

Karl Krukow · Mogens Nielsen

Trust Structures

Denotational and Operational Semantics

Abstract A general formal model for trust in dynamic networks is presented. The model is based on the trust structures of Carbone, Nielsen and Sassone: A domain theoretic generalisation of Weeks' framework for credential based trust management systems, e.g., KeyNote and SPKI. Collections of mutually referring trust policies (so-called "webs" of trust) are given a precise meaning in terms of an abstract domain-theoretic semantics. A complementary concrete operational semantics is provided using the well-known I/O-automaton model. The operational semantics is proved to adhere to the abstract semantics, effectively providing a distributed algorithm allowing principals to compute the meaning of a "web" of trust policies. Several techniques allowing sound and efficient distributed approximation of the abstract semantics are presented and proved correct.

Keywords Trust Management, Trust Structures, Foundations, Denotational and Operational Semantics, I/O Automata

1 An Introduction to Trust Management

The unique dynamic properties of the Internet imply that traditional theories and mechanisms for resource access control are often inappropriate as they are of a too static nature. For example, traditional access control consists of a policy specifying which subjects may access which objects, e.g., a user accessing a file in a UNIX file system. Apart from inflexibility and lack of expressive power, this approach assumes that resources are only accessed by a static set of known subjects (and that resources themselves are fixed); an assumption incompatible with open dynamic systems.

K. Krukow · M. Nielsen
BRICS: Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.
University of Aarhus, Denmark
E-mail: (krukow, mn)@brics.dk

Many distributed systems use a combination of access control lists and user authentication (usually implemented via some public key authentication protocol), i.e., deciding how to respond to a request to perform an action is done by authenticating the public key, effectively linking the key to a user identity, and then looking up in the access control list to see whether that identity is authorised to perform the action [3]. The security properties of current systems is often not verified, i.e., proofs of soundness of the security mechanism are lacking (e.g., statements of the form "if the system authorises an action requested by a public key then the key is controlled by user U and U is authorised to perform that action according to the access control list").

In Internet applications there is an extremely large set of entities that make requests, and this set is in constant change as entities join and leave networks. Furthermore, even if we could reliably decide *who* signed a given request, the problem of deciding whether or not access should be granted is not obvious: Should all requests from unknown entities be denied?

Blaze et al. [3] present a number of reasons why the traditional approach to authorisation is inadequate:

- *Authorisation is not necessarily Authentication + Access-Control-List: Authentication* deals with establishing identity. In traditional static environments, e.g., operating systems, the identity of an entity is well-known. In Internet applications this is often not the case. This means that if an access control list (ACL) is to be used, some form of authentication must first be performed. In distributed systems, often public-key based authentication protocols are used, which usually relies on centralised and global certification authorities.
- *Delegation*: Since the global scale of the Internet implies that each entity's security policy must encompass billions of entities, delegation is necessary to obtain scalable solutions. Delegation implies that entities may rely on other (semi) trusted entities for deciding how to respond to requests. In traditional approaches either delegation is not supported, or it

is supported in an inflexible manner where security policy is only specified at the last step of a delegation chain.

- *Expressive power and Flexibility*: The ACL-based approach to authorisation has proved not to be sufficiently expressive (with respect to desired security policies) or extensible. The result has been that security policies are often hard-coded into the application code, i.e., using the general purpose programming language that the application is written in. This has a number of implications: Changes in security policy often means rewriting and recompilation, and security reasoning becomes hard as security code is intertwined with application code (i.e., violating the principle of ‘separation of concerns’).
- *Locality*: Different Internet entities have different trust requirements and relationships. Hence, entities should be able to specify local security and trust policies, and security mechanisms should not enforce uniform and implicit policies or trusting relations.

Credential-based Trust Management. In contrast to the “access control list”-approach to authorisation, trust management is naturally distributed, and consists of a unified and general approach to specifying security policies, credentials and trusting relationships; it is backed up by general (application-independent) algorithms to implement these policies. The trust management approach is based on programmable security policies that specify access-control restrictions and requirements in a domain-specific programming language, leading to increased flexibility and expressive power.

Given a request r signed by a key k , the question we really want to answer is the following: “Is the information about key k such that the request r should be granted?” In principle, we do not care about *who* signed r , only whether or not sufficient information can be inferred to grant the request. In the trust management approach, one does not need to resolve the actual identity (e.g., the human-being believed to be performing the request); instead, one deals with the following question, known as the compliance-checking problem: “Does the set C of credentials prove that the request r complies with the local security policy σ ?” [3,5]. Let us elaborate: A request r can now be accompanied by a set C of *credentials*. Credentials are signed policy statements, e.g., of the form “public key k is authorised to perform action a .” Each entity that receives requests has its own security policy σ that specifies the requirements for granting and denying requests. Policies can directly authorise requests, or they can delegate to credential issuers that the entity expects have more detailed information about the requester.

Policy is separated from mechanism: A trust management *engine* takes as input a request r , a set of credentials C and a local policy σ ; it outputs an authorisation decision (this could be ‘yes’/‘no’, but also more

general statements about, say, *why* a request is denied, or *what would be further needed* to grant the request). This separation of concerns supports security reasoning needed in distributed applications.

1.1 Two lines of research

So far we have considered only a single notion of ‘trust’ in computer science, namely the concept of ‘trust management’ coined by Blaze, Feigenbaum and Lacy [5]. In fact, there is a whole different strand of research on trust distinct from the technical notion of Blaze et al., which has resulted in the term ‘trust’ being overloaded within computer science. This other “strand” deals with a computational formalisation of the *human notion of trust*, i.e., trust as a sociological, psychological and philosophical concept. Broad surveys in this field includes Jøsang et al. (2006) [12], Sabater and Sierra (2005) [26] and Jennings et al. (2004) [25]. Also the PhD thesis of Abdul-Rahman [1] contains a vast survey (mostly) of the human notion of trust in computer science, including insights from social sciences.

We shall call this other strand of research “experience-based trust management.” In experience-based trust management systems, an entity’s trust in another is based, in part, on the other’s past behaviour or evaluations of such past behaviour (similarly to the human notion of trust). (This also covers many so-called *reputation systems* or *reputation-based* trust management systems, which are often used in peer-to-peer (P2P) systems.) Note that at this level of abstraction, the question “Is the information about key k such that the request r should be granted?” still applies; the difference is what is meant by the word ‘information.’ In credential-based systems, the information ranges over credentials and local security-policies; in experience-based systems one still has local security-policies, but instead of credentials one has information about the past behaviour of the requester. This view brings closer the two distinct strands of research.

In fact, the trust structure framework [8] can be seen as an attempt to merge the two types in a “unified” theory of trust which combines the strengths of both notions.¹ Ideally we would like to combine the *rigour* of traditional trust management with the general view on *information* of the experience-based notion. Let us elaborate: Traditional trust management deals with credentials, policies, requests and the compliance-checking problem. Rigorous security *reasoning* is possible: The intended meaning of a trust management engine is formally specified, correctness proofs are feasible, and many security questions are effectively decidable [21]. In contrast, (to our knowledge) we have yet to see a experience-based system which, with realistic assump-

¹ We shall return to this framework in Section 1.4.

tions, guarantees any sort of rigorous security property.² On the other hand, such systems are capable of making *intuitively* reasonable decisions based on information such as evidence of past behaviour, reputation, recommendations and probabilistic models. A combination of these two approaches could lead to powerful frameworks for decision-making which incorporates more general information than credentials, yet which remains tractable to rigorous reasoning.

1.2 Formal Models

The traditional trust management approach was born from an engineering perspective: Important concepts were identified and prototype systems were built. However, no foundational models were developed initially. Although the original notion of proof of compliance (POC) was subject to theoretical investigation and rigorous definition [6], the definition itself is also considered to be ad-hoc [19]. The notion of POC (and trust management in general) has developed and is now better founded on existing theory.

Formal models make precise the notion of proof of compliance. For example, Li and Mitchell et al. [20] propose Constraint Datalog, *Datalog^C*, as the formal foundation for trust management languages and engines. *RT₁^C* is a constraint-based extension of the *RT₁* language of the *RT* family of languages. It is an example of a trust management language based on constraint Datalog. Each statement of *RT₁^C* is translated into a rule of *Datalog^C* with certain tractable domains, called linearly decomposable domains. This enables rigorous definition of POC, and POC checking in polynomial time. We consider the *RT* family of languages, specifically its extension *RT₁^C* with constraints [20], to be the state-of-the-art for credential-based trust management systems.

Stephen Weeks developed a very general formal model of credential-based trust management [27]; many concrete systems are instances of this general framework, e.g., SPKI [9], KeyNote [4] and logical systems such as that of Li et al. [18]. The advantages of general mathematical frameworks for trust management are many: It increases the understanding of trust management systems; techniques developed for the general framework are applicable in all its instances; and, the notion of proof-of-compliance is formalised in a very general way.

In Weeks' framework a trust management system is expressed as a complete lattice (D, \preceq) of possible *authorisations*, a set of principal names \mathcal{P} , and a language for specifying so-called *licenses*. The lattice elements $d, e \in D$ express the authorisations relevant for a particular system, e.g. access-rights, and $d \preceq e$ then means that e authorises at least as much as d . An *assertion* is a

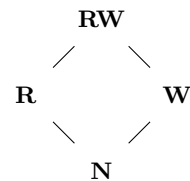


Fig. 1 Example authorisation lattice.

pair $a = \langle p, l \rangle$ consisting of a principal $p \in \mathcal{P}$, the *issuer*, and a monotonic function $l : (\mathcal{P} \rightarrow D) \rightarrow D$, called a *license*. In the simplest case l could be a constant function, say d_0 , and then a states that p authorises d_0 . In the general case the interpretation of a is the following: Given that all principals authorise as specified in the *authorisation map*, $m : \mathcal{P} \rightarrow D$, then p authorises as specified in $l(m)$. This means that a license such as $l(m) = m(A) \vee m(B)$ expresses a policy saying “give the least upper bound in (D, \preceq) of what A says and what B says.” Weeks showed that a collection of assertions $L = \langle p_i, l_i \rangle_{i \in I}$ gives rise to a monotonic function $L_\lambda : (\mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$, with the property that a coherent authorisation map representing the authorisations of the involved principals is given by the least fixed point, $\text{lfp } L_\lambda$.³

For the sake of clarity, we present a simple example, taken from Weeks' 2001-paper [27]. Consider $D = \{\mathbf{N}, \mathbf{R}, \mathbf{W}, \mathbf{RW}\}$ with \mathbf{N} least, \mathbf{RW} greatest and \mathbf{R} and \mathbf{W} unrelated, i.e., the Hasse diagram in Figure 1. Lattice D represents file access rights for a particular principal Alice, i.e., Alice may read (\mathbf{R}), write (\mathbf{W}), both (\mathbf{RW}) or Alice has no rights (\mathbf{N}). For a collection of licenses, L , the least fixed point $\text{lfp } L_\lambda$, represent how principals give authorisations to Alice. For some example licenses and fixed point computations, we refer to Weeks [27].

In the framework, the compliance-checking problem can be elegantly specified as: “Given a set of assertions C , i.e., $C \subseteq \mathcal{P} \times ((\mathcal{P} \rightarrow D) \rightarrow \mathcal{P})$, a request $r \in D$ and a principal identity $p \in \mathcal{P}$, does $r \preceq \text{lfp } C_\lambda(p)$ hold?” A great advantage of this approach is that existing theory of fixed points and algorithms for fixed point computation can be used in trust management engines. Weeks shows also how this generality can even lead to more efficient algorithms for compliance checking [27].

The two models, i.e., Constraint Datalog and Weeks' model, are related. Both models require that “policies” (licenses and rules) are monotonic, and both are based on fixed points. In fact, it should be simple to show that

³ If L contains two (or more) entries $\langle p, l_1 \rangle$ and $\langle p, l_2 \rangle$, then the licenses are combined to a single assertion $\langle p, l_1 \vee l_2 \rangle$. Hence the collection L gives rise to a function $L_\lambda : (\mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$ which maps an authmap m and a principal p to $l(m)$ where l is the *unique* license with $\langle p, l \rangle \in L$ (if no such entry exists the constant \perp license is used).

² See also our position paper, presented at FAST 2006 [15].

the fixed point semantics of Constraint Datalog can be obtained as a set-based instance of Weeks' model.

1.3 Trust Structures: An Introduction

As we have seen, the model of Weeks is a general framework for expressing credential-based trust management systems. The trust structure framework of Carbone et al. [8], attempts to generalise Weeks' framework by focusing on *information* in contrast to *authorisation*. A trust structure T is a triple $T = (D, \preceq, \sqsubseteq)$, where \preceq and \sqsubseteq are binary orderings on a set of values D . This set of values correspond to, but generalise, Weeks' authorisation lattices. For example, the set D might represent event counts, e.g., $D = E \rightarrow \mathbb{N}$, where E is a set of events. In this example, a value $t \in E \rightarrow \mathbb{N}$ represents information about the past behaviour of a principal, e.g., $t(e) = 42$ means that 42 occurrences of event e has been observed in the history interactions. The ordering \preceq on values corresponds to the ordering of Weeks: $t \preceq s$ means that s represents a higher degree of trust than t , i.e., if principal p is associated with value t and principal q is associated with value s then q would be considered more trustworthy than p (in particular if p is authorised to perform an action then q would be too). However, another ordering on values, the *information ordering* \sqsubseteq , is introduced too; this is where the framework generalises Weeks model. Suppose that at one point in time principal p is associated with value t ; if one obtains more information about p then another value, say s , could be associated with p . Depending on the nature of the information p may now be more or less trusted (or it may not make sense to compare the levels); this is represented by \preceq . In contrast, the ordering \sqsubseteq represents that s denotes a higher degree of *information* than does t ; alternatively one may think of $t \sqsubseteq s$ as the statement that t can be refined into s .

