BRICS RS-05-6 Krukow & Twigg: Distributed Approximation of Fixed-Points in Trust Structures

BRICS

Basic Research in Computer Science

Distributed Approximation of Fixed-Points in Trust Structures

Karl Krukow
Andrew Twigg

BRICS Report Series **RS-05-6**

ISSN 0909-0878 **February 2005**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/05/6/

Distributed Approximation of Fixed-Points in Trust Structures

Karl Krukow*†
krukow@brics.dk

Andrew Twigg‡
Andrew.Twigg@cl.cam.ac.uk

February 2005

Abstract

Recently, Carbone, Nielsen and Sassone introduced the trust-structure framework: a semantic model for trust-management in global-scale distributed systems. The framework is based on the notion of trust structures: a set of "trust-levels" ordered by two distinct partial orderings. In the model, a unique global trust-state is defined as the least fixed-point of a collection of local policies assigning trust-levels to the entities of the system. However, the framework is a purely denotational model; it gives precise meaning to the global trust-state of a system, but without specifying a way to compute this abstract mathematical object.

This paper complements the denotational model of trust structures with operational techniques. It is shown how the least fixed-point can be computed using a simple, totally asynchronous distributed-algorithm. In addition, two efficient protocols for approximating the least fixed-point are provided. These enable sound reasoning about the global trust-state without computing the exact fixed-point. Finally, dynamic algorithms are presented for safe reuse of information between computations, in face of dynamic trust-policy updates.

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

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

‡Computer Laboratory, University of Cambridge, Cambridge, UK

1 Introduction

The need for flexible security mechanisms in emerging Internet-based distributed-systems is evident. However, the diversity and scale of these systems, combined with the lack of centralized authority, means that traditional mechanisms for security decision-making, e.g. access-control lists, are often too restrictive and complex to deploy [2]. The concept of trust management, introduced by Blaze *et al.* [4], was presented as a solution to the problems with authorization in large-scale distributed systems. Traditional trust-management systems make security decisions based on locally controlled security-policies, dealing with authorization by deciding the so-called compliance-checking problem: given a *request* to perform a certain action, together with a set of *credentials*; does the request comply with the local *security policy*, given the credentials?

In *dynamic* trust-management-systems [15,16,21], trust-specifications are often based on the *past behaviour* of principals, which gives rise to a different, more flexible notion of trust than that of traditional trust-management systems. The traditional systems often take an "all or nothing" approach, in which no or partial credentials necessarily means no interaction. By broadening the range of specifications of trust-levels, one may encourage interaction in situations where the traditional approach would be too restrictive.

While the traditional notion of trust management is well understood, e.g. Mitchell *et al.* [8, 17, 18], and, to a large extent, captured concisely in a mathematical framework of Weeks [23]; a lot of the "broader" dynamic systems lack such foundation in formal methods (this point is illustrated by the wide range of related systems in the survey [13]). This lack prompted the development of a mathematical framework for trust [7], inspired by that of Weeks, but departing from Weeks by emphasizing the concept of *information* in contrast to *authorization*. The framework, which was introduced by Carbone *et al.* [7], discussed also by Nielsen *et al.* [19] and Krukow [15], is the focus of this paper. Our motivation and contributions are to complement this model with a operational techniques, and consequently, we recall the model now.

1.1 The Trust-Structure Framework

A trust model should be generic enough to be instantiated to support authorization in a variety of distributed computing systems. The trust-structure framework [7] is a generic model, parameterized by a set X

of possible *trust values* representing distinct levels or degrees of trust, relevant for a particular application. The framework is aimed at global-computing environments, and is based on a domain-theoretic modelling of trust information. The goal is to provide a *unique global trust-state* for every set $\mathcal{P}$ of principal identities, where each principal $p \in \mathcal{P}$ defines a trust policy π_p which quantifies for any principal identity $q \in \mathcal{P}$ the level of trust that p has in q.

Trust structures. In the framework, trust is something which exists between *pairs of principals*; it is *quantified* and *asymmetric* in that we care of "how *much*" or "to what *degree*" principal p trusts principal q (which may not be to the same degree that q trusts p). Each application instance of the framework defines a so-called *trust structure*, $T = (X, \preceq, \sqsubseteq)$, which consists of a set X of *trust values*, together with two relations on X; the trust ordering ($\preceq$) and the information ordering ($\sqsubseteq$). The elements $s, t \in X$ express the levels of trust that are relevant for the particular instance, and $s \preceq t$ means that t denotes at least as high a trust-level as s. In contrast, the information ordering introduces a notion of precision or information. The key idea is that the elements of X embody various degrees of uncertainty. One may think of assertion $x \sqsubseteq y$ as the statement that x can be refined into y, or that x approximates y. Krukow has considered a categorical axiomatization of trust structures [15], but here we are concerned with trust structures in their most general form, given by the following definition.

Definition 1.1 (Trust Structure). *A* trust structure *is a triple $T = (X, \preceq, \sqsubseteq)$, consisting of a set X of* trust values*, ordered by two binary relations: $\preceq \subseteq X \times X$ called the* trust ordering *of T, and $\sqsubseteq \subseteq X \times X$ called the* information ordering *of T. The trust ordering is a pre-order on X, meaning that it is reflexive and transitive, and the information ordering makes $(X, \sqsubseteq)$ an ω-complete partial order with a least element, denoted $\perp_\sqsubseteq$. For any $\sqsubseteq$-ω-chain, $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$, the least upper-bound in $(X, \sqsubseteq)$ is denoted by*

$$\bigsqcup_{i\in\omega} x_i$$

In simple cases, the trust values are just symbolic, e.g. $\mathsf{unknown} \sqsubseteq \mathsf{low} \preceq \mathsf{high}$, but they may also have more internal structure. As a simple example of a trust structure, consider the so-called "MN" trust-structure T_{MN} [15]. In this structure, trust values are pairs (m, n) of (extended) natural numbers, representing $m + n$ interactions with a principal; each interaction classified as either "good" or "bad". In a trust value (m, n), the first component, m, denotes the number of "good" interactions, and the second, the number of "bad" ones. The information-ordering is given by: $(m, n) \sqsubseteq (m', n')$ only if one can refine (m, n) into (m', n') by adding zero or more good interactions, and, zero or more bad interactions, i.e., iff $m \leq m'$ and $n \leq n'$. In contrast, the trust ordering is given by: $(m, n) \preceq (m', n')$ only if $m \leq m'$ and $n \geq n'$. Nielsen *et al.* [7, 20], have considered several additional examples of trust structures.

Trust Policies. Given a fixed trust structure $T = (X, \preceq, \sqsubseteq)$, a *global trust-state* of the system is a function $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to X$. The interpretation is that gts represents the trust state where p's trust in q (formalized as an element of X) is given by $\mathsf{gts}(p)(q)$. The goal of the framework is to uniquely define a global trust-state, denoted $\overline{\mathsf{gts}}$. Each principal $p \in \mathcal{P}$ autonomously controls a trust policy, denoted π_p, which then determines p's trust within the unique global trust-state, i.e. $\overline{\mathsf{gts}}(p)$.

Definition 1.2 (Trust Policy). *Let $T = (X, \preceq, \sqsubseteq)$ be a trust structure. A* trust policy *in T, is a function $\pi : (\mathcal{P} \to \mathcal{P} \to X) \to \mathcal{P} \to X$, which is continuous with respect to the point wise extension of $\sqsubseteq$.*[1] *This continuity property is called* information continuity*.*

In the simplest case, π_p could be a constant function, ignoring its first argument $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to X$. As an example, $\pi_p(\mathsf{gts}) = \lambda q.t_0$ (for some $t_0 \in X$) defines p's trust in any $q \in \mathcal{P}$ as the constant t_0. In general, policy π_p may refer to other policies (π_z, $z \in \mathcal{P}$), and the general interpretation of π_p is the following. *Given that* all principals assign trust values as specified in the global trust-state $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to X$, *then* p assigns trust values as specified in function $\pi_p(\mathsf{gts}) : \mathcal{P} \to X$. For example, function $\pi_p(\mathsf{gts}) = \lambda q \in \mathcal{P}.(\mathsf{gts}(A)(q) \vee_{\preceq} \mathsf{gts}(B)(q)) \wedge_{\preceq} \mathsf{medium}$, represents a policy saying "for any $q \in \mathcal{P}$, the trust in q is the least upper-bound in $(X, \preceq)$ of what A and B say, but no more than the constant $\mathsf{medium} \in X$."[2] Such *policy references* are very similar to the concept of *delegation*, known from traditional trust management.

[1] We overload $\sqsubseteq$ (respectively $\preceq$) to denote also the pointwise extension of $\sqsubseteq$ ($\preceq$) to the function space $X^{\mathcal{P}} = \mathcal{P} \to X$ as well as to $(X^{\mathcal{P}})^{\mathcal{P}} = \mathcal{P} \to \mathcal{P} \to X$.

[2] Assuming that $(X, \preceq)$ is a lattice, and that $\vee_{\preceq}$ is an information-continuous operation (which is often the case). We always denote information least upper bounds by "square" symbols $\sqcup$, and trust least-upper-bounds/greatest-lower-bounds by $\vee/\wedge$.

Unique Trust-State. The collection of the trust policies of all principals, denoted $\Pi = (\pi_p : p \in \mathcal{P})$, thus "spins a global web-of-trust" in which the trust policies mutually refer to each other. Since trust policies Π may give rise to cyclic policy-references, a crucial requirement is that the information ordering makes $(X, \sqsubseteq)$ a complete partial order (cpo) with a bottom element. Since all policies are information-continuous, there exists a unique information-continuous function $\Pi_\lambda = \langle \pi_p : p \in \mathcal{P} \rangle$, of type $X^{\mathcal{P}^{\mathcal{P}}} \rightarrow X^{\mathcal{P}^{\mathcal{P}}}$ with the property that $\mathtt{Proj}_p \circ \Pi_\lambda = \pi_p$ for all $p \in \mathcal{P}$, where $\mathtt{Proj}_p$ is the p'th projection.[3] Since Π_λ is information-continuous and $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow X, \sqsubseteq)$ is a cpo with bottom, Π_λ has a (unique) least fixed-point [24] which we denote $\mathsf{lfp}_\sqsubseteq \Pi_\lambda$ (or simply $\mathsf{lfp}\, \Pi_\lambda$):

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

Hence, for any collection of trust policies Π, we can define the unique global trust-state induced by that collection, as $\overline{\mathsf{gts}} = \mathsf{lfp}\, \Pi_\lambda$, which has the type of global trust-states, $\mathcal{P} \rightarrow \mathcal{P} \rightarrow X$. This unique trust-state thus satisfies the following fixed-point equation:

$$\begin{array}{rcll} \forall p \in P. \quad m(p) & = & \mathtt{Proj}_p(m) & \\ & = & \mathtt{Proj}_p(\Pi_\lambda(m)) & \qquad \text{(since } \Pi_\lambda(m) = m\text{)} \\ & = & \pi_p(m) & \end{array}$$

Reading this from the left to the right, any function $m : \mathcal{P} \rightarrow \mathcal{P} \rightarrow X$ satisfying this equation is *consistent* with the policies $(\pi_p \mid p \in \mathcal{P})$, i.e. *any* fixed point of Π_λ is consistent with all policies π_p. Consider now two mutually referring functions π_p and π_q, given by $\pi_p = \lambda m.\mathtt{Proj}_q(m)$, and $\pi_q = \lambda m.\mathtt{Proj}_p(m)$. Intuitively, there is no information present in these functions; p delegates all trust-questions to q and similarly q delegates to p. In this case, we would like the global trust-state gts induced by the functions to take the value $\bot_\sqsubseteq$ on any entry $z \in \mathcal{P}$ for both p and q, i.e., for both $x = p$ and $x = q$ and for all $z \in \mathcal{P}$ we should have $\mathsf{gts}(x)(z) = \bot_\sqsubseteq$. This is exactly what is obtained by choosing the information-*least* fixed-point of Π_λ. We summarize as a definition.

