

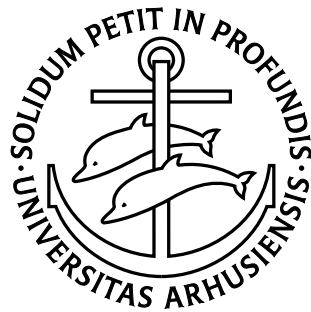
# Towards a Theory of Trust for the Global Ubiquitous Computer

Karl Krukow

---

---

PhD Dissertation



Department of Computer Science  
University of Aarhus  
Denmark



# Towards a Theory of Trust for the Global Ubiquitous Computer

A Dissertation  
Presented to the Faculty of Science  
of the University of Aarhus  
in Partial Fulfilment of the Requirements for the  
PhD Degree

by  
Karl Krukow  
July 27, 2006



# Abstract

This dissertation contributes to a relatively young field in computer science research, the field of *trust-based security*. Primarily, *theories* of computational trust are considered, *i.e.*, principles, models and associated reasoning techniques. It is argued that one or several theories of trust should constitute *a* corner-stone in the answer to one of the so-called ‘Grand Challenges’ for computer science research: ‘Science for Global Ubiquitous Computing.’ A central implication is that trust management is not purely an engineering task: To be compatible with the vision of a science for global ubiquitous computing, it must be amenable to rigorous reasoning; hence, *mathematical models* as well as *languages and tools* are necessary to achieve the ultimate goal.

The scientific contributions of this dissertation are primarily theoretical, and lie within each of these three areas: We develop mathematical models for trust management, associated domain-specific policy languages as well as tools (in a theoretical sense, *e.g.*, algorithms and operational semantics). At a more practical level, it is shown how a general formal approach to reputation-based trust management has applications in a research area previously not linked to computational trust: A prototype software framework for *history-based access control* in Java is implemented based on models, languages and tools developed in this dissertation.



# Acknowledgements

There are many people that I want to thank:

Vladimiro Sassone for excellent supervision and encouragement before, during and after my stays at University of Sussex;

my friends, both from university and those who go back longer;

my beloved girlfriend, Inger: Thanks for your love and support, you know I love you;

the staff and students of DAIMI and BRICS for creating a stimulating and inspiring environment; particularly, I want to thank the inspiring teachers we have been blessed with at DAIMI: Ivan, Olivier, Erik . . . to name a few;

the SECURE project consortium; particularly, I thank Marco and Andy for cooperation, discussion and fun;

my co-students at University of Sussex, especially Damiano, Jan, David, Giovanni, Phillipe and Mikkel;

my family, especially my mother, for support and continuous(!) interest.

Last *but most*, I want to thank my supervisor, Mogens: No PhD student could wish for a better or more devoted and passionate supervisor! I never stopped to be amazed that you can find the *time* and energy to support, read, advise and generally be a good colleague and friend (in spite of strange ideas and endless printouts for “home-reading”): Excellent!

*Karl Krukow,  
Århus, July 27, 2006.*



# Preface

This dissertation is structured in two parts. Part I gives an overview of the contributions of my research in context. This is done from several angles:

*(i)* I will identify a long-term research agenda to which my research may be seen as contributing, hence, giving a broader picture; *(ii)* a survey of relevant research in the area of trust management is provided; *(iii)* each of my contributions are discussed at an abstract level, giving the reader a taste of the contents of each of the research papers included in Part II of this dissertation; and finally, *(iv)* my view on an outlook for future theoretical research within the field of trust management is presented. Part II consists of three chapters, each corresponding roughly to a group of related published papers. Additionally, the first Chapter in Part II (Chapter 4) contains new material which has not yet been published.



# List of Publications

The following is a list of publications of conference and journal papers, where I am either author or co-author, produced during my PhD studies. The list is presented in reverse chronological order.

- [55] K. Krukow and M. Nielsen. Trust Structures: Denotational and operational semantics. Journal version. Accepted for publication in the International Journal of Information Security, special edition on Formal Aspects of Security and Trust. Available online <http://www.brics.dk/~krukow>, 2006.
- [58] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for reputation systems. Journal version. Submitted. Available online [www.brics.dk/~krukow](http://www.brics.dk/~krukow), 2006.
- [61] K. Krukow and A. Twigg. The complexity of fixed point models of trust in distributed networks. To be published in TCS – Special issue on Semantic and Logical Foundations of Global Computing, Elsevier, 2006.
- [53] K. Krukow. An operational semantics for trust policies. Presented at Workshop on Issues in the Theory of Security, 2006 (WITS'06), March 25–26 2006, Vienna, Austria, co-located with ETAPS 2006. No published proceedings yet.
- [56] K. Krukow, M. Nielsen, and V. Sassone. A formal framework for concrete reputation-systems with applications to history-based access control. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 260–269, New York, NY, USA, 2005. ACM Press.
- [59] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. In *Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 805–814. IEEE, 2005.
- [82] M. Nielsen and K. 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 on the Occasion of His 70th Birthday*, volume 3113 of *Lecture Notes in Computer Science*, pages 192–204. Springer Verlag, 2004.
- [16] V. Cahill, E. Gray, K. Krukow *et al.* Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.
- [81] M. Nielsen and K. 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.



# Non-standard Notation

We suggest that the reader consults this section in case of ambiguities or imprecision in our notation.

**Probability theory.** We subscribe to the so-called Bayesian view on probability (as advocated by many, e.g., Jaynes, Jeffreys, Laplace [43]). Mostly, we follow the notational conventions of Jaynes [43]. In the view of Jaynes (and Jeffreys), probability theory is an extension of logic, operating on formal propositions; the notions of “random variables” and “random experiments” are not fundamental. If  $\Theta = \{A, B, C, \dots\}$  is a finite set of base propositions, and  $X, Y$  are propositional logical formulas over the base propositions, then

$$P(X | Y)$$

denotes the probability of  $X$  given information  $Y$ . When  $X$  and  $Y$  are logical formulas,  $XY$  denotes the conjunction of  $X$  and  $Y$ , and  $X + Y$  denotes their disjunction;  $\bar{X}$  denotes the negation of  $X$ . For example, we might write  $P(\bar{A} + B | CD)$ . The probability  $P(X | Y)$  is similar to (but quantitatively generalises) the judgement  $Y \models X$  of propositional logic; in particular, if  $Y \models X$  then  $P(X | Y) = 1$  is satisfied, and if  $Y \models \bar{X}$  then  $P(X | Y) = 0$ .

Sometimes we are less formal and write statements like  $P(m | \lambda)$  where  $m$  might be a number, and  $\lambda$  a mathematical structure, i.e., a model; what is meant is  $P(X_m | Y_\lambda)$  where  $X_m$  and  $Y_\lambda$  are appropriate formal propositions encoding the intended meaning, e.g.,  $X_m$  might state that a certain variable  $V$  equals the number  $m$ , and  $Y_\lambda$  could state information about  $m$ , e.g.,  $1 \leq m \leq 42$  (for further clarification see Jaynes (2005) [43]).

If  $f(\theta | \lambda)$  is a (continuous) probability density function (pdf) for a real parameter  $\theta$  (see Jaynes [43]), defined on an interval  $I = [a, b] \subseteq \mathbb{R}$ , then the *expectation* or *expected value* of  $\theta$  given  $f$ , written  $\mathbf{E}_{f(\theta|\lambda)}(\theta)$  is defined as

$$\mathbf{E}_{f(\theta|\lambda)}(\theta) = \int_a^b d\theta \theta f(\theta | \lambda).$$

Note that, following Jaynes, we write the variable that is integrated *just after* the integral symbol, as opposed to the (perhaps) more common:

$$\int_a^b \theta f(\theta | \lambda) d\theta.$$

Similarly, if  $X$  is a variable quantity that can take on finitely many values  $(x_1, x_2, \dots, x_n)$  in  $n$  mutually exclusive and exhaustive situations, and  $\lambda$  is a model that assigns probabilities  $p_i = P(X = x_i | \lambda)$  to each of them, then the expectation of  $X$  given  $\lambda$ , written  $\mathbf{E}_{P(X|\lambda)}(X)$ , is defined as:

$$\mathbf{E}_{P(X|\lambda)}(X) = \sum_{i=1}^n x_i p_i.$$

**Sets and functions.** Let  $X, Y, U, W$  be sets from some universe  $\mathcal{U}$ ; and let  $f : X \rightarrow_* U$  and  $g : Y \rightarrow_* W$  be a (possibly partial) functions. We write  $f/g$ , pronounced *f over g*, for the (partial) function of type  $X \cup Y \rightarrow_* U \cup W$ , given by

$$f/g(v) \stackrel{(\text{def.})}{=} (f/g)(v) \stackrel{(\text{def.})}{=} \begin{cases} f(v) & \text{if } v \in X \\ g(v) & \text{if } v \notin X \end{cases}$$

(in particular, if  $v \in X \cap Y$  then  $f/g(v) = f(v)$ ).

# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>Preface</b>	<b>ix</b>
<b>List of Publications</b>	<b>xi</b>
<b>Non-standard Notation</b>	<b>xiii</b>
<b>I Overview</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 The Grand Challenges Exercise . . . . .	4
1.2 Understanding the Internet? . . . . .	5
1.3 Science for Global Ubiquitous Computing . . . . .	7
1.3.1 The Challenge: A Theoretical Hierarchy . . . . .	7
1.4 The Role of Trust in SGUC . . . . .	9
1.4.1 Relevance ( <i>i</i> ) . . . . .	10
1.4.2 Modelling and Reasoning ( <i>ii</i> )+( <i>iii</i> ): Trust Management and Science for the GUC . . . . .	14
<b>2 Research Context</b>	<b>17</b>
2.1 Introduction . . . . .	17
2.2 Credential-based Trust Management . . . . .	18
2.2.1 Influential Systems . . . . .	18
2.2.2 Fundamental Models . . . . .	19
2.3 Experience-based Trust Models . . . . .	22
2.3.1 General Principles . . . . .	23

2.3.2	Non-probabilistic Approaches . . . . .	25
2.3.3	Probabilistic Approaches . . . . .	27
2.4	A Hybrid Model . . . . .	33
2.4.1	The Trust Structure Framework . . . . .	33
2.5	Concluding Remarks . . . . .	36
<b>3</b>	<b>Outlook and Extended Abstracts of Contributions</b>	<b>39</b>
3.1	The SECURE Trust Model . . . . .	39
3.1.1	Conclusion . . . . .	44
3.2	Operational Techniques for Trust Structures . . . . .	45
3.2.1	An Operational Semantics . . . . .	46
3.2.2	Approximation Techniques for $\preceq$ -Monotonic Policies . . . . .	47
3.2.3	Conclusion . . . . .	48
3.3	A Logical Framework for Reputation Systems . . . . .	50
3.3.1	Conclusion . . . . .	52
3.4	An Outlook . . . . .	53
3.4.1	Towards Comparing Probabilistic Trust-based Systems . . . . .	54
3.4.2	Towards a Formal Model of Dynamic Behaviour . . . . .	60
3.4.3	Towards the SGUC Challenge . . . . .	62
<b>II</b>	<b>Papers</b>	<b>65</b>
<b>4</b>	<b>The SECURE Trust Model</b>	<b>67</b>
4.1	Introduction . . . . .	69
4.2	Event Structures for Interaction . . . . .	72
4.2.1	Confusion-free Event Structures . . . . .	75
4.3	A Probabilistic Framework . . . . .	78
4.3.1	The Dirichlet Distribution . . . . .	79
4.3.2	Dirichlets on Cells . . . . .	81
4.4	Trust Values and Orderings . . . . .	82
4.5	Transferring Information . . . . .	85
4.5.1	Morphisms . . . . .	85
<b>5</b>	<b>Operational Techniques for Trust Structures</b>	<b>89</b>
5.1	Introduction . . . . .	91
5.1.1	The Trust-Structure Framework . . . . .	92
5.1.2	The Operational Problem . . . . .	96
5.2	A Basic Language for Trust Policies . . . . .	97
5.3	An Operational Semantics . . . . .	101

5.3.1	The I/O Automaton Model . . . . .	101
5.3.2	$\llbracket \cdot \rrbracket^{\text{op}}$ Translation: An Operational Semantics . . . . .	105
5.3.3	Reasoning about the Semantics: Cause and Effect . . . . .	113
5.4	Bertsekas Abstract Asynchronous Systems . . . . .	115
5.4.1	The Asynchronous Convergence Theorem . . . . .	118
5.4.2	$\llbracket \cdot \rrbracket^{\text{op-abs}}$ Translation: An Abstract Operational Semantics . . . . .	119
5.4.3	Correspondence of Abstract and Concrete Operational Semantics . . . . .	120
5.5	Correspondence of Denotational and Operational Semantics . . . . .	122
5.5.1	Practical remarks about the operational semantics . . . . .	125
5.6	Approximation Techniques . . . . .	126
5.6.1	Proof-carrying Requests . . . . .	127
5.6.2	I/O-Automaton Version . . . . .	129
5.6.3	A Snapshot-based Approximation Technique . . . . .	131
5.7	Proofs . . . . .	137
<b>6</b>	<b>A Logical Framework for Reputation Systems</b> . . . . .	<b>147</b>
6.1	Introduction . . . . .	149
6.1.1	Contributions and Outline . . . . .	151
6.2	Observations as Events . . . . .	153
6.2.1	The Event Structure Framework . . . . .	154
6.3	A Language for Policies . . . . .	158
6.3.1	Formal Description . . . . .	159
6.3.2	Example Policies . . . . .	160
6.4	Dynamic Model Checking . . . . .	163
6.4.1	An Array-based Implementation . . . . .	164
6.4.2	An Automata-based Implementation . . . . .	166
6.4.3	Remarks . . . . .	169
6.5	Language Extensions . . . . .	170
6.5.1	Quantification . . . . .	171
6.5.2	References and Quantitative Properties . . . . .	179
6.6	A Java Security Manager . . . . .	182
6.6.1	Design . . . . .	183
6.7	Related Work . . . . .	187
	<b>Bibliography</b> . . . . .	<b>191</b>



**Part I**

**Overview**



# Chapter 1

## Introduction

*A Grand Challenge for research in science or engineering pursues a goal that is recognised one or two decades in advance; its achievement is a major milestone in the advance of knowledge or technology, celebrated not only by the researchers themselves but by the wider scientific community and the general public.*

— Tony Hoare and Robin Milner, 2004  
Grand Challenges in Computing: Research.

This dissertation is concerned with *trust management for global ubiquitous computing*. The purpose of the present chapter is to present “the bigger picture,” i.e., to *identify the role* and *motivate the relevance* of this topic in the context of contemporary computer science research. It is argued that a coherent theory of trust should form a cornerstone in one of the so-called Grand Challenges for computer science research: Science for Global Ubiquitous Computing (SGUC). Problems in the domain of Global Computing have guided the direction of my PhD research; however, the actual SGUC challenge to which it is referred was not formulated clearly until 2004 [76].

To be clear, this dissertation is by no means an attempt to “solve” a particular Grand Challenge or even a sub-challenge; the challenges are *way* to difficult. Instead, we shall use the challenge ‘Science for Global Ubiquitous Computing’ [76, 3] exactly as was intended with the Grand Challenges: To give *focus* to our research (and to this dissertation in particular) by *(i) identifying and placing* our research contributions in a wide context in terms of a long-term grand research objective; *(ii) motivating* the relevance of our research; and *(iii) letting* the challenge serve as a *guideline* and sanity-check for our use of assumptions, and of existing theory and tools of computer science.

In the following, the Grand Challenges Exercise is briefly presented along with the specific challenge Science for Global Ubiquitous Computing. We then proceed to identify the role of theories of trust as a component of a successful answer to the challenge.

## 1.1 The Grand Challenges Exercise

In 2002 the UK Computing Research Committee (UKCRC) initiated the Grand Challenges Exercise. An open ‘call for ideas’ was issued, leading to a workshop, held in Edinburgh in November 2002, which spun-off a number of public discussions and follow-up workshops, culminating in a list of (currently) six Grand Challenges for research in computer science.<sup>1</sup> A Grand Challenge is an ambitious and highly desirable goal that a significant scientific community can commit to working towards, and agree seems feasible within a reasonable though long-term time-scale. The topics of the Grand Challenges have emerged as a consensus among a significant and general scientific community, and serve as a focus for research.

A set of criteria of maturity of a challenge has been established, and we list a select few below.<sup>2</sup> There is no ordering on the criteria, and a particular challenge is not expected to fulfil all of them.

- It arises from scientific curiosity about the foundation, the nature or the limits of a scientific discipline.
- It will be obvious how far and when the challenge has been met (or not).
- It promises to go beyond what is initially possible, and requires development of understanding, techniques and tools unknown at the start of the project.
- It decomposes into identified intermediate research goals, whose achievement brings scientific or economic benefit, even if the project as a whole fails.

We would like to bring focus to the last item. A mature challenge can often be decomposed into a number of sub-challenges to be accomplished, together

---

<sup>1</sup>The present list can be found at [http://www.ukcrc.org.uk/grand\\_challenges/current/index.cfm](http://www.ukcrc.org.uk/grand_challenges/current/index.cfm).

<sup>2</sup>The full list of criteria can be found at [http://www.ukcrc.org.uk/grand\\_challenges/about/criteria.cfm](http://www.ukcrc.org.uk/grand_challenges/about/criteria.cfm).

with an understanding of how the sub-challenges are interrelated and how they interact. In this dissertation we will focus on a specific challenge, ‘Science for Global Ubiquitous Computing’, and argue that one of its sub-challenges is a coherent *theory of trust*.<sup>3</sup>

## 1.2 Understanding the Internet?

Spyware, Phishing, Trojans, Viruses, Worms, SPAM, Botnets, Rootkits, Distributed Denial of Service Attacks, Active-X Weaknesses, Bad Passwords, Zer0-day exploits, Vulnerabilities in Wireless Security, Mobile Viruses; the list of threats goes on — even security software such as Anti-Virus software contains vulnerabilities!<sup>4</sup> The Internet as it exists today is dangerous. Every time we turn on our computers and connect to it, we are exposing ourselves to risks. Vulnerabilities in applications, operating systems and protocols, and flaws in their implementations expose our computers and thus our personal data to compromise. Seemingly innocent e-mail may contain executable content such as viruses, or may attempt to trick us by spoofing otherwise semi-trusted identities, e.g., using techniques known as ‘Phishing’ (“fishing”) and ‘Pharming’ (“farming”) hackers currently impersonate trusted identities, e.g., our banks, attempting to lure us into submitting personal information (See Figure 1.1). Even disclosure of information to trusted Internet entities may also lead to such compromise: The entities may not adequately protect our data, they may even deliberately disclose it to third parties, and hackers may be listening in on the conversation.

The Internet is also useful: It provides, among *many* other things, an open global network storing massive amounts of information on practically any topic, often free, and which is even efficiently searchable to some extent. It supports new forms of global communication, often real-time, e.g., e-mail, USENET, instant-messaging, web-based forums, ‘BLOGs’, video conferencing, etc.; it enables enhancement or re-implementation of existing technology, often providing free or cheap alternatives, e.g., in tele communication:

---

<sup>3</sup>The grand challenges exercise is still very active, and the current proposed challenges constantly changing and moving. The challenge, ‘Science for Global Ubiquitous Computing,’ has changed into ‘Global Ubiquitous Computing: Science & Design’ and later ‘Ubiquitous Computing: Experience, Design and Science’ [21]. For this dissertation, we will adopt the name ‘Science for Global Ubiquitous Computing’ (emphasising the scientific and theoretic part of the challenge) although we are referring to both new and old documents reporting on the challenge.