For clarity, let us consider a *very simple example* of a trust structure. First, suppose there are two possible events $E = \{\text{good}, \text{bad}\}$ that can occur each time we interact with a fixed principal p . An event count, i.e., a function $t : E \rightarrow \mathbb{N}$, can be represented by a pair, $(m, n) \in \mathbb{N}^2$; hence we consider the example where $D = \mathbb{N}^2$. This trust structure is called the "MN trust-structure," and it is denoted T_{MN} . With this set of trust values, one potential trust ordering could be the following:

$$(m, n) \preceq (m', n') \text{ iff } m \leq m' \text{ and } n \geq n'$$

(since m represents the number of 'good' events). Note, initially one might have no information about the past behaviour of p , hence p would be associated with the value $(0, 0)$; in other words $(0, 0)$ should be the least element in the *information* ordering. Given that p is associated with an element (m, n) , this information could

be refined by adding a number of 'good' events and a number of 'bad' events. This leads us to define:

$$(m, n) \sqsubseteq (m', n') \text{ iff } m \leq m' \text{ and } n \leq n'.$$

We explain trust structures in more detail in the following section. For even more detailed descriptions, we refer to the bibliography [8, 16, 14].

1.4 Trust Structure: Formally

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 $D = \{\text{high}, \text{mid}, \text{low}, \text{unknown}\}$, one could have, $\text{low} \preceq \text{mid} \preceq \text{high}$, perhaps $\text{low} \preceq \text{unknown} \preceq \text{high}$, and $\text{unknown} \sqsubseteq \text{low}, \text{mid}, \text{high}$; whereas high , low and 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 $\overline{\text{gts}} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ (the function space $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ is written GTS). Principals control how this function is defined by specifying their *trust policies*. Formally, each principal p specifies a trust policy, π_p , which is a function of type $\text{GTS} \rightarrow (\mathcal{P} \rightarrow D)$ (the space $\mathcal{P} \rightarrow D$ is written LTS). The 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 $\pi_p(\text{gts}) : \mathcal{P} \rightarrow D$. Note that trust policies π_p map *global* trust states to *local* trust states, and hence p 's policy may depend on other principals' 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 $\overline{\text{gts}}$. Clearly, $\overline{\text{gts}}$ should be consistent with each policy, π_p . This amounts to requiring that it should satisfy the following fixed-point equation: $\overline{\text{gts}}(p) = \pi_p(\overline{\text{gts}})$ for all $p \in \mathcal{P}$; or equivalently:

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

where Π_λ is the product function $\Pi_\lambda = \langle \pi_p \mid p \in \mathcal{P} \rangle$. Any $\text{gts} : \text{GTS}$ satisfying this equation is *consistent* with the policies $(\pi_p \mid p \in \mathcal{P})$. This means that *any* fixed-point

of Π_λ is consistent with all policies π_p . But arbitrary Π_λ , 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 $\pi_p : \text{GTS} \rightarrow \text{LTS}$ be *information continuous*, i.e. continuous with respect to \sqsubseteq . Since this implies that Π_λ is also information-continuous, and since $(\text{GTS}, \sqsubseteq)$ is a cpo with bottom, standard theory [28] tells us that Π_λ 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 gts , $\text{lfp}_{\sqsubseteq} \Pi_\lambda \sqsubseteq \text{gts}$). Hence, for any collection Π of trust policies, we define the *global trust-state induced by that collection*, as $\overline{\text{gts}} = \text{lfp} \Pi_\lambda$, which is well-defined by uniqueness.

Example. For a second simple example, we show that Weeks’ framework is an instance of the trust structure framework. This means that all theory developed for trust structures applies to Weeks’ framework; hence, it applies to *many* existing trust management systems.

Consider a complete lattice (D, \leq) in Weeks’ framework (e.g., D might represent the authorisations from KeyNote). We obtain a trust structure T which encodes (D, \leq) by taking $T = (D, \leq, \leq)$, i.e., both the information ordering and the trust ordering are \leq ; in this sense the trust structures generalise Weeks’ framework. Notice that the fixed-point of any collection of policies, and hence the semantics of the trust management system, coincide in (D, \leq) and T .

2 Motivation and Contributions

We have introduced the trust structure framework, motivating it by arguing that it unifies credential-based and experience-based trust management; hence, any techniques developed for the trust structure framework are immediately applicable in both worlds. However, there are some problems with the framework.

2.1 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 .

In fact, there are a number of reasons why computing the exact global trust state, $\overline{\text{gts}}$, would often be infeasible. When the cpo (D, \sqsubseteq) is of finite height h , the cpo $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D, \sqsubseteq)$ has height $|\mathcal{P}|^2 \cdot h$.⁴ 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 $(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \Pi_\lambda(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \Pi_\lambda^2(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \dots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p. \lambda q. \perp_{\sqsubseteq})$ [28]. 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 $|\mathcal{P}|$ 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 $\text{lfp} \Pi_\lambda$ before the policies have changed to Π' .

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. Our motivation is to counter this by showing that the situation is not as hopeless as suggested. The rest of the paper presents a collection of techniques for *approximating* the *idealised* fixed-point $\text{lfp} \Pi_\lambda$.

Contributions. Our work essentially deals with the operational problems left as “future work” by Carbone *et al.* [8]. Preliminary versions of this work has appeared as a conference paper (Krukow and Twigg (2006) [16]) and two technical reports [17, 13]. This paper should be viewed as archival-versions of these papers.

Firstly, techniques for distributed computation of approximations to the idealised trust-values over a global, highly dynamic, decentralised network. We start by showing that although it may be infeasible to compute the global trust-state, $\overline{\text{gts}} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$, one can instead compute so-called *local* fixed-point values. We take the practical point-of-view of a specific principal p , wanting to reason about its trust value for a fixed principal q . The basic idea is that instead of computing the entire state $\overline{\text{gts}}$, and *then* “looking up” value $\overline{\text{gts}}(p)(q)$ to learn p ’s trust in q , one may instead compute this

⁴ The height of a cpo is the size of its longest chain.

value directly. In Sections 3 and 4, we describe a general language for specifying trust policies, and provide a compositional operational semantics. The semantics of a collection of policies will be defined by translation into an I/O automaton [23, 22], providing a formal operational foundation for trust policies. Our main theorem (Theorem 2) proves in this *formal* model, that even in infinite height cpos the I/O automata will converge towards the local least fixed-point, as intended.

Secondly, often it is infeasible and even unnecessary to compute the *exact* denotation of a set of policies. In many cases it is sufficient (in order to make a trust-based decision) to know that a certain property of this value is satisfied. In Section 7, we take very mild assumptions on the relation between the two orderings in trust structures. This enables us to develop two efficient protocols for safe approximation of the least fixed-point. Often this allows principals to take security-decisions without having to compute the exact fixed-point. For example, suppose we know a function $I : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ with the property that $I \preceq \overline{\text{gts}}$. In many trust structures it is the case that if I is sufficient to authorise a given request, so is the actual fixed-point. Furthermore, we provide a formal description of the two approximation protocols, and sketch a proof of their correctness, also in terms of I/O automata.

As mentioned, our primary algorithmic model is that 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 of the semantics (Theorem 2) does not mention the abstract model, its proof uses this model together with the “simulation,” to prove its statement about the actual operational semantics.

3 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. [8], 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.

$$\begin{aligned} \pi &::= \star : \tau \\ &\quad | p : \tau, \pi \\ \tau &::= d \\ &\quad | p?q \\ &\quad | \text{op}_n^i(\tau_1, \tau_2, \dots, \tau_n) \end{aligned}$$

Fig. 2 Syntax

Syntax. We assume a countable collection of n 'ary function symbols op_n^i for each $n > 0$. These are meant to denote functions $[\text{op}_n^i]^{\text{den}} : D^n \rightarrow D$, continuous with respect to \sqsubseteq . The syntax is given in Figure 2. 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 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 (or role) 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 $\text{op}_n^i(\tau_1, \dots, \tau_n)$ is the application of an n 'ary operator to the trust specifications (τ_1, \dots, τ_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.

Denotational Semantics. The denotational semantics of the basic policy language is given in Figure 3, 4 and 5. We assume that for each of the function symbols op_n^i , $[\text{op}_n^i]^{\text{den}}$ is a \sqsubseteq -continuous function of type $D^n \rightarrow D$. The semantics follows closely the ideas presented in the introduction, and a more detailed explanation is given by Carbone et al. [8]. For a collection $\Pi = (\pi_p \mid p \in \mathcal{P})$, the semantics of each π_p is an information-continuous function $[\pi_p]^{\text{den}}$ of type $\text{GTS} \rightarrow \text{LTS}$.

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

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

Fig. 3 Denotational Semantics, π

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

Fig. 4 Denotational Semantics, τ

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

Fig. 5 Denotational Semantics, Π

An example. We present a small and very simple example, intended only to illustrate the denotational semantics presented above. Let us consider an example with 3 principals, named R, A and B. The example is meant to illustrate the situation where R wants to compute its trust value in a certain fixed subject S (which won't be involved in the computation). We will use the so-called “MN trust structure” $T_{MN} = (D, \preceq, \sqsubseteq)$, where trust values are pairs of (extended) natural numbers, i.e., $D = (\mathbb{N} \cup \{\infty\})^2$, and $(m, n) \in D$ intuitively represents a history of $m + n$ interactions with a principal with m interactions classified as “good” and n as “bad.” Recall that 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 $n \leq n'$; the trust ordering is given by: $(m, n) \preceq (m', n')$ only if $m \leq m'$ and $n \geq n'$.

In this example, the policies of the principals are as follows.

$$\begin{aligned} \pi_R &= S : A?S \vee (0, 0), \star : (0, \infty) \\ \pi_A &= S : B?S \sqcup (4, 2), \star : (0, 0) \\ \pi_B &= S : A?S \sqcup (6, 1), \star : (0, 0) \end{aligned}$$

The constants, e.g., $(4, 2)$, is meant to represent local data obtained by the principals via past interactions, i.e., A has interacted 6 times with S for which four interactions were “good” and two were “bad.” It is not hard to see that both (D, \preceq) and (D, \sqsubseteq) are lattices. Operators \vee and \sqcup are the joins of \preceq respectively \sqsubseteq ; they are given by the following formulae: For any $(m, n), (m', n') \in D$ we have:

$$(m, n) \vee (m', n') = (\max(m, m'), \min(n, n'))$$

Table 1 Example centralised fixed-point computation.

iteration	R	A	B
0	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}
1	$(0, 0)$	$(4, 2)$	$(6, 1)$
2	$(4, 0)$	$(6, 2)$	$(6, 2)$
3	$(6, 0)$	$(6, 2)$	$(6, 2)$

and

$$(m, n) \sqcup (m', n') = (\max(m, m'), \max(n, n'))$$

Intuitively, R's policy π_R says that for principal S, the value is at least $(0, 0)$ (i.e., $\lambda p \lambda q. \perp_{\sqsubseteq}$ or “unknown”), but may become \preceq -larger if principal A has some positive information about S.

Let us illustrate the *synchronous* or *central* least fixed-point computation. This is described by Table 1 containing the “synchronous” entries of $\Pi_{\lambda}^i(\lambda p \lambda q. \perp_{\sqsubseteq})$ (i.e., $\lambda p \lambda q. \perp_{\sqsubseteq}, \Pi_{\lambda}(\lambda p \lambda q. \perp_{\sqsubseteq}), \Pi_{\lambda}(\Pi_{\lambda}(\lambda p \lambda q. \perp_{\sqsubseteq})), \dots$).⁵ In the table, column x of row $i + 1$ is obtained by applying policy π_x (for S) to row i , e.g., the value $(4, 0)$ in column R of row 2 is obtained by π_R and row 1, as illustrated by the following informal “calculation:”

$$A?S(\text{“row 1”}) \vee (0, 0) = (4, 2) \vee (0, 0) = (4, 0)$$

It is easy to verify that the last row in the table is the least fixed-point of the policies (i.e., iterating round 4 will give the same row as iteration 3).

4 An Operational Semantics

Clearly, the *synchronous* algorithm illustrated in the previous section is not feasible as a general algorithm for computing the meaning of a collection of policies large, open distributed systems. In this section, we develop an *asynchronous* counterpart to the synchronous algorithm. This is done in terms of I/O automata, a natural model of asynchronous distributed computation. We then proceed to give the operational semantics of the trust policy language. More specifically, in the operational semantics, a principal-indexed collection of policies Π is translated into an I/O Automaton, denoted $\llbracket \Pi \rrbracket^{\text{op}}$. I/O Automata are a form of labelled transition systems, suitable for modelling and reasoning about distributed discrete event systems [22, 23]. Later, 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

⁵ Note, we only show the entries for principal S.

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 $\llbracket H \rrbracket^{\text{op}}$.

4.1 The I/O automaton model

We review the basic definitions of I/O automata. For a more in-depth treatment, we refer to Lynch's book [22]. An I/O automaton is a (possibly infinite) state automaton, where transitions are labelled with so-called actions.

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

Definition 1 (I/O Automaton) An *input/output automaton* A , consists of five components:

$$A = (\text{sig}(A), \text{states}(A), \text{start}(A), \text{steps}(A), \text{part}(A))$$

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

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.

A *run* r of an I/O automaton A is a finite sequence $r = s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n$ or an infinite sequence $r = s_0 a_1 s_1 a_2 s_2 \cdots$, satisfying that $s_0 \in \text{start}(A)$ and for all i , $(s_i, a_{i+1}, s_{i+1}) \in \text{steps}(A)$. 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 $\text{part}(A)$, 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 $\text{part}(A)$ 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



Fig. 6 Interface of the $\text{Channel}(p, q)$ I/O automaton.

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

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

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

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

1. $\text{sig}(A) = \prod_{i \in I} \text{sig}(A_i)$, i.e., $\text{sig}(A)$ is the composition signature of $(S_i \mid i \in I)$.
2. $\text{states}(A) = \prod_{i \in I} \text{states}(A_i)$. We use \bar{s} to denote elements of the Cartesian product. If $\bar{s} \in \text{states}(A)$ then \bar{s}_i refers to the i th component of \bar{s} .
3. $\text{start}(A) = \prod_{i \in I} \text{start}(A_i)$
4. $\text{steps}(A)$ is the set of triples (\bar{s}, a, \bar{s}') so that, for all $i \in I$, if $a \in \text{acts}(S_i)$ then $(\bar{s}_i, a, \bar{s}'_i) \in \text{steps}(A_i)$, and if $a \notin \text{acts}(S_i)$ then $\bar{s}_i = \bar{s}'_i$.
5. $\text{part}(A) = \bigcup_{i \in I} \text{part}(A_i)$

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 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 $p, q \in \mathcal{P}$, $\text{Channel}(p, q)$ 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 $\text{sig}(\text{Channel}(p, q)) = S$ is given by the following. We have $\text{int}(S) = \emptyset$, and $\text{acts}(S) = \text{in}(S) \cup \text{out}(S)$. The input actions are

$$\text{in}(S) = \{\text{send}(p, q, v) \mid v \in V\}$$

and the output actions are

$$\text{out}(S) = \{\text{recv}(q, p, v) \mid v \in V\}$$

The signature is illustrated graphically in Figure 6.