[3] $\mathtt{Proj}_p$ is given by: for all $m : \mathcal{P} \rightarrow \mathcal{P} \rightarrow X$. $\mathtt{Proj}_p(m) = m(p)$.

Definition 1.3 (Global Trust-state). *Let $T = (X, \preceq, \sqsubseteq)$ be a trust structure, $\mathcal{P}$ a set of principal identities, and $\Pi = (\pi_p \mid p \in \mathcal{P})$ be a collection of trust policies in T, indexed by the principal identities. Let $\Pi_\lambda = \langle \pi_p \mid p \in \mathcal{P} \rangle$. The* global trust-state induced by Π*, denoted $\overline{\mathsf{gts}}$, is given by*

$$\overline{\mathsf{gts}} = \mathsf{lfp}_{\sqsubseteq} \Pi_\lambda$$

1.2 The Operational Problem

Many interesting systems are instances of the trust-structure framework [7,15], but one could argue against its usefulness as a basis for the actual construction of trust-management systems. In order to make security decisions, each principal p will need to reason about its trust in others, that is, the values of $\overline{\mathsf{gts}}(p)$. While the framework does ensure the existence of a unique (theoretically well-founded) global trust-state, it is not "operational" in the sense of providing a way for principals to actually *compute* the trust values. Furthermore, as we shall argue in the following, the standard way of computing least fixed-points is inadequate in our scenario.

When the cpo $(X, \sqsubseteq)$ is of finite height h, the cpo $(\mathcal{P} \to \mathcal{P} \to X, \sqsubseteq)$ 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 $(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda^2(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \cdots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p.\lambda q.\bot_\sqsubseteq)$ [24]. 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 (typically, each principal stores its own policy). More importantly, even if the height h is finite, the number of principals $|\mathcal{P}|$, though finite, will be *very* large. Furthermore, even if resources were available to make this computation, we can not assume that any central authority is present to perform it. Finally, since each principal p defines its trust policy π_p autonomously, an inherent problem with trying to compute the fixed point is the fact that p might decide to change its policy π_p to π_p' at any time. Such a policy update would be likely to invalidate data obtained from a fixed-point computation done with global function Π_λ, i.e., one might not have time to compute $\mathsf{lfp}\,\Pi_\lambda$ before the policies have changed to Π'.

The above discussion indicates that exact computation of the fixed point is infeasible, and hence that the framework is not suitable as an

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

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 *idealized* fixed-point $\mathsf{lfp}\,\Pi_\lambda$.

1.3 Technical Contents

Our work essentially deals with the operational problems left as "future work" by Carbone *et al.* [7]. Firstly, techniques for actual distributed computation of approximations to the idealized trust-values, over a global, highly dynamic, decentralized network. We start by showing that although it may be infeasible to compute the global trust-state, $\overline{\mathsf{gts}} : \mathcal{P} \to \mathcal{P} \to X$, one can instead try to compute so-called *local* fixed-point values. We take the practical point-of-view of a specific principal R, wanting to reason about its trust value for a fixed principal q. The basic idea is that instead of computing the entire state $\overline{\mathsf{gts}}$, and *then* "looking up" value $\overline{\mathsf{gts}}(R)(q)$ to learn R's trust in q, one may instead compute this value directly. We prove a convergence result that enables us to apply a robust totally-asynchronous distributed algorithm of Bertsekas [1] for local fixed-point computation. This is developed in Section 2.

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 3, we take very mild assumptions on the relation between the two orderings in trust structures. This enables us to prove the soundness of 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 $\bar{p} : \mathcal{P} \to \mathcal{P} \to X$ with the property that $\bar{p} \preceq \overline{\mathsf{gts}}$. In many trust structures it is the case that if $\bar{p}$ is sufficient to authorize a given request, so is the actual fixed-point.

Finally, the inherently dynamic nature of the envisioned systems requires algorithms that explicitly deal with the dynamic updating of trust policies (rather than implicitly dealing with updates by doing a *complete* re-computation of the trust-state). In Section 4 we address the problem of dynamic policy-changes. We provide algorithms that reuse information from "old" computations, when computing the "new" fixed-point values. For specific (but commonly occurring) types of updates this is very efficient. For fully general updates we have an algorithm which is better than the naive algorithm in many cases.

Future and related work is discussed in the concluding section.

2 Computation of Local Least-Fixed-Points

In this section, we show how to compute the *local* fixed-point value $\overline{\mathsf{gts}}(R)(q)$ for two fixed principals R and q, without computing the complete global trust-state $\overline{\mathsf{gts}}$. The reason for computing local values is twofold. First, we can benefit from distributing the computational- and storage-burdens, so that instead of centrally computing the complete state $\overline{\mathsf{gts}}$, node will R maintain "entry" $\overline{\mathsf{gts}}(R)(q)$ in the "distributed matrix" $\overline{\mathsf{gts}}$. Second, although the semantics of trust policies are functions of type $(\mathcal{P} \to \mathcal{P} \to X) \to \mathcal{P} \to X$ 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 R and q, the set of principals that R's policy *actually* depends on in its entry for q, is often a significantly smaller subset of $\mathcal{P}$. For example, consider our policy from the previous section.

$$\pi_R(\mathsf{gts}) = \lambda q \in \mathcal{P}.(\mathsf{gts}(A)(q) \vee_{\preceq} \mathsf{gts}(B)(q)) \wedge_{\preceq} \mathsf{medium}$$

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

We first compute (distributedly) a dependency graph which contains *only the dependencies relevant for the computation of* $\overline{\mathsf{gts}}(R)(q)$, thus excluding a (hopefully) large set of principals that do not need to be involved in computation. We then proceed with computation of $\overline{\mathsf{gts}}(R)(q)$ by showing that the conditions of a general algorithmic convergence-theorem of Bertsekas [1] are satisfied, and hence we can appeal to previous results on the convergence of a certain totally asynchronous algorithm.

We present our problem in the more abstract setting of a distributed computation of the least fixed-point of a continuous endo-function on a cpo. We show that this indeed models our practical scenario (and of course, many others).

Abstract setting. We are given a cpo $(X, \sqsubseteq)$ of finite height h, and a natural number $n \in \mathbb{N}$. Writing $[n]$ for the set $\{1, 2, \ldots, n\}$, we have also a collection $C = (f_i : i \in [n])$ of n continuous functions, each of type $f_i : X^{[n]} \to X$. These functions induce a unique, continuous, global function $F = \langle f_i : i \in [n] \rangle : X^{[n]} \to X^{[n]}$ which has a unique least-fixed-point, $\mathsf{lfp}\ F \in X^{[n]}$. Define a dependency graph $G = ([n], E)$, where $[n]$ is the set of nodes, and the edges, given as a function $E : [n] \to \mathbf{2}^{[n]}$, model (possibly an over-approximation of) the dependencies of the functions in C, i.e., have $j \notin E(i)$ implies that function f_i does *not* depend on the value of "variable" j. We consider the nodes $[n]$ as network nodes that have memory and computational power. Each node $i \in [n]$ is associated with function f_i, and we assume that each node knows all nodes that it depends on, i.e., node i knows all edges $E(i)$.

Computational problem. Let $R \in [n]$ denote a designated node, called the *root*. The computational problem is for the root to compute the *local fixed-point value* $(\mathsf{lfp}\, F)_R$.

Concrete setting. We translate the trust-structure setting into our abstract setting by defining function f_R as policy π_R's entry for principal q. One then finds the dependencies of f_R by looking at which other policies this expression depends on. If f_R depends on entry w in π_z, then z is a node in the graph, and the function f_z is given by π_z's entry for w, with the dependencies of f_z given by the dependencies in the expression for w in π_z, and so on. From now on, we shall work in the abstract setting as it simplifies notation.

Note, that this translation might lead to a node z appearing several times in the dependency graph, e.g. with entries for principals w and y in π_z. We shall think of these as distinct nodes in the graph, although a concrete implementation would have node z play the role of two nodes, z_w and z_y. Note also, that the (minimal) dependency-graph is *not* modeling any network topology. Although the nodes of the graph represent concrete nodes in a physical communication-network, its edges do not represent any communication-links.

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. The assumptions of non-failure and correct order of delivery ease the exposition, but the fixed-point algorithm we apply is highly robust [1].

Our algorithm for fixed-point computation consists of two stages. In the first stage, the dependency graph $G = ([n], E)$ is distributedly computed so that each node knows the set of nodes that depend on it for the computation. In the second stage, this information is used in an asynchronous algorithm, performing the actual fixed-point computation.

2.1 Computing Dependencies

In this sub-section, we describe how the nodes distributedly compute the dependency graph described above. Two goals are to be fulfilled by the dependency computation. First, each node must obtain a list of the nodes that depend on it for the computation. Second, we want to compute a spanning tree $T_R \leq G_R$ with root R, so that each node knows its parent and its children in this tree. We denote $T_R = ([n], S)$, $S : [n] \to \mathbf{2}^{[n]}$, with $S(i) \subseteq E(i)$ for all $i \in [n]$. Note that we are not making use of this tree until Section 3.

For any node i, we denote the set $E(i)$ by i^+, and the set of nodes k for which $i \in E(k)$ (i.e. $E^{-1}(\{i\})$), by i^-. So to summarize, after the computation, any node i knows i^+ and i^-, and it knows its parent p_i and its children $S(i)$ in a spanning tree T_R rooted at R. Node i will store i^+ and i^- in variables of the same name, and will store p_i and $S(i)$ in variables $i.p$ and $i.S$.

The distributed algorithm for the dependency computation is described by a process that runs at each node. We use syntax inspired by process calculi to describe these processes. The semantics should be clear, perhaps except for the two constructs $||_I$ and **join-then**. Let L denote a set of labels (e.g. A,B,...). The construct $||_I$, for $I \subseteq L$, describes the parallel execution of $|I|$ processes (e.g. threads), where the behaviour of process $i \in I$ is described by an expression $i : \mathit{Proc}$, where Proc is a process. The construct **join** J **then** Proc, with $J \subseteq L$, waits until each process $j \in J$ has terminated, and then executes process Proc. Note also that non-prefixed, capital, italicized letters (e.g. X, Y, M, not $i.S$) are variables that become bound at reception of a message, e.g. **receive** (`mark`) **from** X is executed as soon as the reception of a `mark` message occurs, and in the following code, X is bound to the (identity of the) sender of that message.