<sup>4</sup>See the article “Security Absurdity: The Complete, Unquestionable, and Total Failure of Information Security.”(<http://www.securityabsurdity.com/failure.php/>).

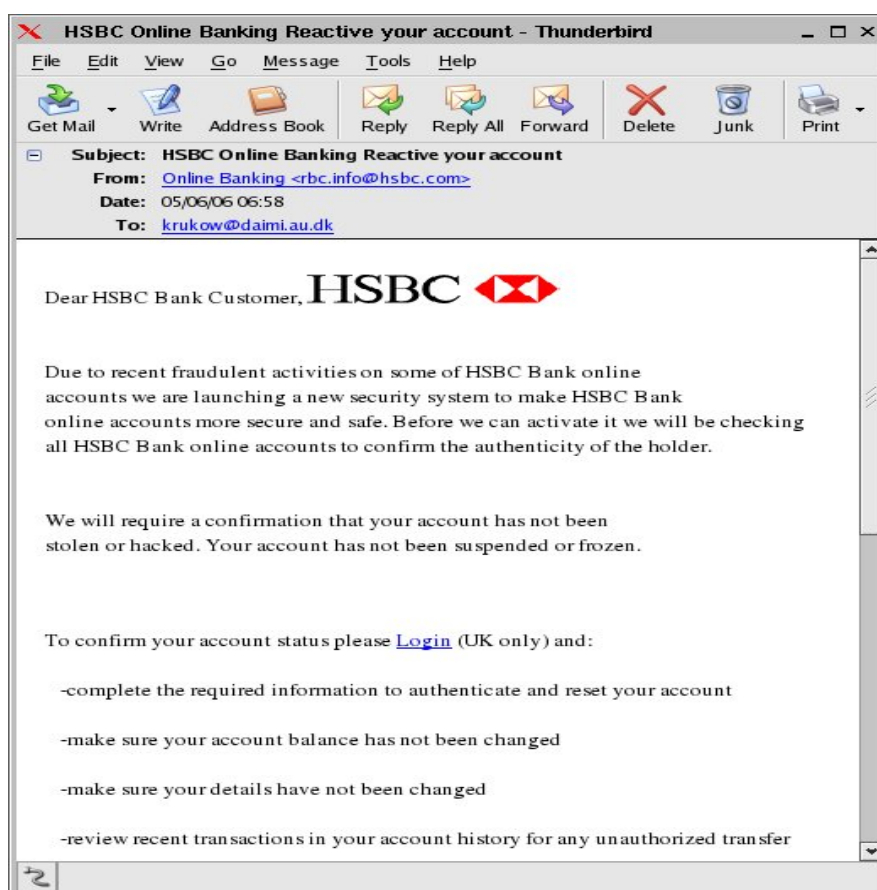


Figure 1.1: A ‘Phishing’ e-mail attempts to trick `krukow@daimi.au.dk` to follow the URL pointed to by the [Login](#) link. In fact, this URL does not point to HSBC; instead, the user is redirected to a malicious site which looks just like HSBC’s website.

Internet-enabled mobile phones, PDAs, and the Skype system.

It is an understanding of the Internet that enables us (the aware Internet users) to make our informed decisions about how (and if) we want to use particular Internet software. However, the Internet and the software we connect to it has grown complex to the point where we do no longer adequately understand it. The growth has been unmanaged, driven by market demands: It is *not* the result of some grand design scheme. In this sense, the Internet can be viewed, in part, as a natural phenomenon, and in part

as an artefact, i.e., something which has been carefully engineered by human craftsmanship [3]. Hence, the Internet itself should be an object of study through a still undeveloped science: a ‘Science for Global Ubiquitous Computing.’

### 1.3 Science for Global Ubiquitous Computing

The notion of the Global Ubiquitous Computer, the GUC, is intended to cover the Internet together with abstractions built upon it and the collection of devices that are connected to it, e.g., overlay networks as well software agents and hardware devices. To illuminate the concept of the GUC and the kind of questions that the challenge deals with, we shall use another quotation, also taken from the introductory section of a recent report on the Grand Challenges Exercise [76]:

*How many computers will you be using, wearing or have installed in your body in 2020? How many other computers will they be talking to? What will they be saying about you, doing for you or to you? By that time, computers will be ubiquitous and globally connected. It is better not to count them individually, but to regard them collectively as a single Global Ubiquitous Computer (GUC). Who will then program the GUC, and how? Shall we be in control of it or even understand it?*

— Marta Kwiatkowska and Vladimiro Sassone,  
*Science for Global Ubiquitous Computing*,  
in: Grand Challenges in Computing – Research,  
edited by Tony Hoare and Robin Milner, 2004.

An *understanding* of the GUC entails many notions; Sassone et al. [3] uses the term to cover, e.g., *analysis, diagnosis, documentation, evaluation, formalisation, specification, validation, . . .* Even less ambitiously: Do we understand the Internet now? How many computers are you using today? How many other computers are they talking to? What are they revealing about you? In fact, many of the questions posed are hard to answer, even when stated about the present; thinking about global scale of the Internet today just emphasises their difficulty, extent and relevance in the future.

#### 1.3.1 The Challenge: A Theoretical Hierarchy

The GUC is characterised as a highly distributed composition of entities (soft- or hardware) that are *dynamic* to the extreme yet reliable and operate

securely: They are *mobile*, moving dynamically between networks, capable of meaningful operation even if disconnected from preferred networks; they are *connected* to an *open* global-scale network (i.e., they are part of the GUC); they are capable of performing *computations*, although many have limited computational and storage capacity; they are *autonomous and ubiquitous*, hence they must make decisions on their own and cannot rely on human intervention; they operate under *incomplete information*: Entities cannot be assumed to have precise (or even any) information about other entities with whom they interact.

Sassone et al. use two main observations for justifying the challenge, i.e., the need for a *Science* for the GUC [3]. Firstly, if industry and market forces alone drive soft- and hardware creation, greatly damaging consequence are bound to occur, as shown by a number of examples in the present and past.<sup>5</sup> Secondly, a sufficiently mature science, would inform the software engineering practice, as do mathematical and physical sciences in other engineering practices, allowing us to design and adapt devices, protocols, components and services that are context-aware, highly distributed, autonomous, ubiquitous, supporting mobility, and, importantly, which are *verifiably* correct, secure and reliable.

A central goal for the challenge is “To develop a coherent informatic science whose concepts, calculi, theories and automated tools allow descriptive and predictive analysis of the GUC at each level of abstraction; ...” [3]. To develop a rigorous understanding necessary of a science, sufficiently advanced and complete formal models are necessary. A proposed approach is the following three principles [3]:

- To express theories for the GUC as a hierarchy of models and languages, assigning each relevant concept to a certain level in the hierarchy;
- To specify, for each model  $M$ , how a system description in  $M$  may be realised in models  $M_1, M_2, \dots, M_n$  lying “below”  $M$  in this hierarchy;
- To devise methods and tools for reasoning at each level, as well as between levels.

The structure of the hierarchy represents abstraction: Intuitively, being “above” in the hierarchy means being more abstract. This provides, at a very high level, a reasonable decomposition of the challenge into sub-challenges, as desired by one of the maturity criteria. Ideally, the idea is that one may

---

<sup>5</sup>See Sassone et al. [3] for a few such examples.

identify the concepts that are relevant to the science, and assign each of these concepts to a suitable model or set of related models in the hierarchy (i.e., a “level”). For example, let us consider the concept of mobility (a concept clearly relevant for ubiquitous computing): At a high level one might have specifications in terms of spacial temporal logics; at a medium level one might have distributed calculi like  $D\pi$  or Ambients, while at an even lower level, one deals with general purpose programming languages (e.g., Java), networks, and protocols like TCP. Examples of reasoning techniques that operate “at a level” could then be types and simulation relations, while techniques for reasoning “between levels” could be model checking and theorem proving, e.g., model checking a spacial temporal logic formula with respect to an ambient process.

This brings us to another important point in the SGUC challenge: We are not starting from scratch! While we have not yet successfully developed complete or unified theories and models for the GUC, there have been major theoretical advances in modelling many of the relevant concepts. A principle of the proposed strategy for answering the challenge is the following [3]: When developing new concepts and models we start from the platform of computer science, i.e., the large body of well-developed existing work both within the theoretical and the experimental discipline. To support the challenge, multiple researchers have contributed to a collection, the so-called Platform Paper [75], containing an impressive survey of concepts and models relevant to the challenge. The collection describes a survey of the state-of-the-art and expected advances within a few years for eight different fields: Space and mobility; Security; Boundaries, resources and trust; Distributed data; Game semantics; Hybrid systems; Stochastics; and Model checking. We shall only deal with one of these fields, namely trust; however, this is done to a much greater extent and detail, and it is hoped that the present and next two chapters may contribute to future versions of the platform paper for this specific field.

## 1.4 The Role of Trust in SGUC

The purpose of this section is to establish trust management as a sub-challenge of the SGUC Challenge. As described in the maturity criteria, such a decomposition consists of “(...) intermediate research goals, whose achievement brings scientific or economic benefit, even if the project as a whole fails.” In the terminology of the previous section: *Ideally*, we would like to (i) argue that the concept of trust management is relevant for the

GUC; *(ii)* assign the relevant concepts from trust management to levels in the theoretical hierarchy (a precondition for this includes developing the appropriate models); and *(iii)* develop reasoning techniques “at each level” and “between levels,” so that desirable security properties can be stated and verified in actual systems. We shall refer to a collection of models together with suitable reasoning techniques as a *theory of trust*. We pose this as an ideal or desired *goal*: We do not claim that it has been accomplished. Item *(i)* is developed to a large extent; it is already recognised by many (e.g., the Platform Paper [75] and the SGUC Manifesto [3]) that trust management is bound to have a role in the SGUC; we elaborate on this here. Items *(ii)* and *(iii)* are less developed: While we shall argue that theories of trust must lie relatively high in the hierarchy, there has not yet been developed sufficiently abstract and complete models nor do sufficient reasoning techniques exist. Our technical contributions are later argued to provide a small delta in these directions.

#### 1.4.1 Relevance *(i)*

Other researchers have already argued convincingly for the relevance of trust management in distributed systems security (cf., Blaze, Feigenbaum et al. [8]). We recapture some of these arguments, and try to highlight a number of properties of the GUC which make the trust management approach even more appealing, even if the limitations of the traditional technology and models must first be overcome.

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

Many modern distributed systems use a combination of access control lists and user authentication, usually implemented via some public key authentication protocol, i.e., deciding a request to perform an action is done by authenticating the public key, effectively linking the key to a user identity, and then looking up in the access control list to see whether that identity

is authorised to perform the action [8]. Furthermore, the security of current systems is often not verified, i.e., proofs of soundness of the security mechanism are lacking (e.g., statements of the form “if the system authorises an action requested by a public key then the key is controlled by user  $U$  and  $U$  is authorised to perform that action according to the access control list”).

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

Blaze et al. [8] present a number of reasons why the traditional approach to authorisation is inadequate:

- *Authorisation = Authentication + Access Control List: Authentication* deals with establishing identity. In traditional static environments, e.g., operating systems, the identity of an entity is well-known. In Internet and GUC applications this is often not the case. This means that if an access control list (ACL) is to be used, some form of authentication must first be performed. In distributed systems, often public-key based authentication protocols are used, which usually relies on centralised and global certification authorities.
- *Delegation*: Since the global scale of the GUC implies that each entity’s security policy must encompass billions of GUC entities, delegation is necessary to obtain scalable solutions. Delegation implies that entities may rely on other (semi) trusted entities for deciding how to respond to requests. In traditional approaches either delegation is not supported, or it is supported in an inflexible manner where security policy is only specified at the last step of a delegation chain.
- *Expressive power and Flexibility*: The traditional ACL-based approach to authorisation has proved not to be sufficiently expressive (with respect to desired security policies) or extensible. The result has been that security policies are often hard-coded into the application code, i.e., using the general purpose programming language that the application is written in. This has a number of implications: Changes in security policy often means rewriting and recompilation, and security reasoning becomes hard as security code is intertwined with application code (i.e., violating the principle of ‘separation of concerns’).

- *Locality*: The autonomy of GUC entities means that different entities have different trust requirements and relationships with other entities. Hence, GUC entities should be able to specify local security and trust policies, and security mechanisms should not enforce uniform and implicit policies or trusting relations.

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

Given a request  $r$  signed by a key  $k$ , the question we really want to answer is the following: “Is the knowledge about key  $k$  such that the request  $r$  should be granted?” In principle, we do not care about *who* signed  $r$ , only whether or not sufficient information can be inferred to grant the request. The trust management approach does not need to resolve the actual identity (e.g., the human-being believed to be performing the request) but, instead, deals with the following question, known as the compliance-checking problem: “Does the set  $C$  of credentials prove that the request  $r$  complies with the local security policy  $\sigma$ ?” [8, 10]. Let us elaborate: A request  $r$  can now be accompanied by a set  $C$  of *credentials*. Credentials are signed policy statements, e.g., of the form “public key  $k$  is authorised to perform action  $a$ .” Each entity that receives requests has its own security policy  $\sigma$  that specifies the requirements for granting and denying requests. Policies can directly authorise requests, or they can delegate to credential issuers that the entity expects have more detailed information about the requester.

Policy is separated from mechanism: A trust management *engine* takes as input a request  $r$ , a set of credentials  $C$  and a local policy  $\sigma$ ; it outputs an authorisation decision (this could be ‘yes’/‘no’, but also more general statements about, say, *why* a request is denied, or *what would be further needed* to grant the request). This separation of concerns supports security reasoning needed in distributed applications, and we shall elaborate on this later (Section 1.4.2) as we believe it to be an important feature for the SGUC challenge.

**Security in the GUC.** The above reasoning argues that traditional authorisation mechanisms are inadequate in modern distributed systems. When

one considers GUC applications, we can add further to this list. Most of the dynamic properties of GUC entities (e.g., mobility, autonomy, ubiquity, global connectivity, ...) affect their security requirements. For example, mobility implies that a GUC entity might find itself in a hostile environment, disconnected from its preferred security infrastructure, e.g., certification authorities. Further, the autonomy requirement means that even in this scenario, it must be able to assign privileges to other GUC entities, privileges that are meaningful based on the usually incomplete information that the assigning entity has about the assigned entity.

- *Active decisions*: Trust management systems focus on deciding how to respond to *requests*. However, GUC entities do not only need to respond meaningfully to requests, i.e., taking *passive* security decisions, but often need to *actively* and autonomously select among equivalent services provided by a number of apparently similar providers. Such decisions may also affect security: Interaction often entails exposing personal data, as well as requiring resources like time, computation, battery and storage. When taking active decisions, there are usually no credentials available; hence, other information, e.g., reputation information, must be taken into account to make meaningful decisions.
- *Information vs. credentials*: Traditional trust management systems focus on *credentials* as the main source of proving compliance of a request with a policy. However, even when no delegation chain may establish sufficient information about a requesting entity, sometimes, collaboration may still be the most beneficial action. Notions of *risk* of an interaction, and *cost/benefit* of the outcome of an interaction are relevant concepts that are not considered in traditional trust management policies. For example, histories, that is, memory of past interactions with an entity, may contain enough information to risk interaction. This entails that trusting relationships change dynamically, based on information about the history of an entity.
- *Probability*: Incomplete information leads naturally to probabilistic decision-making. Trust management systems that focus on information could consider probabilities explicitly (as an alternative to, or, to complement the establishing of credentials in the traditional sense). Ideally, security policies would be amenable to probabilistic yet rigorous reasoning which leads to more generality and flexibility. Additionally, as factors such as cost and benefit of interactions enters the

equation, a notion of *risk* emerges as a product of cost/benefit and probability.

So far we have considered only a single notion of ‘trust’ in computer science, namely the concept of ‘trust management’ coined by Blaze, Feigenbaum and Lacy [10]. In fact, there is a whole different strand of research on trust distinct from the technical notion of Blaze et al., which has resulted in the term ‘trust’ being overloaded within computer science. This other “strand” deals with a computational formalisation of the *human notion of trust*, i.e., trust as a sociological, psychological and philosophical concept. However, the human concept of trust is elusive and its many facets make it hard to define formally [74, 16]. We believe that to live up to the SGUC challenge, it is necessary that the two concepts be merged in a “unified” theory of trust which combines the strengths of both notions. To be more precise, our ideal would be to combine the *rigour* of traditional trust management with the *dynamics* and *flexibility* of the human notion. Let us elaborate: Traditional trust management deals with credentials, policies, requests and the compliance-checking problem. Rigorous security *reasoning* is possible: The intended meaning of a trust management engine is formally specified, correctness proofs are feasible, and many security questions are effectively decidable [71]. In contrast, (to our knowledge) we have yet to see a system based on the human notion of trust which, with realistic assumptions, guarantees any sort of rigorous security property. On the other hand, such systems are capable of making *intuitively* reasonable decisions based on information such as evidence of past behaviour, reputation, recommendations and probabilistic models. A combination of these two approaches would lead to powerful frameworks for decision-making which incorporates more general information than credentials, yet which remains tractable to rigorous reasoning.

#### 1.4.2 Modelling and Reasoning *(ii)+(iii)*: Trust Management and Science for the GUC

While the trust management approach to security has led to significant progress with respect to the flexibility, generality and expressive power, the traditional notion of proof of compliance could be generalised: Instead of asking “Does the set  $C$  of credentials prove that the request  $r$  complies with the local security policy  $\sigma$ ?” one might ask “Does the *information*  $I$  justify taking action  $a$  given security policy  $\sigma$ ?” or “among potential providers  $p_1, p_2, \dots, p_n$  who (if any) can I expect to supply the best service, given

their credentials, information  $I$  and local security policy  $\sigma$ ?” This would retain the rigorous flavour of trust management, yet generalise credentials, requests and policies.

As mentioned earlier, the natural separation of policy and mechanism inherent in the trust management approach fits naturally with the theoretical hierarchy of the SGUC challenge. Consider again the question “Does the information  $I$  justify taking action  $a$  given security policy  $\sigma$ ?” At the high level one could have languages for *specifying* security policies, and one would have formal semantics providing relations that specify which information justifies an action, e.g.,  $I, \sigma \models a$ ; at a lower level one would then have distributed, high-level programming languages and distributed programs *implementing* trust management engines. Since policies deal with specification it is natural that policies are written in declarative and high-level languages, e.g., logics. At a lower level, but still relatively high in the theoretical hierarchy, one would then have more operational models where a distributed process  $P$  could then define a judgement  $I, \sigma \vdash a$ ; process  $P$  would then be sound if  $I, \sigma \vdash a$  implies  $I, \sigma \models a$  and complete if the other implication holds.