- $states(\mathbf{Channel}(p, q)) = V^*$, 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).
- $start(\mathbf{Channel}(p, q)) = \epsilon$ (the empty sequence).
- $steps(\mathbf{Channel}(p, q))$ is given by the following. For any state $s \in V^*$ and any $v \in V$, we have

$$(s, \mathbf{send}(p, q, v), s \cdot v) \in steps(\mathbf{Channel}(p, q))$$

For any $v_0 \in V$ and any sequence $s = v_0 \cdot s' \in V^+$, we have

$$(s, \mathbf{recv}(q, p, v_0), s') \in steps(\mathbf{Channel}(p, q))$$

- $part(\mathbf{Channel}(p, q))$ is the trivial partition where all $\mathbf{recv}(q, p, v)$ 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 [10,11], 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 initialises 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 $\mathbf{Channel}(p, q)$ 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 2 (Composition) Continuing from the previous example, consider now an automaton A which will represent principal p . Suppose A has the following signature $sig(A)$: $acts(A) = \{\mathbf{send}(p, q, v) \mid v \in V\} \cup I$, $in(A) = \emptyset$, $int(A) = I$, and $out(A) = \{\mathbf{send}(p, q, v) \mid v \in V\}$, where I is some set of internal actions (disjoint from any other set of actions in this example). Then A and $\mathbf{Channel}(p, q)$ are compatible automata, and their composition $A \times \mathbf{Channel}(p, q)$ is illustrated in Figure 7. Notice that the composition has no input actions, but output actions $\{\mathbf{send}(p, q, v) \mid v \in V\} \cup \{\mathbf{recv}(q, p, v) \mid v \in V\}$.

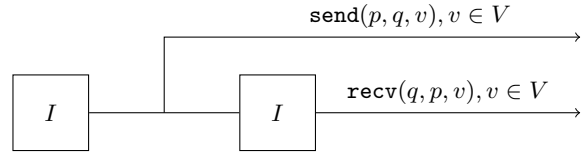


Fig. 7 The interface of $A \times \mathbf{Channel}(p, q)$.

4.2 $[[\cdot]]^{\text{op}}$ 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 $[[II]]^{\text{den}}$. Each principal $p \in \mathcal{P}$ will be computing its local values (i.e., $[[II]]^{\text{den}}(p)(q)$ for each $q \in \mathcal{P}$). We present now such an algorithm, inspired by the work of Bertsekas [2]. We then give an I/O automata version of this algorithm, which is used for formal proofs. We use the following “dot notation:” if A is an I/O-automaton $A.v$ refers to variable v of A .

Algorithm. The asynchronous algorithm is executed in a network of nodes, each denoted pq for $p, q \in \mathcal{P}$. A node pq represent a component of principal p , which is responsible for computing the value $[[II]]^{\text{den}}(p)(q)$. 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, denoted by $pq.gts$, of type $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$. Initially, $pq.t_{cur} = pq.t_{old} = \perp_{\square}$, and the matrix is also initialised 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.

Although we have presented the algorithm so that each principal has an approximation for *every* other principal (i.e., the arrays $pq.gts$), it is only necessary for principal p to store approximations for the principals that p depends on (in its policy). Similarly, pq only needs to send updates to nodes xy that depend on p ’s value for q . While the semantics of trust policies are functions of type $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$ which (due to policy referencing) in general may depend on the trust values of *all* principals, we expect that in practice, policies will not be written in this way. Instead,

policies are likely to refer to a few known (and usually “trusted”) principals. For fixed p and q , the set of principals that p ’s policy *actually* depends on in its entry for q , is often a significantly smaller subset of \mathcal{P} . For example, consider the policy:

$$\llbracket \pi_p \rrbracket^{\text{den}}(\text{gts}) = \lambda q \in \mathcal{P}. \text{gts}(A)(q) \vee_{\leq} \text{gts}(B)(q)$$

This policy is independent of all entries of gts except for those of principals A and B . This means that in order to evaluate π_p with respect to some principal q , p needs only information from A and B .

It is easy to convert the proposed algorithm into one that takes dependencies into account (for details, including a distributed algorithm for computing a dependency graph from a set of policies, consult the technical report [17]).

Communication model. We use an asynchronous communication model, assuming no known bound on the time it takes for a sent message to arrive. We assume that communication is reliable in the sense that any message sent eventually arrives, exactly once, unchanged, to the right node, and that messages arrive in the order in which they are sent. We assume (in the spirit of the global-computing vision) an underlying physical communication-network allowing any node to send messages to any other node. Furthermore, we assume that all nodes are willing to participate, and that they do not fail (however, we believe that the fixed-point algorithm we apply is highly robust [2]).

4.2.1 An example

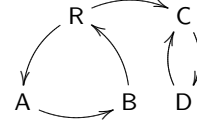
In this subsection, we give a small example of a run of the asynchronous algorithm. Let us consider an extended version of the simple example from Section 3. Our example has 5 principals, named R, A, B, C and D . The example is meant to illustrate the situation where R wants to compute its trust value in a certain fixed subject S (which won’t be involved in the computation). We will use the MN trust-structure T_{MN} (Section 1.3) in the example.

Policies. The policies of the principals are the following:

$$\begin{aligned} \pi_R &= \star : (A? \star \vee C? \star) \sqcup \text{Loc}_R(\star) \\ \pi_A &= \star : B? \star \sqcup \text{Loc}_A(\star) \\ \pi_B &= \star : R? \star \sqcup \text{Loc}_B(\star) \\ \pi_C &= \star : D? \star \sqcup \text{Loc}_C(\star) \\ \pi_D &= \star : C? \star \sqcup \text{Loc}_D(\star) \end{aligned}$$

The constructs Loc_p (for $p \in \mathcal{P}$) are special unary operators: $\text{Loc}_p(q)$ refers to the trust-value derived from the local observations made by principal p about the subject $q \in \mathcal{P}$ (this construct is discussed also by Nielsen and Krukow [24]). For example, R ’s policy for the subject S is to take the \leq -join in T_{MN} of the values that

Fig. 8 Example dependency-graph.



A and C specify for the subject, and then the \sqsubseteq -join of this value and the trust-value given by the local observations made by R about the subject.

The dependency graph derived from the policies is given in Figure 8.

Local data. We assume that the principals have the following local data, representing observations made about the subject. E.g., A has recorded one ‘good’, but five ‘bad’ observations about subject S .

	R	A	B	C	D
$\text{Loc}_p(S)$	(0, 0)	(1, 5)	(3, 0)	(2, 5)	(4, 6)

Synchronous computation. Let us first illustrate the least fixed-point of the policies by showing sequence of computations corresponding to the “synchronous” iterations (i.e., $\perp_{\sqsubseteq}, \llbracket \Pi_{\lambda} \rrbracket^{\text{den}}(\perp_{\sqsubseteq}), \llbracket \Pi_{\lambda} \rrbracket^{\text{den}}(\llbracket \Pi_{\lambda} \rrbracket^{\text{den}}(\perp_{\sqsubseteq})), \dots$). In the table below, column x of row $i+1$ is obtained by applying policy π_x to row i , e.g., the value (3, 5) in column A of row 2 is obtained by π_A and row 1, illustrated by the following informal “calculation:”

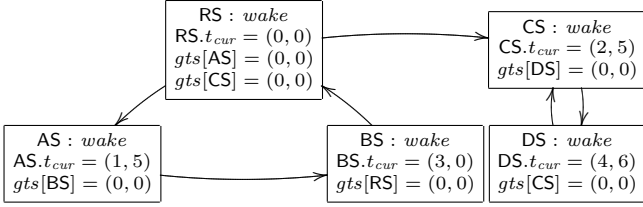
$$B?S(\text{“row 1”}) \sqcup \text{Loc}_A(S) = (3, 0) \sqcup (1, 5) = (3, 5)$$

It is easy to verify that the last row in the table below is the least fixed-point of the policies (i.e., iterating round 6 will give the same row as iteration 5).

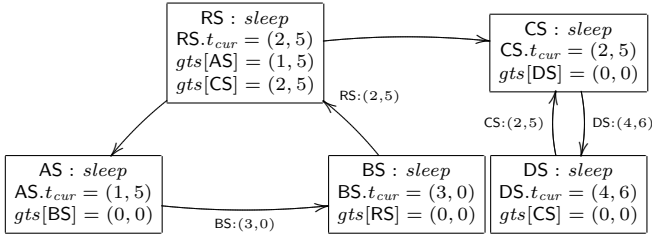
iteration	R	A	B	C	D
0	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}	\perp_{\sqsubseteq}
1	(0, 0)	(1, 5)	(3, 0)	(2, 5)	(4, 6)
2	(2, 5)	(3, 5)	(3, 0)	(4, 6)	(4, 6)
3	(4, 5)	(3, 5)	(3, 5)	(4, 6)	(4, 6)
4	(4, 5)	(3, 5)	(4, 5)	(4, 6)	(4, 6)
5	(4, 5)	(4, 5)	(4, 5)	(4, 6)	(4, 6)

An asynchronous run. We now show a possible run of the asynchronous algorithm for the same set of policies as above. We illustrate the algorithm by showing the local-states of the nodes in the network at various points in time. The nodes are denoted by boxes describing the local state in terms of values of arrays $xy.\text{gts}$, and the values of $xy.t_{\text{cur}}$. Furthermore, messages that are in transit are visible on the “dependency” edges between the nodes. Note that messages “flow against” the direction of the arrowhead since arrows denote dependencies. Note also, only nodes xy of form xS are considered (since these are the only relevant nodes for the computation).

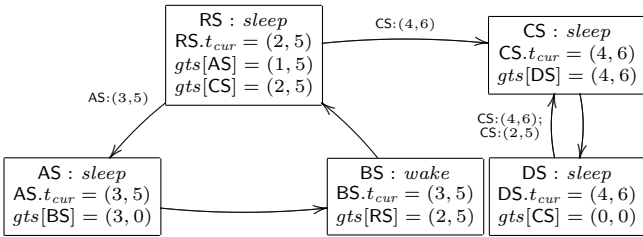
Network Snapshot 1. We assume that the initial states of the nodes are given by the following. All nodes are *wake*, the arrays are initialised to $\perp_{\square} = (0, 0)$. Each node xy has $xy.t_{cur} = \llbracket \pi_x \rrbracket^{\text{den}}(xy.gts)(y)$.



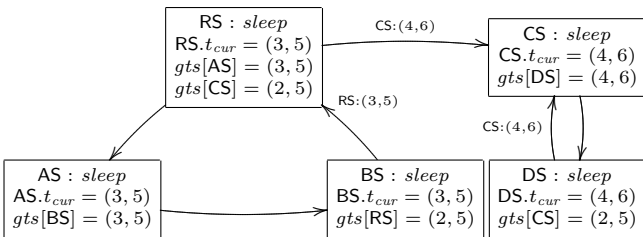
Network Snapshot 2. Here RS has received value (1, 5) from AS and (2, 5) from CS. Further values are in transit, e.g. value RS : (2, 5) “on” edge BS \rightarrow RS represents a message in transit from RS to BS.



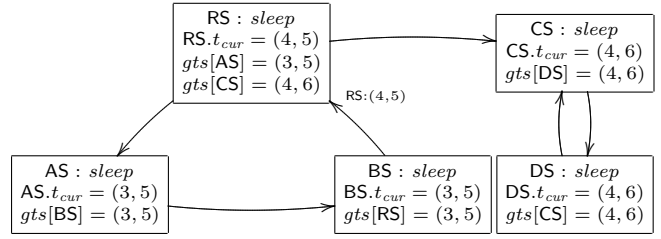
Network Snapshot 3. Two messages are in transit on the (presumably slow) path from CS to DS (we are assuming a reliable network, so the first sent will also arrive first). BS has just finished computing $\llbracket \pi_B \rrbracket^{\text{den}}(BS.gts)(S) = (3, 5)$, but has not yet sent this value.



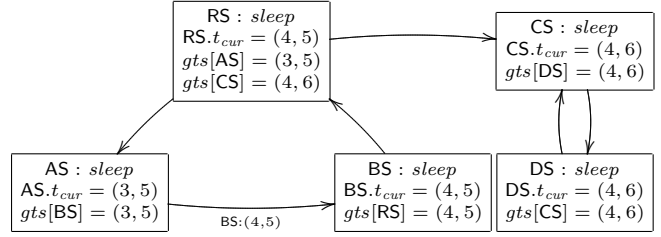
Network Snapshot 4.



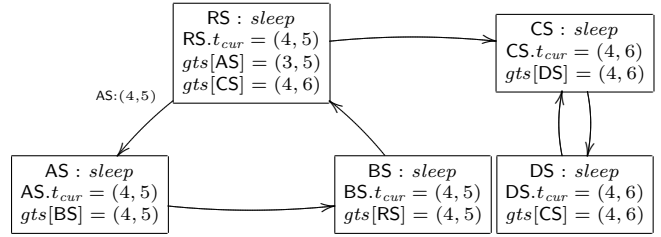
Network Snapshot 5. Notice that the component consisting of CS and DS has converged. No more messages are exchanged between them for the remainder of the algorithm; this is in contrast to the globally synchronous iteration.



Network Snapshot 6.



Network Snapshot 7. When RS receives the final value from AS, the algorithm has converged.



4.2.2 Formalisation via I/O Automata

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 (technically, principal p is modelled as the automaton-composition of each of the component-automata pq for $q \in \mathcal{P}$). The sending of a message d from component pq to, say rs , is represented by the I/O-automaton action $\text{send}(p, r, q, d)$ (note this is independent of s because all components rs will receive this update simultaneously). Note that the formal version of the algorithm does not consider policy-dependencies. While it is not too hard to change the algorithm to one which does [17], we choose to leave this out here as it is not too interesting (technically), and it adds complexity to the presentation.