Non-root nodes run a process given by Figure 1. The root node runs a special process which similar to that of Figure 1, but it has no parent,

```
Process: Dependency Algorithm for non-root node $i$
    receive (mark) from $X$;
    $i^- \leftarrow \{X\}$; $i.p \leftarrow X$; $i.S \leftarrow \emptyset$;
    $||_{\{A,B,C\}}$
        A : replicate
            [ receive (mark) from $Y$;
              $i^- \leftarrow i^- \cup \{Y\}$;
              send (ack, ok) to $Y$]

        B : $||_{c \in i^+}$
                $c$ : send (mark) to $c$;
                    receive (ack, M) from $c$;
                    if (M=marker) then $i.S \leftarrow i.S \cup \{c\}$
        C : || join $i^+$ then send (ack, marker) to $X$
```

Figure 1: Dependency Algorithm - Generic node behaviour

and it will initiate the computation. One way to think of the algorithm is as a simple distributed graph-marking algorithm: the initial message that a node i receives from a node j "marks" the node i, and j is then the "marker" for i. The edges between "marker" and "marked" nodes, will constitute the spanning tree T_R. Furthermore, once a node is marked it starts a "server" sub-process (labelled `A`) which accepts `mark`-messages from any node Y, adds Y to its dependency set i^-, and acknowledges with an "`ok`" message. A sub-process running in parallel (`B`), notifies all nodes that i depends on (i.e. i^+) of this dependency, and waits for each node to acknowledge. This acknowledgment is either "`ok`" in case i is not the marker, or "`marker`" in case i *is* the marker. Finally, when an acknowledgment has been received from each child, i acknowledges its "marker". Once the root node has received acknowledgment from each of its children, the algorithm terminates.

The following statements hold. The number of messages sent is $O(|E|)$, each message of bit length $O(1)$. This follows from the observation that for each edge in the graph there flows at most two messages, one `mark` and one acknowledgment. When the root node R has received acknowledgment from all its children then every node i, which is reachable from R, stores in the variable i^-, the set i^- (by abuse of notation), stores in variable $i.S$, the children of i in T_R, and in variable $i.p$, i's parent in T_R. Note, that we only mark the nodes that are reachable from R, which amounts to excluding any node that R does not depend on (directly or by transitivity) for computing its trust value for $q \in \mathcal{P}$.

2.2 An Asynchronous Algorithm

In this section, we assume that the dependency graph has already been computed. We show that we a now in a situation in which we can apply existing work of Bertsekas for computation of the least fixed-point. Bertsekas has a class of algorithms, called totally asynchronous (TA) iterative fixed-point algorithms, and a general theorem which gives conditions ensuring that a specific TA fixed-point algorithm will converge to the desired result. In our case, "converge to" means that each principal $i \in \mathcal{P}$ will compute a sequence of values $\perp_{\sqsubseteq} = i.t_0 \sqsubseteq i.t_1 \sqsubseteq \cdots \sqsubseteq i.t_k = (\mathsf{lfp}\ F)_i$. The general theorem is called the "Asynchronous Convergence Theorem" (ACT), and we use this name to refer to Proposition 6.2.1 of Bertsekas' book [1]. 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 $\perp_{\sqsubseteq} \sqsubseteq F(\perp_{\sqsubseteq}) \sqsubseteq \cdots$ converges to the least fixed-point, which is true. 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. As a consequence of $\sqsubseteq$-monotonicity of the policies, the conditions of the Asynchronous Convergence Theorem are satisfied (the following Proposition 2.1), and so, we can deploy a TA distributed algorithm.

We now describe the algorithm and argue for its correctness. We will assume that each node i allocates variables $i.t_{cur}$ and $i.t_{old}$ of type X, which will later record the "current" value and the last computed value in X. Each node i has also an array, denoted by $i.m$. The array $i.m$ is of type X array, and will be indexed by the set i^+. Initially, $i.t_{cur} = i.t_{old} = \perp_{\sqsubseteq}$, and the array is also initialized with $\perp_{\sqsubseteq}$. For any nodes i and $j \in i^+$, when i receives a message from j (which is always a value $t \in X$), it stores this message in $i.m[j]$.

Asynchronous algorithm. 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 i repeats the following: it starts by assigning to variable $i.t_{cur}$ the result of applying its function f_i to the values in $i.m$, i.e., node i executes assignment $i.t_{cur} \leftarrow f_i(i.m)$. If there is no change in the resulting value of $f_i(i.m)$ (compared to the last value computed, which is stored in $i.t_{old}$), it will go to the *sleep* state unless a message was received since $f_i(i.m)$ was computed. Otherwise, if a new value resulted from the computation (i.e., if $t_{old} \neq f_i(i.m)$), this value is sent to all nodes in i^-. Concurrently with this we can run a termination detection algorithm, which will detect when all nodes are in the *sleep*-state and no messages are in transit. Bertsekas has already addressed this problem with his termination-detection algorithm [1], which directly applies, yielding only a constant overhead in the message complexity.

The Asynchronous Convergence Theorem. We recall the definition of the Synchronous Convergence Condition (SCC) and the Box Condition (BC) (Section 6.2 [1]). Let X be any set, and $F : X^{[n]} \to X^{[n]}$ be any function with $F = \langle f_1, f_2, \ldots, f_n \rangle$.

Definition 2.1 (SCC and BC). *Let $\{X(k)\}_{k=0}^{\infty}$ be a sequence of subsets $X(k) \subseteq X^{[n]}$ satisfying $\forall k.X(k+1) \subseteq X(k)$.*

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

$$\forall x \in X(k).F(x) \in X(k+1)$$

and furthermore, if $\{y^k\}_{k\in\omega}$ is a sequence with $y^k \in X(k)$ for all k, then every limit point of $\{y^k\}_{k\in\omega}$ is a fixed-point of F.

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

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

We state a version of the Asynchronous Convergence Theorem of Bertsekas which matches our notation. We need some preliminary terminology. Assume that before starting the asynchronous algorithm, the arrays of the nodes are initialized with a vector $\bar{x} \in X^{[n]}$, called the initial solution estimate. That is, for all nodes $i \in [n]$, and all $j \in i^+$ assume that $i.m[j] = \bar{x}_j$ and that $i.t_{old} = \bar{x}_i$.

A *limit point of the asynchronous algorithm* (with initial estimate $\bar{x}$) is a vector $\hat{x} \in X^{[n]}$ which can be written $\hat{x}_i = i.t_{cur}$, where there exists some state of the distributed system in which the algorithm has converged, and $i.t_{cur}$ is the current value of node i in this state (the algorithm has converged when all nodes are in the *sleep*-state, and no messages are in transit).

Theorem 2.1 (ACT [1]). *Assume there exists a sequence of subsets $X(k) \subseteq X^{[n]}$ with $\forall k.X(k+1) \subseteq X(k)$, satisfying the SCC and the BC. Assume that the arrays of the nodes are initialized with $\bar{x} \in X(0)$, called the initial solution estimate. Then every limit point of the asynchronous algorithm is a fixed point of F.*

Note that the ACT ensures that the algorithm converges to a fixed point of F. In our specific scenario, $X^{[n]}$ is ordered by $\sqsubseteq$, and, furthermore, we want the algorithm to converge to the $\sqsubseteq$-least fixed-point of F.

Correctness. To prove correctness of the asynchronous algorithm, one might attempt to prove that the assumption of the ACT is satisfied when all nodes initialize their trust-values ($i.m$ and $i.t_{old}$) to $\perp_{\sqsubseteq}$. We instead prove a slightly more general convergence-result which is useful when considering the interplay between the asynchronous-algorithm and policy-updates. We must also prove that any limit point of the algorithm is the *least* fixed-point of F. The following concept of an *information approximation* is central.

Definition 2.2 (Information Approximation). *Let $F : X^{[n]} \to X^{[n]}$ be continuous. Say that a value $\bar{t} \in X^{[n]}$, is an* information approximation *for F if $\bar{t} \sqsubseteq \mathsf{lfp}\, F$ and $\bar{t} \sqsubseteq F(\bar{t})$.*

The following Proposition 2.1 shows that we can indeed appeal to the ACT. In the following $(X, \sqsubseteq)$ is a finite-height cpo, and $F : X^{[n]} \to X^{[n]}$ is continuous.

Proposition 2.1 (Correctness). *Let $\bar{t}$ be any information approximation for F. Assume that the arrays of the nodes are initialized with $\bar{t}$. Then there exist a decreasing sequence $\{X(k)\}$ of subsets of $X^{[n]}$ with $\bar{t} \in X(0)$, satisfying the synchronous convergence condition and the box condition. Furthermore, any limit point of the asynchronous algorithm with initial estimate $\bar{t}$ is $\mathsf{lfp}\, F$, and hence, limit points are unique.*

Proof. Define a sequence of subsets of $X^{[n]}$, $X(0) \supseteq X(1) \supseteq \cdots \supseteq X(k) \supseteq X(k+1) \supseteq \cdots$ by

$$X(k) = \{m \in X^{[n]} \mid F^k(\bar{t}\,) \sqsubseteq m \sqsubseteq \mathsf{lfp}\ F\}$$

Note that $X(k{+}1) \subseteq X(k)$ follows from the fact that $F^k(\bar{t}\,) \sqsubseteq F^{k+1}(\bar{t}\,)$ for any $k \in \mathbb{N}$, which, in turn, holds since $\bar{t}$ is an information approximation. For the synchronous convergence condition, assume that $m \in X(k)$ for some $k \in \mathbb{N}$. Since $F^k(\bar{t}) \sqsubseteq m \sqsubseteq \mathsf{lfp}\ F$, we get by monotonicity $F^{k+1}(\bar{t}\,) \sqsubseteq F(m) \sqsubseteq F(\mathsf{lfp}\ F) = \mathsf{lfp}\ F$. Let $(y_k)_{k\in\omega}$ be so that $y_k \in X(k)$ for every k.

Since $\bar{t}$ is an information approximation, we have $\bigsqcup_i F^i(\bar{t}\,) = \mathsf{lfp}\,F$. Since X is of finite height there exists $k_h \in \mathbb{N}$ so that for all $k' \geq k_h$, $X(k') = \{\mathsf{lfp}\,F\}$. Thus any such $(y_k)_{k\in\omega}$ converges to $\mathsf{lfp}\,F$. The box condition is also easy:

$$X(k) = \prod_{i=1}^{n} \{m(i) \in X \mid m \in X^{[n]} \text{ and } F^k(\bar{t}\,) \sqsubseteq m \sqsubseteq \mathsf{lfp}\,F\ \}$$

□

To prove correctness of our algorithm, we simply invoke Proposition 2.1 in the case of the trivial information-approximation $\bar{t} = \bot^n_{\sqsubseteq}$. The asynchronous convergence theorem ensures that the asynchronous algorithm converges towards the right values at all nodes, and, because of our assumption of finite height cpos, the distributed system will eventually reach a state which is stable. In this state, each node i will have computed $(\mathsf{lfp}\,F)_i$.

Remarks. Since any node sends values only when a change occurs, by monotonicity of f_i, node i will send at most $h \cdot |i^-|$ messages, each of size $O(\log|X|)$ bits.[5] Node i will receive at most $h \cdot |i^+|$ messages, each message (possibly) triggering a computation of f_i. Globally, the number of messages is $O(h \cdot |E|)$ each of bit size $O(\log|X|)$. Hence, the communication complexity of our algorithm is linear in the height of the lattice used by the policies. An important global invariant in this algorithm is that any value computed locally at a node (by the assignment $i.t_{cur} \leftarrow f_i(i.m)$) is a component in an information approximation for F. That is, it holds everywhere, at any time, that (1) $i.t_{cur} \sqsubseteq (\mathsf{lfp}\,F)_i$ and

[5] In fact, there will be *only* $O(h)$ *different* messages, each sent to all of i^-. Consequently, a broadcast mechanism could implement the message delivery efficiently.

(2) $i.t_{cur} \sqsubseteq f_i(i.m)$. To see this, note that $(1, 2)$ hold initially, and that both properties are preserved by the update $i.t_{cur} \leftarrow f_i(i.m)$ whenever $i.m[y] \sqsubseteq (\mathsf{lfp}\, F)_y$ for all $y \in i^+$ (which is always true). We state this fact as a lemma, as it becomes very useful in the next section where we consider fixed-point approximation-algorithms.

