N} \rightarrow \mathbb{N}$ , so that the following holds.

- If action  $k$  is an **eval** action, as we are not interested in linking **eval** actions with previous events, we have  $\text{cause}_{r_c}(k) = 0$ .
- If action  $k$  is of form **send**( $p, r, q, d$ ) for some  $p, q, r \in \mathcal{P}$ ,  $d \in D$ , then the causing event  $j$  is an **eval**( $p, q$ ) event, and we have  $s_{j-1}.pq.t_{\text{cur}} \neq s_j.pq.t_{\text{cur}}$ , and  $s_j.pq.wake = \text{true}$ .
- If action  $k$  is of form **recv**( $p, r, s, d$ ) for some  $p, r, s \in \mathcal{P}$ ,  $d \in D$ , then the causing event  $j$  is a **send**( $r, p, s, d$ ) event, and we have  $s_j.rs.t_{\text{cur}} = d$ , and  $s_j.rs.send(p) = \text{false}$ .

#### 4.3 $\llbracket \cdot \rrbracket^{\text{op-abs}}$ 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  $\llbracket \Pi \rrbracket^{\text{abs-op}}$  consists of the set  $D$  of trust values, a collection of  $n = |\mathcal{P}|^2$  functions  $f_{pq} : D^n \rightarrow D$  (the functions are indexed by pairs  $pq$  where  $p, q \in \mathcal{P}$ ). The functions  $f_{pq}$  are given by the policies,

$$f_{pq}(\text{gts}) = \llbracket \pi_p \rrbracket^{\text{den}} \text{gts } q$$

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

In the rest of this paper, we shall not distinguish between  $[n] = \{1, 2, \dots, n\}$  and the set  $\mathcal{P} \times \mathcal{P}$ , nor shall we distinguish between  $D^n$  and  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . 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  $\Pi_\lambda$ ).

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 Relating the abstract and concrete operational semantics

The two translations  $\llbracket \cdot \rrbracket^{\text{op}}$  and  $\llbracket \cdot \rrbracket^{\text{op-abs}}$  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  $\llbracket \Pi \rrbracket^{\text{op}}$  to “corresponding” runs of  $\llbracket \Pi \rrbracket^{\text{op-abs}}$ .

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

1. For any  $p, q \in \mathcal{P}$ ,  $T^{pq}$  is defined as  $\{k - 1 \mid k \in \mathbb{N}, a_k = \text{eval}(p, q)\}$ . That is, the update-times of  $pq$  are the indexes of pre-states of  $\text{eval}(p, q)$  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  $\tau_{pq}^{pq} : \mathbb{N} \rightarrow \mathbb{N}$  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, i.e.,  $x^{pq}(t)_{pq} = x_{pq}(\tau_{pq}^{pq}(t)) = x_{pq}(t)$ .
3. For each  $p, q \in \mathcal{P}$  and each  $r, s \in \mathcal{P}$  with  $(r, s) \neq (p, q)$ , the function  $\tau_{rs}^{pq} : \mathbb{N} \rightarrow \mathbb{N}$  is given by the following. Let  $t \in \mathbb{N}$  be arbitrary but fixed. Let  $k \leq t$  be the *largest*, with the property that  $a_k = \text{recv}(p, r, s, d)$  for some  $d \in D$ .
  - (a) If no such index exists, then  $\tau_{rs}^{pq}(t)$  is defined as the largest  $j \leq t$  with the property that for all  $j'$  with  $0 \leq j' \leq j$  we have  $s_{j'}.rs.t_{cur} = \perp_{\square}$ .
  - (b) If such  $k$  exists, let  $k' = \text{cause}_{r_c}(k)$ . Note that  $a_{k'} = \text{send}(r, p, s, d)$ . Define  $\tau_{rs}^{pq}(t)$  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 that  $0 \leq \tau_{rs}^{pq}(t) \leq t$  is satisfied.
4. The value functions  $x_{pq} : \mathbb{N} \rightarrow D$  are given inductively. We have  $x_{pq}(0) = \perp_{\square}$ . 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 BAASs.

**Lemma 3.** *For any infinite fair run  $r_c$  of  $\llbracket II \rrbracket^{\text{op}}$ , the corresponding run of  $r_c$  is an infinite fair run of  $\llbracket II \rrbracket^{\text{op-abs}}$ .*

*Abstract state.* For a run  $r_a$  of  $\llbracket II \rrbracket^{\text{op-abs}}$  and a time  $t \in \mathbb{N}$ , we let  $\text{state}_{abs}(r_a, t)$  be the following (estimate-value) pair:  $\text{state}_{abs}(r_a, t) = (E^{\text{abs}}, V^{\text{abs}})$ , where

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

Similarly, for a run  $r_c = s_0 a_1 s_1 \dots$  of  $\llbracket II \rrbracket^{\text{op}}$ , and an index  $0 \leq k < |r_c|$ , we let  $\text{state}_{con}(r_c, k)$  be the following pair:  $\text{state}_{con}(r_c, k) = (E^{\text{con}}, V^{\text{con}})$ , where

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

Let us call  $\text{state}_{con}$  and  $\text{state}_{abs}$  the “abstract state.” The following lemma relates concrete and abstract runs via the abstract state.