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 parametrised I/O automata: $\text{Channel}(p, q, r)$ (modelling a communication medium for sending values from pq to r), and automaton $\text{IOTemplate}(p, q, f_{pq})$, where $p, q, r \in \mathcal{P}$ and $f : (\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow D$ is a continuous function. The latter $\text{IOTemplate}(p, q, f_{pq})$ is the component we denoted “ pq ” when f_{pq} is principal p 's policy for principal q , i.e. the entry for q in π_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 $B = \prod_{p, r, q \in \mathcal{P}} \text{Channel}(p, r, q)$ is a composition of channel automata. For $p, r, q \in \mathcal{P}$, the channel automaton $\text{Channel}(p, r, q)$ represent a reliable FIFO communication channel, and will be used by the automaton $\text{IOTemplate}(p, q, f_{pq})$, to communicate trust-values of p about principal q to principal r .

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

The Channel automaton is described syntactically 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 $f_{pq}(\text{gts}) = \llbracket \pi_p \rrbracket^{\text{den}} \text{gts } q$. The component $\text{IOTemplate}(p, q, f_{pq})$, denoted simply as “ pq ”, is responsible for computing principal p ’s trust value for principal q , i.e., value $\overline{\text{gts}}(p)(q)$. The semantics of each policy π is given by the following.

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

The most important automata are the IOTemplate automata. The parametrised 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, $\text{send}(p, r, q, pq.t_{cur})$ is always eventually executed when variable $pq.send(r)$ 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
  \forallall r, s : P (gts(r)(s) = bot)
  \forallall 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 /\ \forallall 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 Π are well-formed, then all the automata occurring in the definition of $\llbracket \Pi \rrbracket^{\text{op}}$ have compatible signatures, and, hence, are composable.*

Proof Simple inspection shows disjointness of all relevant actions. \square

4.3 Reasoning about the Semantics: Cause and Effect

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

For a (finite or infinite) run $r_c = s_0 a_1 s_1 a_2 s_2 \dots$ of $\llbracket \Pi \rrbracket^{\text{op}}$, we write $ActIndex(r_c)$ for the set $\{j \in \mathbb{N} \mid 0 < j < |r_c|\}$ of action indexes of r_c . Define the function $cause_{r_c} : ActIndex(r_c) \rightarrow \mathbb{N}$ inductively.

$$cause_{r_c}(1) = 0$$

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

- Case $a_{k+1} = \text{eval}(p, q)$ for some $p, q \in \mathcal{P}$. As we are not interested in “causes” of eval events, we simply define $cause_{r_c}(k+1) = 0$.
- Case $a_{k+1} = \text{send}(p, r, q, d)$ for some $p, q, r \in \mathcal{P}$, and $d \in D$.

$$\begin{aligned} cause_{r_c}(k+1) &= \max\{j+1 \mid 0 \leq j < k, \\ &\quad s_j.pq.send(r) = \text{false}, \\ &\quad s_{j+1}.pq.send(r) = \text{true}\} \end{aligned}$$

Note that, writing $j+1$ for $cause_{r_c}(k+1)$, a_{j+1} must be an $\text{eval}(p, q)$ event, and that we must have $s_j.pq.t_{cur} \neq s_{j+1}.pq.t_{cur}$, and $s_{j+1}.pq.wake = \text{true}$.

- Case $a_{k+1} = \mathbf{recv}(p, r, s, d)$ for some $p, r, s \in \mathcal{P}$, and $d \in D$. Let $R_0 = \{j \mid j < k+1, a_j = \mathbf{recv}(p, r, s, d)\}$, and let $S_0 = \{j \mid j < k+1, a_j = \mathbf{send}(r, p, s, d)\}$. S_0 is a candidate set of indices for the result. Now let

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

Define

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

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

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

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

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

Lemma 2 (Simple properties of cause) *For any run r_c , function cause_{r_c} satisfies the following.*

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

Proof See Appendix A. \square

It is easy to show that the channels are reliable, and act in a FIFO manner (see Appendix A). Further we can show that if $a_k = \mathbf{send}(p, r, q, d)$ then there is a unique j with $\mathit{cause}_{r_c}(j) = k$, which means 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 $a_m = \mathbf{recv}(r, p, q, d)$ then $\mathit{effect}_{r_c}(\mathit{cause}_{r_c}(m)) = m$.

5 Bertsekas abstract asynchronous systems

In this section we will define another translation, the abstract operational semantics, $\llbracket \Pi \rrbracket^{\text{op-abs}}$, mapping a collection of policies Π into what we call a Bertsekas Abstract Asynchronous System (BAAS). We prove the existence of a close correspondence between runs of the

concrete semantics $\llbracket \Pi \rrbracket^{\text{op}}$ and runs of the abstract semantics. We first introduce the abstract model, and then proceed to relate the two semantics.

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's. 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 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 $x^i = (x_1^i, x_2^i, \dots, x_n^i)$ 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_j^i is updated at node j when receiving an update from node i).

Examples of BAAS's include distributed optimisation, numerical and dynamic programming algorithms [2].

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 must satisfy $x_i(0) = \hat{x}_i$, and we use $x(t)$ to denote the following vector.

$$x(t) = (x_1(t), x_2(t), \dots, x_n(t))$$

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 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 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 $\lim_{k \rightarrow \infty} \tau_j^i(t_k) = \infty$ 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:

- $x^i(t_i^*)_j = x_j(t_i^*)$ 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 *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, $x_j(\tau_j^i(t_i^*)) = x_j(t_i^*)$ for all $j \in [n]$. Hence $f_i(x^i(t_i^*)) = f_i(x(t_i^*))$. 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

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

□

5.1 The asynchronous convergence theorem

The Bertsekas abstract asynchronous systems are models of asynchronous distributed algorithms. The so-called Asynchronous Convergence Theorem (ACT) (Proposition 6.2.1 of Bertsekas' book [2]) 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 $\perp \sqsubseteq f(\perp) \sqsubseteq \dots$ 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 [2]). 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 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 1 (ACT, Bertsekas [2])

Let $B = ((X_i)_{i=1}^n, (f_i)_{i=1}^n)$ be a Bertsekas Abstract Asynchronous System, $X = \prod_{i=1}^n X_i$, 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 .

5.2 $\llbracket \cdot \rrbracket^{\text{op-abs}}$ translation, an abstract operational semantics

We map a collection of policies $\Pi = (\pi_p \mid p \in \mathcal{P})$ to a BAAS, in a way similar to the concrete operational semantics. 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 $[n] \stackrel{(\text{def})}{=} \{1, 2, \dots, n\}$ 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 Π_λ).

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

5.3 Correspondence of 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 (finite or infinite, fair or not) run $r_c = s_0 a_1 s_1 a_2 s_2 \dots$ of the concrete I/O-automaton $\llbracket \Pi \rrbracket^{\text{op}}$ to a run r_a of the BAAS $\llbracket \Pi \rrbracket^{\text{op-abs}}$, 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.
 - (a) Let $k \leq t$ be the *largest*, with the property that $a_k = \text{recv}(p, r, s, d)$ for some $d \in D$. 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 $s_{j'}.rs.t_{cur} = \perp_{\square}$. If such k exists, let $k' = \text{cause}_{r_c}(k)$. Note that $a_{k'} = \text{send}(r, p, s, d)$.

- (b) We then define $k'' = \text{cause}_{r_c}(k')$. Note that $a_{k''} = \text{eval}(r, s)$, and that we must have $s_{k''}.rs.t_{cur} = d$.
- (c) Finally, 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, in particular $s_j.rs.t_{cur} = d$.

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 BAAS's.

Lemma 4 For any $p, q, r, s \in \mathcal{P}$, function τ_{rs}^{pq} is monotonically increasing.

Proof See Appendix A. □

Abstract state. For a run r_a of $\llbracket \Pi \rrbracket^{\text{op-abs}}$ 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^{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^{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 $\llbracket \Pi \rrbracket^{\text{op}}$, 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^{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^{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 state_{con} and state_{abs} the “abstract state.” The following lemma relates concrete and abstract runs via the abstract state.

Lemma 5 (Corresponding runs) Let $\Pi = (\pi_p \mid p \in \mathcal{P})$ be a collection of policies. Let r_c be a run of $\llbracket \Pi \rrbracket^{\text{op}}$, and let r_a be the corresponding run. Then,

$$\forall k. 0 \leq k < |r_c| \Rightarrow \text{state}_{con}(r_c, k) = \text{state}_{abs}(r_a, k)$$

Proof See Appendix A. □

Lemma 6 For any fair run r_c of $\llbracket \Pi \rrbracket^{\text{op}}$, let r_a denote its corresponding run. Then,

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

Proof See Appendix A. □

6 Correspondence between the Denotational and Operational Semantics

In this section, we present the main theorem of the operational semantics: 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:* $\forall t' \leq t. x^{pq}(t') \sqsubseteq x^{pq}(t)$

Proof See Appendix A. □

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

Definition 4 (Information Approximation)

Let (X, \sqsubseteq) be a CPO with bottom \perp_\square . Let $f : X \rightarrow X$ be any continuous function. An element $x \in X$ is an *information approximation* for f if

$$x \sqsubseteq \text{lfp } f \quad \text{and} \quad x \sqsubseteq f(x)$$

Lemma 7 *Let (X, \sqsubseteq) be a CPO with bottom \perp_\square . Let $f : X \rightarrow X$ be any continuous function, and $\hat{x} \in X$ an information approximation for f . Then the set, $\{f^k(\hat{x}) \mid k \in \mathbb{N}\}$ is a chain and*

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

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

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

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

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

□

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 an information approximation for II_λ . 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 $\lim_t x(t) = \text{lfp } II_\lambda$.*

Proof First we must show that the sequence $\{x(t)\}_t$ actually has a limit. We show that $x(0) \sqsubseteq x(1) \sqsubseteq \dots \sqsubseteq x(t) \sqsubseteq \dots$, i.e., $\{x(t)\}_t$ is an increasing omega chain. This follows from Proposition 1 since

$$x(t) = (\dots, x_{pq}(t), \dots) = (\dots, x^{pq}(t)_{pq}, \dots),$$

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 II_λ , we shall invoke the Asynchronous Convergence Theorem of Bertsekas. Define a sequence of subsets of X^n , $X(0) \supseteq X(1) \supseteq \dots \supseteq X(k) \supseteq X(k+1) \supseteq \dots$ by

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

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 $\Pi_\lambda^k(\hat{d}) \sqsubseteq y \sqsubseteq \text{lfp } II_\lambda$, we get by monotonicity $\Pi_\lambda^{k+1}(\hat{d}) \sqsubseteq \Pi_\lambda(y) \sqsubseteq \Pi_\lambda(\text{lfp } II_\lambda) = \text{lfp } II_\lambda$.

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 $\Pi_\lambda^k(\hat{d}) \sqsubseteq y^k \sqsubseteq \text{lfp } II_\lambda$. This implies that $\text{lfp } II_\lambda = \bigsqcup_k \Pi_\lambda^k(\hat{d}) \sqsubseteq \bigsqcup_k y^k \sqsubseteq \text{lfp } II_\lambda$, and hence $\bigsqcup_k y^k = \text{lfp } II_\lambda$. This means that $\lim_k y^k$ is a fixed point of II_λ .

The box condition is easy:

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

However, this only proves that the system converges to some fixed point $x^* = \lim_t x(t)$ of II_λ . But note that the invariance property (Proposition 1) implies that $x^{pq}(t) \sqsubseteq \text{lfp } II_\lambda$ for all t . Hence, x^* is a fixed point of II_λ , and $x^* \sqsubseteq \text{lfp } II_\lambda$. So we must have $x^* = \text{lfp } II_\lambda$. □

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 $\llbracket II \rrbracket^{\text{op}}$ “computes” the least fixed-point of the function II_λ , 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 $\{V^k \mid k \in \mathbb{N}\}$ is a chain in $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow X, \sqsubseteq)$, and*

$$\bigsqcup_{k \in \mathbb{N}} V^k = \llbracket II \rrbracket^{\text{den}}.$$

Proof First, map r_c to its corresponding run r_a . This is a fair run of $\llbracket \Pi \rrbracket^{\text{op-abs}}$ by Lemma 6. By Lemma 5, $\{x(t)\}_{t \in \mathbb{N}} = \{V^k\}_{k \in \mathbb{N}}$, and by the Proposition 2 $\{x(t)\}_{t \in \mathbb{N}}$ has a limit which is $\text{lfp } \Pi_\lambda = \llbracket \Pi \rrbracket^{\text{den}}$. \square

Corollary 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 $\llbracket \Pi \rrbracket^{\text{op}}$ is finite, and if N is the length of r , and $\text{state}_{\text{con}}(r, N-1) = (E, V)$, then $V = \llbracket \Pi \rrbracket^{\text{den}}$.*

Proof Let r' denote the corresponding run of the concrete run $r = s_0 a_1 s_1 \dots$. Proposition 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 5 then for all t with $t_0 \leq t < |r_c|$ we have $s_t.pq.t_{\text{cur}} = s_{t+1}.pq.t_{\text{cur}}$ 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 $\sqcup x(t) = x(t_0)$ the correspondence theorem implies that $V = \llbracket \Pi \rrbracket^{\text{den}}$. \square

6.1 Practical remarks about the operational semantics

We have presented an operational semantics $\llbracket \cdot \rrbracket^{\text{op}}$ for trust policies. The semantics formalises a distributed algorithm for computing local trust-values $\overline{\text{gts}}(p)(q)$. As we have mentioned, the operational semantics is a simplification of the intended algorithm: A practical algorithm would also do dependency analysis of the trust policies so that **send**-messages are not global broadcasts, but instead, are sent only to principals that really depend on this information.

Regarding complexity, we have the following results. Assume that trust structure $(D, \sqsubseteq, \preceq)$ is finite with (D, \sqsubseteq) of height, $h \in \mathbb{N}$. Since any node sends values only when a change in its current value occurs, by monotonicity of policies, each node pq will send at most $h \cdot M_{pq}$ messages, where M_{pq} denotes the number of principals that depend on p 's entry for q . Each message is of size $O(\log |D|)$ bits.⁶ Node pq will receive at most $h \cdot N_{pq}$ messages, where N_{pq} denotes the number of principals that p depends on in its entry for q . Each message (possibly) triggers an evaluation of π_p .