Lemma 2.1. *Any value $i.t_{cur} \in X$ computed by any node $i \in [n]$, at any time in the algorithm by the statement $i.t_{cur} \leftarrow f_i(i.m)$, is a part of an information approximation for F, in the sense that $i.t_{cur} \sqsubseteq (\mathsf{lfp}\, F)_i$ and $i.t_{old} \sqsubseteq i.t_{cur}$.*

2.3 An example computation

In this subsection, we give a small example of a run of the asynchronous algorithm. Let us consider an example with 5 principals, named $\mathsf{R}, \mathsf{A}, \mathsf{B}, \mathsf{C}$ 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) in the example.

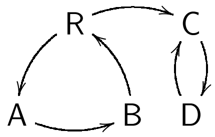
Policies. The policies of the principals have the following entries for S.

$$\begin{aligned}
\pi_{\mathsf{R}} &= (\ulcorner \mathsf{A} \urcorner(S) \vee \ulcorner \mathsf{C} \urcorner(S)) \sqcup \mathtt{Loc}(S) \\
\pi_{\mathsf{A}} &= \ulcorner \mathsf{B} \urcorner(S) \sqcup \mathtt{Loc}(S) \\
\pi_{\mathsf{B}} &= \ulcorner \mathsf{R} \urcorner(S) \sqcup \mathtt{Loc}(S) \\
\pi_{\mathsf{C}} &= \ulcorner \mathsf{D} \urcorner(S) \sqcup \mathtt{Loc}(S) \\
\pi_{\mathsf{D}} &= \ulcorner \mathsf{C} \urcorner(S) \sqcup \mathtt{Loc}(S)
\end{aligned}$$

The construct $\ulcorner \cdot \urcorner$ is policy-reference (as in Section 3.1), $\vee$ is $\preceq$-join and $\sqcup$ is $\sqsubseteq$-join. The construct $\mathtt{Loc}(S)$ is a special construct for the T_{MN} trust structure; it refers to the trust-value derived from the local observations made by a principal about the subject in question (this construct is discussed also by Nielsen and Krukow [20]). For example, R's policy for the subject is to take the $\preceq$-join in T_{MN} of the values that 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.

It is not hard to see that both $(T_{MN}, \preceq)$ and $(T_{MN}, \sqsubseteq)$ are lattices, and that the joins are given by the following formulas; for any $(m, n), (m', n') \in$

Figure 2: Example dependency-graph.



T_{MN} we have:

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

The dependency graph derived from the policies is given in Figure 2.

Local data. We assume that the principals have the following local data, representing observations made about the subject.

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

E.g., A has recorded one 'good', but five 'bad' observations about subject S.

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., $\bot_\sqsubseteq, \Pi_\lambda(\bot_\sqsubseteq), \Pi_\lambda(\Pi_\lambda(\bot_\sqsubseteq)), \ldots$). 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:"

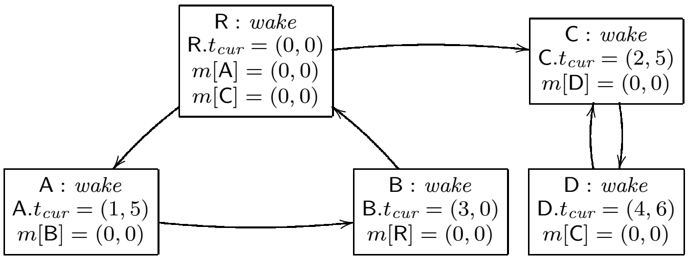
$$\ulcorner \mathsf{B} \urcorner(S)(\text{"row 1"}) \sqcup \mathtt{Loc}(S) = (3,0) \sqcup (1,5) = (3,5)$$

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).

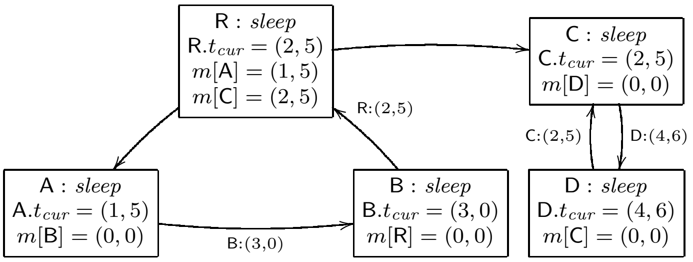
iteration	R	A	B	C	D
0	$\bot_\sqsubseteq$	$\bot_\sqsubseteq$	$\bot_\sqsubseteq$	$\bot_\sqsubseteq$	$\bot_\sqsubseteq$
1	$(0,0)$	$(1,5)$	$(3,0)$	$(2,5)$	$(4,6)$
2	$(2,5)$	$(3,5)$	$(3,0)$	$(4,6)$	$(4,6)$
3	$(4,5)$	$(3,5)$	$(3,5)$	$(4,6)$	$(4,6)$
4	$(4,5)$	$(3,5)$	$(4,5)$	$(4,6)$	$(4,6)$
5	$(4,5)$	$(4,5)$	$(4,5)$	$(4,6)$	$(4,6)$

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 $i.m$, and the values of $i.t_{cur}$. 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.

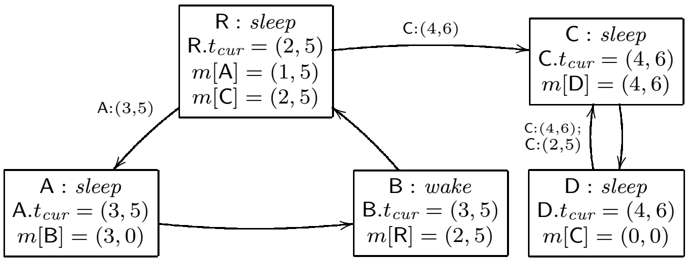
Network Snapshot 1. We assume that the initial states of the nodes are given by the following. All nodes are *wake*, the arrays $(i.m)$ are initialized to $\bot_{\sqsubseteq} = (0, 0)$. Each node has $i.t_{cur} = \pi_i(i.m)$.



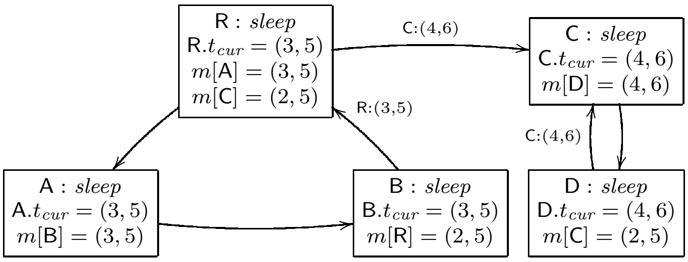
Network Snapshot 2. Here R has received value $(1, 5)$ from A and $(2, 5)$ from C. Further values are in transit, e.g. value R : $(2, 5)$ "on" edge B $\rightarrow$ R represents a message in transit from R to B.



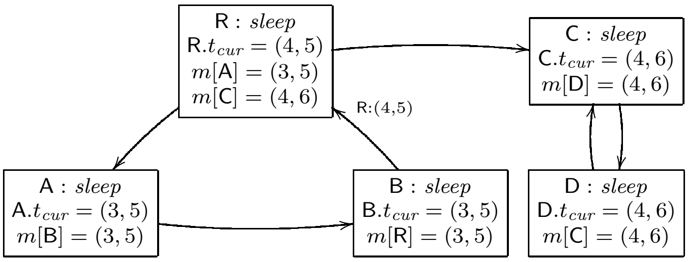
Network Snapshot 3. Two messages are in transit on the (presumably slow) path from C to D (we are assuming a reliable network, so the first sent will also arrive first). B has just finished computing $\pi_{\mathsf{B}}(\mathsf{B}.m) = (3, 5)$, but has not yet sent this value.



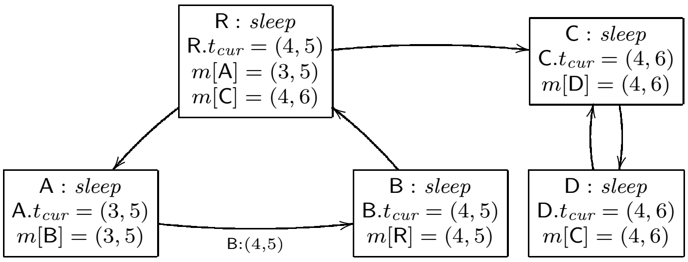
Network Snapshot 4.



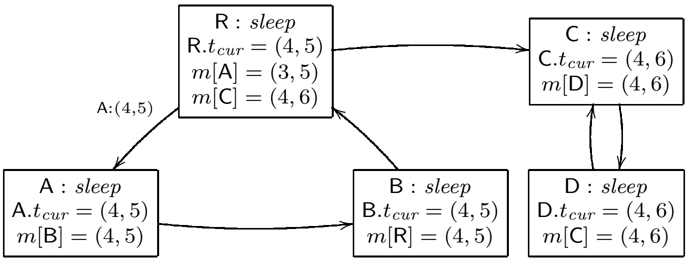
Network Snapshot 5. Notice that the component consisting of C and D has converged. No more messages are exchanged between them for the remainder of the algorithm; this is in contrast to the globally synchronous iteration.



Network Snapshot 6.



Network Snapshot 7. When R receives the final value from A, the algorithm has converged.



3 Approximation techniques

In this section, we present two techniques for safe and efficient approximation of the fixed-point. Consider a situation in which a client principal p wants to access a resource controlled by server v. Assume that the access-control policy of v is that, to allow access, its trust in p should be trust-wise above some threshold $t_0 \in X$, i.e., the fixed-point should satisfy $t_0 \preceq (\mathsf{lfp}\,\Pi_\lambda)(v)(p)$. The goal of the approximation techniques is to allow the server to (soundly) make its security decision without having to actually compute the exact fixed-point value. Instead, the server is able to efficiently compute an approximating global trust-state $\bar{p} : \mathcal{P} \to \mathcal{P} \to X$ which is related to the fixed point in such a way that the desired property can be asserted.

We need some preliminary terminology. Let $T = (X, \preceq, \sqsubseteq)$ be a trust structure, i.e. $(X, \sqsubseteq)$ is a cpo with bottom $\perp_\sqsubseteq$ and $(X, \preceq)$ is a partial order (not necessarily complete). We assume also that $(X, \preceq)$ has a least element, denoted $\perp_\preceq$. If for any countable $\sqsubseteq$-chain $C = \{x_i \in X \mid i \in \mathbb{N}\}$ and any $x \in X$ we have (i) $x \preceq C$ implies $x \preceq \bigsqcup C$ and (ii) $C \preceq x$ implies $\bigsqcup C \preceq x$, then $\preceq$ can be said to be $\sqsubseteq$-*continuous.*

3.1 Bounding "Bad Behaviour"

This first technique lets a client convince a server that its trust in the client is (trust-wise) above a certain level. The technique is based on the following proposition.