One of our contributions is an example of how this can be done for a specific formal model of trust (see Chapter 5). We use a high-level model (based on domain theory) for trust policies and rules of decision making. At a lower, more operational level we have I/O automata (a formal model of distributed event systems), and we prove that the I/O automata correctly implement the high-level policies. Our example also illustrates a theory of trust that builds strongly on the Platform of theoretical computer science, combining domain theory and formal semantics of distributed systems with reasoning techniques originally developed for numerical methods.

Further work along these lines is needed. For example, while there is a significant body of research on building trust management systems that are probabilistic (e.g., systems that try to give probabilities of a GUC entity well-behaving in the next interaction given its past behaviour), again, to live up to the SGUC challenge, we must move from engineering to theory (and back again); from ad-hoc, common-sense algorithms to well-founded rigorous modelling and reasoning; from *simulations* to *theorems*. We shall return to this point when we consider potential future research on theories of trust in Chapter 3.



# Chapter 2

## Research Context

*If you want to make an apple pie from scratch, you must first create the universe.*

— Carl Sagan, *Cosmos* (1980).

The preceding chapter has identified trust management, specifically theories of trust, as a cornerstone in the SGUC Grand Challenge. Hence, it considers the *interface* between a theory of trust and the Challenge itself. In this chapter, we give an overview of the *internal* state-of-the-art within the field of trust management. The purpose is to provide a survey of existing research on trust within computer science, *focusing on that which is particularly relevant to the contributions of this dissertation*. This includes existing theoretical work on *models* of computational trust, significant ideas and some of the influential systems that have been developed.

### 2.1 Introduction

It is not our intention to survey the entire collection of works on trust in computer science; this is too comprehensive and there are already surveys of large extent, e.g., Grandison and Sloman (2000) [41] (dealing mostly with the traditional notion of trust), Jøsang et al. (2006) [47], Sabater and Sierra (2005) [92] and Jennings et al. (2004) [89] (dealing mostly with the human notion). Also the PhD thesis of Abdul-Rahman [1] contains a vast survey (mostly) of the human notion of trust in computer science, including insights from social sciences. Since our focus is trust management in the context of the SGUC challenge, we survey the models and systems that we consider the most relevant for addressing this challenge.

## 2.2 Credential-based Trust Management

### 2.2.1 Influential Systems

**Blaze & Feigenbaum et al.** Blaze et al. developed the traditional notion of trust management [10] and developed also the first prototype trust management system, PolicyMaker (See Blaze, Feigenbaum et al. (1999) for a good overview of PolicyMaker and traditional trust management [8]). The most general form of proof of compliance (POC) in PolicyMaker is undecidable and several natural restrictions are NP hard [11]. PolicyMaker considers a version of the POC problem which requires that all assertions are *monotonic* (assertions are fully programmable functions that are part of credentials). This leads to a restricted notion of proof of compliance which is decidable in polynomial time [11]. KeyNote [9, 7], the successor of PolicyMaker, restricts the language of assertions to a simple domain-specific language so that resource-usage is proportional to program size [8]. KeyNote is less general than PolicyMaker but has simpler syntax and semantics, and requires less computational power. Architectural tradeoffs between PolicyMaker and KeyNote is considered by Blaze, Feigenbaum and Keromytis [9]. A number of applications using PolicyMaker and KeyNote has also been developed [13, 12].

Suppose, given credentials  $C$ , request  $r$  and policy  $\sigma$ , that  $C, \sigma \not\vdash r$ , i.e., that  $C$  does *not* prove that  $r$  complies with  $\sigma$ . Suppose also that  $c$  is an additional credential such that  $C \cup \{c\}, \sigma \vdash r$ . It would be desirable that a trust management engine which is presented with only  $C, r$  and  $\sigma$  would be able to “fetch” the needed credential  $c$  and then proceed to verify that  $C \cup \{c\}, \sigma \vdash r$ . This feature is called distributed credential discovery. REFEREE [23] is able to do this as well as performing the appropriate cryptographic signature verification. Also, REFEREE can have non-monotonic policies and credentials. Rigorous specification and complexity analysis, such was done with the PolicyMaker-notion of POC, has not been done for REFEREE.

**Li & Feigenbaum et al.** Delegation Logic (DL) [66, 65, 64] and its monotonic version D1LP is a language for trust management credentials, policies and requests; it extends the logic programming language Datalog with expressive delegation constructs [66]. D1LP is implemented by translation in to ordinary logic programs which enables use of existing logic programming technology, e.g., Prolog. An important feature of DL and D1LP is that the notion of proof of compliance is well founded in the roots of logic. D1LP sup-

ports the concepts of authenticated attributes and decentralised attribute authority. *Decentralised attributes* allow an entity to assert that another has a certain attribute (i.e., satisfies a certain property, e.g., “being a university student”). D1LP is declarative, expressive and tractable (compliance checking is polynomial in the size of credentials, policies and requests). Li, Feigenbaum and Grosz argue that no previous trust management system (PolicyMaker, KeyNote and SKPI/SDSI) satisfies their requirements of expressive power (e.g., “attributes”), efficiency and clear well-founded semantics of POC (in DL/D1LP: this would be the model-theoretic semantics).

**Li & Mitchell et al.** The RT (role-based trust-management) framework is a family of languages for policies and credentials which combines the strengths of role-based access control and trust management [68, 70]. The RT framework consists of the languages together with an engine which works by translating credentials into Datalog rules, similarly to the DL languages. This enables POC checking in polynomial time. Apart from supporting role-based features, Li et al. argue that RT is more convenient than D1LP although both support attribute-based access control. RT supports concepts of intersection roles, manifold roles and delegation of role activation; these enhance the expressive power, compared to other frameworks, e.g., D1LP [70]. Furthermore, RT supports distributed credentials and distributed credential discovery [68]. RT is monotonic, and it has been argued that non-monotonicity requires complete information which is unrealistic in distributed systems [70, 66]. We consider the RT family of languages, specifically its extension  $RT_1^C$  with constraints [67] (also, see below), to be the state-of-the-art for credential-based trust management systems.

**Honourable mentions.** There is a range of other systems that have appeared in the literature that we do not consider here. Examples include SPKI/SDSI [34, 24, 69], SD3 [44] and Binder [31]; the latter two which also are based on Datalog. Etalle et al. [25] present a version of *RT* with non-monotonic features. Appel and Felten [4] use a logic-based approach to authentication. Their logic is higher-order and undecidable. In analogy with proof-carrying code, the requester must submit a proof that the request should be allowed; the proof is then efficiently *checkable* in the framework.

### 2.2.2 Fundamental Models

The traditional trust management approach was born from an engineering perspective: Important concepts were identified and prototype systems were

built. However, no foundational models were developed initially. Although the original notion of proof of compliance was subject to theoretical investigation and rigorous definition [11], the definition itself is also considered to be ad-hoc [64]. The notion of proof of compliance (and trust management in general) has developed and is now better founded on existing theory. We consider two formal models that give well-founded notions of proof-of-compliance.

**Constraint Datalog.** Li and Mitchell proposed an extension of Datalog which incorporates a notion of constraints as a promising operational foundation for trust management [67]. As we have seen, a number of existing trust management systems are “equivalent” to a subset of Datalog. Li and Mitchell argue that standard Datalog “(…) is not sufficiently expressive for fine-grained control of structured resources” [67]. They show how Datalog can remain tractable when extended with so-called linearly decomposable unary constraint domains, and that permissions associated with structured resources are expressible within this framework. Based on Li and Mitchell (2003) [67], we briefly recapture the model in the following.

A constraint domain consists of a set of object, e.g., numbers, and a language for “speaking about” those objects. A constraint domain is a triple  $\Phi = (\Sigma, \mathcal{D}, \mathcal{L})$  where  $\Sigma$  is a signature (in the sense of first-order logic),  $\mathcal{D}$  is a structure for  $\Sigma$  (i.e., a structure in the sense of first-order logic), and  $\mathcal{L}$  is a class of quantifier-free first-order formulas over  $\Sigma$ , called the primitive constraints.

For example, the domain of linear constraints is given by the following. Signature  $\Sigma$  has function symbols  $+$  and  $*$ , and predicates  $\{=, \neq, <, >, \leq, \geq\}$ . Primitive constraints are of the form  $c_1x_1 + c_2x_2 + \dots + c_nx_n\theta b$ , where  $c_i$  are constants,  $x_i$  are variables,  $\theta$  is a predicate and  $b$  is also a constant. The interpretation could be the reals with addition, multiplication and the usual relations.

A constraint (Datalog) rule has the form:

$$R_0(x_{0,1}, \dots, x_{0,k}) : -R_1(x_{1,1}, \dots, x_{1,k_1}), \dots, R_n(x_{n,1}, \dots, x_{n,k_n}), \psi_0$$

where the  $R_i$  are relation symbols,  $x_{i,j}$  are variables and  $\psi_0$  is a constraint in the set of variables in the rule. When  $n = 0$  the constraint rule is called a (constraint) fact. The process of generating new facts from old facts and rules requires a notion of quantifier elimination which is only possible for some constraint domains. An associated *least fixed point* algorithm generates a certain set of facts, called the constraint least fixed point of a *Datalog<sup>C</sup>*

program (a collection of (constraint) facts and rules). A constraint domain  $\Phi$  is said to be tractable when the evaluation of any *Datalog*<sup>C</sup> program with constraints in  $\Phi$  has polynomial time complexity in the size of the program (when the size of each rule is bounded by a fixed number which is often the sum of arities of the predicates in a rule).

$RT_1^C$  is a constraint-based extension of the  $RT_1$  language of the  $RT$  family of languages. It is an example of a trust management language based on constraint Datalog. Each statement of  $RT_1^C$  is translated into a rule of *Datalog*<sup>C</sup> with certain tractable domains, called linearly decomposable domains. Again, we refer to Li and Mitchell for concrete examples [67].

**A fixed point model.** Stephen Weeks presented a mathematical framework for trust management systems based on the existence of least fixed points of monotonic functions on complete lattices [100]. This gives a general semantic model for trust management systems and for the notion of proof of compliance; further, it is shown that the general framework can be instantiated to obtain a number of existing systems, e.g., KeyNote and SPKI.

In Weeks' framework a trust management system is expressed as a complete lattice  $(D, \preceq)$  of possible *authorisations*, a set of *principal names*  $\mathcal{P}$ , and a language for specifying so-called *licenses*. The lattice elements  $d, e \in D$  express the authorisations relevant for a particular system, e.g. access-rights, and  $d \preceq e$  then means that  $e$  authorises at least as much as  $d$ . An *assertion* is a pair  $a = \langle p, l \rangle$  consisting of a principal  $p \in \mathcal{P}$ , the *issuer*, and a monotonic function  $l : (\mathcal{P} \rightarrow D) \rightarrow D$ , called a *license*. In the simplest case  $l$  could be a constant function, say  $d_0$ , and then  $a$  states that  $p$  authorises  $d_0$ . In the general case the interpretation of  $a$  is the following: Given that all principals authorise as specified in the *authorisation map*,  $m : \mathcal{P} \rightarrow D$ , then  $p$  authorises as specified in  $l(m)$ . This means that a license such as  $l(m) = m(A) \vee m(B)$  expresses a policy saying “give the least upper bound in  $(D, \preceq)$  of what  $A$  says and what  $B$  says.” Weeks showed that a collection of assertions  $L = \langle p_i, l_i \rangle_{i \in I}$  gives rise to a monotonic function  $L_\lambda : (\mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$ , with the property that a coherent authorisation map representing the authorisations of the involved principals is given by the least fixed point,  $\text{lfp } L_\lambda$ .<sup>1</sup>

---

<sup>1</sup>If  $L$  contains two (or more) entries  $\langle p, l_1 \rangle$  and  $\langle p, l_2 \rangle$ , then the licenses are combined to a single assertion  $\langle p, l_1 \vee l_2 \rangle$ . Hence the collection  $L$  gives rise to a function  $L_\lambda : (\mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$  which maps an authmap  $m$  and a principal  $p$  to  $l(m)$  where  $l$  is the *unique* license with  $\langle p, l \rangle \in L$  (if no such entry exists the constant  $\perp$  license is used).

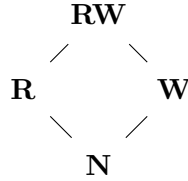


Figure 2.1: Example authorisation lattice.

For the sake of clarity, we present a simple example, taken from Weeks’ 2001-paper [100]. Consider  $D = \{\mathbf{N}, \mathbf{R}, \mathbf{W}, \mathbf{RW}\}$  with  $\mathbf{N}$  least,  $\mathbf{RW}$  greatest and  $\mathbf{R}$  and  $\mathbf{W}$  unrelated, i.e., the Hasse diagram in Figure 2.1. Lattice  $D$  represents file access rights for a particular principal Alice, i.e., Alice may read ( $\mathbf{R}$ ), write ( $\mathbf{W}$ ), both ( $\mathbf{RW}$ ) or Alice has no rights ( $\mathbf{N}$ ). A number of example licenses are given in Table 2.1, and fixed point computations are presented in Table 2.2. In the fixed point computation, the first example illustrates a chain of delegations, and the second example, two mutually recursive licenses.

In the framework, the compliance-checking problem can be elegantly specified as: “Given a set of assertions  $C$ , i.e.,  $C \subseteq \mathcal{P} \times ((\mathcal{P} \rightarrow D) \rightarrow \mathcal{P})$ , a request  $r \in D$  and a principal identity  $p \in \mathcal{P}$ , does  $r \preceq \text{lfp } C_\lambda(p)$  hold?” A great advantage of this approach is that existing theory of fixed points and algorithms for fixed point computation can be used in trust management engines. Weeks shows also how this generality can even lead to more efficient algorithms for compliance checking [100].

**Remarks.** The two formal models, i.e., Constraint Datalog and Weeks’ fixed point model, are related. Both models require that “policies” (licenses and rules) are monotonic, and both are based on fixed points. In fact, it should be simple to show that the fixed point semantics of Constraint Datalog can be obtained as a set-based instance of Weeks’ model: A constraint rule can be seen as a monotonic function mapping a set of facts to another set of facts.

### 2.3 Experience-based Trust Models

We use the term ‘experience-based’ to cover the models where an entity’s trust in another is based, in part, on past behaviour or evaluations of past

Table 2.1: Example licenses for permission to principal “Alice” [100].

License	Intended meaning
$\lambda m. \mathbf{W}$	Alice may write the file.
$\lambda m. m(\mathit{Bob})$	Alice may do whatever <i>Bob</i> allows.
$\lambda m. \mathbf{W} \vee m(\mathit{Bob})$	Alice may write and do whatever <i>Bob</i> allows.
$\lambda m. \mathbf{W} \wedge m(\mathit{Bob})$	Alice may write only if <i>Bob</i> allows so.
$\lambda m. \text{if } \mathbf{W} \preceq m(\mathit{Bob}) \text{ then } \mathbf{R} \text{ else } \mathbf{N}$	Alice may read if <i>Bob</i> says she may write.
$\lambda m. \text{if }  \{p \mid \mathbf{R} \preceq m(p)\}  \geq 2 \text{ then } \mathbf{R} \text{ else } \mathbf{N}$	Alice may read if at least two principals say so.

behaviour (similarly to the human notion of trust). This also covers many so-called *reputation systems* or *reputation-based* trust management systems, which are often used in peer-to-peer (P2P) and eCommerce applications.

The amount of literature on experience-based trust models, including reputation systems, has quickly grown very extensive. In our view, these systems are based on a few fundamental principles, and even fewer fundamental models. Hence, we do not claim to be complete: We focus on the fundamental models, and only *selected examples* of systems deploying those models.

### 2.3.1 General Principles

Consider a set  $\mathcal{P}$  of principal identities. From time to time, principals will interact in a pair-wise manner, and such interactions result in each principal observing a set of time-stamped events. In the following we make a number of simplifications, but stay general enough to capture most of the principles of existing experience-based systems: We assume that each time  $p$  interacts with another principal, say  $q \in \mathcal{P}$ , the interaction generates only a single event  $e$ , drawn from some set  $E$  of events (left unspecified here).

**A general formal model.** Let  $(T, \leq)$  be an totally ordered set of time-stamps, e.g.,  $T = \{0, 1, \dots\}$  for discrete time. Principal  $p$  records its interactions with other principals so that at each point in time,  $t_0 \in T$ , there is

Table 2.2: Two example fixed-point computations for permissions to principal “Alice” [100].

Name	License	Computation			Iteration
		<i>Bob</i>	<i>Carl</i>	<i>Dave</i>	
<i>Bob</i>	$\lambda m. \mathbf{W}$	<b>N</b>	<b>N</b>	<b>N</b>	(0)
<i>Carl</i>	$\lambda m. m(\textit{Bob})$	<b>W</b>	<b>N</b>	<b>N</b>	(1)
<i>Dave</i>	$\lambda m. m(\textit{Carl})$	<b>W</b>	<b>W</b>	<b>N</b>	(2)
		<b>W</b>	<b>W</b>	<b>W</b>	(3)
<i>Bob</i>	$\lambda m. \mathbf{W} \vee m(\textit{Carl})$	<b>N</b>	<b>N</b>		(0)
<i>Carl</i>	$\lambda m. \textit{if } \mathbf{W} \preceq m(\textit{Bob}) \textit{ then } \mathbf{R} \textit{ else } \mathbf{N}$	<b>W</b>	<b>N</b>		(1)
		<b>W</b>	<b>R</b>		(2)
		<b>RW</b>	<b>R</b>		(3)

a set  $Hist^p(t_0)$  consisting of triples  $(q, t, e)$  where  $q \in \mathcal{P}$ ,  $e \in E$ ; and  $t \in T$  satisfies  $t \leq t_0$ . To be clear, a triple  $(q, t, e) \in Hist^p(t_0)$  represents that: “In an interaction between  $p$  and  $q$ , principal  $p$  has observed event  $e$  at time  $t$ .” We write  $Hist_q^p(t)$  for the  $q$ -projection, i.e., the set of pairs  $(t', e)$  such that  $(q, t', e) \in Hist^p(t)$ .

At any point in time,  $t \in T$ , the sets  $(Hist^p(t) \mid p \in \mathcal{P})$  constitutes the basic or *direct data of an experience-based system at time  $t$* . When  $p$  needs to make a decision at time  $t$ , e.g., about a principal  $q$ , principal  $p$  does this based on information about the direct data of the system at time  $t$ . Usually, this information is incomplete: While  $p$  (often) knows  $Hist^p(t)$ , the sets  $Hist^r(t)$  for  $r \neq p$  may not be known exactly. This may be due to several reasons:  $p$  may only have  $Hist^r(t')$  for some  $t' < t$ ; when asked about  $Hist^r(t')$ ,  $r$  may lie; principal  $p$  may not be able to obtain any information about  $Hist^r(t')$ ; principal  $p$  may only see some abstracted version  $Abs(Hist^r(t'))$  of the  $r$ 's direct data; and any combinations of the above, etc.