Another practical issue concerns dynamics. Suppose that a policy π_p changes to π'_p at some point in time (during computation or not). Mathematically, this is not an issue: The result is a new well-defined global trust-state given by the fixed point of the collection $(\pi_x \mid x \in \mathcal{P} \setminus \{p\}) \cup (\pi'_p)$. However, in practice, how would principals compute the new local values given the

⁶ In fact, there will be *only* $O(h)$ different messages, each sent to all of the nodes that depend on pq . Consequently, a broadcast mechanism could implement the message delivery efficiently.

information about the old fixed-point? When π_p and π'_p satisfies $\pi_p \sqsubseteq \pi'_p$ (i.e., π'_p is more precise than π_p), this presents no problems: Principal p can change its policy locally, and continue the algorithm without notifying anyone; the correct fixed-point will be computed. For more general scenarios, the technical report of Krukow and Twigg [17] suggest some simple algorithms to reuse as much of the old information as possible. However, in the worst-case a complete recomputation is necessary. For this reason we do not present these minor results here.

Finally, in our I/O-automata version of the algorithm, each principal, say p , is represented as a collection of nodes $(pq \mid q \in \mathcal{P})$; the reason for this is that then we have a node responsible for computing $\overline{\text{gts}}(p)(q)$ for any fixed q (and only this entry). In a physical communication network, there would only be one communication node representing p . The nodes pq would be represented as virtual communication points in the node, p , e.g., in TCP/IP this could be ports on the IP-address for p ; in this case the subcomponents of p could also share one single matrix for their approximation of $\overline{\text{gts}}$.

7 Approximation techniques

In the previous sections we showed how a generic language for trust policies can be given an ideal abstract meaning (its denotational semantics), and a practical concrete meaning (its operational semantics). Further, we showed that the I/O automata of the concrete semantics form a labelled transition system in which every run converges to the ideal meaning. In effect we have obtained a correct asynchronous distributed algorithm for computing the trust state. In this section, we formally develop other distributed algorithms and protocols for approximating and reasoning about the ideal global trust state.

Recall that, in a trust management scenario with trust structure $T = (D, \preceq, \sqsubseteq)$, when a principal v needs to make a security decision about another principal p , this decision is made depending on the ideal trust value for p , i.e., $\overline{\text{gts}}(v)(p)$ (as determined uniquely by the collection of policies Π). Let us define a (trust-based) *security policy* as a function $\sigma : \mathcal{P} \times D \rightarrow \{\perp, \top\}$, with the following interpretation: If v has security policy σ , then v allows interaction with a principal p is only if the ideal global trust state satisfies $\sigma(p, \overline{\text{gts}}(v)(p)) = \top$. An important class of security policies are the monotonic policies: σ is *monotonic* if for all $d, d' \in D$ with $d \preceq d'$ we have $\sigma(p, d) \Rightarrow \sigma(p, d')$ for all $p \in \mathcal{P}$. Many natural security policies are monotonic; for example, the simple threshold policies: A policy σ of the form $\sigma(p, d) = \top \iff t_p \preceq d$ for some threshold $t_p \in D$. For example, in the security lattice from Figure 1, a threshold security-policy for reading a file could be $\sigma(p, d) = \top \iff \mathbf{R} \preceq d$; in the MN structure, one might have

$\sigma(p, d) = \top \iff (m, n) \preceq d$, i.e., allow interaction if there are *at least* m ‘good’ events and *at most* n ‘bad’ events.

In this section, we formally develop techniques for safely and efficiently evaluating monotonic security policies, i.e., evaluating $\sigma(v, \overline{\text{gts}}(v)(p))$, *without* explicitly computing the trust-value $\overline{\text{gts}}(v)(p)$. In practice, designers of trust management systems will need a range of such techniques in order to develop functional systems. The reader will notice that the way we have presented the overall idea lends itself to attempts of exploiting techniques from the well established area of *abstract interpretation*, and we are convinced that this would be a fruitful path to follow.

In this section, we make the following non-restrictive assumptions about the trust structure $T = (D, \preceq, \sqsubseteq)$: (D, \preceq) has a least element, denoted \perp_{\preceq} . For a subset $D_0 \subseteq D$ and $d \in D$ we write $d \preceq D_0$ only if $d \preceq d_0$ for all $d_0 \in D_0$ (similarly for $D_0 \preceq d$ and for \sqsubseteq). We assume that \preceq is \sqsubseteq -continuous, meaning that for any countable \sqsubseteq -chain $D_0 = \{d_i \in D \mid i \in \mathbb{N}\}$ and any $d \in D$ we have:

1. $d \preceq D_0$ implies $d \preceq \bigsqcup D_0$; and
2. $D_0 \preceq x$ implies $\bigsqcup D_0 \preceq d$.

7.1 Proof-carrying Requests

In the first approximation technique, we consider a so-called “prover” p (say, wanting to access a resource) and a “verifier” v . We assume that v ’s security policy, σ , is monotonic. The prover will send information I to v , which is used to convince her that $\sigma(p, \overline{\text{gts}}(v, p)) = \top$, hence, allowing v to make a safe decision about p . The following proposition provides the theoretical basis for the proof-carrying request protocol.

Proposition 3 (Proof-carrying requests) *Assume that $(D, \preceq, \sqsubseteq)$ is a trust structure where \preceq is \sqsubseteq -continuous. Let $I \in D^n$, and $f : D^n \rightarrow D^n$ be \sqsubseteq -continuous and \preceq -monotonic. If $I \preceq \perp_{\preceq}^n$ and $I \preceq f(I)$, then $I \preceq \text{lfp}_{\sqsubseteq} f$.*

Proof Since $I \preceq \perp_{\preceq}^n$ we have $f(I) \preceq f(\perp_{\preceq}^n)$ by the monotonicity of f . Since $I \preceq f(I)$,

$$I \preceq f(\perp_{\preceq}^n).$$

Continuing, we obtain $I \preceq f^i(\perp_{\preceq}^n)$, for all $i \geq 0$. Since \preceq is \sqsubseteq -continuous we get that

$$I \preceq \bigsqcup_{i \geq 0} f^i(\perp_{\preceq}^n) = \text{lfp}_{\sqsubseteq} f.$$

□

As in Section 5.2, we assume that f is given by coordinate functions f_{pq} for $p, q \in [n]$ (which are the policies of principals). Returning to the approximation protocol, the prover p first sends I of type $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ to v . If v can verify that I satisfies the properties $I(x)(y) \preceq \perp_{\preceq}$

and $I(x)(y) \preceq f_{xy}(I)$ (for all $x, y \in \mathcal{P}$), then we can invoke the proposition to obtain $I \preceq \text{lfp}_{\sqsubseteq} f$. This means that if $\sigma(p, I(v)(p)) = \top$, also $\sigma(p, \overline{\text{gts}}(v)(p)) = \top$, and v can make a safe decision based on I .

In principle, to check $I \preceq f(I)$, one would need to evaluate all functions f_{xy} in the network. However, in practice the information I is chosen so that $I(x)(y) = \perp_{\preceq}$ for all but a few x and y , effectively making the checks $I(x)(y) \preceq \perp_{\preceq}$ and $I(x)(y) \preceq f_{xy}(I)$ redundant for most x and y . We represent a function $I : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ where most entries are $I(x)(y) = \perp_{\preceq}$ as the subset $I \subseteq \mathcal{P} \times \mathcal{P} \times D$ of non- \perp_{\preceq} entries.

7.1.1 An Example

Consider for simplicity the “MN” trust-structure T_{MN} from Section 1, which satisfies the information-continuity requirement. Recall that, in this structure, trust values are pairs (m, n) of natural numbers, representing a number, $m+n$, of past interactions; m of which where classified ‘good’, and n , classified as ‘bad’.⁷ The orderings are given by $(m, n) \sqsubseteq (m', n') \iff m \leq m'$ and $n \leq n'$, and $(m, n) \preceq (m', n') \iff m \leq m'$ and $n \geq n'$.

Suppose principal p wants to efficiently convince principal v , that v ’s trust value for p is a pair (m, n) with the property that n is less than some fixed bound $N \in \mathbb{N}$ (i.e., giving v an upper bound on the amount of recorded “bad behaviour” of p). Let us assume that v ’s trust policy π_v is monotonic, also with respect to \preceq , and that it depends on a large set S of principals. Assume also that it is sufficient that principals a and b in S have a reasonably “good” trust-value for p , to ensure that v ’s trust-value for p is not too “bad”. An example policy with this property could be:

$$\pi_v = \star : (a? \star \wedge b? \star) \vee \bigwedge_{s \in S \setminus \{a, b\}} s? \star.$$

The construct $e \vee e'$ represents least upper-bound in the trust-ordering (intuitively, “trust-wise maximum” of e and e'), and similarly \wedge represents greatest lower-bound (“trust-wise minimum”). Thus, (informally) the above policy says that any principal p should have “high trust” with a and b , or, with *all of* $s \in S \setminus \{a, b\}$, for the v to assign “high trust” to p . Now, if p knows that it has previously performed well with a and b , and knows also that v depends on a and b in this way, it can engage in the following protocol.

Protocol. Principal p sends to v the information:

$$I = [(v, p, (0, N)), (a, p, (0, N_a)), (b, p, (0, N_b))],$$

which can be thought of as a “proof” (analogous to a ‘proof-of-compliance’) or a “claim” made by p , stating

⁷ To be precise, the set \mathbb{N}^2 is completed by allowing also value ∞ as “ m ” or “ n ” or both.

that $(0, N) \preceq \Pi_\lambda(v)(p)$ (and similarly for a and b). Upon reception, v first extends I to a global trust state, which is the extension of I to a function of type $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$, given by

$$\bar{I} = \lambda x \in \mathcal{P} \lambda y \in \mathcal{P}. \begin{cases} (0, N) & \text{if } x = v \text{ and } y = p \\ (0, N_a) & \text{if } x = a \text{ and } y = p \\ (0, N_b) & \text{if } x = b \text{ and } y = p \\ (0, \infty) & \text{otherwise} \end{cases}$$

To check the proof, principal v must verify that \bar{I} satisfies the conditions of Proposition 3. First, v must check that $\bar{I}(x)(y) \preceq \perp_{\square} = (0, 0)$ for all x, y . But this holds trivially if $y \neq p$ or $x \neq v, a, b$ because then $\bar{I}(x)(y) = (0, \infty) = \perp_{\preceq}$. For the other few entries it is simply an order-theoretic comparison $\bar{I}(x)(y) \preceq (0, 0)$. Now v tries to verify that $\bar{I} \preceq \Pi_\lambda(\bar{I})$. To do this, v verifies that $(0, N) \preceq \llbracket \pi_v \rrbracket^{\text{den}}(\bar{I})(p)$. If this holds then v sends the information I to a and b , and ask a and b to perform a similar verification (e.g. $(0, N_a) \preceq \llbracket \pi_a \rrbracket^{\text{den}}(\bar{I})(p)$). Then a and b reply with ‘yes’ if this holds and ‘no’ otherwise. If both a and b reply ‘yes’, then p is sure that $\bar{I} \preceq \Pi_\lambda(\bar{I})$: By the checks made by v, a and b , we have that $\bar{I}(x)(y) \preceq \Pi_\lambda(\bar{I})(x)(y)$ holds for pairs $(x, y) = (v, p), (a, p), (b, p)$, but for all other pairs it holds trivially since \bar{I} is the \preceq -bottom on these. By Proposition 3, we have $\bar{I} \preceq \llbracket \Pi \rrbracket^{\text{den}}$, and so, v is *ensured* that its trust value for p is \preceq -greater than $(0, N)$.

7.2 I/O-Automaton Version

We now describe an I/O automaton implementing the verifiers role in the proof-carrying request protocol. Consider the following automaton **Verify**.

```

automaton Verify(v : P,
  f_v : (P -> P -> D) -> P -> D)
signature
  input   proof(I : Set[P x P x D])
          reply-ok(x : P)
          reply-fail(x : P)

  output  check(x : P, Ip : Set[P x P x D])
          reject
          accept

state
  Ip: Set[P x P x D] := {}
  abort: Boolean := false
  pending : Set[P] := {}
  sent : Set[P] := {}
  ok : Set[P] := {}
transitions
  input proof(I)
  eff
    Ip := I;
    foreach ((x,y,d) in I) do
      if (x = v) then
        abort := abort \vee not ( d <= f_v(I)(y)
                               /\ d <= bot)
      else
        pending.add(x)
    fi
  od
  input reply-ok(x)
  eff ok.add(x)

```

```

input reply-fail(x)
  eff abort := true
output check(x, Ip)
  pre x \in pending /\ not (x \in sent)
  eff sent.add(x);
output reject
  pre abort
output accept
  pre ok = pending
partition {check(x,Ip) | x : P, Ip : Set[P x P x D]};
{reject}; {accept}

```

Verify takes an identity $v \in \mathcal{P}$ and v 's policy $\llbracket \pi_v \rrbracket^{\text{den}}$ as parameters. The verifier has an input-action **proof**(I), which models the reception of information $I \subseteq \mathcal{P} \times \mathcal{P} \times D$. We can view I as a claim made by a prover: for all $(x, y, d) \in I$ we have $d \preceq \text{gts}(x)(y)$. It is now the job of the verifier to check this by checking that I satisfies the assumptions of Proposition 3. First all $(x, y, d) \in I$ with $x = v$ are checked locally. For all (x, y, d) with $x \neq v$ we add x to a set *pending* of non-local checks that are to be performed. Eventually, an output action **check**(x, I) is performed for each such $x \in \text{pending}$. This models the verifier sending a request to principal x to check that all “ x -entries” of I satisfy the assumptions of the proposition. For each such message sent, the verifier expects to receive one of two possible replies from x : **reply-ok**(x) or **reply-fail**(x). If all $x \in \text{pending}$ send **reply-ok**(x) and the local checks succeeded, then the verifier performs the **accept** action to model the acceptance of the proof; otherwise **reject** is performed.