Proposition 3.1. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{p} \in X^{[n]}$, and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. If we have $\bar{p} \preceq (\lambda k \in [n].\perp_\sqsubseteq)$ and $\bar{p} \preceq F(\bar{p})$, then $\bar{p} \preceq \mathsf{lfp}_\sqsubseteq F$.*

Proof. We have $\bar{p} \preceq \lambda k.\perp_\sqsubseteq$ which implies $F(\bar{p}) \preceq F(\lambda k.\perp_\sqsubseteq)$ by $\preceq$-monotonicity. Since $\bar{p} \preceq F(\bar{p})$, transitivity implies that $\bar{p} \preceq F(\bar{p}) \preceq F(\lambda k.\perp_\sqsubseteq)$. So again by $\preceq$-monotonicity of F and transitivity

$$\bar{p} \preceq F(\bar{p}) \preceq F^2(\bar{p}) \preceq F^2(\lambda k.\perp_\sqsubseteq)$$

Now since for all $i \geq 0$ we have $\bar{p} \preceq F^i(\lambda k.\perp_\sqsubseteq)$, the fact that $\preceq$ is $\sqsubseteq$-continuous implies that

$$\bar{p} \preceq \bigsqcup_i F^i(\lambda k.\perp_\sqsubseteq) = \mathtt{lfp}_\sqsubseteq F$$

□

Note that the conclusion of the proposition is an assertion that is useful for authorization; if the server knows a $\bar{p} \in X^{[n]}$ which is sufficient to allow an authorization, and knows also that $\bar{p} \preceq \mathsf{lfp}_{\sqsubseteq} F$, then since the ideal global trust-state is *trust-wise* above $\bar{p}$, then it is a sound decision to allow the authorization. This idea is the basis of an efficient protocol for a kind of "proof-carrying" authorization.

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 $m + n$ past interactions; m of which where classified 'good', and n, classified as 'bad'.[6] 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 written in the language of Carbone *et al.* [7] as

$$\pi_v \equiv \lambda x : \mathcal{P}.(\ulcorner a \urcorner(x) \wedge \ulcorner b \urcorner(x)) \vee \bigwedge_{s \in S \setminus \{a,b\}} \ulcorner s \urcorner(x)$$

The construct $\ulcorner \cdot \urcorner$ represents *policy reference* or *delegation*, e.g., if a and x are principal identities then expression $\ulcorner a \urcorner(x)$ "evaluates" to the value that a's trust policy specifies for x. 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").[7] 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.

[6] To be precise, the set $\mathbb{N}^2$ is completed by allowing also value ∞ as "m" or "n" or both.

[7] The example policy assumes that $(X, \preceq)$ is a lattice, meaning that for any $x, y \in X$ both $x \vee y$ and $x \wedge y$ exist. Furthermore operations $\vee$ and $\wedge$ must be continuous *also with respect to the information ordering*. In many trust-structure this is often the case [7].

Protocol. Principal p sends to v the "trust-state"

$$t = [(v,p) \mapsto (0,N), (a,p) \mapsto (0,N_a), (b,p) \mapsto (0,N_b)]$$

which can be thought of as a "proof" (analogous to a 'proof-of-compliance') or a "claim" made by p, stating that $(0,N) \preceq (\mathsf{lfp}\,\Pi_\lambda)(v)(p)$ (and similarly for a and b). Upon reception, v first extends t to a global trust state, which is the extension of t to a function $\bar{p}$ of type $\mathcal{P} \to \mathcal{P} \to T_{MN}$, given by

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

To check the proof, principal v must verify that $\bar{p}$ satisfies the conditions of Proposition 3.1. First, v must check that $\bar{p}(x)(y) \preceq \bot_{\sqsubseteq} = (0,0)$ for all x, y. But this holds trivially if $y \neq p$ or $x \neq v, a, b$ because then $\bar{p}(x)(y) = (0,\infty) = \bot_{\preceq}$. For the other few entries it is simply an order-theoretic comparison $\bar{p}(x)(y) \preceq (0,0)$. Now v tries to verify that $\bar{p} \preceq \Pi_\lambda(\bar{p})$. To do this, v verifies that $(0,N) \preceq \pi_v(\bar{p})(p)$. If this holds then v sends the value t to a and b, and ask a and b to perform a similar verification (e.g. $(0,N_a) \preceq \pi_a(\bar{p})(p)$). 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 $\bar{p} \preceq \Pi(\bar{p})$: by the checks made by v, a and b, we have that $\bar{p}(x)(y) \preceq \Pi_\lambda(\bar{p})(x)(y)$ holds for pairs $(x,y) = (v,p), (a,p), (b,p)$, but for all other pairs it holds trivially since $\bar{p}$ is the $\preceq$-bottom on these. By Proposition 3.1, we have $\bar{p} \preceq \mathsf{lfp}\,\Pi_\lambda$, and so, v is *ensured* that its trust value for p is $\preceq$-greater than $(0,N)$.

We have illustrated the main idea of the protocol by way of an example, but the general technique for verifying a proof should be clear. In general, the proof $\bar{p}$ may include a larger number of principals, which would then have to be involved in the verification process.

Remarks. Our approximation protocol has very much the flavour of a proof-carrying authorization: the requester (or prover) must provide a proof that its request should be granted. It is then the job of the service-provider (or verifier) to check that the proof is correct. The strength of this protocol lies in replacing an entire fixed-point computation with a few local checks made by the verifier, together with a few checks made by a subset of the principals that the verifier depends on. An interesting property of this protocol is that part of the information that the prover needs to supply should already be known to the prover; it should already know who it has performed well with in the past (e.g. in our example above, p could know the bounds N_a and N_b because of its previous interaction with a and b). There are, however, two important restrictions to this approach. First, as in the example, in order to construct its proof, the prover needs information about the verifiers trust policy and of the policies of those whom the verifier depends on. If policies are secret, it is not clear how the verifier would construct this proof. Second, because of the requirement in Proposition 3.1 that $\bar{p} \preceq \bot_{\sqsubseteq}$, the protocol can usually only be used to prove properties stating "not too much bad behaviour," and *not* properties guaranteeing sufficiently "good" behaviour.

Notice that the protocol for exploiting Proposition 3.1 has a message complexity which is independent of the height of the cpo; in particular, it works also for infinite height cpos. In contrast, the algorithm for computing fixed-points has message complexity $O(h \cdot |E|)$.

We present now another approach which requires more computation and communication, but does not have the two mentioned restrictions.

3.2 Exploiting Information Approximations

The approximation technique developed in this section is different from that of the "proof-carrying" protocol in the previous section. In this section, we not require the "prover" (client) to provide any information. Instead, we derive an approximation from a "snapshot" of the state of the asynchronous fixed-point algorithm from Section 2.2. The "verifiers" (servers) are then able make a collection of local checks on this snapshot, allowing them to infer that the fixed-point value must be trust-wise above the snapshot-value. The technique is based on the following proposition.

Proposition 3.2. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{t} \in X^{[n]}$, and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that $\bar{t}$ is an information approximation for F. If $\bar{t} \preceq F(\bar{t})$ then $\bar{t} \preceq \mathsf{lfp}\, F$.*

Proof. Since $\bar{t}$ is an information approximation for F, we have by easy induction that for all $k \in \mathbb{N}$, $F^k(\bar{t}) \sqsubseteq F^{k+1}(\bar{t}) \sqsubseteq \mathsf{lfp}\ F$, and so by continuity of F, $\bigsqcup_{k\in\mathbb{N}} F^k(\bar{t}) = \mathsf{lfp}\ F$. Since $\bar{t} \preceq F(\bar{t})$, an easy induction gives $\bar{t} \preceq F^k(\bar{t})$ for all k. Then the information continuity of $\preceq$ implies that $\bar{t} \preceq \mathsf{lfp}\ F$. □

This proposition is very useful because, by Lemma 2.1, a global invariant in the asynchronous fixed-point algorithm is that all values computed are information approximations for F. This means that we can combine the algorithm with a protocol that, intuitively, implements the check for the condition $\bar{t} \preceq F(\bar{t})$ in the above proposition.

Imagine that during the execution of the asynchronous algorithm, there is a point in time, in which no messages are in transit, all nodes i have computed their function f_i, and sent the value $f_i(i.m)$ to all that depend on it. Thus we have a "consistent" state in the sense that for any node x and any node $y \in x^+$ we have $x.m[y] = y.t_{cur}$. In particular if x and z both depend on y, then they agree on y's value: $x.m[y] = y.t_{cur} = z.m[y]$. In this ideal state, there is a consistent vector $\bar{t}$ which, by Lemma 2.1, is an information approximation for F, i.e. $\bar{t}$ contains the values $\bar{t}_i = i.t_{cur}$ for nodes $i \in [n]$. If the state of the distributed system was frozen at this point, and all nodes x, simultaneously make the check $x.t_{cur} \preceq f_x(x.m)$, then vector $\bar{t}$ satisfies $\bar{t} \preceq F(\bar{t})$. Since $\bar{t}$ is an information approximation for F, by Proposition 3.2, the root node R knows that $\bar{t}_R \preceq \mathsf{lfp}\ F_R$, which is what we want.

Of course, the ideal situation described above would rarely occur in a real execution. The aim of the approximation technique in this section, is to enforce, during execution of the asynchronous algorithm, a consistent view of such an ideal situation. In so-called snapshot-algorithms (see Bertsekas [1]), 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.

We describe now a distributed algorithm implementing this. We assume that the asynchronous algorithm is running, and at some point the root node decides to run the approximation check (e.g. because it has computed a (non fixed-point) value $R.t_{cur}$ which is sufficient to allow access). We assume that each node $i \in \mathcal{P}$ has additional variables $i.t_{app} : X$ and $i.m_{app} : X$ array, indexed by i^+. The array will eventually store only consistent values. The algorithm, as usual, consists of a special process run by the root, and another similar process running at non-root nodes, given by Fig. 3.

Recall, that the dependency-graph algorithm has generated also a spanning tree T_R, rooted at R. The root initiates the approximation algorithm. It starts by sending an init message to each of it's children

```
Process: non-root nodes $i$
$||_{\{A,B,C\}}$
    A : receive (init);
        $||_{\{A1,A2\}}$
            A1: $||_{c \in i.S}\ c$ : send (init) to $c$;
            A2: [ //$w$ait until consistent state];
                $i.t_{app} \leftarrow i.t_{cur}$;
                $||_{j \in i^-}\ j$ : send (copy) to $j$;

    B : $||_{\{B1,B2\}}$
            B1: $||_{k \in i^+}$
                    $k$ : receive (copy) from $k$;
                        $i.m_{app}[k] \leftarrow i.m[k]$;
            B2: join {B1, A2} then
                $i.b$ : bool $\leftarrow (i.t_{app} \preceq f_i(i.m_{app}))$;
    C : $||_{\{C1,C2\}}$
            C1: $||_{c \in i.S}$
                    $c$ : receive ($i.b_c$ : bool) from $c$;
            C2: join {C1, B2} then
                send $(i.b \wedge (\bigwedge_{c \in i.S} i.b_c))$ to $i.p$;
```

Figure 3: Snapshot Algorithm - Generic node behaviour

listed in $R.S$ (Fig. 3, label A1). Now it waits until it is in a *locally consistent* state (A2), which means that, in the asynchronous algorithm, it has just computed $R.t_{cur} \leftarrow f_R(R.m)$, and (if necessary) has sent that value to each of R^-. Once in such a state, R saves the value by doing $R.t_{app} \leftarrow R.t_{cur}$ – this value will become the value for R in the consistent vector we are seeking. R now sends a copy message to each node in R^- (A2). A node $y \in R^-$ which receives a copy message from R will copy the last value received from R into its approximation array, i.e. $y.m_{app}[R] \leftarrow y.m[R]$ (B1). Since we are assuming a reliable network, the copied value is $R.t_{app}$, and so we are propagating consistent values. Root R now waits until each node $z \in R^+$ has sent a copy message, and computes then $R.t_{app} \preceq f_R(R.m_{app})$ (B2). Finally, the root waits for all children in the spanning tree to have replied with a boolean, and if all of these are true and the check succeeded (C1, C2), then the root is ensured that $R.t_{app} \preceq (\mathsf{lfp}\ F)_R$. Non-root nodes i, once initiated, do almost the same. The only difference is that after the check has been made, and all children in the spanning tree have replied with a boolean, i sends value true to its parent $i.p$ only if all $i.S$ sent true and i's own check succeeded.