**Experience-based systems.** Most experience-based systems work on some abstracted version of the direct data, denoted  $AbsHist^p(t)$ . Some systems are central, so that (abstract versions of) the direct data are stored on a global server, whereas other system are distributed. In the following we focus on models, not architectures (e.g., centralised vs. distributed). Given our general model, an experience-based system is designed by (i) choosing if and how to abstract (or aggregate) the sets  $Hist^p(t)$  to obtain the “abstract”

sets  $AbsHist^p(t)$ ; (ii) choosing if and how each principal  $p$  will obtain information about  $Hist^q(t)$  for  $q \neq p$ ; (iii) (optionally) choosing how principals combine personal data with the data of others; and (iv) designing an architecture and algorithms (possibly distributed) to implement the system. The optional step (iii) often works in the following way: Principal  $p$  computes for each other principal, say  $q$ , a “score” or “rating”,  $T_{pq} \in D$ , for some set  $D$  of possible scores. The score  $T_{pq}$  is usually computed from some of the abstract sets  $(AbsHist_q^r(t) \mid r \in I \subseteq \mathcal{P})$ , and represents  $q$ ’s trustworthiness (or reputation), seen from the point-of-view of  $p$ . Some systems have a uniform mechanism where  $T_{pq} = T_{p'q}$  for all  $p, p' \in \mathcal{P}$ , i.e.,  $q$  has a unique “global” score.

A common example of an abstraction is the following. At time  $t_0$ , principal  $p$  is interested in information about principal  $q$ . Each record  $(q, t, e)$  is evaluated as either ‘positive’ or ‘negative’, and time is ignored; hence,  $Hist_q^p(t_0)$  is abstracted to a pair consisting of the number of ‘positive’ interactions and the number of ‘negative’ interactions. Principals then obtain information about  $AbsHist_q^r(t)$  by asking a central repository. Sets of records  $(AbsHist_q^r(t) \mid r \in I \subseteq \mathcal{P})$  are combined into a single pair by adding-up the total number of ‘negative’ interactions, and similarly adding-up total number of ‘positive’ interactions. This example system is much like the eBay system (see Chapter 6).

In the following, we consider two types of systems: Non-probabilistic and probabilistic.

### 2.3.2 Non-probabilistic Approaches

#### Shmatikov and Talcott

We use the term ‘concrete’ reputation systems for experience-based systems where the sets  $Hist^p(t)$  undergo little or no abstraction [56]. Shmatikov and Talcott [94] define a concrete reputation system that is very close to our general formal model (in fact, our model was inspired by theirs). The system is centred around a notion of ‘licenses’. A license formalises restrictions and obligations, i.e., what principals *must* do and what they *cannot* do. Essentially, one can view a license  $l$  as specifying three functions:  $l.\mathbf{permits}$ ,  $l.\mathbf{violated}$  and  $l.\mathbf{done}$ . The functions  $l.\mathbf{violated}$  and  $l.\mathbf{done}$  both take as input a set  $Hist_q^p(t)$  and returns a boolean; the interpretation is that  $l.\mathbf{violated}$  returns *true* when the history  $Hist_q^p(t)$  violates the license (e.g., neglected obligation); similarly,  $l.\mathbf{done}$  returns *true* if license  $l$  has expired in the history. The function  $l.\mathbf{permits}$  takes as input an event  $e$  and a history

$Hist_q^p(t)$ , it outputs *true* only if adding  $(q, t_0, e)$  to  $Hist_q^p(t)$  is permitted in the license. Licenses are completely programmable so that, in principle, any computable function can be used to define each of the three functions. This gives much flexibility in defining licenses.

Licenses are used when principals want to access resources owned by other principals. The owner of resources specify, for each resource  $r$ , method  $r.\mathbf{useOk}$ , which takes as input a license  $l$  and an event history, say  $Hist_q^\times(t)$ , and output a boolean: *true* if the agent  $q$  can access  $r$  with license  $l$  given history  $Hist_q^\times(t)$ . The resource owner can specify this method using any computable function, but typically the owner would check if the license permits this use and if it has expired, etc. This gives a reputation system where decision are made based on exact criteria on past histories. Hence, if one can reason about the license methods and the  $\mathbf{useOk}$  method, then it is possible to reason about the security guarantees provided by the system.

### Kamvar et al.

Kamvar et al. [49] present a reputation system for P2P systems, called EigenTrust (also known as EigenRep), based on the existence of stationary distributions for Markov Chains. In contrast to the framework of Shmatikov and Talcott, EigenTrust abstracts away much information: Events  $(q, t, e)$  are evaluated as either ‘satisfactory’ or ‘unsatisfactory’; time is ignored; each peer  $i$  computes a value for other peers  $j$  as  $s_{ij} = sat(i, j) - unsat(i, j)$  (that is, the number of interactions between  $i$  and  $j$  where  $i$  rated  $j$ ’s performance as ‘satisfactory’ minus the number of ‘unsatisfactory’ ones); and finally, these values are then normalised for each peer  $j$  (by comparison to other peers performance):  $c_{ij} = \frac{\max(s_{ij}, 0)}{\sum_j \max(s_{ij}, 0)}$ . The normalised values define the abstracted histories  $AbsHist_j^i(t)$ , which give rise to a Markov chain (given by  $[c_{ij}]$ ) that has a stationary distribution  $(t_j)_{j \in \mathcal{P}}$ . This distribution is computed using an iterative, synchronous algorithm; the value  $t_j$  then represents the “global score” of  $j$ ,  $T_{ij}$  (uniformly for all  $i$ ).<sup>2</sup>

One problem with EigenTrust is that no meaningful semantic interpretation of the value  $t_j$  exists (only “the larger the better.”) Furthermore, temporal aspects are ignored, and information is thrown away with the normalisation. Hence, it is hard to do formal reasoning about the system, e.g., what is guaranteed about the behaviour of  $j$  if  $t_j$  is greater than, say, .9?

---

<sup>2</sup>This works in a manner similar to Google’s Pagerank.

### Honourable mentions

With his PhD thesis, Stephen Marsh was among the first to formalise a computational human notion of trust in computer science [74]. Abdul-Rahman & Hailes [2, 1] was also among the first to consider a simplified practical model similar to Marsh’s. Xiong and Liu present PeerTrust [106] featuring a complex trust metric, but which only has an intuitive justification. In our opinion, these systems exhibit the same problems as EigenTrust: Rigorous reasoning about the past behaviour seems impossible given only the abstracted information.

### 2.3.3 Probabilistic Approaches

The probabilistic systems work by assuming a particular probabilistic model, say  $\lambda$ , for the behaviour of principals. The goal is to predict the behaviour of principals in future interactions, given their behaviour in past interactions and the model  $\lambda$ , i.e., computing a probability  $P(\text{‘next’} \mid \text{‘past’}, \lambda)$ . The abstractions, i.e.,  $AbsHist(t)$ , are then chosen to be as efficient as possible while preserving as much information as is relevant with respect to the model. For example,  $\lambda$  may specify that each principal (intrinsically) is either ‘good’ or ‘bad’, and that interaction with ‘good’ principals always results in event  $e$ , whereas interaction with ‘bad’ principals always results in event  $f$ . In this model, one only needs to interact with a principal once to know if he is a ‘good’ type or ‘bad’ type. Hence, the sets  $AbsHist_q^p(t)$  need only have three values to preserve sufficient information: ‘good’, ‘bad’ or ‘unknown’.

### Aberer & Despotovic

Despotovic et al. [29, 30] propose a probabilistic system and an estimation algorithm based on maximum likelihood. Aberer & Despotovic assume that peers interact with each other in a binary way: In each interaction they can either ‘be honest’ or ‘cheat’. Furthermore, peers can report to other peers on past behaviour (and they are allowed to lie in their reports).

**The model.** The probabilistic model of Aberer and Despotovic, say  $\lambda_{AD}$ , assumes that each principal  $j \in \mathcal{P}$  is “probabilistic” in the sense that there is a fixed probability  $\theta_j \in [0, 1]$  of peer  $j$  acting honestly in any interaction. Note, this assumes that  $j$  is always honest with probability  $\theta_j$  independently of any other information we might have (e.g., the time, the past, etc.). The parameters,  $\theta_j$ , are unknown and the goal is to estimate them. Furthermore, each principal  $k \in \mathcal{P}$  can report on its past interactions with  $j$ ; it is assumed

that  $k$ 's report is also probabilistic so that the probability of observing a report  $y_k \in \{0, 1\}$  ('0' means cheated, '1' means honest) from principal  $k$  is given by

$$P(Y_k = y_k \mid \theta_j, l_k) = \begin{cases} l_k(1 - \theta_j) + (1 - l_k)\theta_j & \text{if } y_k = 1; \\ l_k\theta_j + (1 - l_k)(1 - \theta_j) & \text{if } y_k = 0 \end{cases}$$

where  $l_k$  (like  $\theta_j$ ) are fixed parameters specifying the probability of  $k$  submitting a false report. Hence for each principal  $j$ , there are two parameters that probabilistically decides its behaviour:  $\theta_j$  and  $l_j$ .

**The system.** Let us write  $AbsHist_j^\times(t)$  for the collection of information that a particular principal has about  $j$ . In the system, this collection consists of a number of reports  $((y_1, p_1), (y_2, p_2), \dots, (y_m, p_m))$  where  $y_i \in \{0, 1\}$ ,  $p_i \in \mathcal{P}$ , and  $(y_i, p_i)$  means that principal  $p_i$  has filed report  $y_i$  (we do not consider here how reports are obtained). Hence, time is abstracted away and events are "rated" in a binary fashion.

Now given  $AbsHist_j^\times(t) = \mathbf{Y} = ((y_1, p_1), (y_2, p_2), \dots, (y_m, p_m))$  of independent reports, the so-called likelihood function is:

$$L(\theta_j, l) = P(\mathbf{Y} \mid \theta_j, l, \lambda_{AD}) = P(Y_1 = y_1 \mid \theta_j, l_{p_1}) \cdots P(Y_m = y_m \mid \theta_j, l_{p_m})$$

(note this expression depends also on  $l$  which is not clear from the authors presentation [29]). Given current estimates for  $l$  and the data  $\mathbf{Y}$  the goal is to estimate the behaviour of principal  $j$ , i.e., to estimate  $\theta_j$ .

The system uses a maximum likelihood procedure which seeks to find a  $\theta_j$  which maximises the likelihood expression. In the computation of likelihood function, estimates for the  $l_k$  are based on past interactions, but it is unspecified exactly how these are computed. Authors also present an approach based on normal distributions instead of the fixed  $\theta_j$ 's. Similarly, the maximum likelihood techniques are used to estimate the parameters of the normal distribution.

### Beta-based Bayesian Systems.

Jøsang et al. [46] and Mui et al. [77] were among to first to (independently) develop reputation systems based on a Bayesian probabilistic approach with beta priors. In the following we recall the beta distribution and explain the underlying theoretical model for the beta-based reputation systems.

**The beta distribution.** The beta family  $Beta(\cdot, \times)$  is a parameterised collection of continuous probability density functions (pdfs) defined on the interval  $[0, 1]$ . There are two parameters  $\alpha > 0$  and  $\beta > 0$  that select a specific beta distribution from the family. The pdf  $Beta(\alpha, \beta)$  is given by

$$f(\theta | \alpha, \beta) = \frac{1}{\mathbf{B}(\alpha, \beta)} \theta^{\alpha-1} (1 - \theta)^{\beta-1} = \frac{1}{\int_0^1 dt t^{\alpha-1} (1 - t)^{\beta-1}} \theta^{\alpha-1} (1 - \theta)^{\beta-1}$$

where  $\mathbf{B}$  is the beta function, and  $\mathbf{B}(\alpha, \beta)^{-1}$  is a normalising constant. The expected value and variance are given by

$$\mathbf{E}_{f(\theta|\alpha,\beta)}(\theta) = \frac{\alpha}{\alpha + \beta}, \quad \sigma_{f(\theta|\alpha,\beta)}^2(\theta) = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

The beta distributions provide a so-called *family of conjugate prior distributions* for the family of distributions for Bernoulli trials. To explain the notion of conjugate priors, consider the general problem of estimating a parameter  $\theta$  given some data  $x$  and background information  $I$ . Let  $H_\theta$  be some hypothesis about parameter  $\theta$ . The Bayesian approach (see the excellent book of Jaynes [43]), is to compute the posterior  $P(H_\theta | xI)$  (i.e., the probability after seeing the data) from the prior  $P(H_\theta | I)$  (the *a priori* probability, given only information  $I$ ) and the likelihood function  $H_\theta \mapsto P(x | H_\theta I)$ , using Bayes' Theorem:

$$P(H_\theta | xI) = P(H_\theta | I) \frac{P(x | H_\theta I)}{P(x | I)}$$

Different priors  $P(H_\theta | I)$  may make this probability more or less difficult to calculate, but certain choices of the prior lead to the the posterior  $P(H_\theta | xI)$  having the same algebraic form as the prior. Now, a family of conjugate prior distributions for the family of distributions  $H_\theta \mapsto P(x | H_\theta I)$  is a collection of distributions such that when ever the prior  $P(H_\theta | I)$  belongs to the family, the posterior  $P(H_\theta | xI)$  is also in that family (one might say that the family Bayes-closed, i.e., is closed under the application of Bayes' Theorem).

**What has this got to do with trust and reputation?**<sup>3</sup> Consider again sequences of independent experiments with binary outcomes, each yielding one of the outcomes with some fixed probability (i.e., Bernoulli trials). In

---

<sup>3</sup>The following justification for using the Bayesian beta-based approach in reputation systems represents our personal explanation; it is not explicitly written out in such detail in the papers describing beta-systems [46, 77, 15, 98], and, hence, the authors may have different opinions.

systems where principal-interactions consists of binary outcomes (or where interactions are rated on a binary scale, e.g., ‘cooperate’ or ‘defect’; ‘success’ or ‘failure’), one can model repeated interaction (or repeated ratings) as Bernoulli trials. Let us be more precise: Let  $p, q \in \mathcal{P}$  be principals, and assume that  $p$  and  $q$  have interacted  $n$  times; that in each interaction  $q$  takes an action; and that the whole interaction is given a binary rating by  $p$  (which depends only on  $q$ ’s action). Let  $X_i^{pq} \in \{0, 1\}$ , for  $i = 1, 2, \dots, n$ , be  $p$ ’s rating (i.e., subjective evaluation) of the  $i$ th interaction with  $q$ . Let us assume that principal  $q$ ’s behaviour is so that there is a fixed parameter such that at each interaction we have, *independently of anything we know about other interactions*, the probability  $\theta$  for a ‘success’ and therefore probability  $1 - \theta$  for ‘failure.’ This gives us a probabilistic model, and let us call it the *beta model*. Note, this is like the model  $\lambda_{AD}$  of Aberer and Despotovic, except for the parameters  $l_k$  for  $k \in \mathcal{P}$ . Let  $\lambda_{\mathbf{B}}$  denote a formal proposition representing the beta model, i.e., the assumptions about the behaviour of  $q$ ; also, let  $\theta \in [0, 1]$  be the parameter determining success in the  $i$ th trial. Finally, let  $\mathbf{X}$  be the conjunction of statements of the form

$$Z_i \equiv (X_i^{pq} = 0); \text{ or } Z_i \equiv (X_i^{pq} = 1)$$

so  $\mathbf{X} = \bigwedge_{i=1}^n Z_i$ , and let there be  $f$  statements of the first form and  $s$  statements of the second form (there is one statement for each  $i$ , so  $s + f = n$ ). Then, by definition of our model  $\lambda_{\mathbf{B}}$ , we have the following likelihood.

$$P(\mathbf{X} \mid \theta \lambda_{\mathbf{B}}) = \prod_{i=1}^n P(Z_i \mid \theta \lambda_{\mathbf{B}}) = \theta^s (1 - \theta)^f$$

Hence, we can obtain the posterior pdf as

$$\begin{aligned} g(\theta \mid \mathbf{X} \lambda_{\mathbf{B}}) &= g(\theta \mid \lambda_{\mathbf{B}}) \frac{P(\mathbf{X} \mid \theta \lambda_{\mathbf{B}})}{P(\mathbf{X} \mid \lambda_{\mathbf{B}})} = g(\theta \mid \lambda_{\mathbf{B}}) \frac{\theta^s (1 - \theta)^f}{\int_0^1 d\theta P(\mathbf{X} \mid \lambda_{\mathbf{B}} \theta) g(\theta \mid \lambda_{\mathbf{B}})} \\ &= g(\theta \mid \lambda_{\mathbf{B}}) \frac{\theta^s (1 - \theta)^f}{\int_0^1 d\theta \theta^s (1 - \theta)^f g(\theta \mid \lambda_{\mathbf{B}})} \end{aligned}$$

(where  $g(\theta \mid \lambda_{\mathbf{B}})$  is the prior pdf for  $\theta$  (see Jaynes [43])). If we choose the prior pdf  $g(\theta \mid \lambda_{\mathbf{B}})$  to be *Beta*( $\theta \mid \alpha_0, \beta_0$ ) (e.g.,  $(\alpha_0, \beta_0) = (1, 1)$  gives the

uniform distribution), then we can compute the posterior:

$$\begin{aligned}
g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}}) &= g(\theta \mid \lambda_{\mathbf{B}}) \frac{\theta^s(1-\theta)^f}{\int_0^1 d\theta \theta^s(1-\theta)^f g(\theta \mid \lambda_{\mathbf{B}})} \\
&= \frac{1}{\mathbf{B}(\alpha_0, \beta_0)} \frac{\theta^{\alpha_0-1}(1-\theta)^{\beta_0-1} \theta^s(1-\theta)^f}{\int_0^1 d\theta \theta^s(1-\theta)^f \frac{1}{\mathbf{B}(\alpha_0, \beta_0)} \theta^{\alpha_0-1}(1-\theta)^{\beta_0-1}} \\
&= \frac{1}{\int_0^1 d\theta \theta^{s+\alpha_0-1}(1-\theta)^{f+\beta_0-1}} \theta^{\alpha_0+s-1}(1-\theta)^{\beta_0+f-1} \\
&= \frac{1}{\mathbf{B}(\alpha_0 + s, \beta_0 + f)} \theta^{\alpha_0+s-1}(1-\theta)^{\beta_0+f-1}
\end{aligned}$$

which means that  $g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})$  is  $Beta(\theta \mid \alpha_0 + s, \beta_0 + f)$ .

**The predictive probability.** Now let  $Z_{n+1} \equiv (X_{n+1}^{pq} = 1)$ , i.e., the statement that the  $(n+1)$ 'st interaction is rated as a 'success', then  $P(Z_{n+1} \mid \mathbf{X}\lambda_{\mathbf{B}})$  is a predictive probability: Given no direct knowledge of  $\theta$ , but only past evidence ( $\mathbf{X}$ ) and the model ( $\lambda_{\mathbf{B}}$ ), then  $P(Z_{n+1} \mid \mathbf{X}\lambda_{\mathbf{B}})$  is the probability that the next interaction will be a "good" one. We can compute it as follows.

$$\begin{aligned}
P(Z_{n+1} \mid \mathbf{X}\lambda_{\mathbf{B}}) &= \int_0^1 d\theta P(Z_{n+1} \mid \mathbf{X}\lambda_{\mathbf{B}}\theta) g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}}) \\
&= \int_0^1 d\theta \theta g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}}) \\
&= \mathbf{E}_{g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})}(\theta)
\end{aligned}$$

Now recall the expectation of beta distributions; then

$$P(Z_{n+1} \mid \mathbf{X}\lambda_{\mathbf{B}}) = \mathbf{E}_{g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})}(\theta) = \frac{\alpha_0 + s}{\alpha_0 + s + \beta_0 + f} \quad (2.1)$$

since  $g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})$  is  $Beta(\theta \mid \alpha_0 + s, \beta_0 + f)$ .