**Lemma 4 (Corresponding runs).** *Let  $\Pi = (\pi_p \mid p \in \mathcal{P})$  be a collection of policies. Let  $r_c$  be a run of  $\llbracket II \rrbracket^{\text{op}}$ , and let  $r_a$  be the corresponding run. Then,*

$$\forall k. \quad \text{state}_{con}(r_c, k) = \text{state}_{abs}(r_a, k).$$

## 5 Correspondence between the Denotational and Operational Semantics

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

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

- *approximation:  $x^{pq}(t) \sqsubseteq \llbracket II \rrbracket^{\text{den}}$ ,*
- *increasing:  $x_{pq}(t) \sqsubseteq f_{pq}(x^{pq}(t))$ , and*
- *monotonic:  $\forall t' \leq t. x^{pq}(t') \sqsubseteq x^{pq}(t)$*

We are now able to prove that the abstract operational semantics of  $II$  converges to  $\text{lfp } II_\lambda$ .

**Proposition 2 (Convergence of  $\llbracket \cdot \rrbracket^{\text{op-abs}}$ ).** *Let  $II = (\pi_p \mid p \in P)$  be a collection of policies. Let  $\hat{d} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$  be  $\hat{d}(p)(q) = \perp_{\sqsubseteq}$ . Let  $r$  be any fair run of  $\llbracket II \rrbracket^{\text{op-abs}}$  with initial solution estimate  $x(0) = \hat{d}$ . Then the sequence  $\{x(t)\}_{t \in \mathbb{N}}$  has a limit point, and  $\lim_t x(t) = \text{lfp } II_\lambda$ .*

This, in turn, lets us prove the main theorem of this paper: the operational semantics is correct in the sense that the I/O automaton  $\llbracket II \rrbracket^{\text{op}}$  “computes” the least fixed-point of the function  $II_\lambda$ , and, hence, the operational and denotational semantics agree.

**Theorem 2 (Correspondence of semantics).** *Let  $II$  be any collection of policies, indexed by a finite set  $\mathcal{P}$  of principal identities. Let  $r = s_0 \pi_1 s_1 \pi_2 s_2 \dots$  be any fair run of the operational semantics of  $II$ ,  $\llbracket II \rrbracket^{\text{op}}$ . Let  $\text{state}_{\text{con}}(r, k) = (E^k, V^k)$ , then we have*

- $\{V^k \mid k \in \mathbb{N}\}$  *is a chain in  $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D, \sqsubseteq)$ .*
- $\bigsqcup_{k \in \mathbb{N}} V^k = \llbracket II \rrbracket^{\text{den}}$ .

*Proof.* First, map  $r_c$  to its corresponding run  $r_a$ . This is a fair run of  $\llbracket II \rrbracket^{\text{op-abs}}$  by Lemma 3. By Lemma 4,  $\{V^k\}_{k \in \mathbb{N}} = \{x(t)\}_{t \in \mathbb{N}}$ , and by the Proposition 2  $\{x(t)\}_{t \in \mathbb{N}}$  has a limit which is  $\text{lfp } II_\lambda = \llbracket II \rrbracket^{\text{den}}$ .

## 6 Conclusion

We have provided a formal operational semantics for a simple, yet general, language of trust policies. The operational semantics agrees with the denotational semantics of the trust structure framework. Furthermore, the operational semantics is practical: it is based on I/O automata, a formal low-level model of distributed algorithms; and the semantics itself coincides with the asynchronous algorithm of Bertsekas (as described *informally* by Krukow and Twigg [5]). The proof uses translation into BAAS's for proving correctness of algorithms formalized with I/O automata. This technique may be applicable for other algorithms. A simple continuation of this work is to also formalize the approximation techniques of Krukow and Twigg [5] in the I/O automata setting which we feel is a very appropriate model for algorithms in open distributed systems (e.g., global computing scenarios), as it provides a simple formal model of distributed computation as well as rigorous, compositional reasoning techniques.

## References

1. Carbone, M., Nielsen, M., Sassone, V.: A formal model for trust in dynamic networks. In: Proceedings from Software Engineering and Formal Methods (SEFM'03), IEEE Computer Society Press (2003)
2. Nielsen, M., Krukow, K.: On the formal modelling of trust in reputation-based systems. In Karhumäki, J., Maurer, H., Paun, G., Rozenberg, G., eds.: Theory Is Forever: Essays Dedicated to Arto Salomaa. Volume 3113 of Lecture Notes in Computer Science. Springer Verlag (2004) 192–204
3. Winskel, G.: Formal Semantics of Programming Languages : an introduction. Foundations of computing. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts (1993)
4. Krukow, K.: An operational semantics for trust policies. Technical Report RS-05-30, BRICS, University of Aarhus (2005) Available online: <http://www.brics.dk/RS/05/30>.
5. Krukow, K., Twigg, A.: Distributed approximation of fixed-points in trust structures. In: Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), IEEE (2005) 805–814
6. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM Press (1987) 137–151
7. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA (1996)
8. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice-Hall International Editions. Prentice-Hall, Inc. (1989)
9. Garland, S.J., Lynch, N.A.: Using I/O automata for developing distributed systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press, NY (2000) 285–312