Since there is a constant number of messages sent for each edge in G_R, the message complexity of the snapshot algorithm is $O(|E|)$ messages, each of size $O(1)$ bits.

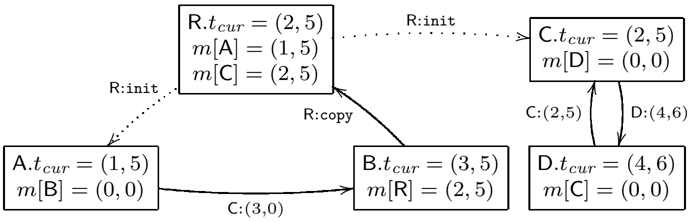
A useful property of this algorithm is that it can be run concurrently with the asynchronous fixed-point algorithm - there is no reason to stop! One may simply allocate a thread implementing the approximation-check, which runs concurrently with the asynchronous fixed-point algorithm.

Note that, the style of this protocol is different than that of the previous section. In the previous protocol the client presents a "proof" t which the servers then verifies. It is not clear how one could use Proposition 3.2 in this style. In particular, if a client presented a "proof" t, then it is not clear how the servers would check that $t \sqsubseteq \mathsf{lfp}\ F$ without already knowing $\mathsf{lfp}\ F$.

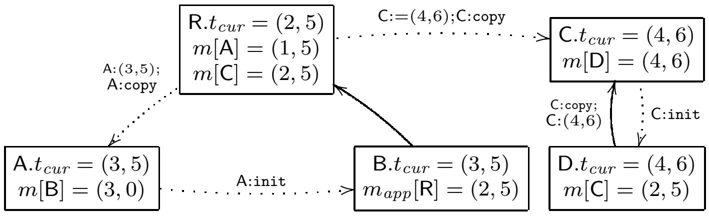
3.3 An example run

We illustrate the snapshot-algorithm on the policies from Section 2.3. Again we illustrate the algorithm by a sequence of network-states. We assume that the asynchronous algorithm has been running for some time (as in Section 2.3). Let us assume that R now wants to run the snapshot algorithm with respect to its current value, $(2, 5)$.

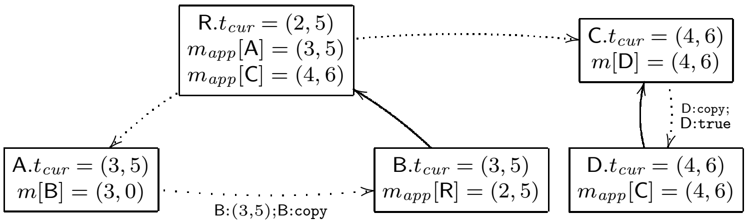
Network snapshot 1. R initiates the algorithm by sending `init` messages to A and C. Concurrently it sends a `copy` message to B to indicate that the last message received by B from R should be used for approximation. (Note the it is not a problem that B receives the `copy` message before it is initiated).



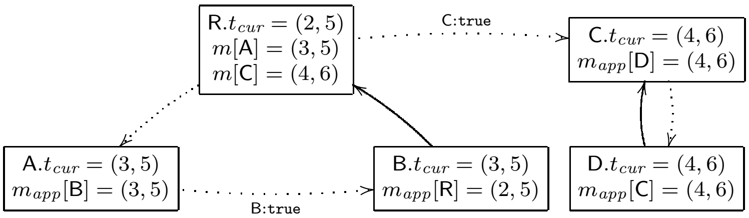
Network snapshot 2. Here C has received message D : $(4,6)$. It then receives the R : `init` message. C proceeds by sending its current value to R and D, followed by `copy`-messages, and an `init`-message (only) to D. A behaves similarly.



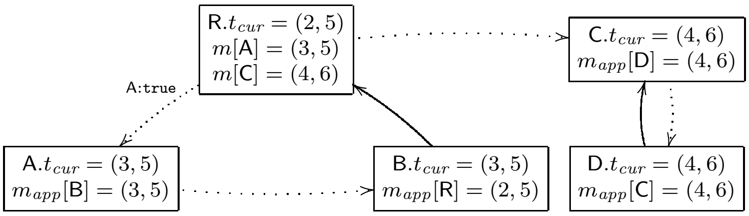
Network snapshot 3. Here D has asserted that $\mathsf{D}.t_{cur} \preceq \pi_{\mathsf{D}}(\mathsf{D}.m_{app})$ is true. Since it has no children in the spanning tree, it immediately sends value `true` to its (spanning-) parent C (illustrated by the dotted edges).



Network snapshot 4. B has made its assertion, and similarly, once C receives `true` from D, it makes assertion $\mathsf{C}.t_{cur} \preceq \pi_{\mathsf{C}}(\mathsf{C}.m_{app})$, and sends `true` to R.



Network snapshot 5. Finally, A makes its assertion, and once R receives value `true`, R knows that $(2,5) \preceq \mathsf{lfp}(\Pi_\lambda)(\mathsf{R})(\mathsf{S})$ (recall that S is the subject of the trust-computation).


$\mathsf{R}.t_{cur} = (2,5)$
$m[\mathsf{A}] = (3,5)$
$m[\mathsf{C}] = (4,6)$
$\mathsf{C}.t_{cur} = (4,6)$
$m_{app}[\mathsf{D}] = (4,6)$
A:true
$\mathsf{A}.t_{cur} = (3,5)$
$m_{app}[\mathsf{B}] = (3,5)$
$\mathsf{B}.t_{cur} = (3,5)$
$m_{app}[\mathsf{R}] = (2,5)$
$\mathsf{D}.t_{cur} = (4,6)$
$m_{app}[\mathsf{C}] = (4,6)$

3.4 Dual Propositions and Generalization

Note that both the propositions in this section have “dual” versions.

Proposition 3.3. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{p} \in X^{[n]}$, and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. If $\bot_\sqsubseteq \preceq \bar{p}$ and $F(\bar{p}\,) \preceq \bar{p}$ then $\mathsf{lfp}\, F \preceq \bar{p}$.*

The dual of Proposition 3.2 is the following.

Proposition 3.4. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{t} \in X^{[n]}$, and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that $\bar{t}$ is an information approximation for F. If $F(\bar{t}\,) \preceq \bar{t}$ then $\mathsf{lfp}\, F \preceq \bar{t}$.*

We can deploy similar algorithms for the duals. At first sight Proposition 3.3 does not seem as useful as its dual. The conclusion $\mathsf{lfp}\, F \preceq \bar{p}$ can usually only be used to *deny* a request, and a prover in the protocol for Proposition 3.3 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. [5, 20]), and that $\bar{p} \preceq \bar{p}'$ expresses (informally) that, when interacting with a certain principal, the probability of a specific outcome given $\bar{p}$, is lower than the probability of that outcome given $\bar{p}'$. In this case, an assertion of the form $\mathsf{lfp}\, F \preceq \bar{p}$, can convince the verifier that when interacting with the prover, the probability of a “bad” outcome is *below* a certain threshold.

Essentially, we can use the same algorithm as that of Section 3.2 for exploiting Proposition 3.4. Servers can incorporate the check $F(\bar{t}\,) \preceq \bar{t}$ together with the dual check $\bar{t} \preceq F(\bar{t}\,)$.

Interestingly, it turns out that the two propositions of this section are actually instances of a more general theorem, which gives rise to a generalized approximation-protocol, that can be seen as a combination of the two techniques presented in this section.

Proposition 3.5. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{p} \in X^{[n]}$, and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that $\bar{p}$ satisfies $\bar{p} \preceq F(\bar{p})$. If there exists an information approximation $\bar{t} \in X^{[n]}$ for F, with property that $\bar{p} \preceq \bar{t}$, then $\bar{p} \preceq \mathsf{lfp}\, F$.*

Proof. The proof of Proposition 3.5 is similar to that of Proposition 3.1. We use the diagram:

$$\begin{array}{ccccccccc}
\bar{p} & \preceq & F(\bar{p}) & \preceq & \cdots & \preceq & F^i(\bar{p}) & \preceq & \cdots \\
\rotatebox{-90}{$\preceq$} & & \rotatebox{-90}{$\preceq$} & & & & \rotatebox{-90}{$\preceq$} & & \cdots \\
\bar{t} & \sqsubseteq & F(\bar{t}) & \sqsubseteq & \cdots & \sqsubseteq & F^i(\bar{t}) & \sqsubseteq & \cdots
\end{array}$$

By continuity of $\preceq$ we have $\bar{p} \preceq \bigsqcup_i F^i(\bar{t})$. □

Note that one obtains Proposition 3.1 with the trivial information approximation $\bar{t} = \bot_{\sqsubseteq}$, and Proposition 3.2 by taking the proof to be the approximation, i.e. $\bar{p} = \bar{t}$.

In fact, this proposition can be used for a protocol, which can be seen as a merger of the ideas of proof-carrying-authorization and the snapshot protocol. The prover p sends a proof $\bar{p}$ to the verifier v. The condition $\bar{p} \preceq \Pi(\bar{p})$ can be checked in the same manner as in the proof-carrying authorization protocol. Now v needs to assert the existence of an information approximation $\bar{t}$, with $\bar{p} \preceq \bar{t}$. This can be done by running the asynchronous algorithm until property $\bar{t}_i \preceq \bar{p}_i$ is satisfied locally at each node i, which can be checked in a manner similar to that in the snapshot algorithm for Proposition 3.2. This protocol is analogous to that in Section 3.1 without the restriction $\bar{p} \preceq \bot_{\sqsubseteq}$, but requiring more work for the verifiers to check the proof (in the worst case they must compute their local fixed-point-values).

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.* [7] 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 $\bot_{\preceq}$). Several natural examples of non-interval domains can also be seen to have the required properties [15]. The requirement that all policies π_p are monotonic also with respect to $\preceq$ is not unrealistic. Intuitively, it amounts to saying that if everyone raises their trust-levels in everyone, then policies should not assign lower trust levels to anyone.

4 Dynamics

In this section, we briefly outline what can be done in the case of some function f_i changing to f'_i (denoted $f_i \mapsto f'_i$). Denote by $F' = F[f'_i/i] = \langle f_1, f_2, \ldots, f_{i-1}, f'_i, f_{i+1}, \ldots f_n\rangle$ the updated 'global function'. Suppose that the fixed-point computation has terminated, i.e. each node i knows its value $i.t_{cur}$ along with values $i.m[j]$ for $j \in i^+$, so that for all i, $i.t_{cur} = (\mathsf{lfp}\ F)_i$. Suppose now that some node i makes a policy update, i.e. changes its function f_i to f'_i, e.g. due to new information being available. One could now let node i broadcast a "reset"-message to all nodes, and computation, including dependency graph discovery, could restart. However, in this approach there is a lot of information which is unnecessarily discarded. In many systems it is likely that observing interactions between principals will cause information-increasing changes in policies [15,20]. An update $F \mapsto F'$ is information increasing if for all $\bar{t} \in X^{[n]}, F(\bar{t}) \sqsubseteq F'(\bar{t})$. This constitutes a very important restricted class of updates which can be resolved very efficiently. It is easy to see that, in the case of information increasing updates, the current values $(i.t_{cur})_{i\in[n]}$ are an information approximation also to F'. This means that one can invoke Prop. 2.1, and so the asynchronous fixed-point algorithm can continue with the old values.