To summarise, given the assumptions of the beta model, one can compute the probability of a success in the next interaction as the expectation of the beta pdf  $g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})$  which results via Bayesian updating given the past history  $\mathbf{X}$ . Hence, the beta based systems (that deploy the technique we have described here) are mathematically well-founded on probability theory.

**The concrete systems.** Jøsang et al. [46], Mui et al. [77], Buchegger et al. [15], Jennings et al. [98] all present systems based on the beta model. Buchegger et al. and Jennings et al. also propose mechanisms for dealing with lying reputation sources. Technically, all the systems work by maintaining the two parameters  $(\alpha, \beta)$  of the current pdf  $g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})$ . However, the systems (except for Mui et al. and Jennings et al.) deviate from the model  $\lambda_{\mathbf{B}}$  in the following sense: The parameters  $(\alpha, \beta)$  are adjusted as time passes, e.g., Jøsang uses exponential decay where  $\alpha$  and  $\beta$  are multiplied by a constant  $0 < u < 1$  each time parameters are updated (or a fixed time limit is exceeded). The intuition is that somehow information about more recent interactions should be considered more important than information about older interactions.

We would like to emphasise the following point, as we shall return to it later: (a) Under the assumption of the beta model ( $\lambda_{\mathbf{B}}$ ) there is a unique correct probability of success in the next interaction given a history of past interactions (Eqn. (2.1)); (b) any computation which obtains a different result is either incorrect (with respect to the model) or is *really* dealing with different behavioural assumptions about the entity in question. Of course, Jøsang and Buchegger, when introducing decay functions, have in mind a more complex time-dependent model, e.g., there might be a range of parameters  $(\theta_i)_{i \in I}$  such that success at time  $i$  is determined by  $\theta_i$ , and with some relation between the  $\theta_i$ 's. Unfortunately, neither Jøsang et al. or Buchegger et al. are explicit about their new more complex model, and their systems are based on intuitive (yet ad-hoc) exponential decay functions.

### Honourable mentions

Several models are based on a notion of “belief theory” which is related to probability theory: Yu et al. developed a distributed reputation system [107], and Jøsang developed the subjective logic of opinions [45]. Indeed, the subjective logic is closely linked to the probabilistic beta model [45].

There are a number of “economic” reputation-system models based on the theory of games, e.g., a “reputation effect” occurs in rational strategies when modelling interaction as a finitely repeated Prisoner’s Dilemma game [52, 102]. For a good overview of this area, see the work of Dellarocas [27, 28]. We do not discuss these models here as they are decision-theoretic extensions of the probabilistic models, featuring game-theoretic notions of utility and strategies (e.g., Dash et al. [26]). We do believe that there is potential future research in this area, but as of now, we stay away from this area as we believe that the existing probabilistic models for trust and

reputation must be better understood before we can move to distributed economic models. The notion of (computational) mechanism design is also relevant for this area. For mechanism design see Papadimitriou [84], and Feigenbaum and Shenker [36]; with respect to trust and mechanism design, see Dash et al. [26].

## 2.4 A Hybrid Model

In the following we present a “hybrid” model: It is not really a credential-based model (although it may be instantiated to obtain certain credential-based systems), and on the other hand, it is not really an “experience-based” model (although it may be instantiated to certain simple experience-based system). Some of our contributions are based on this framework (cf. Chapter 5), so we present it in some detail.

### 2.4.1 The Trust Structure Framework

Carbone et al. [19, 18] define the trust structure framework, a formal model for trust based on the work of Weeks, but focusing on information rather than authorisation. The trust structure framework also considers a set  $\mathcal{P}$  of principal *identities* which assign certain degrees of trust to each other. These degrees are drawn from a set  $D$  of possible trust values, which is a parameter of the framework, with different instantiations in different applications. A *trust structure* is a triple  $T = (D, \sqsubseteq, \preceq)$  consisting of a set  $D$  of such trust values, ordered by two partial orderings: the trust ordering ( $\preceq$ ) and the information ordering ( $\sqsubseteq$ ). Intuitively,  $c \preceq d$  means that  $d$  denotes at-least as high a trust degree as  $c$ . Instead, the information ordering introduces a notion of refinement;  $c \sqsubseteq d$  is intended to mean that  $c$  can be refined into  $d$ . For  $D = \{\text{high}, \text{mid}, \text{low}, \text{unknown}\}$ , one could have,  $\text{low} \preceq \text{mid} \preceq \text{high}$ , perhaps  $\text{low} \preceq \text{unknown} \preceq \text{high}$ , and  $\text{unknown} \sqsubseteq \text{low}, \text{mid}, \text{high}$ ; whereas  $\text{high}, \text{low}$  and  $\text{mid}$  would be unrelated by  $\sqsubseteq$ .

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

Note that trust policies  $\pi_p$  map *global* trust states to *local* trust states, and hence  $p$ 's policy may depend on other principals' policies.

**Security decisions.** When a principal  $v$  needs to make a security decision about another principal  $p$ , this decision is made depending on the ideal trust value for  $p$ , i.e.,  $\overline{\text{gts}}(v)(p)$  (which is determined uniquely by the collection of all policies, as shown in the next paragraph). Let us define a (trust-based) *security policy* as a function  $\sigma : \mathcal{P} \times D \rightarrow \{\perp, \top\}$ , with the following interpretation: If  $v$  has security policy  $\sigma$ , then  $v$  allows interaction with a principal  $p$  if-and-only-if the ideal global trust state satisfies  $\sigma(p, \overline{\text{gts}}(v)(p)) = \top$ . An important class of security policies are the monotonic policies:  $\sigma$  is *monotonic* if for all  $d, d' \in D$  with  $d \preceq d'$  we have  $\sigma(p, d) \Rightarrow \sigma(p, d')$  for all  $p \in \mathcal{P}$ . Many natural security policies are monotonic; for example, the simple threshold policies: a threshold policy is a policy  $\sigma$  of the form  $\sigma(p, d) = \top \iff t_p \preceq d$  for some threshold  $t_p \in D$ .

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

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

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

The trust structure framework solves this problem by turning to simple domain theory. A crucial requirement in the trust-structure framework is that the information ordering  $\sqsubseteq$  makes  $(D, \sqsubseteq)$  a complete partial order (cpo) with a least element (this element is denoted  $\perp_{\sqsubseteq}$ , and can be thought of as a value representing “unknown”). The framework then requires that all policies  $\pi_p : \text{GTS} \rightarrow \text{LTS}$  be *information continuous*, i.e. continuous with respect to  $\sqsubseteq$ . Since this implies that  $\Pi_\lambda$  is also information-continuous, and since  $(\text{GTS}, \sqsubseteq)$  is a cpo with bottom, standard theory [104] tells us that  $\Pi_\lambda$  has a (unique) least fixed-point which we denote  $\text{lfp}_{\sqsubseteq} \Pi_\lambda$  (or simply  $\text{lfp} \Pi_\lambda$ ):

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

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

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

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

$$\begin{aligned} \pi_R(m)(x) &= \begin{cases} \pi_A(m)(S) \vee (0, 0) & \text{for } x = S \\ (0, \infty) & \text{for } x \neq S \end{cases} \\ \pi_A(m)(x) &= \begin{cases} \pi_B(m)(S) \sqcup (4, 2) & \text{for } x = S \\ (0, 0) & \text{for } x \neq S \end{cases} \\ \pi_B(m)(x) &= \begin{cases} \pi_A(m)(S) \sqcup (6, 1) & \text{for } x = S \\ (0, 0) & \text{for } x \neq S \end{cases} \end{aligned}$$

The constants, e.g.,  $(4, 2)$ , is meant to represent local data obtained by the principals via past interactions, i.e.,  $A$  has interacted 6 times with  $S$  for which four interactions were "good" and two were "bad." It is not hard to see that both  $(T_{MN}, \preceq)$  and  $(T_{MN}, \sqsubseteq)$  are lattices, and that the joins are given by the following formulas: for any  $(m, n), (m', n') \in T_{MN}$  we have:

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

and

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

Table 2.3: Example centralised fixed-point computation.

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

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

Let us illustrate the *synchronous* or *central* least fixed-point computation. This is described by Table 2.3 containing the “synchronous” entries of  $\Pi_{\lambda}^i(\perp_{\square})$  (i.e.,  $\perp_{\square}, \Pi_{\lambda}(\perp_{\square}), \Pi_{\lambda}(\Pi_{\lambda}(\perp_{\square})), \dots$ ), but only for principal S. In the table, column  $x$  of row  $i + 1$  is obtained by applying policy  $\pi_x$  for S to row  $i$ , e.g., the value (4, 0) in column R of row 2 is obtained by  $\pi_R$  and row 1, as illustrated by the following informal “calculation:”

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

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

## 2.5 Concluding Remarks

We have presented what we perceive to be the most relevant models and ideas in current research on trust management with regard to answering the SGUC challenge. In our view, one crucial requirement with regard to the challenge is that the systems should incorporate more general information than “just” credentials while remaining amenable to rigorous reasoning. The system of Shmatikov and Talcot satisfies this requirement to some extent, but no general reasoning techniques are available since the crucial functions are arbitrary computable functions. There is a parallel here to the first *credential-based* trust management systems: In PolicyMaker assertions were completely programmable which gave much flexibility but made reasoning hard. Later systems restricted the expressive power in favour of supporting well-founded semantics and reasoning techniques. Similarly, our logical framework for reputation systems (Chapter 6) may be seen as an attempt

to support reasoning in a model similar to that of Shmatikov and Talcott by restricting the expressive power of policies.

A clear advantage of the probabilistic approach is that rigorous probabilistic reasoning about principals' behaviour is possible. The disadvantage, of course, is that such reasoning is only valid in actual systems where the assumptions are not violated. For example, the beta-based systems are rigorous in the sense that they are based on a formal model ( $\lambda_{\mathbf{B}}$ ), and that one can prove that principals compute the exact probability of 'success' in the next interaction, given their present information; this can be considered a security or correctness guarantee. However, the basic assumptions of the beta systems are unrealistic in many applications: The behaviour of principals may change depending on time and on their present information; describing it by a single fixed parameter ( $\theta$ ) does not suffice. Furthermore, only binary outcomes are assumed, and often, the exact meaning of 'success' or 'failure' is not clear and may vary between principals.



# Chapter 3

## Outlook and Extended Abstracts of Contributions

*The purpose of models is not to fit the data but to sharpen the questions.*

— Samuel Karlin, 1983.

This chapter presents our contributions in a *non-technical* way, emphasising novel contributions and ideas as well as weaknesses; the full papers are presented in Part II. The contributions are centred around three topics: The SECURE trust model (cf., Chapter 4), operational semantics and approximation techniques for the trust structure framework (cf., Chapter 5), and a logical approach to concrete reputation systems (cf., Chapter 6). To conclude the first part of this dissertation, we reflect on possible future research directions for achieving a coherent theory of trust, compatible with the SGUC challenge.

### 3.1 The SECURE Trust Model

The SECURE project (Secure Collaboration among Ubiquitous Roaming Entities) investigated the design of security mechanisms for global computing based on (the human notion of) trust [16]. The heart of the SECURE project is a computational model of trust, which provides the basis for reasoning about trust and for the deployment of trust-based security policies [17]. The trust model used in SECURE is based on the trust structure framework ([16, 19], see also Chapter 2), but specialised to use a specific

class of trust structures. As we shall explain more precisely below, the contributions described in this section consist of the specification of this special class of trust structures, together with probabilistic reasoning techniques.

In SECURE, each principal  $p$  has its own decision-making framework which is invoked when an application needs to make a decision involving another principal. The decision-making framework contains three primary components: The risk engine, the trust engine, and the collaboration monitor. At the most abstract level, the collaboration monitor records the behaviour of principals with whom  $p$  interacts. This information together with a trust policy defines how  $p$  assigns trust values to each other principal. The trust information, in turn, serves as a basis for a risk analysis of each interaction. In fact, an important part of the SECURE approach is the explicit modelling of *risk* as well as trust.

In SECURE, each interaction is modelled as having a set of possible outcomes; the outcome that *actually* occurs in an interaction depends on the behaviour of  $q$ . The (perceived) risk of each outcome depends on the (perceived) probability of the outcome when interacting with  $q$ , and the intrinsic cost (or benefit) of the outcome.<sup>1</sup> Costs are not always known exactly; hence, in SECURE, costs are represented by probability density functions on some range of cost values, typically, an interval  $[c_0, c_1] \subseteq \mathbb{R}$ .

Since outcomes of interactions depend on the behaviour  $q$ , the decision of how to interact is based on the trust in  $q$ . Notice that in this set-up, it is necessary that the trust value for  $q$  carries enough information that estimation of the likelihood of each of the outcomes is possible. This is the reason that one cannot use an arbitrary trust structure,  $T = (D, \sqsubseteq, \preceq)$ : How would one convert a trust value  $d \in D$  for principal  $q$  into a probability distribution on outcomes (and how are outcomes even modelled in  $T$ )? Of course, if such probabilistic estimation is possible, one may start reasoning about risk, e.g., the *expected* cost of an interaction. For example, suppose that principal  $p$  can interact with principal  $q$ , and that such an interaction has three mutually exclusive and exhaustive potential outcomes  $o_1, o_2$  and  $o_3$ , with costs  $c_1, c_2$  and  $c_3$ , respectively. Suppose also that, based on its past interactions with  $q$ , principal  $p$  assigns probability  $p_1$  to outcome  $o_1$ ; the risk associated with outcome  $o_1$  is then  $c_1 p_1$ . The expected cost of the interaction would be  $c_1 p_1 + c_2 p_2 + c_3 p_3$ .

---

<sup>1</sup>The term cost should be understood more generally as cost or *benefit*. If costs are represented as non-negative numbers, one might represent benefits as negative numbers.

**Motivation.** Recall that the trust structure framework is parameterised by a trust structure  $T = (D, \sqsubseteq, \preceq)$ , and that trust policies (semantically) are functions:  $\pi : (\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$ , where  $\mathcal{P}$  is the set of principals. The unique global trust state associated with a collection of policies,  $\Pi$ , is the least fixed point of  $\Pi_\lambda$ ; hence, its type is  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . Principal  $p$ 's trust in  $q$  is represented as the element,  $\text{lfp } \Pi_\lambda(p)(q)$ , of  $D$ . Initially, the SECURE trust model was identical to the trust structure framework [16, 19]. However, as mentioned, it was later found that having an arbitrary trust structure  $(D, \sqsubseteq, \preceq)$  is too abstract to derive probabilities in a general and meaningful way. In order to do this, we need a model which explicitly considers a structure,  $Out$ , modelling the possible outcomes. Furthermore, the model must associate each outcome  $o \in Out$  with an object representing the past evidence for that particular outcome. This means that we would like a trust structure  $T = (D, \sqsubseteq, \preceq)$  where  $D$  has the general form  $D = Out \rightarrow Ev$  where  $Ev$  represents such ‘‘evidence.’’ Further, we would like a general procedure to convert trust values,  $t \in Out \rightarrow Ev$  into probabilities of outcomes: Intuitively,  $P(o | t)$  for  $o \in Out$ , should represent the probability of outcome  $o$  in a future interaction with  $q$ , given the trust information  $t$  about  $q$ .

**Outcomes.** In Chapter 4 we propose Event Structures [103, 83] for modelling outcomes in SECURE. An event structure consists of a set of events,  $E$ , and two binary relations  $\#$  (conflict) and  $\leq$  (causality) on  $E$ . The relations specify which events can occur together and in which order such events can occur. We take an example from Chapter 4 to illustrate the idea: the event structure in Figure 3.1 could model a small scenario where a principal may ask a bank for the transfer of electronic cash from its bank account to an electronic wallet. After making the request, the principal observes that the request is either rejected or granted. After a successful transaction, the principal could observe that the cash sent in the transaction is forged or perhaps run an authentication algorithm to establish that it is authentic. Also, the principal could observe a withdrawal from its bank account with the present transaction's id, and this withdrawal may or may not be of the correct amount.

Notice, the use of conflict and causality in Figure 3.1 which represents knowledge about the structure of the interaction context, for example, that one cannot observe both `authentic` and `forged` (these are in conflict), and that `correct` cannot be observed unless `grant` has been first observed (causality).

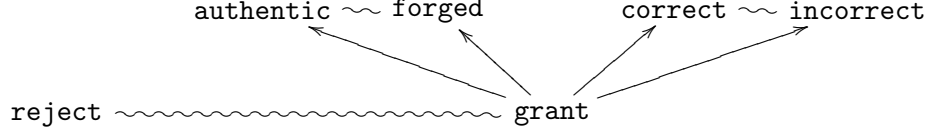


Figure 3.1: An example event structure. The curly lines  $\sim$  describe the immediate conflict relation and pointed arrows, the causality relation.

Event structures can be used to model protocols for interaction. At each point in time during the run of such a protocol, each principal has observed a set of events  $x \subseteq E$ ; this set, which is conflict-free and respects causality, is called a *configuration* of the event structure. The set of configurations of an event structure  $ES$  is denoted  $\mathcal{C}_{ES}$ . In our example, the following are examples of configurations:  $\emptyset$ ,  $\{\text{reject}\}$ ,  $\{\text{grant}, \text{forged}\}$  and  $\{\text{grant}, \text{authentic}, \text{correct}\}$ ; the following sets are not configurations:  $\{\text{reject}, \text{grant}\}$  and  $\{\text{authentic}\}$ . Hence, we see that configurations can model (*partial*) *knowledge* about the outcome of an interaction. For this reason, in the SECURE trust model, the set of outcomes of a particular type of interaction,  $Out$ , is given by the configurations of an event structure  $ES$ , i.e.,  $Out = \mathcal{C}_{ES}$ . Notice that a mutually exclusive and exhaustive set of outcomes is given by the set of *maximal* configurations of  $ES$ ; further, also partial outcomes can be modelled by non-maximal configurations.

**Evidence.** Our modelling has enabled us to be *precise* about what is meant by outcomes (cf. the opening quotation of this chapter). We need also be precise about what is meant by *evidence* of an outcome (the set  $Ev$ ). In Chapter 4, we present the specific structure for evidence used in SECURE. It turns out that, a simple event-*counting* structure works in many cases. The set  $Ev = \mathbb{N}$  can represent (counts of) past evidence regarding a particular configuration. In the model we consider, it suffices to count events: We do not need to keep track of the actual outcomes of past interactions, only how many times each event has occurred. Hence, our trust structure will have values  $E \rightarrow \mathbb{N}$ , where  $ES = (E, \leq, \#)$  is a special type of finite event structure, called a confusion-free event structure. We equip the values with a simple information ordering  $\sqsubseteq$ ; however, it turns out that having a single trust ordering doesn't make sense: Which configurations are 'good' and which are 'bad' depends on principals' interpretations; if  $f, g : E \rightarrow \mathbb{N}$ , what should  $f \preceq g$  mean? We suggest to generalise the trust structure framework

so that principals agree on the set of trust values and agree on the information ordering, but instead of having a single trust ordering, there is a whole collection  $(\preceq_i)_{i \in I}$  of different trust orderings. Another way to think about this structure is that each principal subjectively decides which values  $f$  represent “more trust” than  $g$ . In the event structure model, the orderings are indexed by configurations: Intuitively,  $f \preceq_x g$  means that  $f$  represents more *evidence in favour of configuration  $x$*  than does  $g$ .