Let us assume that for each $x \in \mathcal{P}$, A_x is an I/O automaton with signature S , $\text{in}(S) = \{\text{check}(x, I) \mid I \subseteq \mathcal{P} \times \mathcal{P} \times D\}$,

$$\text{out}(S) = \{\text{reply-ok}(x), \text{reply-fail}(x)\},$$

and $\text{int}(S) = \emptyset$. Assume that any fair run $r_x = s_0 a_1 s_1 \dots$ of A_x satisfies the following property: action **reply-ok**(x) occurs if and only if there is exactly one previous occurrence of **check**(x, I) and I satisfies: for all $(a, b, d) \in I$ with $a = x$ we have $d \preceq \perp_{\square}$ and $d \preceq \llbracket \pi_x \rrbracket^{\text{den}}(I)(b)$. Similarly action **reply-fail**(x) occurs if and only if there is exactly one previous occurrence of **check**(x, I) and I does not satisfy this property. The collection $(A_x \mid x \in \mathcal{P})$ models an environment of the verifier which is willing to make local checks on the information I . We then obtain the following correctness lemma.

Lemma 8 (PCR Protocol) Let $I \subseteq \mathcal{P} \times \mathcal{P} \times D$, and let $r = s_0 a_1 s_1 \dots$ be any fair run of the composite automaton $\text{Verify}(v, \llbracket \pi_v \rrbracket^{\text{den}}) \times \prod_{x \in \mathcal{P} \setminus v} A_x$. Assume that $a_1 = \text{proof}(I)$. Then there exists an index j so that $a_j = \text{accept}$ or $a_j = \text{reject}$. Further $a_j = \text{accept}$ if-and-only-if I satisfies: for all $(x, y, d) \in I$ we have $d \preceq \llbracket \pi_x \rrbracket^{\text{den}}(I)(y)$ and $d \preceq \perp_{\square}$. Hence if $a_j = \text{accept}$ then $I \preceq \text{gts}$.

Proof (sketch) Since, $a_1 = \text{proof}(I)$ the state s_1 satisfies **abort** = **false** if-and-only-if $\forall (x, y, d) \in I : x = v \Rightarrow d \preceq \perp_{\square}, f_v(I)(y)$. If **abort** = **true** in this state

we are done, as **reject** is scheduled. Otherwise, for each $(x_0, y, d) \in I$ with $x_0 \neq v$ action **check** (x_0, I) is scheduled, and by fairness, eventually performed. By the assumptions about the A_x , action **reply-ok** (x_0) occurs if $\forall(x, y, d) \in I : x = x_0 \Rightarrow d \preceq \perp_{\square}, f_{x_0}(I)(y)$; otherwise **reply-fail** (x_0) occurs. Assuming all x reply with “ok,” we obtain $ok = pending$ and **accept** is eventually performed, which is correct because then we have $\forall(x, y, d) \in I : d \preceq \perp_{\square}, f_x(I)(y)$. Otherwise, some **reply-fail** (x) action was performed, and **reject** is eventually performed. \square

As noted previously, the PCR protocol gives an efficient way of asserting $\sigma(p, \overline{gts}(v)(p)) = \top$ without computing $\overline{gts}(v)(p)$. However, two essential practical problems arise with this “proof-carrying” approach. First, to construct proof I , the prover needs information about the verifiers trust policy and of the policies of those whom the verifier depends on. If policies are not public, it is not clear how the verifier would construct I . Second, because of the requirement in Proposition 3 that $I \preceq \perp_{\square}^n$, the protocol can usually only be used to prove properties limiting the amount of previous “bad behaviour,” and *not* properties guaranteeing sufficient previous “good behaviour.” For example, in the T_{MN} trust structure $I \preceq \perp_{\square}^n = (0, 0)^n$ means that we can only allow information I with entries of the form $(x, y, (0, N))$, i.e., stating that x ’s trust in y has no more than N “bad” interactions.

7.3 A Snapshot-based Approximation Technique

The second technique also enables the efficient assertion of $\sigma(p, \overline{gts}(v)(p)) = \top$ without computing \overline{gts} . However, it is not “proof based,” and does not restrict the possible trust values to be less than \perp_{\square} . Instead the technique in this section can be viewed as a simple snapshot algorithm, performed on the I/O automaton of the operational semantics. The resulting snapshot is a trust-state $I : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ which is an information approximation to $\llbracket II \rrbracket^{\text{den}}$. The principals are then able to make a distributed check that $I \preceq \overline{gts}$, and if the check succeeds and $\sigma(p, I(v)(p)) = \top$ then also $\sigma(p, \overline{gts}(v)(p)) = \top$. The theoretical basis for the technique is the following proposition.

Proposition 4 (Snapshot) *Let (D, \preceq, \square) be a trust structure in which \preceq is \square -continuous. Let $I \in D^n$, and $f : D^n \rightarrow D^n$ be any function that is \square -continuous and \preceq -monotonic. Assume that I is an information approximation for f . If $I \preceq f(I)$ then $I \preceq \text{lfp}_{\square} f$.*

Proof Since $I \preceq f(I)$, monotonicity gives $I \preceq f^k(I)$ for all $k \geq 0$. I is an information approximation for f , so Lemma 7 implies $\bigsqcup_{k \in \mathbb{N}} f^k(I) = \text{lfp}_{\square} f$. Finally, information continuity of \preceq implies $I \preceq \text{lfp}_{\square} f$. \square

The above proposition requires that I be an information approximation. Fortunately, the invariance property of the operational semantics (Proposition 1) implies that we can, indeed, compute a snapshot I which is an information approximation. Once I is computed, each principal pq checks that $I(p)(q) \preceq f_{pq}(I)$. If all principals succeed with the check, Proposition 4 implies $I \preceq \text{lfp} f$ as desired.

Imagine that during a run of the operational semantics there is a point in time in which no messages are in transit, all nodes pq have computed their function f_{pq} , and sent the value to all nodes. Thus we have a “consistent” state in the sense that for any node pq and any node rs we have $pq.gts(r)(s) = rs.t_{cur}$. In particular any pq and xy agree on rs ’s value: $pq.gts(r)(s) = rs.t_{cur} = xy.gts(r)(s)$. In this ideal state, there is a consistent vector I which, by Proposition 1 (and Lemma 5), is an information approximation for f . If all nodes pq , can make the local check $I(p)(q) = pq.t_{cur} \preceq f_{pq}(I)$, then I satisfies $I \preceq f(I)$, and hence $I \preceq \overline{gts}$.

The approximation algorithm computes a consistent view that corresponds to such an ideal situation, and incorporates the local checks. In so-called snapshot algorithms (see Lynch [22] or Bertsekas [2]), the (local views of the) global state of the system is recorded during execution of an algorithm. Our problem is slightly less complicated since we are not interested in the status of communication links, but slightly more complicated since each snapshot-value must be propagated to a specific set of nodes.

Consider the following I/O automaton, **Snapshot**, for computing a snapshot of the operational semantics.

```

automaton Snapshot(p : P, q : P, Dep : Set[P x P])
signature
  input ping(x : P, y : P, const p, const q)
        reply(x : P, y : P, const p,
              const q, d : D)
        reply-par(x : P, y : P, const p,
                  const q, d : D, b : Boolean)
  output ping(const p, const q, x : P, y : P)
         reply(const p, const q, x : P,
              y : P, d : D)
         reply-par(const p, const q, x : P,
                  y : P, d : D, b : Boolean)

state
  val : D;
  parent : (P x P) + {undef} := undef;
  Iapp: P -> P -> D;
  ok : Boolean;
  reply_to: Set[PxP] := {};
  sent: Set[PxP] := {};
  recvd: Set[PxP] := {};
  initially
    \forall p, q \in P (Iapp(p)(q) = bot)

transitions
  input ping(x,y,p,q)
    eff if (parent = undef)
      then parent := (x,y);
         val := t_cur;
         Iapp(p)(q) := val;
         ok := true
      else reply_to.add(x,y)
    fi
  input reply(x,y,p,q,d)
    eff
      Iapp(x)(y) := d;

```

```

    recvd.add(x,y)
input reply-par(x,y,p,q,d,b)
  eff
    Iapp(x)(y) := d;
    recvd.add(x,y);
    ok := ok /\ b
output ping(p,q,x,y)
  pre parent != undef /\
    (x,y) \in Dep /\ not ((x,y) \in sent)
  eff sent.add(x,y)

output reply(p,q,x,y,d)
  pre (x,y) \in reply_to /\ (x,y) != parent
    /\ d = val
  eff reply_to.remove(x,y)

output reply-par(p,q,x,y,d,b)
  pre (x,y) = par /\ recvd = Dep /\ d = val
    /\ b = (ok /\ (val <= f_pq(Iapp)))
  eff par := undef

partition {ping(p,q,x,y) | x,y : P};
  {reply(p,q,x,y,d) | x,y : P,d:D};
  {reply-par(p,q,x,y,d,b) | x,y : P,
    d:D,
    b:Boolean};

```

The $\text{Snapshot}(p, q, \dots)$ automaton is supposed to be viewed as an extension of the $\text{IOTemplate}(p, q, f_{pq})$ automaton in the sense that it must have access to its local state of pq and policy f_{pq} . Our I/O automaton implementation differs from the algorithm sketched by Krukow and Twigg [16] in that it does not assume a known spanning tree on the nodes. Instead, we assume simply that each node pq knows the set $\text{Dep} \subseteq \mathcal{P}$ of principals that its policy f_{pq} depends on.⁸

The $\text{Snapshot}(p, q, \text{Dep})$ automaton is initiated by another node rs when pq first receives a $\text{ping}(r, s, p, q)$ message from rs . We then say that rs is the *parent* of pq (in fact, it is the parent of pq in a spanning tree rooted at the node which initiates the snapshot algorithm). Subsequent $\text{ping}(x, y, p, q)$ messages received simply register the pair (x, y) which later results in pq performing action $\text{reply}(p, q, x, y, d)$ with d being the value of pq in the snapshot I . Upon the first reception of a ping-message, pq stores its current value t_{cur} (as obtained from the IOTemplate algorithm) in a variable val . Automaton pq then proceeds to request the approximation value of rs for each $rs \in \text{Dep}$; this is implemented by scheduling the $\text{ping}(p, q, r, s)$ action. This action results in a reply, either of type $\text{reply}(r, s, p, q, d)$ meaning $I(r)(s) = d$; or of type $\text{reply-par}(r, s, p, q, d, b)$ meaning $I(r)(s) = d$ and “you’re my parent in the spanning tree, and my check $I(r)(s) \preceq f_{rs}(I)$ succeeded only if $b = \text{true}$.” When pq has received replies from all rs in Dep it can evaluate $I(p)(q) \preceq f_{pq}(I)$, hence, action $\text{reply-par}(p, q, r, s, val, \text{true})$ (where rs is pq ’s parent) is performed if all its children in the spanning tree replied with true and the check succeeded.

We assume that there is a special initiating automaton, $\text{Init}(p, v, \text{Dep}_{pv})$ (denoted simply pv), which is like the snapshot automaton, but doesn’t have a parent (it

⁸ A good approximation to this set can be computed from the syntax of the policy, simply by inspecting which principals are referenced.

is, in fact, the parent of the spanning tree to be formed). Automaton $\text{Init}(p, v, \text{Dep}_{pv})$ initiates the computation by performing output action $\text{ping}(p, v, x, y)$ for each $(x, y) \in \text{Dep}_{pv}$, the dependencies of f_{pv} . Further, pv performs output action accept if it receives a message $\text{reply-par}(\dots, \text{true})$ from each of its children in the spanning tree, and its local check succeeds; otherwise output action reject is performed. We have the following lemma, which establishes the correctness of the protocol.

Lemma 9 (Snapshot Protocol) Let $r = s_0 a_1 s_1 \dots$ be any fair run of

$$\text{Init}(p, v, \text{Dep}_{pv}) \times \prod_{(x,y) \in \mathcal{P}^2 \setminus \{(p,v)\}} \text{Snapshot}(x, y, \text{Dep}_{xy}).$$

Then there exists an index j so that $a_j = \text{accept}$ or $a_j = \text{reject}$. Further, if $a_j = \text{accept}$ then there exists a vector $I \in D^n$ which is an information approximation for f and for which $I \preceq f(I)$. Furthermore, $I(p)(v) = pv.t_{cur}$: the current value of pv in $\llbracket \Pi \rrbracket^{\text{op}}$ at the time the snapshot protocol is initiated.

Proof (sketch) We assume for simplicity that $\text{Dep}_{pv} = \{xy\}$, i.e., that pv only depends on a single node xy (the proof idea works also in the general case). Let us assume that pv initiates the protocol by performing action $\text{ping}(p, v, x, y)$ and records its value in the operational semantics, $pv.t_{cur}$, at time 1 in r . Hence $xy.parent = (p, v)$ and $pv.val = pv.t_{cur}$. Eventually, the set of *parent* “pointers” will form a spanning tree, rooted at pv . Further, each node rs in this tree is associated a value v_{rs} at the time it performs the input action $\text{ping}(r, s, z, w)$ for some zw . For each such v_{rs} we have $v_{rs} = x_{rs}(t_{rs})$ (using the notation of Section 4), for some time t_{rs} . Hence there is a consistent vector $I = (\dots, v_{rs}, \dots) \in D^n$, consisting of all these values; the vector is consistent because each node zw ’s view of the vector will agree. It follows from Proposition 1 that I is an information approximation (this is actually non-trivial). Each node zw in the spanning tree will make the check $v_{zw} \preceq f_{zw}(I)$, and if all the children of zw reply with $\text{reply-par}(\dots, \text{true})$, zw will perform action $\text{reply-par}(z, w, zw.parent, zw.val, \text{true})$; otherwise, $\text{reply-par}(z, w, zw.parent, zw.val, \text{false})$ is performed. Hence, if pv receives “ true ” from each of its children and its check succeeded, action accept is performed correctly. Otherwise, if the check fails or some child replies with “ true ,” reject is performed. \square

Remarks. Note that the approximation propositions have “dual” versions.

Proposition 5 Let $(D, \preceq, \sqsubseteq)$ be a trust structure in which \preceq is \sqsubseteq -continuous. Let $I \in D^n$, and $f : D^n \rightarrow D^n$ be any function that is \sqsubseteq -continuous and \preceq -monotonic. If $\perp_{\sqsubseteq}^n \preceq I$ and $f(I) \preceq I$ then $\text{lfp } f \preceq I$.