Clearly, also more general types of updates will occur, but we conjecture that in many systems they will be less frequent, i.e. policy changes are rare whereas obtaining new information about behaviour is not, but both trigger a change in the trust-policy function. In the case of general updates, there are two issues which can largely be considered independently.

Firstly, an update $f_i \mapsto f'_i$ may or may not change the structure of the dependency graph. It may add new edges and/or delete edges. This can easily be dealt with, essentially by running the dependency-graph algorithm from Section 2.1, but with the updating node i running in the role of the *root* node. The only interesting difference is the case where i initiates a new node j, which has an edge to a node k, which was already in the old dependency graph. In this case, there is no reason for j to assume $\perp_{\sqsubseteq}$ for k, instead, k acknowledges and sends its current value, $k.t_{cur}$.

The second issue is the fact that when f_i changes to f_i', in general, any node j that has a path to i in the dependency graph will no longer have a valid estimate of the (updated) trust values in variables $j.t_{cur}$ and $j.m$. Our idea is to ensure that any such node j, takes on an information approximation for the *updated function* F' for all entries in its array $j.m[k]$ where $k \in [n]$ has a path to i. If this can be ensured, we can invoke our Proposition 2.1 with respect to the updated function F', which ensures that the asynchronous algorithm, when run with the updated functions F', will converge to $\mathsf{lfp}\ F'$. An important simple observation is that if $j \in [n]$ does *not* have a path to i, then j is unaffected by the update. This means that *any* node $k \in [n]$, does not need to reset its entries $k.m[j]$ for such unaffected nodes j.

4.1 Edge-adding Updates

Consider an update, $f_i \mapsto f_i'$, where the dependencies of f_i are contained in the dependencies of f_i', i.e. node i adds additional edges to the dependency graph, but deletes none. Clearly, one must extend the current dependency graph to a larger graph in order to proceed with computation. Furthermore, since function f_i has changed, any node j that depends on i, either directly or indirectly, will have inconsistent values in their arrays $j.m$. The idea in our algorithm is for those nodes (and only those nodes!) to take on a safe approximation to the updated function. Since the update can be arbitrary, we choose value $\perp_{\sqsubseteq}$ as our approximation, to ensure that this value is, in fact, an information approximation. Once all these nodes have taken a safe approximation, we can invoke our Proposition 2.1, to ensure that the Asynchronous Convergence Theorem is satisfied, and computation can proceed in the updated graph.

Concretely, node i will now run two algorithms concurrently. Firstly, i will run, in the role of a root node, a generalized version of the dependency-graph algorithm from Section 2. More specifically, it is the same algorithm, except that we allow generalized acknowledgment-messages which carry values from X. The reason is that i might initiate a new node j, which has an edge to a node k, which was already in the old dependency graph (see Figure 4). In this case, there is no reason for j to assume $\perp_{\sqsubseteq}$ for k, instead, k acknowledges and sends its current value, $k.t_{cur}$. This leads to a sound approach when combined with the second algorithm.

In the second algorithm, the node i ensures that all nodes that de-

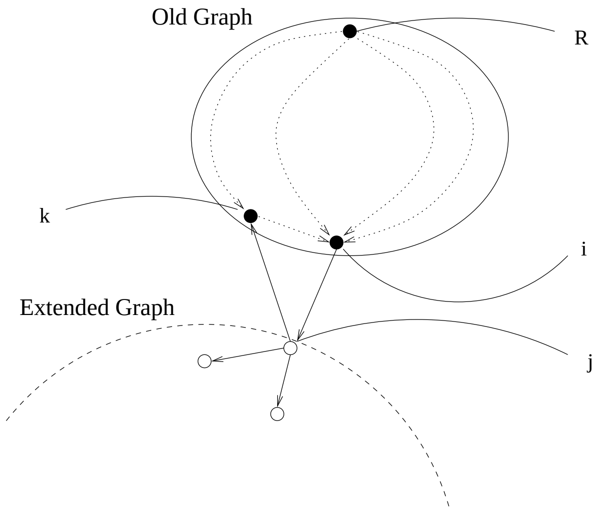


Figure 4: Situation where node i initiates a new node j which has an edge back to an old node k. Old nodes are black and new nodes white. Edges in the new graph are full lines, whereas dotted lines denote paths in the old graph.

pend on it, either directly or indirectly, take on a safe approximation to the new function. This algorithm is given by Figure 5. Node i will set $i.t_{cur} \leftarrow i.t_{old} \leftarrow \bot_{\sqsubseteq}$, while leaving its array $i.m$ unchanged. The algorithm will terminates by a principle similar to that of the dependency-graph algorithm. In contrast to that algorithm, the messages "flow" in the opposite direction of the edges, e.g. from i to i^-. Node i starts by sending `reset` messages to each node $j \in i^-$. It expects to receive a reply from each of these nodes, and once received, the algorithm is terminated. Any node j (including possibly i itself) which receives a `reset` message from a node k, starts by setting its entry for k, $j.m[k] \leftarrow \bot_{\sqsubseteq}$. It then sets $j.t_{old} \leftarrow j.t_{cur} \leftarrow \bot_{\sqsubseteq}$, which is a safe approximation of its value, with respect to the updated fixed point. Unless $i = j$, node j then propagates the `reset` message to each $j' \in j^-$. The propagation is only done for the first `reset` message received. Subsequent message are simply acknowledged immediately after the array entry has been updated. Once j has received acknowledgments from each of its "parents" (i.e. j^-), it either sends an acknowledgment to the first node which sent it the `reset` message, or, in case of $i = j$, it stops. The effect of this algorithm is the following. For each node $j \in [n]$, j will receive a `reset` message from

```
Process: nodes $j(\neq i)$
    receive (reset) from $X$;
    $j.t_{old} \leftarrow j.t_{cur} \leftarrow j.m[X] \leftarrow \bot_{\sqsubseteq}$;
    $||_{\{A,B\}}$
        A : replicate
            [ receive (reset) from $Y$;
              $j.m[Y] \leftarrow \bot_{\sqsubseteq}$;
              send (ack) to $Y$]

        B : $||_{k \in j^-}$
              $k$ : send (reset) to $k$;
                  receive (ack) from $k$;
    || join $j^-$ then send (ack) to $X$
```

Figure 5: Reset Algorithm - Generic node behaviour

each $l \in j+$ which has a path to i. Each of the array entries for these nodes, $j.m[l]$, is set to $\bot_{\sqsubseteq}$ in order to take a safe approximation to the updated fixed point. Note however, that any node n which does not have a path to i, will never receive a `reset` message, and thus any $n' \in n^-$ which depends on n, will not reset its entry for n.

Finally, one must consider how the two algorithms work concurrently. Say that a node is "new" if it is in the dependency graph for the updated function F' but not in that for F. Similarly an "old" node is one that is in the dependency graph for F. Suppose first that i initiates a new node j, which depends on an old node k. If j informs k of the dependency before k "is reset" (receives a `reset` message), then j will receive the current value of k, which can later be reset to $\bot_{\sqsubseteq}$ if the `reset`-algorithm requires it (see Figure 4). Suppose instead that k has received its `reset` message before it is informed of j's dependency. In this case j will receive $\bot_{\sqsubseteq}$. In either case, when *both* algorithms have terminated, then we have extended the dependencies to the new dependency graph, and furthermore, any node j in the extended graph stores in its current value, and in its array $j.m$ only information approximations to the fixed point value. In this case, by Prop. 2.1 the conditions of the Asynchronous Convergence Theorem are satisfied, and computation with respect to the new function $F' = F[f'_i/f_i]$ can proceed.

In the worst case, in which every other node depends on i, this algorithm will reduce to the trivial "reset" algorithm, in which the `reset` message is broadcast to all nodes. Is not hard to see that the global number of messages sent in this algorithm is $O(|E'|)$, where E' denotes the edges in the extended graph. Each message has bit-size $O(1)$, or $O(\log_2 |X|)$ in case of generalized acknowledgments.

4.2 Non Edge-adding Updates

Consider an update, $f_i \mapsto f'_i$, where the dependencies of f_i are a super set of the dependencies of f'_i, i.e. node i deletes edges, but adds none. In some ways, this case is simpler. One could just let i send a `delete` message to each j that "was deleted". Then i simply runs the "reset"-algorithm described in the previous section. This is safe, but one might argue that we might still have redundant "dangling" edges in the graph. This occurs in the case where a deleted edge ij disconnects from the root node, a set of nodes k, which are reachable only from j. If such disconnected k has a dependency back to a node l, which is still reachable from the root, then l will be sending values to k even though k is really not needed in order for the root to compute its value. It is not clear, short of a complete re-computation of the dependency graph (which could be acceptable), how to efficiently (dynamically) deal with this problem.

4.3 General Updates

It should be clear that a combination of these two algorithms can be used to recover from arbitrary updates, leaving still the problem of "dangling" edges. One simply initiates, a concurrent execution of the algorithms for updating the dependency graph (deletion and addition of edges), together with the reset-algorithm described previously.

4.4 An example run

We illustrate a simple policy update on the functions from Section 2.3. Let us assume that the root node R decides to update π_{R} from

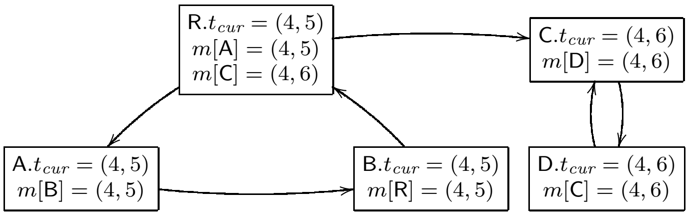
$$\pi_{\mathsf{R}} \quad = \quad (\ulcorner \mathsf{A} \urcorner(S) \vee \ulcorner \mathsf{C} \urcorner(S)) \sqcup \mathtt{Loc}(S)$$

to policy π'_{R} by replacing the trust-join by the more restrictive trust-meet.

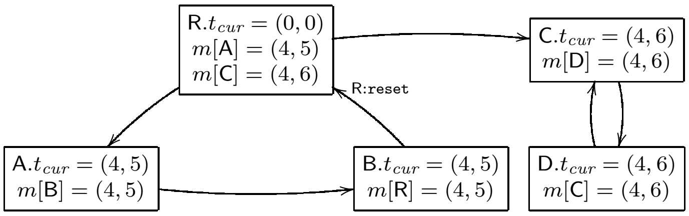
$$\pi'_{\mathsf{R}} \quad = \quad (\ulcorner \mathsf{A} \urcorner(S) \wedge \ulcorner \mathsf{C} \urcorner(S)) \sqcup \mathtt{Loc}(S)$$

Its dependencies are unchanged. Again we illustrate the algorithm with a sequence of network-states, starting with the converged state from Section 2.3.

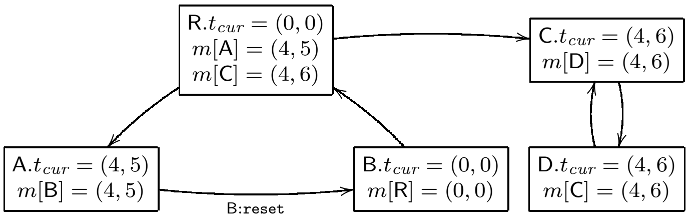
Network snapshot 1. Initial state, corresponding to the fixed point of the old policies.