We define a probabilistic trust model,  $\lambda_{DES}$ , which generalises the beta model: Instead of having binary ratings we use event structures to model outcomes; also, the assumptions about principal behaviour are similar to (but generalise) those of the beta model (see Chapter 2). Based on so-called Dirichlet distributions, a generalisation of the beta family, we derive equations for the predictive probability,  $P(o \mid t\lambda_{DES})$ , i.e., the probability in the model  $\lambda_{DES}$  of an outcome  $o$  in the next interaction given trust value  $t$ .

**Results.** The main results of Chapter 4 consist of (i) the idea of modelling outcomes as configurations of event structures; (ii) definition of a behavioural model  $\lambda_{DES}$  and derivation of equations for the various probabilities of interest; (iii) definition of a generalised type of trust structure  $(D, \sqsubseteq, (\preceq_i)_{i \in I})$ , where  $(D, \sqsubseteq)$  is a cpo and  $(D, \preceq_i)$  is a pre-order for all  $i \in I$ ; (iv) considering a specific instance of the generalised structure, the SECURE model, which uses event structures for outcomes and event-counts as evidence, hence, yielding a specific trust structure  $(E \rightarrow \mathbb{N}, \sqsubseteq, (\preceq)_{x \in \mathcal{C}_{ES}})$ .

Additionally, Chapter 4 contains some further minor ideas. It has often been mentioned by several researchers that trust in one context does not necessarily transfer to trust in another. However, clearly, sometimes contexts are related so that information about the trustworthiness of an entity in one context may be useful in another. For a crude example, suppose  $q$  is a supplier of books and movies, and that  $p$  cares about quality and shipping-time for both products. Suppose also that  $p$  has information which indicates that the shipping time for  $q$ 's books is short. Clearly, the relation between the contexts means that the same information would make it probable that the shipping-time for  $q$ 's movies is also short; however, the information would not make it more or less likely that the quality of neither  $q$ 's books or movies is high. We present a notion of composable morphisms of event structures which serve as customisable information transfer functions between contexts in SECURE. The morphisms of event structures lift naturally to mappings from trust values in one event structure to trust values in another, hence, providing a means to transfer SECURE trust values.

### 3.1.1 Conclusion

The strength of the contribution in Chapter 4 is *modelling*. We consider the event structure model to be natural and useful: Event structures were originally used as denotational models in semantics of concurrency [103, 105, 99]; hence, modelling interaction protocols in SECURE as event structures seems reasonable. The SECURE model only deals with finite event structures, but, in principle, also infinite event structures could be considered; for example, one might use parameterised event structures, as defined in Chapter 6. A useful property of event structures is that given a probability distribution  $P(x \mid I)$  on the set of maximal configurations, one can automatically get a distribution on *any* set of mutually exclusive and exhaustive configurations: For a non-maximal  $y$ ,  $P(y \mid t)$  is simply the sum of  $P(x \mid t)$  for all maximal  $x$  which contain  $y$  (for details, Varacca et al. [99]).

Chapter 4 also presents formal behavioural model  $\lambda_{DES}$ , and a way of computing probabilities of outcomes from trust values of the form  $t : E \rightarrow \mathbb{N}$ . However, the behavioural model may not be appropriate for some applications: It is assumed that the behaviour of each principal is given by a fixed distribution, and does not change with time (or with the information that the principal has). For example, when principals  $p$  and  $q$  interact, often, the behaviour of  $q$  depends on the past behaviour of  $p$  and vice versa. Hence, assuming a fixed distribution is not always realistic. We return to this point in Section 3.4 of this chapter.

**Connection to the SGUC challenge.** The SECURE project represents a step towards incorporating more general information into trust management than credentials: Policies for security decisions are based on assessment of risks of interaction, derived from probabilities of various outcomes, which, in turn, are derived from information about past interactions with principals. The derivation of probabilities is founded on the formal probabilistic model of principal behaviour ( $\lambda_{DES}$ ). The SECURE framework supports policy-driven autonomic decision-making, and it is reasonably light-weight so that it can run on devices of limited computational power. In SECURE, also active decisions are supported. For example, it is possible to specify a policy choosing between a number of service providers by selecting one with a maximal expected benefit.

There are a number of drawbacks. As we discuss in Section 3.4, the  $\lambda_{DES}$  model is still unrealistic in many scenarios. Further, there is a clear need for technology to verify security policies: While each component in the SECURE framework is responsible for ensuring some kind of security property, little

work has been done to consider how these properties compose. In other words, an answer to the question: Exactly what security properties does one get when deploying the SECURE framework?

## 3.2 Operational Techniques for Trust Structures

**Motivation.** Recall the trust structure framework, presented briefly in Chapter 2: Given a trust structure,  $T = (D, \sqsubseteq, \preceq)$ , and a collection of policies,  $\Pi = (\pi_p \mid p \in \mathcal{P})$ , there exists a unique global trust-state  $\overline{\text{gts}} = \text{lfp } \Pi_\lambda$ , defining for any  $p, q \in \mathcal{P}$ ,  $p$ 's trust in  $q$  as  $\overline{\text{gts}}(p)(q)$ . Recall also that  $p$ 's (trust-based) *security policy* is an abstraction mapping  $\sigma_p : \mathcal{P} \times D \rightarrow \{\top, \perp\}$  so that interaction with  $q$  is allowed only if  $\sigma_p(q, \overline{\text{gts}}(p)(q)) = \top$ . Hence, to make security decisions,  $p$  must be able to compute the value  $\overline{\text{gts}}(p)(q)$ .

There are a number of reasons why computing the global trust state is infeasible: When the cpo  $(D, \sqsubseteq)$  is of *finite* height  $h$ , the cpo  $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D, \sqsubseteq)$  has height  $|\mathcal{P}|^2 \cdot h$ .<sup>2</sup> In this case, the least fixed-point can, *in principle*, be computed by finding the first identity in the chain of approximants  $(\lambda p. \lambda q. \perp \sqsubseteq) \sqsubseteq \Pi_\lambda(\lambda p. \lambda q. \perp \sqsubseteq) \sqsubseteq \Pi_\lambda^2(\lambda p. \lambda q. \perp \sqsubseteq) \sqsubseteq \dots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p. \lambda q. \perp \sqsubseteq)$  [104]. However, in the environment envisioned, such a computation is infeasible. The functions  $(\pi_p : p \in \mathcal{P})$  defining  $\Pi_\lambda$  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  $\pi_p$  autonomously, an inherent problem with trying to compute the fixed point is the fact that  $p$  might decide to change its policy  $\pi_p$  to  $\pi'_p$  at any time. Such a policy update would be likely to invalidate data obtained from a fixed-point computation done with global function  $\Pi_\lambda$ , i.e., one might not have time to compute  $\text{lfp } \Pi_\lambda$  before the policies have changed to  $\Pi'$ .

The above discussion indicates that exact computation of the fixed point is infeasible, and hence that the framework is not suitable as an operational model. To counter this, Chapter 5 presents several techniques for *approximating*  $\overline{\text{gts}}$ ; we give an overview of this work in the following.

---

<sup>2</sup>The height of a cpo is the size of its longest chain.

### 3.2.1 An Operational Semantics

As we have argued, it is infeasible to compute the complete global trust state,  $\overline{\text{gts}}$ . However, for a principal  $p$  to make a decision about one particular principal  $q$ ,  $p$  needs only know  $\overline{\text{gts}}(p)(q)$ , i.e., one particular entry of the global trust state; such a value is called a local (least) fixed-point value. Computing local fixed-point values is one way of approximating the global trust state; this section describes our work with proving the correctness of an algorithm for local fixed-point computation.

Carbone et al. [19] described a language for trust policies, and provided this language with the denotational fixed-point semantics. In Chapter 5, we consider a similar language and provide an operational semantics to complement the denotational semantics. The semantics of a collection of policies will be defined by translation into an I/O automaton [73, 72], providing a formal operational foundation for trust policies. Our main theorem proves in this formal model, that even in infinite height cpos the I/O automata will converge towards the least fixed-point, i.e., that the denotational and operational semantics agree.

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

**Results.** Chapter 5 provides an operational semantics for a simple, but generic trust-policy language. Based on the work of Bertsekas and Tsitklis, we introduce the formal model of Bertsekas abstract asynchronous systems; we then show that there is a simulation-like relation from the concrete semantics to the abstract model. This allows us to prove that the operational semantics agrees with the denotational semantics of Carbone et al. [19].

### 3.2.2 Approximation Techniques for $\preceq$ -Monotonic Policies

Let us say that a security policy  $\sigma : \mathcal{P} \times D \rightarrow \{\top, \perp\}$  is *monotonic* if for all  $q$ , and for all  $d, d'$  with  $d \preceq d'$  we have  $\sigma(q, d) = \top$  implies  $\sigma(q, d') = \top$ . If  $\sigma_p$  is monotonic, and we know that (i) there is a value  $d \in D$  with  $d \preceq \overline{\text{gts}}(p)(q)$  and (ii) that  $\sigma_p(q, d) = \top$ ; then we also know that  $\sigma_p(q, \overline{\text{gts}}(p)(q)) = \top$ . In Chapter 5, we present three techniques for computing a value  $d$  satisfying (i) and (ii); the techniques are valid whenever all trust policies are also monotonic with respect to the trust ordering  $\preceq$ .

**A ‘proof-carrying’ approach.** Consider a situation in which a client principal  $p$  wants to access a resource controlled by server  $v$ . The client is called the “prover” and the server is called the “verifier”. We assume that the verifier’s security policy,  $\sigma$ , is monotonic. The prover will send information  $I$  to  $v$ , which is used to convince her that  $\sigma(p, \overline{\text{gts}}(v, p)) = \top$ , hence, allowing  $v$  to make a safe decision about  $p$ . To verify  $I$ , the prover will inspect  $I$ , but also communicate with other principals (which its trust policy depends on), and ask them to check properties of  $I$ ; hence, we obtain a distributed proof-checking algorithm. The following proposition provides the theoretical basis for the proof-carrying request protocol.

**Proposition 3.1 (Proof-carrying requests)** *Assume that  $(D, \preceq, \sqsubseteq)$  is a trust structure where  $\preceq$  is continuous with respect to  $\sqsubseteq$ . Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. If  $I \preceq \perp_{\sqsubseteq}^n$  and  $I \preceq f(I)$ , then  $I \preceq \text{lfp}_{\sqsubseteq} f$ .*

The information  $I$  is a list of claims: Each entry  $j$  of  $I$  refers to some fixed-point value, say  $\overline{\text{gts}}(a)(b)$ , the claim then is  $I_j \preceq \overline{\text{gts}}(a)(b)$ . One of the entries, say  $v$ , will correspond to  $I_v \preceq \overline{\text{gts}}(v)(p)$ , and, hence, if the claim can be verified, the verifier knows that  $\overline{\text{gts}}(v)(p)$  is above  $I_v$ . The proposition says that if  $I \preceq \perp_{\sqsubseteq}$ , then the claim can be checked by just checking  $I_k \preceq \pi_k(I)$  for each relevant  $k$ .

The scheme has very much the flavour of a “proof-carrying” code: 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.

**A snapshot-based approach.** The approximation technique discussed now is different from that of the “proof-carrying” protocol: 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. 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 (Snapshot)** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. Assume that  $I$  is an information approximation for  $f$ . If  $I \preceq f(I)$  then  $I \preceq \text{lfp}_{\sqsubseteq} f$ .*

An information approximation  $I$  for  $f$  satisfies  $I \sqsubseteq \text{lfp} f$  and  $I \sqsubseteq f(I)$ . It turns out that we obtain an information approximation  $I$  by taking a snapshot of the state of the asynchronous algorithm described in the previous section. By the above proposition, we need only check, for each entry  $j$  of  $I$ , that  $I_j \preceq \pi_a(I)(b)$ , for appropriate  $a, b \in \mathcal{P}$ .

**Results.** Chapter 5 presents two approximation protocols, formalised using I/O automata, that allow a principal  $p$  to (distributedly) compute a trust value  $d$  so that  $d \preceq \text{lfp} (p)(q)$  for any fixed principal  $q$ . The techniques are based on two distinct approaches: One requires the client to supply information, whereas, in the other, servers cooperate in a snapshot computation; however, although distinct, the two techniques turn out to be instances of a more general technique, based on the following theorem:

**Proposition 3.3 (Generalised Approximation)** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $J \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. Assume that  $J$  satisfies  $J \preceq f(J)$ . If there exists an information approximation  $I \in D^n$  for  $f$ , with the property that  $J \preceq I$ , then  $J \preceq \text{lfp} f$ .*

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

### 3.2.3 Conclusion

There are two important restrictions to the proof-carrying approach. First, as in the example, in order to construct its proof, the prover needs infor-

mation about the verifier’s 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 that  $I \preceq \perp_{\square}$ , the protocol can usually only be used to prove properties stating “not too much bad behaviour,” and *not* properties guaranteeing sufficiently “good” behaviour (however, this depends on the trust structure used). In contrast to the proof-carrying approach, the snapshot-based approach does not suffer from the two mentioned restrictions. The price to pay is more communication and computation: In the worst case, the snapshot-checks fail, and the asynchronous algorithm has to compute the exact (local) fixed-point value. The generalised protocol suffers from the first problem, i.e., the prover needs information about the verifier’s policy, but not the second, i.e.,  $I \preceq \perp_{\square}$ .

We would like to emphasise that the developed techniques are generic: They can be used in any trust structure satisfying the monotonicity assumptions. For example, the asynchronous algorithm can be used also in Weeks’s fixed point model as a distributed way to check proof-of-compliance.

There are still unresolved issues. If principals may dynamically update their policies, a fixed-point computation can be invalidated if a principal updates its policy during the actual asynchronous algorithm. There is an exception though: If an update  $\pi_p \mapsto \pi'_p$  is information increasing, i.e.,  $\pi_p(m) \sqsubseteq \pi'_p(m)$  for all  $m : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ , then the fixed point computation is not affected. Many important policy updates are information increasing, e.g., updating behavioural information in the SECURE trust structure.

**Connection to the SGUC challenge.** The trust structures of Carbone et al. [19] represent a formal model of trust, “living” at a high-level in the theoretical hierarchy of the challenge. Concepts such a “ $p$ ’s trust in  $q$ ” are formally defined in the model (fixed-point semantics), and a high-level policy language with an abstract denotational semantics is defined. The contribution of the work described in this section, in relation to the challenge, is the following: (i) Showing how the high-level model for policies can be realised in the lower-level model of I/O automata, closer to implementation; (ii) development of reasoning techniques for the high-level policies, i.e., the approximation protocols. In this sense, this work is truly in the spirit of the theoretical hierarchy of the SGUC challenge. However, this work is only applicable in practice if one can define useful trust structures for specific GUC applications. Not much work using trust structures in actual systems has been done apart from the SECURE project, so the question of validity of trust structures remains.

### 3.3 A Logical Framework for Reputation Systems

**Motivation.** As we have seen in Chapter 2, most existing reputation systems, or experience-based trust management systems, perform some kind of abstraction on the direct data of the system. This has the effect that several quite different concrete behaviours are collapsed in to the same “equivalence class” of recorded behaviours. Abstract representations of behavioural information have their advantages (e.g., numerical values are often easily comparable, and require little space to store), but clearly, information is lost in the abstraction process. For example, in EigenTrust, value ‘0’ may represent both “no previous interaction” and “many unsatisfactory previous interactions” [49]. Consequently, one cannot verify exact properties of past behaviour given only the reputation information. The only non-example of information abstraction (that we are aware of) is the framework of Shmatikov and Talcott; however, as we discussed in Chapter 2, the framework lacks support in the form of reasoning techniques. The work presented in Chapter 6 is the result of a desire for a reputation system where one may reason rigorously about principals’ past behaviour, hence providing a form of security guarantees expressed in terms of properties of such past behaviour.

**Concrete reputation.** Chapter 6 presents a formal framework for a class of simple reputation systems in which, as opposed to most “traditional” systems, behavioural information is represented in a very concrete form. The advantage of our concrete representation is that sufficient information is present to check precise properties of past behaviour. In our framework, such requirements on past behaviour are specified in a declarative policy-language, and the basis for making decisions regarding future interaction becomes the verification of a behavioural history with respect to a policy. This enables us to define reputation systems that provide a form of provable “security” guarantees, intuitively, of the form: “If principal  $p$  gains access to resource  $r$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi_r$ .”

To get the flavour of such requirements, we preview an example policy from a declarative language formalized in the following sections. Edjlali *et al.* [33] consider a notion of history-based access control in which unknown programs, in the form of mobile code, are dynamically classified into equivalence classes of programs according to their behaviour (e.g. “browser-like” or “shell-like”). This dynamic classification falls within the scope of our very broad understanding of reputation systems. The following is an example of a policy written in our language, which specifies a property similar to that

of Edjlali *et al.*, used to classify “browser-like” applications:

$$\begin{aligned} \psi \equiv & \neg F^{-1}(\text{modify-file}) \wedge \\ & \neg F^{-1}(\text{create-subprocess}) \wedge \\ & G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))]) \end{aligned}$$

Informally, `modify-file`, `create-subprocess`, `open(x)` and `create(x)` are *events* which are observable by monitoring an entity’s behaviour. The latter two are *parameterised* events, and the quantification ‘ $\forall x$ ’ ranges over the possible parameters of these. Operator  $F^{-1}$  means “at some point in the past,”  $G^{-1}$  means “always in the past,” and constructs  $\wedge$  and  $\neg$  are conjunction and negation, respectively. Thus, clauses  $\neg F^{-1}(\text{modify-file})$  and  $\neg F^{-1}(\text{create-subprocess})$  require that the application has never modified a file, and has never created a sub-process. The final, quantified clause  $G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))])$  requires that whenever the application opens a file, it must previously have created that file. For example, if the application has opened the local system-file `"/etc/passwd"` (i.e. a file which it has not created) then it cannot access the network (a right assigned to the “browser-like” class). If, instead, the application has previously only read files it has created, then it will be allowed network access.

**Results.** Chapter 6 describes our formal declarative language for policies. Based on the framework of event structures, behavioural information is modelled as sequences of sets of events. Such linear structures can be thought of as (finite) models of linear temporal logic (LTL) [86]. Indeed, our basic policy language is based on a (pure-past) variant of LTL. We give the formal syntax and semantics of our language, and provide several examples illustrating its naturality and expressiveness. We are able to encode several existing approaches to history-based access control, e.g. the Chinese Wall security policy [14] and a restricted version of so-called ‘one-out-of- $k$ ’ access control [33].

An interesting new problem is how to re-evaluate policies efficiently when interaction histories change as new information becomes available. It turns out that this problem, which can be described as dynamic model-checking, can be solved very efficiently using an algorithm adapted from that of Havelund and Roşu, based on the technique of dynamic programming, used for runtime verification [42]. Interestingly, although one is verifying properties of an *entire* interaction history, one needs not store this complete history in order to verify a policy: Old interaction can be efficiently summarised relative to the policy.