Proposition 6 *Let $(D, \preceq, \sqsubseteq)$ be a trust structure in which \preceq is \sqsubseteq -continuous. Let $I \in D^n$, and $f : D^n \rightarrow D^n$ be any function that is \sqsubseteq -continuous and \preceq -monotonic. Assume that I is an information approximation for f . If $f(I) \preceq I$ then $\text{lfp } f \preceq I$.*

Proposition 5 may not seem as useful as its dual. The conclusion $\text{lfp } f \preceq I$ can usually only be used to *deny* a request, and a prover in the protocol for Proposition 5 would probably not be interested in supplying information which would help refuting its request! However, this is not always so. For example, suppose one is using trust structures conveying probabilistic information (e.g., [7, 24]), and that $I \preceq J$ expresses (informally) that, when interacting with a certain principal, the probability of a specific outcome given I , is lower than the probability of that outcome given J . In this case, an assertion of the form $\text{lfp } f \preceq I$, can convince the verifier that when interacting with the prover, the probability of a “bad” outcome is *below* a certain threshold.

In fact, it turns out that both approximation propositions are instances of a more general theorem, that can be seen as a combination of the two propositions presented in this section.

Proposition 7 (Generalised Approximation) *Let $(D, \preceq, \sqsubseteq)$ be a trust structure in which \preceq is \sqsubseteq -continuous. Let $J \in D^n$, and $f : D^n \rightarrow D^n$ be any function that is \sqsubseteq -continuous and \preceq -monotonic. Assume that J satisfies $J \preceq f(J)$. If there exists an information approximation $I \in D^n$ for f , with the property that $J \preceq I$, then $J \preceq \text{lfp } f$.*

Proof The proof of Proposition 7 is similar to that of Proposition 3. Follow the path from J via \preceq to $f^i(I)$ in the following diagram.

$$\begin{array}{ccccccc} J & \preceq & f(J) & \preceq & \cdots & \preceq & f^i(J) & \preceq & \cdots \\ \wedge & & \wedge & & & & \wedge & & \cdots \\ I & \sqsubseteq & f(I) & \sqsubseteq & \cdots & \sqsubseteq & f^i(I) & \sqsubseteq & \cdots \end{array}$$

By continuity of \preceq we have $J \preceq \bigsqcup_i f^i(I)$. \square

One obtains Proposition 3 with the trivial information approximation $I = \perp_{\sqsubseteq}$, and Proposition 4 by taking the proof to be the approximation, i.e. $J = I$.

One could imagine a protocol based on the generalised approximation proposition: The prover would send a proof J to a verifier, which would then ‘check’ the proof by distributedly verifying that (a) $J \preceq f(J)$ and (b) computing an information approximation I (via the operational semantics), and check that it satisfies $J \preceq I$. Note that this is more general than the ‘proof carrying request’ protocol as we do not require $J \preceq \perp_{\sqsubseteq}$. On the other hand, the protocol requires more computation and communication to obtain I .

We note finally that the \sqsubseteq -continuity property required of \preceq in our propositions is satisfied for all interesting trust-structures we are aware of: Theorem 3 of Carbone *et al.* [8] implies that the information-continuity condition is satisfied for all interval-constructed structures. Furthermore, their Theorem 1 ensures that interval-constructed structures are complete lattices with respect to \preceq (thus ensuring existence of \perp_{\preceq}). Several natural examples of non-interval domains can also be seen to have the required properties [24]. Also, the requirement that all policies π_p are monotonic also with respect to \preceq is sensible. It amounts to saying that if everyone raises their trust-levels in everyone, then policies should not assign lower trust levels to anyone.

8 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. The proof uses translation into BAAS’s for proving correctness of algorithms formalised with I/O automata. This technique may be applicable for other algorithms. We have also formalised and proved correct a number of other approximation techniques in the I/O automata setting (which we feel is a very appropriate model for algorithms in open distributed systems as it provides a simple formal model of distributed computation as well as rigorous, compositional reasoning techniques).

Our work shows how the trust structure framework can be used in operational trust management systems, not just in theory. However, the trust structures have seen relatively few applications since their conception in 2001. Work is needed to clarify whether or not trust structures are useful as the theoretical underpinnings of actual trust management systems.

Acknowledgements. We thank the anonymous reviewers for helping to significantly improve this paper. Furthermore we thank Andrew Twigg for substantial contributions to foundation of this paper. Also, we thank Vladimiro Sassone and Marco Carbone for useful insights regarding trust structures.

References

1. Abdul-Rahman, A.: A framework for decentralised trust reasoning. Ph.D. thesis, University of London, Department of Computer Science, University College London, England (2005)

2. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice-Hall International Editions. Prentice-Hall, Inc. (1989)
3. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The role of trust management in distributed systems security. In: J. Vitek, C.D. Jensen (eds.) Secure Internet Programming: Security Issues for Mobile and Distributed Objects, *Lecture Notes in Computer Science*, vol. 1603, pp. 185–210. Springer (1999)
4. Blaze, M., Feigenbaum, J., Keromytis, A.D.: KeyNote: Trust management for public-key infrastructures. In: Proceedings from Security Protocols: 6th International Workshop, Cambridge, UK, April 1998, vol. 1550, pp. 59–63 (1999)
5. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings from the 17th Symposium on Security and Privacy, pp. 164–173. IEEE Computer Society Press (1996)
6. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance checking in the policymaker trust management system. In: Proceedings from Financial Cryptography: Second International Conference (FC'98), Anguilla, British West Indies, February 1998, pp. 254–274 (1998)
7. Cahill, V., Gray *et al.*, E.: Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing* **2**(3), 52–61 (2003)
8. 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)
9. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomsa, B., Ylonen, T.: SPKI Certificate Theory. RFC 2693, ftp-site: <ftp://ftp.rfc-editor.org/in-notes/rfc2693.txt> (1999)
10. Garland, S.J., Lynch, N.A.: Using I/O automata for developing distributed systems. In: G.T. Leavens, M. Sitaraman (eds.) Foundations of Component-Based Systems, pp. 285–312. Cambridge University Press, NY (2000)
11. Garland, S.J., Lynch, N.A., Tauber, J., Vaziri, M.: IOA user guide and reference manual. Tech. Rep. MIT-LCS-TR-961, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), Cambridge, MA (2004)
12. Jøsang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. *Decision Support Systems*, (to appear, preprint available online: <http://sky.fit.qut.edu.au/~josang/>) (2006)
13. Krukow, K.: An operational semantics for trust policies. Tech. Rep. RS-05-30, BRICS, University of Aarhus (2005). Available online: <http://www.brics.dk/RS/05/30>
14. Krukow, K.: Towards a theory of trust for the global ubiquitous computer. Ph.D. thesis, University of Aarhus, Denmark (2006). ; available online (submitted): <http://www.brics.dk/~krukow>
15. Krukow, K., Nielsen, M.: From simulations to theorems: A position paper on research in the field of computational trust. In: To be published in Proceedings from Formal Aspects in Security and Trust (FAST 2006). Springer (2006)
16. 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), pp. 805–814. IEEE (2005)
17. Krukow, K., Twigg, A.: Distributed approximation of fixed-points in trust structures. Tech. Rep. RS-05-6, BRICS, University of Aarhus (2005). Available online: <http://www.brics.dk/RS/05/6>
18. Li, N., Feigenbaum, J., Grosf, B.: A logic-based knowledge representation for authorization with delegation. In: Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99), pp. 162–174. IEEE Computer Society Press (1999)
19. Li, N., Grosf, B., Feigenbaum, J.: A logic-based knowledge representation for authorization with delegation. In: Proceedings of the 9th Computer Security Foundations Workshop (CSFW'99), pp. 162–174. IEEE Computer Society (1999)
20. Li, N., Mitchell, J.: Datalog with constraints: A foundation for trust-management languages. In: Proceedings from the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003), *Springer Lecture Notes in Computer Science*, vol. 2562, pp. 58–73. Springer (2003)
21. Li, N., Mitchell, J.C., Winsborough, W.H.: Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM* **52**(3), 474–514 (2005)
22. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA (1996)
23. 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), pp. 137–151. ACM Press (1987)
24. Nielsen, M., Krukow, K.: On the formal modelling of trust in reputation-based systems. In: J. Karhumäki, H. Maurer, G. Paun, G. Rozenberg (eds.) Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday, *Lecture Notes in Computer Science*, vol. 3113, pp. 192–204. Springer Verlag (2004)
25. Ramchurn, S.D., Huynh, D., Jennings, N.R.: Trust in multi-agent systems. *The Knowledge Engineering Review* **19**(1), 1–25 (2004)
26. Sabater, J., Sierra, C.: Review on computational trust and reputation models. *Artificial Intelligence Review* **24**(1), 33–60 (2005). DOI 10.1007/s10462-004-0041-5
27. Weeks, S.: Understanding trust management systems. In: Proceedings from the 2001 IEEE Symposium on Security and Privacy, pp. 94–106. IEEE Computer Society Press (2001)
28. Winskel, G.: Formal Semantics of Programming Languages : an introduction. Foundations of computing. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts (1993)

A Proofs

Lemma 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 $\mathbf{send}(p, r, q, d)$ action in r_c is caused by a unique $\mathbf{eval}(p, q)$ action, and each $\mathbf{recv}(p, r, s, d)$ action in r_c is caused by a unique $\mathbf{send}(r, p, s, d)$ action.
- The $cause_{r_c}$ function is injective when restricted to \mathbf{recv} actions. That is, for any indices k, k' with $k \neq k'$, if $a_k = \mathbf{recv}(\dots)$ and $a_{k'} = \mathbf{recv}(\dots)$, 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 $k < k'$, $a_k = \mathbf{recv}(p, r, s, d)$ and $a_{k'} = \mathbf{recv}(p', r', s', d')$. Let $j = cause_{r_c}(k)$ and $j' = cause_{r_c}(k')$, then by the above, $a_j = \mathbf{send}(r, p, s, d)$ and $a_{j'} = \mathbf{send}(r', p', s', d')$. 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 $S_0 = \{j \mid j < k, a_j = \mathbf{send}(r, p, s, d)\}$, $S'_0 = \{j \mid j < k', a_j = \mathbf{send}(r, p, s, d)\}$, $R_0 = \{j \mid j < k, a_j = \mathbf{recv}(p, r, s, d)\}$ and $R'_0 = \{j \mid j < k', a_j = \mathbf{recv}(p, r, s, d)\}$. We have $S_0 \subseteq S'_0$ and $R_0 \subsetneq R'_0$, in particular, $k \in R'_0 \setminus R_0$.

$$\text{cause}_{r_c}(k) = \min(S_0 \setminus \text{cause}_{r_c}(R_0))$$

$$\text{cause}_{r_c}(k') = \min(S'_0 \setminus \text{cause}_{r_c}(R'_0))$$

Injectivity follows, as $\text{cause}_{r_c}(k) \in \text{cause}_{r_c}(R'_0)$.

Lemma 10 (FIFO) *Let $r_c = s_0 a_1 s_2 \dots$ be a finite or infinite fair run of $\text{Channel}(p, r, q)$ for $p, r, q \in \mathcal{P}$. Suppose that $a_k = \text{send}(p, r, q, d)$ and $a_{k'} = \text{send}(p, r, q, d')$ for some $d, d' \in D$. If $k \leq k'$ then there exists unique j, j' with $k < j$ and $k' < j'$, so that $j \leq j'$, $a_j = \text{recv}(r, p, q, d)$, $a_{j'} = \text{recv}(r, p, q, d')$, $\text{cause}_{r_c}(j) = k$ and $\text{cause}_{r_c}(j') = k'$.*

Proof Since $a_k = \text{send}(p, r, q, d)$ then we have $s_k \cdot \text{buffer} = u \cdot d$, 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 $1 \leq i \leq N$,
- $a_{k_i} = \text{recv}(r, p, q, u_i)$ for all $1 \leq i \leq N$,
- $a_{k_{N+1}} = \text{recv}(r, p, q, d)$, 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 recv action, i.e., $a_l \neq \text{recv}(p, r, q, d_0)$ for all $d_0 \in D$.

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

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 $\text{cause}_{r_c}(R_0) = S_0^k$. Hence,

$$S_0 \setminus \text{cause}_{r_c}(R_0) = S_0 \setminus S_0^k$$

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

Similar reasoning applies to k' , so let j, j' be so that $a_j = \text{recv}(r, p, q, d)$, $k = \text{cause}_{r_c}(j)$, $a_{j'} = \text{recv}(r, p, q, d')$ and $k' = \text{cause}_{r_c}(j')$. 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 $s_{k'} \cdot \text{buffer} = u_{i+1} u_{i+2} \dots u_N d s' d'$ for some $s' \in D^*$, and hence, $j' = k'_{N'+1} > k_{N+1} = j$.

Lemma 11 (Cause and Effect) *Let $\Pi = (\pi_p \mid p \in \mathcal{P})$ be a collection of policies, and let $r_c = s_0 a_1 s_1 a_2 \dots$ be a finite or infinite fair run of $\llbracket \Pi \rrbracket^{\text{op}}$. The following properties hold of r_c :*

1. Assume that for some $k \geq 0$, we have $s_k \cdot \text{pq.wake} = \text{true}$, then there exists a $k' > k$ so that $a_{k'} = \text{eval}(p, q)$.
2. Assume that $a_{k_0} = \text{eval}(p, q)$ and that $s_{k_0-1} \cdot \text{pq.t}_{\text{cur}} \neq s_{k_0} \cdot \text{pq.t}_{\text{cur}} = d$. Let $k_1 > k$ be least with $a_{k_1} = \text{eval}(p, q)$ (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 $a_{k_r} = \text{send}(p, r, q, -)$. Furthermore, $a_{k_r} = \text{send}(p, r, q, d)$ and $\text{cause}_{r_c}(k_r) = k_0$.

3. Assume that $a_k = \text{send}(p, r, q, d)$ and also that $a_{k'} = \text{send}(p, r, q, d')$. Then, $k < k'$ implies $\text{cause}_{r_c}(k) < \text{cause}_{r_c}(k')$.
4. Assume that $a_k = \text{recv}(r, p, q, d)$ and also that $a_{k'} = \text{recv}(r, p, q, d')$. Then, $k < k'$ implies $\text{cause}_{r_c}(k) < \text{cause}_{r_c}(k')$.