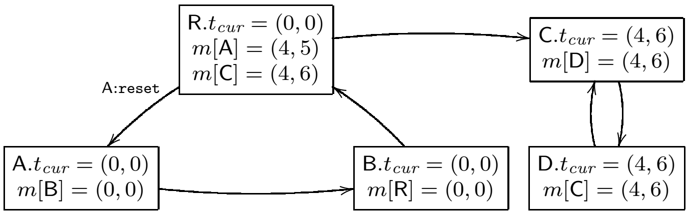
Network snapshot 2. R updates, $\pi_{\mathsf{R}} \mapsto \pi'_{\mathsf{R}}$, and initiates the update algorithm.



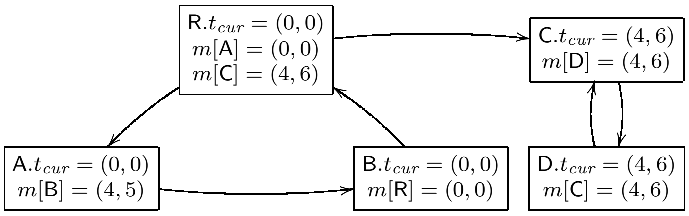
Network snapshot 3. B resets its value for R, and sends a reset message to A.



Network snapshot 4.



Network snapshot 5. No further reset-messages are sent. We do not illustrate the sequence of `ack` messages flowing back.


$\mathsf{R}.t_{cur} = (0,0)$
$m[\mathsf{A}] = (0,0)$
$m[\mathsf{C}] = (4,6)$
$\mathsf{C}.t_{cur} = (4,6)$
$m[\mathsf{D}] = (4,6)$
$\mathsf{A}.t_{cur} = (0,0)$
$m[\mathsf{B}] = (4,5)$
$\mathsf{B}.t_{cur} = (0,0)$
$m[\mathsf{R}] = (0,0)$
$\mathsf{D}.t_{cur} = (4,6)$
$m[\mathsf{C}] = (4,6)$

It is important to notice that the component consisting of C and D is completely unaffected by the update, and, more importantly, that R keeps the old information from C, i.e., entry $\mathsf{R}.m[\mathsf{C}] = (4,6)$. If the asynchronous algorithm is run from this point, it will converge much faster than if run from scratch since this information is immediately incorporated by R.

5 Conclusion

We have presented concrete algorithmic techniques for computation and approximation of the least fixed-point of a collection of continuous functions on trust structures. We have shown that the assumptions of the Asynchronous Convergence Theorem of Bertsekas are satisfied if one initiates computation with a consistent information-approximation, which means that we can apply a well-established asynchronous fixed-point algorithm for both approximation and computation of the least fixed-point. We have considered trust structures in which the two orderings are related in that the information ordering is continuous with respect to the trust ordering. For these trust structures, we have proved two propositions which relate the two orderings, allowing one to reason about the least fixed-point of continuous functions that are also monotonic with respect to the trust ordering. The propositions are theoretically simple, but their novelty lies in that we are relating the two different orderings in a way that gives rise to efficient protocols that allows principals to reason about the fixed-point values without having to compute the exact fixed-point. The second approximation technique (Prop. 3.2) relies on an algorithm which supplies an information approximation. This is used to reason about the trust-relation between the current value ($i.t_{cur}$) and the actual fixed-point value $(\mathsf{lfp}\ F)_i$. This gives a nice connection between the asynchronous fixed-point algorithm, which automatically provides information approximations, and the idea of safe fixed-point approximation. In the final section, we have presented techniques to deal with dynamic updates of the policy functions, f_i. For information-increasing updates, this is very efficient as all current estimates are still valid (invoking Prop. 2.1), in the sense that that the asynchronous algorithm will converge to the correct value. For completely general updates, we have given an algorithm which does not discard the information that is definitely not affected by the update.

One can imagine the system starting from scratch, i.e. there is a collection of nodes, each with a trivial policy $\pi = \bot_{\sqsubseteq}$, implying that no nodes are connected in the dependency graph. As the system evolves, edges are added, reflecting creation of new trusting relationships. One runs the dynamic algorithms for policy updates to ensure that the computations are always consistent. The techniques are dynamic, which allows new nodes to enter and leave the network, without affecting the algorithms.

Apart from its application in implementing trust-structure-based systems, the technique for fixed-point computation presented in this paper is general enough to be used for order-theoretic fixed-point computation in any cpo with bottom, or complete lattice. In particular, the techniques could be the basis of a distributed implementation of a variant of Weeks' model of trust-management systems [23], in which credentials are stored by the issuing authorities instead of being presented by clients. This would allow also for revocation, implemented simply as a trust-policy update at the authority revoking the credential.

Interesting future work is to explore to which extent the area of abstract interpretation [10] can be applied for trust-structure fixed-points, e.g. using widening and narrowing to speed up computation, and allow possibly infinite height cpos. Also, it could be interesting to try and analyze the amortized complexity of our system. For example, if principal R wants to know its trust in q, it can run the algorithm presented in this paper to compute this value. Now, after some time has passed, principals might have made additional observations about q. Supposing that R at some point later wants to compute its trust in q, then since one reuses the information gained from the last computation, the second re-computation would be significantly faster. In general, one might consider the amortized cost of a sequence of operations which are either 'computation of the fixed-point value', or 'policy update'.

5.1 Related Work

As mentioned previously, Weeks has developed a mathematical framework [23] suitable for modelling many traditional trust-management systems (e.g. [3, 4, 9, 11, 12]). The framework is based on defining a global trust-state ("authorization map" [23]) by existence of least fixed-points of monotonic endo-functions on complete lattices. The trust-structure framework [7,19], introduces a notion of *information* into the framework of Weeks. The primary difference between the two frameworks is that, in trust structures, least fixed-points are with respect to *information*, whereas in Weeks' framework they are with respect to *trust* (indeed, there is no notion of 'information ordering', and 'trust' is identified with authorization [23]). Another important difference is that in Weeks' framework, the trust policies (licenses) are carried by clients instead of being stored at the issuing servers. This means that the operational approach is to let clients present, along with their request, a set of licenses, which, together, give rise to what corresponds to function Π_λ. It is now the job of the server to (locally) compute the fixed-point, $\mathsf{lfp}\ \Pi_\lambda$, and decide how to respond. In contrast, in the trust-structure framework, the trust policies are naturally distributed. Each principal p, autonomously controls and stores its policy, π_p. This leads naturally to a distributed approach to computation of fixed-points.

The trust-structure framework has been further developed [15], providing a categorical axiomatization of trust structures, and providing an understanding of the interval construction, a general technique for constructing trust structures [7,19], as a functor, which is the full and faithful left-adjoint in a co-reflection of a new category of trust structures, in a category of complete lattices. The framework has a concrete instance in the SECURE project [5, 6] which deploys a specific class of trust structures, allowing probabilistic information in its modelling of trust [15,20].

The idea of computing *local* fixed-points has been recognized also by Vergauwen *et al.* in the non-distributed context of static program-analysis [22]. Dimitri Bertsekas has developed a substantial body of work on distributed- and parallel-algorithms for fixed points, and this paper applies his asynchronous convergence theorem [1] to prove correctness of a distributed fixed-point algorithm. Finally, the EigenTrust system also defines its global trust-state by existence of unique (non order-theoretic) fixed-points [14], and the basic EigenTrust algorithm is essentially Bertsekas' globally synchronous algorithm.

Acknowledgments. This work was done as part of the SECURE project, EU FET-GC IST-2001-32486, and we would like to thank everyone in the consortium for their cooperation. We thank also Vladimiro Sassone and Mogens Nielsen for useful feedback.

References

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

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1999.

[3] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Proceedings from Security Protocols: 6th International Workshop, Cambridge, UK, April 1998*, volume 1550, pages 59–63, 1999.

[4] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings from the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[5] Vinny Cahill and Elizabeth Gray *et al.* Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.

[6] Vinny Cahill and Jean-Marc Seigneur. The SECURE website. `http://secure.dsg.cs.tcd.ie`, 2004.

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

[8] A. Chander, D. Dean, and J.C. Mitchell. Reconstructing trust management. *Journal of Computer Security*, 12(1):131–164, 2004.

[9] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for (web) applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM-Press, New York.

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomsa, and T. Ylonen. SPKI Certificate Theory. RFC 2693, ftp-site: `ftp://ftp.rfc-editor.org/in-notes/rfc2693.txt`, September 1999.

[12] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings from the 2001 IEEE Symposium on Security and Privacy, Oakland, California*, pages 106–116. IEEE Computer Society Press, 2001.

[13] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation for online service provision. Decision Support Systems, (to appear, preprint available online: `http://security.dstc.edu.au/staff/ajosang`), 2004.

[14] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings from the twelfth international conference on World Wide Web, Budapest, Hungary*, pages 640–651. ACM Press, 2003.

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

[16] Karl Krukow, Mogens Nielsen, and Vladimiro Sassone. A formal framework for concrete reputation-systems with applications to history-based access control. Submitted. Available online: `http://www.brics.dk/~krukow`, 2005.

[17] N. Li and J.C. Mitchell. A role-based trust-management framework. In *Proceedings from DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 201–213. IEEE Computer Society Press, 2003. (vol. 1).

[18] N. Li and J.C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings from the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 89–103. IEEE Computer Society Press, 2003.

[19] Mogens Nielsen and Karl Krukow. Towards a formal notion of trust. In *Proceedings from the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 4–7. ACM Press, 2003.

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

[21] Vitaly Shmatikov and Carolyn Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005.

[22] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In B. Le Charlier, editor, *Proceedings from Static Analysis (SAS'94)*, pages 314–328. Springer, Berlin, Heidelberg, 1994.

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

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

Recent BRICS Report Series Publications

RS-05-6 Karl Krukow and Andrew Twigg. *Distributed Approximation of Fixed-Points in Trust Structures*. February 2005. 41 pp.

RS-05-5 A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations (Preliminary Version). *Dariusz Biernacki and Olivier Danvy and Kevin Millikin*. February 2005. ii+16 pp. Superseded by BRICS RS-05-16.

RS-05-4 Andrzej Filinski and Henning Korsholm Rohde. *Denotational Aspects of Untyped Normalization by Evaluation*. February 2005. 51 pp. Extended version of an article to appear in the FOSSACS 2004 special issue of RAIRO, *Theoretical Informatics and Applications*.

RS-05-3 Olivier Danvy and Mayer Goldberg. *There and Back Again*. January 2005. iii+16 pp. Extended version of an article to appear in *Fundamenta Informaticae*. This version supersedes BRICS RS-02-12.

RS-05-2 Dariusz Biernacki and Olivier Danvy. *On the Dynamic Extent of Delimited Continuations*. January 2005. ii+30 pp.

RS-05-1 Mayer Goldberg. *On the Recursive Enumerability of Fixed-Point Combinators*. January 2005. 7 pp. Superseedes BRICS report RS-04-25.

RS-04-41 Olivier Danvy. *Sur un Exemple de Patrick Greussay*. December 2004. 14 pp.

RS-04-40 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. December 2004. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.

RS-04-39 Olivier Danvy and Lasse R. Nielsen. *CPS Transformation of Beta-Redexes*. December 2004. ii+11 pp. Extended version of an article appearing in *Information Processing Letters*, 94(5):217–224, 2005. Also superseedes BRICS report RS-00-35.

RS-04-38 Olin Shivers and Mitchell Wand. *Bottom-Up β-Substitution: Uplinks and λ-DAGs*. December 2004. iv+32 pp.