Our simple policy language can be extended to encompass policies that are more realistic and practical (e.g., for history-based access control [33, 37, 5, 96], and within the traditional domain of reputation systems: peer-to-peer- and online feedback systems [49, 90]). More specifically, we present two extensions. The first is quantification: We extend the basic language, allowing parameterised events and quantification over the parameters. An algorithm for checking the extended language along with complexity analyses is provided. The second extension covers the two aspects of *information sharing*, and *quantitative properties*. We introduce constructs that allow principals to state properties, not only of their personally-observed behaviour, but also of the behaviour observed by others. Such information sharing is characteristic of most existing reputation systems. Another common characteristic is focus on conveying quantitative information. In contrast, standard temporal logic is qualitative: it deals with concepts such as *before*, *after*, *always* and *eventually*. We show that we can extend our language to include a range of quantitative aspects, intuitively, operators like ‘almost always,’ ‘more than  $N$ ,’ etc.

Finally, as a proof-of-concept we have developed a prototype software framework for history-based access control (HBAC) in Java, based on the theoretic contributions described in Chapter 6. The framework defines an XML language for declaratively specifying HBAC policies. Using the algorithms of Chapter 6, our framework creates a Java security manager which enforces the specified policy. If the monitored Java program is about to violate policy, the security manager throws an exception. The prototype is available as OpenSource software from Sourceforge on the URL: <https://sourceforge.net/projects/javahbac>.

### 3.3.1 Conclusion

Our approach to reputation-systems differs from most existing systems in that reputation information has an exact semantics, and is represented in a very concrete form. In our view, the novelty of our approach is that our instance systems can verifiably provide a form of exact *security guarantees*, albeit non-standard, that relate a *present authorisation* to a precise property of *past behaviour*. Our declarative languages allow for the specification of such security properties, and the applications of our framework extends beyond the traditional domain of reputations systems in that (i) we can explain formally, several existing approaches to “history based” access control; (ii) based on our framework, we have developed JavaHBAC, a prototype security manager for history-based access control in Java.

**Connection to the SGUC challenge.** The logical policies provide a high-level language for specifying the acceptable behaviours of principals. This defines a high-level model in the theoretical hierarchy, and we show how to realise this high-level model in lower level models (i.e., using either the array-based algorithms or the automata-based algorithm). As mentioned previously, the advantage of this framework is that it allows verifiable security policies: A policy  $\psi$  specifies exactly the allowable behaviours, and our proofs show how the algorithms guarantee that policies are not violated. While we do believe that this work is useful in GUC applications, questions of expressive power remain. Further, the constructs for quantitative properties and policy-referencing are rather ad-hoc, and their usefulness in practical applications should be explored.

### 3.4 An Outlook

*The actual science of logic is conversant at present only with things either certain, impossible, or entirely doubtful, none of which (fortunately) we have to reason on. Therefore the true logic for this world is the calculus of Probabilities, which takes account of the magnitude of the probability which is, or ought to be, in a reasonable man's mind.*

— James Clerk Maxwell, 1850

As this opening quotation suggests, we believe that to form part of an answer to the SGUC grand challenge, future computational trust models should be based on “the calculus of Probabilities.” Our main reasons for this belief is that probability theory allows rigorous reasoning based on formal models, and that it is *the* foundation for reasoning under incomplete information [43].

While the purpose of models may not be to fit the data but to sharpen the questions, good models must do both! Our probabilistic models must be more realistic. For example, the beta model of principal behaviour (which we consider to be state-of-the-art) assumes that for each principal  $p$  there is a single fixed parameter  $\theta_p$  so at each interaction, *independently of anything else we know*, there is probability  $\theta_p$  for a ‘good’ outcome and probability  $1 - \theta_p$  for a ‘bad’ outcome. For *some* applications, one might argue that this is unrealistic, e.g.: (i) The parameter  $\theta_p$  is fixed, independent of time, i.e., no dynamic behaviour; and (ii) principal  $p$ 's behaviour when interacting with us is likely to depend on our behaviour when interacting with  $p$ ; let us

call this property ‘recursive behaviour.’ (Note, the same issues are present in the Dirichlet model  $\lambda_{\mathcal{D}}$  and the Dirichlet-Event-Structure model  $\lambda_{DES}$  that we propose). Some beta-based reputation systems attempts to deal with the first problem by introducing so-called “forgetting factors”; essentially this amounts to choosing a number  $0 \leq \delta \leq 1$ , and then each time the parameters  $(\alpha, \beta)$  of the pdf for  $\theta_p$  are updated, they are also scaled with  $\delta$ , e.g., when observing a single ‘good’ interaction,  $(\alpha, \beta)$  become  $(\alpha\delta + 1, \beta\delta)$ . In effect, this performs a form of exponential decay on the parameters. The idea is that, somehow, old interactions weigh less than new ones; however, this represent a departure from the probabilistic beta model, where all interactions “weigh the same.” Since a new model is *not* introduced, i.e., to formalise this preference towards newer information, it is not clear what the exact benefits of forgetting factors are; e.g., why exponential decay as opposed to linear? As far as we know, no-one has considered the ‘recursive behaviour’ problem before.

The notion of context is also relevant for computational trust models, as have been recognised by many. Given a single-context model, one can obtain a multi-context model by instantiating the single-context model in each context. However, as Sierra and Sabater argue [92], this is too naive: The goal of a true multi-context model is not just to model multiple contexts, but to provide the basis for transferring information from one context to another related context. To our knowledge, there are no techniques to handle this problem.

Finally, we believe (as do Sierra and Sabater [92]) that our field is lacking a way of comparing the qualities of the many proposed trust-based systems. Sierra and Sabater propose that our field develop “(...) test-beds and frameworks to evaluate and compare the models under a set of representative and common conditions” [92]. Note that “a set of representative and common conditions” could be a formal probabilistic model. We agree with Sierra and Sabater, and in the following, we sketch ideas towards a what one might call “a theoretical test-bed.” Further, we present some preliminary ideas towards a formal model of dynamic principal behaviour.

### 3.4.1 Towards Comparing Probabilistic Trust-based Systems

We shall propose a generic measure to “score” specific probabilistic trust-based systems in a particular environment (i.e., “a set of representative and common conditions”). The score, which is based on the so-called Kullback-Leibler divergence, is a measure of how well an algorithm approximates the “true” probabilistic behaviour of principals.

Consider a probabilistic model of principal behaviour, say  $\lambda$ . We consider only the behaviour of a single fixed principal  $p$ , and we consider only algorithms that attempt to solve the following problem: Suppose we are given an interaction history  $\mathbf{X} = [(x_1, t_1), (x_2, t_2), \dots, (x_n, t_n)]$  obtained by interacting  $n$  times with principal  $p$ , observing at time  $t_i$ , outcome  $x_i$ . Suppose also that there are  $m$  possible outcomes  $(y_1, \dots, y_m)$  for the next interaction. The goal of a probabilistic trust-based algorithm (PTA), say  $\mathcal{A}$ , is to approximate a distribution on the outcomes  $(y_1, \dots, y_m)$  given this history  $\mathbf{X}$ . That is,  $\mathcal{A}$  satisfies:

$$\mathcal{A}(y_i | \mathbf{X}) \in [0, 1] \text{ (for all } i), \quad \sum_{i=1}^m \mathcal{A}(y_i | \mathbf{X}) = 1.$$

We assume that the probabilistic model,  $\lambda$ , defines the following probabilities:  $P(y_i | \mathbf{X}\lambda)$ , i.e., the probability of  $y_i$  in the next interaction given a past history of  $\mathbf{X}$ ; and  $P(\mathbf{X} | \lambda)$  the a priori probability of observing sequence  $\mathbf{X}$  in the model.<sup>3</sup>

Now,  $(P(y_i | \mathbf{X}\lambda) | i = 1, 2, \dots, m)$  defines the true distribution on outcomes for the next interaction (according to the model); in contrast,  $(\mathcal{A}(y_i | \mathbf{X}) | i = 1, 2, \dots, m)$  attempts to approximate this distribution. The Kullback-Leibler divergence, which is closely related to Shannon entropy, is a measure of the distance from a true distribution to an approximation of that distribution.<sup>4</sup> The Kullback-Leibler divergence from distribution  $\hat{p} = (p_1, p_2, \dots, p_m)$  to distribution  $\hat{q} = (q_1, q_2, \dots, q_m)$  on a finite set of  $m$  outcomes, is given by

$$D_{\text{KL}}(\hat{p} || \hat{q}) = \sum_{i=1}^m p_i \log_2 \left( \frac{p_i}{q_i} \right)$$

(any log-base could be used). The Kullback-Leibler divergence is almost a distance, but symmetry fails; that is,  $D_{\text{KL}}(\hat{p} || \hat{q}) \geq 0$  and  $D_{\text{KL}}(\hat{p} || \hat{q}) = 0$  only if  $\hat{p} = \hat{q}$ . The asymmetry comes from considering one distribution as “true” and the other as approximating.

For each  $n$  let  $\mathbf{O}^n$  denote the set of interaction histories of length  $n$ . Let us define for each  $n$ , the  $n$ 'th *expected Kullback-Leibler divergence in the*

<sup>3</sup>In a way, this model takes into account also the ‘recursive behaviour’ problem: The probabilities  $P(y_i | \mathbf{X}\lambda)$  and  $P(y_i | \lambda)$  are distinguished.

<sup>4</sup>The Kullback-Leibler divergence is commonly attributed to Kullback and Leibler with their paper “On information and sufficiency” [62], although it has been considered earlier [43].

model  $\lambda$ :

$$D_{\text{KL}}^n(\lambda \parallel \mathcal{A}) \stackrel{(\text{def})}{=} \sum_{\mathbf{X} \in \mathbf{O}^n} P(\mathbf{X} \mid \lambda) D_{\text{KL}}(P(\cdot \mid \mathbf{X}\lambda) \parallel \mathcal{A}(\cdot \mid \mathbf{X})),$$

that is,

$$D_{\text{KL}}^n(\lambda \parallel \mathcal{A}) = \sum_{\mathbf{X} \in \mathbf{O}^n} P(\mathbf{X} \mid \lambda) \left( \sum_{i=1}^m P(y_i \mid \mathbf{X}\lambda) \log_2 \left( \frac{P(y_i \mid \mathbf{X}\lambda)}{\mathcal{A}(y_i \mid \mathbf{X})} \right) \right).$$

Note that, for each input sequence  $\mathbf{X} \in \mathbf{O}^n$  to the algorithm, we evaluate its performance as  $D_{\text{KL}}(P(\cdot \mid \mathbf{X}\lambda) \parallel \mathcal{A}(\cdot \mid \mathbf{X}))$ ; however, we accept that some algorithms may perform poorly on very unlikely training sequences. Hence, we weigh the penalty on input  $\mathbf{X}$ , i.e.,  $D_{\text{KL}}(P(\cdot \mid \mathbf{X}\lambda) \parallel \mathcal{A}(\cdot \mid \mathbf{X}))$ , with the intrinsic probability of sequence  $\mathbf{X}$ ; that is, we compute the expected Kullback-Leibler divergence.

The Kullback-Leibler divergence is a well established measure in statistic, however, to our knowledge, the measure  $D_{\text{KL}}^n$  is new. Due to the relation to Shannon’s Information Theory, one can interpret  $D_{\text{KL}}^n(\lambda \parallel \mathcal{A})$  quantitatively as the expected number of bits of information one would gain if one would know the true distribution instead of  $\mathcal{A}$ ’s approximation on  $n$ -length training sequences.

**An example.** For an example of our measure, we compare the beta-based algorithm of Mui et al. [77] with the maximum-likelihood algorithm of Aberer and Despotovic [29]. We can compare these because they both deploy the same fundamental assumptions:

*Assume* that the behaviour of each principal is so that there is a fixed parameter such that at each interaction we have, *independently of anything we know about other interactions*, the probability  $\theta$  for a ‘success’ and therefore probability  $1 - \theta$  for ‘failure.’

This gives us the *beta model*,  $\lambda_{\mathbf{B}}$ , with the following likelihood for any  $\mathbf{X} \in \{s, f\}^n$ :

$$P(\mathbf{X} \mid \lambda_{\mathbf{B}}\theta) = \theta^{N_s(\mathbf{X})} (1 - \theta)^{N_f(\mathbf{X})},$$

where  $N_x(\mathbf{X})$  denotes the number of  $x$  occurrences in  $\mathbf{X}$ .

Let  $\mathcal{A}$  denote the algorithm of Mui et al., and let  $\mathcal{B}$  denote the algorithm of Aberer and Despotovic. Let  $s$  stand for “success” and  $f$  stand for “failure”,

and let  $\mathbf{X} \in \{s, f\}^n$  for some  $n > 0$ . Then,

$$\mathcal{A}(s \mid \mathbf{X}) = \frac{N_s(\mathbf{X}) + 1}{n + 2} \text{ and } \mathcal{A}(f \mid \mathbf{X}) = \frac{N_f(\mathbf{X}) + 1}{n + 2},$$

and it is easy to show that

$$\mathcal{B}(s \mid \mathbf{X}) = \frac{N_s(\mathbf{X})}{n} \text{ and } \mathcal{B}(f \mid \mathbf{X}) = \frac{N_f(\mathbf{X})}{n}.$$

For each choice of  $\theta \in [0, 1]$ , and each choice of training-sequence length, we can compare the two algorithms by computing and comparing  $D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \mid \mathcal{A})$  and  $D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \mid \mathcal{B})$ . For example:

**Theorem 3.1** *If  $\theta = 0$  or  $\theta = 1$  then the algorithm  $\mathcal{B}$  of Aberer and Despotovic [29] computes a better approximation of principal behaviour than the algorithm  $\mathcal{A}$  of Mui et al. [77]. In fact,  $\mathcal{B}$  always computes the exact probability of success on any possible training sequence.*

*Proof.* Assume that  $\theta = 0$ , and let  $n > 0$ . The only sequence of length  $n$  which has non-zero probability is  $f^n$ , and we have  $\mathcal{B}(f \mid f^n) = 1$ ; in contrast,  $\mathcal{A}(f \mid f^n) = \frac{n+1}{n+2}$ , and  $\mathcal{A}(s \mid f^n) = \frac{1}{n+2}$ . Since  $P(s \mid f^n \lambda_{\mathbf{B}}\theta) = \theta = 0 = \mathcal{B}(s \mid f^n)$  and  $P(f \mid f^n \lambda_{\mathbf{B}}\theta) = 1 - \theta = 1 = \mathcal{B}(f \mid f^n)$ , we have

$$D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{B}) = 0.$$

Since  $D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{A}) > 0$  we are done (the argument for  $\theta = 1$  is similar).  $\square$

Now let us compare  $\mathcal{A}$  and  $\mathcal{B}$  with  $0 < \theta < 1$ . Since  $\mathcal{B}$  assigns probability 0 to  $s$  on input  $f^k$  (for all  $k \geq 1$ ) which results in  $D_{\text{KL}}^n(\lambda \parallel \mathcal{B}) = \infty$ , we shall ignore these extreme sequences. For the rest of this section, we change the beta model slightly so that  $s^n$  and  $f^n$  have probability 0; the resulting probability-mass is distributed among all other outcomes in the following way.

Let  $\lambda'_{\mathbf{B}}$  denote a formal proposition representing the modified beta model: We define  $\lambda'_{\mathbf{B}}$  so that for any  $0 < \theta < 1$  and any  $\mathbf{X} \in \{s, f\}^n$ ,

$$\begin{aligned} P(\mathbf{X} \mid \lambda'_{\mathbf{B}}\theta) &\stackrel{(\text{def})}{=} P(\mathbf{X} \mid \neg(s^n + f^n)\lambda_{\mathbf{B}}\theta) \\ &= \frac{P(\mathbf{X} \mid \lambda_{\mathbf{B}}\theta)}{P(\neg(s^n + f^n) \mid \lambda_{\mathbf{B}}\theta)} \\ &= \frac{\theta^{N_s(\mathbf{X})}(1 - \theta)^{N_f(\mathbf{X})}}{1 - (\theta^n + (1 - \theta)^n)} \end{aligned}$$

We have the following theorem which compares  $\mathcal{A}$  and  $\mathcal{B}$  in several different environments.

**Theorem 3.2** *On average, the algorithm  $\mathcal{A}$  of Mui et al. [77] computes a better approximation of principal behaviour than the algorithm  $\mathcal{B}$  of Aberer and Despotovic [29] under all of the the following scenarios:*

- Principals behave as specified in the model  $\lambda'_{\mathbf{B}}\theta$  with

$$\theta \in \{.25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75\}$$

and the algorithms get inputs of length  $n \in [3, 170]$ .

- Similarly, if  $\theta \in \{.20, .80\}$  and  $n \in [9, 170]$ ; if
- $\theta \in \{.15, .85\}$  and  $n \in [13, 170]$ ; if
- $\theta \in \{.10, .90\}$  and  $n \in [23, 170]$ ; and if
- $\theta \in \{.05, .95\}$  and  $n \in [50, 170]$ .

*Proof.* The proof is done by computing and comparing  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A})$  and  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{B})$  for the specified values of  $\theta$  and  $n$ . We have computed these values with a computer program specialised for the model  $\lambda'_{\mathbf{B}}$ ; for each of the specified  $\theta$  and  $n$  we have  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A}) \leq D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{B})$ . The number 170 appears because it is the maximum for which our program can compute  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A})$ .

It is easy to show that  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A})$  is equal to

$$\sum_{i=1}^{n-1} \binom{n}{i} \frac{\theta^i (1-\theta)^{n-i}}{1 - (\theta^n + (1-\theta)^n)} \left[ \begin{aligned} &\theta (\log_2(\theta) - \log_2(i+1) + \\ &\log_2(n+2)) + \\ &(1-\theta) (\log_2(1-\theta) - \log_2(n-i+1) + \\ &\log_2(n+2)) \end{aligned} \right],$$

and, similarly, that  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{B})$  is equal to

$$\sum_{i=1}^{n-1} \binom{n}{i} \theta^i \frac{\theta^i (1-\theta)^{n-i}}{1 - (\theta^n + (1-\theta)^n)} \left[ \begin{aligned} &\theta (\log_2(\theta) - \log_2(i) + \log_2(n)) + \\ &(1-\theta) (\log_2(1-\theta) - \log_2(n-i) + \\ &\log_2(n)) \end{aligned} \right].$$

Hence, we obtain (writing  $D = (1 - (\theta^n + (1 - \theta)^n))(D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{B}) - D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A}))$ ):

$$\begin{aligned}
D &= \sum_{i=1}^{n-1} \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[ \begin{array}{l} \theta(\log_2(i+1) - \log_2(i) + \log_2(n) - \\ \log_2(n+2)) + \\ (1 - \theta)(\log_2(n-i+1) - \log_2(n-i) + \\ \log_2(n) - \log_2(n+2)) \end{array} \right] \\
&= \sum_{i=1}^{n-1} \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[ \begin{array}{l} \theta(\log_2(i+1) - \log_2(i)) + \\ (1 - \theta)(\log_2(n-i+1) - \log_2(n-i)) + \\ \log_2(n) - \log_2(n+2) \end{array} \right]
\end{aligned}$$

Thus  $D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{B}) - D_{\text{KL}}^n(\lambda'_{\mathbf{B}}\theta \parallel \mathcal{A}) \geq 0$  if-and-only-if the term

$$\sum_{i=1}^{n-1} \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[ \theta \log_2 \frac{i+1}{i} + (1 - \theta) \log_2 \frac{n-i+1}{n-i} \right]$$

is greater than or equal to

$$\sum_{i=1}^{n-1} \binom{n}{i} \theta^i (1 - \theta)^{n-i} (\log_2(n+2) - \log_2(n)) = (1 - (\theta^n + (1 - \theta)^n)) \log_2\left(\frac{n+2}{n}\right)$$

(where the last equality follows since  $\sum_{i=0}^n \binom{n}{i} \theta^i (1 - \theta)^{n-i} = 1$ ).