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

1. Assume that $s_k \cdot \text{pq.wake} = \text{true}$. Assume first that for every $r \in \mathcal{P}$ we have $s_k \cdot \text{pq.send}(r) = \text{false}$. Then action $\text{eval}(p, q)$ is enabled. Notice that this action stays enabled until a $\text{eval}(p, q)$ event occurs. Since $\{\text{eval}(p, q)\}$ is an equivalence class, fairness of r_c implies that there exists some $k' > k$ so that $a_{k'} = \text{eval}(p, q)$. Now, suppose instead that for some $r \in \mathcal{P}$ we have $s_k \cdot \text{pq.send}(r) = \text{true}$. Then action $\text{send}(p, r, q, d)$ is enabled for $d = s_k \cdot \text{pq.t}_{\text{cur}}$, and notice that this action stays enabled until action $\text{send}(p, r, q, d)$ occurs. Since $\{\text{send}(p, r, q, c) \mid c \in D\}$ is an equivalence class, and only $\text{send}(p, r, q, d)$ is enabled in the class, fairness of r_c means that for some $k'_0 > k$ we have $a_{k'_0} = \text{send}(p, r, q, d)$. Hence, $s_{k'_0} \cdot \text{pq.send}(r) = \text{false}$. Let k_0 be the least such index, and note that for all j with $k \leq j \leq k_0$ we have $s_j \cdot \text{pq.wake} = \text{true}$ (as no $\text{eval}(p, q)$ action can occur while $\text{pq.send}(r) = \text{true}$). Since this holds for all r , there must exist a $k' > k$ so that $s_{k'} \cdot \text{pq.wake} = \text{true}$ and for all $r \in \mathcal{P}$ we have $s_{k'} \cdot \text{pq.send}(r) = \text{false}$, and we are done by the initial comment.
2. Assume that $a_{k_0} = \text{eval}(p, q)$, and that

$$s_{k_0-1} \cdot \text{pq.t}_{\text{cur}} \neq s_{k_0} \cdot \text{pq.t}_{\text{cur}} = d.$$

Notice that $s_{k_0} \cdot \text{pq.wake} = \text{true}$, and let $k_1 > k_0$ be the (index of the) first occurrence of an $\text{eval}(p, q)$ event after time k_0 . Notice that since no $\text{eval}(p, q)$ event occurs in the interval (k_0, k_1) we have $s_l \cdot \text{pq.t}_{\text{cur}} = d$ for all $l \in [k_0, k_1)$. Let $r \in \mathcal{P}$ be arbitrary. Notice that $\text{send}(p, r, q, d)$ is enabled at time k_0 , and stays enabled until a $\text{send}(p, r, q, d)$ action occurs. By fairness such an action must occur, so let $k_r > k_0$ be the least index so that $a_{k_r} = \text{send}(p, r, q, d)$. Notice that after time k_0 , no $\text{eval}(p, q)$ action can occur before a $\text{send}(p, r, q, d)$ 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 $s_k \cdot \text{pq.send}(r) = \text{true}$ and for k in $[k_r, k_1)$ we have $s_k \cdot \text{pq.send}(r) = \text{false}$. Hence, there can only be one occurrence of a $\text{send}(p, r, q, d)$ event in the time interval (k_0, k_1) . Finally, since for all k with $k_0 < k < k_r$ we have $a_k \neq \text{eval}(p, q)$, it follows that $\text{cause}_{r_c}(k_r) = k_0$.

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

$$\begin{aligned} \{j \mid j < k, s_j \cdot \text{pq.send}(r) = \text{false}, \\ s_{j+1} \cdot \text{pq.send}(r) = \text{true}\} \end{aligned}$$

is contained in

$$\begin{aligned} \{j \mid j < k', s_j \cdot \text{pq.send}(r) = \text{false}, \\ s_{j+1} \cdot \text{pq.send}(r) = \text{true}\} \end{aligned}$$

Hence, $\text{cause}_{r_c}(k) \leq \text{cause}_{r_c}(k')$. Assume for the sake of contradiction that, $\text{cause}_{r_c}(k) = \text{cause}_{r_c}(k') = k_0$. Let k_1 be the first occurrence of $\text{eval}(p, q)$ after time k_0 . Then $k_1 > k$ because by definition of $\text{cause}_{r_c}(k)$, there can be no $\text{eval}(p, q)$ occurrences in the interval $(\text{cause}_{r_c}(k), k] = (k_0, k]$ (because $\text{pq.send}(r) = \text{true}$). Since also $\text{cause}_{r_c}(k_1) = k_0$, 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 $\mathbf{send}(p, r, q, _)$ occurrence in the interval $[k_0, k_1]$, and hence $k = k'$, which contradicts $k < k'$. So, by contradiction, we must have $\mathit{cause}_{r_c}(k) < \mathit{cause}_{r_c}(k')$.

4. Assume that $a_k = \mathbf{recv}(r, p, q, d)$ and also that $a_{k'} = \mathbf{recv}(r, p, q, d')$. Assume $k < k'$.

Then $\mathit{cause}_{r_c}(k) < \mathit{cause}_{r_c}(k')$ follows because if we have $\mathit{cause}_{r_c}(k') \leq \mathit{cause}_{r_c}(k)$. By the FIFO Lemma (10),

$$k' = \mathit{effect}_{r_c}(\mathit{cause}_{r_c}(k')) \leq \mathit{effect}_{r_c}(\mathit{cause}_{r_c}(k)) = k$$

Lemma 4 For any $p, q, r, s \in \mathcal{P}$, function τ_{rs}^{pq} is monotonically increasing.

Proof We must show for all t, u if $t \leq u$ then $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$. If no $\mathbf{recv}(p, r, s, d)$ exists before u , or no $\mathbf{recv}(p, r, s, d)$ exists before t but $\mathbf{recv}(p, r, s, d)$ exists before u , then it is simple to verify that $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$. Let $t \leq u$, and let k, l denote the “ k ’s”, corresponding to t and u respectively, in the definition of τ_{rs}^{pq} , i.e., $a_k = \mathbf{recv}(p, r, s, d)$, $k \leq t$, $a_l = \mathbf{recv}(p, r, s, d')$, $l \leq u$. Because $t \leq u$, clearly $k \leq l$. By Lemma 11, $k' = \mathit{cause}_{r_c}(k) \leq \mathit{cause}_{r_c}(l) = l'$. Similarly, by Lemma 11, $k'' = \mathit{cause}_{r_c}(k') \leq \mathit{cause}_{r_c}(l') = l''$. 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 $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$. If $d \neq d'$ then for all j' with $k'' \leq j' \leq \tau_{rs}^{pq}(t)$, $s_k.rs.t_{cur} = d$, means that $k'' \leq l''$ implies $\tau_{rs}^{pq}(t) < l'' \leq \tau_{rs}^{pq}(u)$.

Lemma 5 (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,

$$\forall k. 0 \leq k < |r_c| \Rightarrow \mathit{state}_{con}(r_c, k) = \mathit{state}_{abs}(r_a, k)$$

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

Inductive step. Assume that for all $k' \leq k$, $\mathit{state}_{con}(r_c, k') = \mathit{state}_{abs}(r_a, k')$, where $k + 1 < |r_c|$.

- Case $a_{k+1} = \mathbf{eval}(p, q)$ for some $p, q \in \mathcal{P}$. Since by the induction hypothesis $\mathit{state}_{con}(r_c, k) = \mathit{state}_{abs}(r_a, k)$, 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 $a_{k+1} = \mathbf{send}(p, q, s, v)$. Notice that send-actions don’t affect the abstract state: $\mathit{state}_{con}(r, k+1) = \mathit{state}_{con}(r, k) = \mathit{state}_{abs}(r, k) = \mathit{state}_{abs}(r, k+1)$.
- Case $a_{k+1} = \mathbf{recv}(p, r, s, v)$. 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 $(v, w) \neq (r, s)$,

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

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 $x^{pq}(k+1)_{rs} = x_{rs}(\tau_{rs}^{pq}(k+1))$, and we simply recall that for any m , if $m = \mathbf{recv}(p, r, s, d)$, then, as noted in the definition

of τ_{rs}^{pq} , $s_{\tau_{rs}^{pq}(m)}.pq.t_{cur} = d$. Now, since $\tau_{rs}^{pq}(k+1) \leq k+1$, there are two cases. If $\tau_{rs}^{pq}(k+1) \leq k$ then the i.h. implies

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

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

$$x_{rs}(k) \stackrel{(i.h.)}{=} s_k.rs.t_{cur} = s_{k+1}.rs.t_{cur} = s_{\tau_{rs}^{pq}(k+1)}.rs.t_{cur} = v$$

Lemma 6 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 $[\Pi]^{op-abs}$.
- If r_c is finite, then r_a is a finite fair run of $[\Pi]^{op-abs}$.

Proof We prove each point separately. In the following “The Lemma” refers to Lemma 11.

- Let $r_c = s_0a_1s_1 \dots$ be an infinite fair run of $[\Pi]^{op}$. We let $p, q \in \mathcal{P}$ be arbitrary, and prove that there are infinitely many $\mathbf{eval}(p, q)$ events in r_c , which implies that T^{pq} is infinite. Note that it suffices to prove that there are infinitely many \mathbf{send} -events in r_c , by the following. Assume there are infinitely many \mathbf{send} -events in r_c , and note that since \mathcal{P} is finite, there must exist $r, s, t \in \mathcal{P}$ so that $\mathbf{send}(r, t, s, _)$ events occur infinitely often (i.o.) in r_c . By The Lemma, cause_{r_c} maps the indexes of these $\mathbf{send}(r, t, s, _)$ events, injectively, to indexes k with $a_k = \mathbf{eval}(r, s)$ 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, also, $\mathbf{send}(r, p, s, _)$ events occur i.o. in r_c , hence, by the FIFO Lemma, $\mathbf{recv}(p, \dots)$ events occur i.o. in r_c . But this implies that $pq.wake = \mathbf{true}$ infinitely often, and hence by The Lemma, we have $\mathbf{eval}(p, q)$ infinitely often.

So let us prove that there are infinitely many \mathbf{send} -events. Assume this is not the case, and let k be an arbitrary index so that there are no \mathbf{send} events after k . Note that by The Lemma it suffices to prove that there is some $k' > k$ so that $a_{k'} = \mathbf{eval}(u, v)$ 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 \mathbf{send} events occur later than k , then there can be only finitely many \mathbf{recv} -events after k . So let $K \geq k$ be arbitrary so that there are no \mathbf{recv} -events after K . By construction there can only be \mathbf{eval} -events after K , but since r_c is infinite there must also be some \mathbf{eval} event with a change in the t_{cur} variable (otherwise all $wake$ variables eventually become \mathbf{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_{rs}^{pq} is the identity function. So assume this is not the case. We know that there are infinitely many $\mathbf{eval}(r, s)$ events in r_c . There are three cases.

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

If there are some but only finitely many k with $a_k = \mathbf{eval}(r, s)$ 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 and the FIFO Lemma, let l be so that $a_l = \mathbf{recv}(p, r, s, v)$ and $\mathit{cause}_{r_c}(\mathit{cause}_{r_c}(l)) = k_0$. Now we get,

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

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

– 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 $x_{rs}(\tau_{rs}^{pq}(t_{pq}^*))$. If $(p, q) = (r, s)$ then τ_{rs}^{pq} is the identity, and hence, by definition of t_{pq}^* , $x_{pq}(t_{pq}^*) = x_{pq}(\tau_{rs}^{pq}(t_{pq}^*))$. So assume that $(p, q) \neq (r, s)$. Assume, for the sake of contradiction, that $x_{rs}(\tau_{rs}^{pq}(t_{pq}^*)) \neq x_{rs}(t^*)$. By Lemma 5, this implies that there must exist an index k with $a_k = \text{eval}(r, s)$ 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 $K \leq j \leq t^*$, $s_j.rs.t_{cur} = x_{rs}(t^*) \stackrel{(\text{def})}{=} d$. By Lemma 11 and the FIFO Lemma, there exists $j_p > k_p > K$ so that $a_{j_p} = \text{recv}(p, r, s, d)$, $a_{k_p} = \text{send}(r, p, s, d)$, $\text{cause}_{r_c}(k_p) = K$ and $\text{cause}_{r_c}(j_p) = k_p$. Note, there cannot be a later index $j' > k_p$ with $a_{j'} = \text{recv}(p, r, s, -)$, because then

$$\text{cause}_{r_c}(\text{cause}_{r_c}(j')) > \text{cause}_{r_c}(\text{cause}_{r_c}(k_p) = K,$$

which contradicts maximality of K . Hence, if $k_p \leq t_{pq}^*$ then $s_{t_{pq}^*}.pq.t_{cur} = d$, and by Lemma 5, $x_{pq}(\tau_{rs}^{pq}(t_{pq}^*)) = d = x_{rs}(t^*)$: a contradiction. But, if $k_p > t_{pq}^*$ then we must have $s_{k_p}.pq.wake = \text{true}$ and by Lemma 11 there must be a later $\text{eval}(p, q)$ event, contradicting maximality of t_{pq}^* .

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

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

Proof By induction in t .

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

Inductive step.

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

$$x^{pq}(t)_{rs} = x_{rs}(\tau_{rs}^{pq}(t))$$

and

$$x^{pq}(t+1)_{rs} = x_{rs}(\tau_{rs}^{pq}(t+1))$$

Note that since τ_{rs}^{pq} is monotonically increasing, we have $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(t+1)$. Note also, that if $\tau_{rs}^{pq}(t+1) \leq t$ we can just refer to the i.h., and we are done. So assume, finally, that $\tau_{rs}^{pq}(t+1) = t+1$. 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., $x_{rs}(\tau_{rs}^{pq}(t)) = x^{rs}(\tau_{rs}^{pq}(t))_{rs} \sqsubseteq x^{rs}(t)_{rs} = x_{rs}(t)$ (the i.h. applies since $\tau_{rs}^{pq}(t) \leq t$). Now we are done, because we have already proved that $x_{rs}(t) \sqsubseteq f_{rs}(x^{rs}(t)) = x_{rs}(t+1)$.