We have written a Java program which tests the inequality

$$\begin{aligned}
&\sum_{i=1}^{n-1} \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[ \theta \log_2 \frac{i+1}{i} + (1 - \theta) \log_2 \frac{n-i+1}{n-i} \right] \\
&\geq (1 - (\theta^n + (1 - \theta)^n)) (\log_2\left(\frac{n+2}{n}\right)),
\end{aligned}$$

for input  $\theta$  and  $n$ . The program has confirmed the specified results.  $\square$

While the above Theorem is not impressive, it is the first *theorem* which compares two algorithms in various environments (to our knowledge); all previous comparisons have been via computer simulations. In fact, we believe (but so-far have been unable to prove) that  $\mathcal{A}$  generally outperforms  $\mathcal{B}$  if sequences are long enough.

**Conjecture** For each  $0 < \theta < 1$  there exists  $K \geq 2$  such that for all  $n \geq K$  we have the inequality  $D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{A}) \leq D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{B})$ .

Furthermore, we believe the following conjecture, which states that  $\mathcal{A}$  outperforms  $\mathcal{B}$  on the average on sequences of size 3 or more when  $\theta$  is  $.25 \leq \theta \leq .75$ , i.e., even on very small training sequences when  $\theta$  is not “too extreme.”

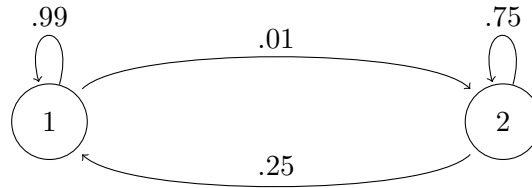
**Conjecture** For each  $.25 \leq \theta \leq .75$  and for all  $n \geq 3$  we have the inequality  $D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{A}) \leq D_{\text{KL}}^n(\lambda_{\mathbf{B}}\theta \parallel \mathcal{B})$ .

It is not so much the comparison of algorithms  $\mathcal{A}$  and  $\mathcal{B}$  that interests us; rather, our message is that probabilistic models enables the possibility of such theoretical comparisons. Notice that without formal probabilistic models we would be unable to even *state* precisely such conjectures.

### 3.4.2 Towards a Formal Model of Dynamic Behaviour

As we have argued, having a single parameter  $\theta_p$  to determine principal  $p$ 's behaviour is unrealistic in some applications. Often, the behaviour of  $p$  depends on the internal state of  $p$ , which is likely to change over time. Suppose we model  $p$  as a probabilistic finite-state system, i.e., there are  $n$  states  $S = \{1, 2, \dots, n\}$  and  $n^2$  transition probabilities  $t_{ij} \in [0, 1]$  with  $\sum_{j=1}^n t_{ij} = 1$ . Then, after at each interaction we assume that  $p$  changes state according to  $t$ : If  $p$  is in state  $i$  then it takes a transition to state  $j$  with probability  $t_{ij}$ ; i.e.,  $p$  behaves as a Markov chain. However, in our environment, state-changes are likely to be unobservable, i.e., principal  $q$  does not know for certain which state principal  $p$  is in, even after interacting with  $p$ . All that  $q$  can observe is the outcome of the interactions with  $p$ . If we accept the finite state assumption and the Markovian transition probabilities, we can incorporate the unobservability of states in our model by using so-called Hidden Markov Models [88].

A (discrete) Hidden Markov Model (HMM) is a tuple  $\lambda = (S, \pi, t, O, s)$  where  $S$  is a finite set of *states*;  $\pi$  is a distribution on  $S$ , called the *initial distribution*;  $t : S \times S \rightarrow [0, 1]$  is the *transition matrix*, i.e.,  $\sum_{j \in S} t_{ij} = 1$ ;  $O$  is a finite set of possible *observations*; and  $s : S \times O \rightarrow [0, 1]$ , called the *signal*, assigns to each state  $j \in S$ , a distribution  $s_j$  on observations, i.e.,  $\sum_{o \in O} s_j(o) = 1$ .



$$\begin{aligned}
 s(1, a) &= .95, s(1, b) = .05 \\
 s(2, a) &= .05, s(2, b) = .95 \\
 \pi(1) &= 1, \pi(2) = 0
 \end{aligned}$$

Figure 3.2: Example Hidden Markov Model.

**An example.** Consider the HMM in Figure 3.2. This models a simple two-state process with two possible observable outputs  $a$  and  $b$ . For example, this could model a channel which can forward a packet or drop it. State 1 models the normal mode of operation, whereas state 2 models operation under high load. Suppose that output  $a$  means “packet forwarded” and output  $b$  means “packet dropped.” Most of the time, the channel is in state 1, and packets are forwarded with probability .95; occasionally the channel will transit to state 2 where packets are dropped with probability .95. This example is not intended to faithfully model channels, but only to illustrate a simple HMM; however, we do believe that by tuning parameters, in fact, Hidden Markov Models can faithfully model many of the dynamic behaviours needed for probabilistic trust-based systems.

Consider now an observation sequence,  $h = aaaaaaaaaabb$ , which is reasonably probable in our model from Figure 3.2. The last two occurrences of  $b$ ’s makes it likely that a state-change has occurred. However, a simple counting algorithm  $\mathcal{A}$  would probably assign probability  $\mathcal{A}(a | h) = \frac{N_a(aaaaaaaaaabb)+1}{|h|+2} = \frac{11}{14} \sim .80$ . Indeed, if the state-change has actually occurred, the real probability would be .05.

Suppose now exponential decay is used, e.g., as in the Beta reputation system [46], with a factor of  $\delta = .5$ . The last observation weighs approximately the same as the rest of the history; in this case, the algorithm would quickly adapt, and probably assign probability  $\mathcal{A}(a | h) \sim .25$ , which is not too bad. However, suppose that we now observe  $bb$  and then another  $a$ . Again this would be reasonably likely, and would make a state-change prob-

able in the model. The exponential forgetting would assign a high weight to  $a$ , but also a high weight to  $b$  (because the last four observations were  $b$ 's); approximately  $\mathcal{A}(a \mid ha) \sim .5$ . In a sense, perhaps the algorithm adapts “too quickly”: It is, perhaps, too sensitive to new observations.<sup>5</sup>

So far, we have not been constructive: We have proposed a more expressive model (HMMs) than, say, the beta model, but we have not proposed any algorithms that do well in this model; this is future work. However, we would like to point to the so-called expectation-maximisation (EM) algorithm (or the Baum-Welch method), which attempts to approximate an HMM from a training sequence. The EM algorithm deals with an arbitrary  $n$ -state HMM, and tries to compute the transition probabilities and the signal probabilities from an input sequence. It can be shown that given an input sequence  $\mathbf{X}$ , the EM algorithm will compute a sequence of models  $\lambda_1, \lambda_2, \dots, \lambda_m = \lambda^*$  such that for all  $1 \leq i < m$  we have  $P(\mathbf{X} \mid \lambda_i) \leq P(\mathbf{X} \mid \lambda_{i+1})$ , so that the output  $\lambda^*$  is a local maximum for the likelihood function.

In certain application, one might use domain-specific knowledge about the structure of the HMM to design special purpose algorithms. This would allow for a type of verifiable correctness properties of the form: “Our trust-based algorithm  $\mathcal{C}$  performs well (formalised in some way, e.g., in terms of  $D_{\text{KL}}^n$ ) on every system where principals behave as  $\phi$ -Hidden-Markov-Models,” i.e., all HMMs satisfying a property  $\phi$ .

### 3.4.3 Towards the SGUC Challenge

As discussed in Chapter 1, a coherent theory of trust should form part of an answer to the SGUC challenge. Trust management systems, whether based on credentials or on experience and reputation, must verifiably provide security guarantees under realistic assumptions about the intended environment. If such guarantees can be provided, we may hope to start considering the interface between trust and other aspects of the SGUC challenge, ultimately, letting us reason about correctness and security properties of complete GUC programs. Regarding experience-based trust management systems, we believe that probabilistic models is the way forward: Probability theory allows us to state precisely the assumptions needed and the guarantees provided by probabilistic trust management systems. For example, in an ideal scenario, a mobile GUC software component could be written in a probabilistic

---

<sup>5</sup>Of course, one might argue, this example depends much on the factor  $\delta$  which can be used to tune the sensitivity. However, we could just as easily have found a model where quick sensitivity is of the essence; the point being that a single “sensitivity” parameter, used uniformly, cannot do well in all cases.

distributed process calculus incorporating a probabilistic trust management component; its specification could be written in a probabilistic special temporal logic; and probabilistic model checking used to verify that the implementation conforms to specification.

As we have argued above, probabilistic models also enable us to compare different systems in various probabilistic environments; this could be valuable if one has to select among several similar systems for a particular application. However, while our models are not yet general enough, Hidden Markov Models may provide a way to model dynamically-changing principal behaviour, which is certainly a necessary step towards more realistic models.



Part II  
Papers



# Chapter 4

## The SECURE Trust Model

*Whenever you can, count.*

— Sir Francis Galton, *The World of Mathematics*, 1956.

The material presented in this chapter is a significantly extended and revised version of a paper published in an invited collection dedicated to Arto Salomaa:

- [82] M. Nielsen and K. 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 on the Occasion of His 70th Birthday*, volume 3113 of *Lecture Notes in Computer Science*, pages 192–204. Springer Verlag, 2004.

This chapter presents the formal trust model used in the SECURE project [17, 16]. We have extended the contributions of the paper [82] in the following ways: We use the (generalised) trust structure  $(E \rightarrow \mathbb{N}, \sqsubseteq, (\preceq_x)_{x \in \mathcal{C}_{ES}})$  (where  $ES = (E, \leq, \#)$  is an event structure) instead of  $(\mathcal{C}_{ES} \rightarrow \mathbb{N}^3, \sqsubseteq, \preceq)$ ; we define the Dirichlet model,  $\lambda_{\mathcal{D}}$ , a Bayesian probabilistic model based on Dirichlet priors, which generalises the beta model; we extend the Dirichlet model to event structures to obtain a formal model  $\lambda_{\mathcal{D}ES}$  in which we can derive probabilities of outcomes as a product of Dirichlets; and finally, we show how morphisms of event structures can transfer trust values in one event structure to values in another. Furthermore, we have removed the section on a trust policy language and its denotational semantics; instead, this material is covered in Chapter 5. There is no concluding section, as this material is covered in Chapter 3.



## The SECURE trust model

### 4.1 Introduction

The Global Computing (GC) vision foresees *vast* numbers of heterogeneous, networked, mobile, autonomous entities interacting in various contexts, seeking to fulfil their respective goals. In GC environments such entities stand to benefit from cooperation, but only if they can assign meaningful privileges to each other.<sup>1</sup>

The classical trust management approach [8], first introduced by Blaze, Feigenbaum and Lacy [10], was proposed as a solution to the inadequacy of traditional security mechanisms in larger decentralised environments. A classical trust management system deals with deciding the so-called compliance checking problem: given a request together with a set of credentials, does the request comply with the local security policy of the provider? The same authors also developed tool-support in the form of PolicyMaker [10, 11] and later KeyNote [9] for handling the trust management problem. Stephen Weeks [100] displayed a simple mathematical framework for trust management, and showed how this framework would instantiate to various existing trust management systems, including KeyNote, SPKI [34] and some logic based systems (see [100] for details), sometimes even leading to more efficient algorithms for the compliance checking problem. The framework expresses a trust management system as a complete lattice  $(D, \leq)$  of possible *authorisations*, a set of *principal names*  $\mathcal{P}$ , and a language for specifying so-called *licenses*. The lattice elements  $d, e \in D$  express the authorisations relevant for a particular system, e.g. access-rights, and  $d \leq e$  means that  $e$  authorises at least as much as  $d$ . An *assertion* is a pair  $a = \langle p, l \rangle$  consisting of a principal  $p \in \mathcal{P}$ , the *issuer*, and a monotone function  $l : (\mathcal{P} \rightarrow D) \rightarrow D$ , called a *license*. In the simplest case  $l$  could be a constant function, say  $d_0$ ,

---

<sup>1</sup>Such entities can be users interacting through hardware devices, e.g. PDAs, Internet-connected PCs or mobile phones, but can also be software, e.g. mobile agents acting on behalf of some user or another program. We will use the term *principal* for such an entity.

meaning that  $p$  authorises  $d_0$ . In the general case the interpretation of  $a$  is: given that all principals authorise as specified in the *authorisation map*,  $m : \mathcal{P} \rightarrow D$ , then  $p$  authorises as specified in  $l(m)$ . This means that a license such as  $l(m) = m(A) \vee m(B)$  expresses a policy saying “give the lub of what  $A$  says and what  $B$  says”. Weeks showed that a collection of assertions  $L = \langle p_i, l_i \rangle_{i \in I}$  gives rise to a monotone function  $L_\lambda : (\mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$ , with the property that a coherent authorisation map representing the authorisations of the involved principals is given by the least fixed point,  $\text{lfp } L_\lambda$ .

The ideas on trust management systems seeded a substantial amount of research in the area of security in large distributed systems, but as noted in [41], which serves as a survey on existing systems anno 2000, the current trust management solutions do not adequately deal with the dynamic aspects of trust: a trusting relationship evolves over time and requires monitoring and reevaluation. In [19, 81] it was argued that while the idea of having mutually referring licenses resolved by fixed points was good, the Weeks-framework for trust would be too restrictive in GC environments. One reason is that principals often do not have sufficient information to specify precise authorisations for all other principals. In the framework this means that any unknown or only partially known principal is always assigned the bottom authorisation. The proposed solution was to have the set  $T$  of “authorisations”, here called *trust values*, equipped with *two* orderings, denoted  $\preceq$  and  $\sqsubseteq$ . Here  $\preceq$ , called the *trust ordering*, corresponds to Weeks’ way of ordering by “more privilege”, whereas  $\sqsubseteq$ , called the *information ordering*, introduces a notion of precision or information. The key idea was that the elements of the set should embody also various degrees of uncertainty, and then  $d \sqsubseteq e$  reflects that  $e$  is more precise or contains more information than  $d$ . In the simplest of cases the trust values could be just symbolic, e.g.  $\text{unknown} \sqsubseteq \text{low} \preceq \text{high}$ , but they might also have more structure, as will become clear in the following sections. It was shown how least fixed points with respect to the information ordering, leads to a way of distinguishing an unknown principal from a known and distrusted one.

The SECURE project [16, 17] aims at providing a framework for decision-making in GC environments, based on the notion of trust. The formal model for trust deployed is that of [19, 81], and a particular application defines a triple  $(T, \sqsubseteq, \preceq)$  of trust values with the two orderings. In this model, trust exists *between principals*, and so for any principals  $P$  and  $Q$  the trust that  $P$  has in  $Q$  is modelled as an element of  $T$ . As in the Weeks-framework this value is defined in terms of a license issued by  $P$  which is called  $P$ ’s *trust policy*. Thus, at any given time the trust-state of the system can be described as function,  $m : \mathcal{P} \rightarrow \mathcal{P} \rightarrow T$ , where  $\mathcal{P}$  is the set of principals,

and the interpretation is that  $m(P)(Q)$  describes  $P$ 's trust in  $Q$ . At any time there is a unique trust-state describing how principals trust, and this state is the  $\sqsubseteq$ -least fixed point of the continuous function induced by the collection of all licenses.

In SECURE, each principal  $P$  has its own decision making framework which is invoked when an application needs to make some decision involving another principal. The decision making framework contains three primary components: the risk engine, the trust engine, and the collaboration monitor. At the most abstract level, the collaboration monitor records the behaviour of principals with which  $P$  has interacted. This information together with a trust policy defines how  $P$  assigns trust values to any other principal. The trust information, in turn, serves as a basis for a risk analysis of any interaction. In fact, with each type of interaction with a principal, say  $Q$ , there is a finite set of possible *outcomes* of the interaction. The outcome that occurs is determined by the behaviour of  $Q$ . Each of these outcomes has an associated cost<sup>2</sup> which could be represented simply as a number, but could also be more complex objects e.g. probability distributions. Since the outcome depends on  $Q$ , the decision of how to interact is based on the trust in  $Q$ . In this set-up it is necessary that the trust value for  $Q$  carries enough information that estimation of the likelihood of each of the outcomes is possible. If this estimation is possible, one may start reasoning about risk, e.g. the *expected* cost of an interaction.

Since outcomes of interactions depend on the behaviour  $q$ , the decision of how to interact is based on the trust in  $q$ . Notice that in this set-up, it is necessary that the trust value for  $q$  carries enough information that estimation of the likelihood of each of the outcomes is possible. This is the reason that one cannot use an arbitrary trust structure,  $T = (D, \sqsubseteq, \preceq)$ : How would one convert a trust value  $d \in D$  for principal  $q$  into a probability distribution on outcomes (and how are outcomes even modelled in  $T$ )? Of course, if such probabilistic estimation is possible, one may start reasoning about risk, e.g., the *expected* cost of an interaction. For example, suppose that principal  $p$  can interact with principal  $q$ , and that such an interaction has three mutually exclusive and exhaustive potential outcomes  $o_1, o_2$  and  $o_3$ , with costs  $c_1, c_2$  and  $c_3$ , respectively. Suppose also that, based on its past interactions with  $q$ , principal  $p$  assigns probability  $p_1$  to outcome  $o_1$ ; the risk associated with outcome  $o_1$  is then  $c_1 p_1$ . The expected cost of the interaction would be  $c_1 p_1 + c_2 p_2 + c_3 p_3$ .

---

<sup>2</sup>The term cost should be understood more generally as cost or *benefit*. If costs are represented as non-negative numbers, one might represent benefit as negative number.

## 4.2 Event Structures for Interaction

As we discussed in the previous section the SECURE architecture brings forward the need for a formal model for trust supporting the approximation of likelihood of interaction outcomes, based on previous observations. The SECURE framework uses event structures to model interaction protocols and outcomes of interaction (for good references on event structures see Winskel et al. [83, 103, 105]). In this section, we recapture the basic definitions of event structures. We also introduce a special case of event structures, confusion-free event structures for which it is particularly simple to adjoin probabilities.

**Definition 4.1 (Event Structure)** *An event structure is a triple  $(E, \leq, \#)$  consisting of a set  $E$  of events which are partially ordered by  $\leq$ , the necessity relation (or causality relation), and  $\#$  is a binary, symmetric, irreflexive relation  $\# \subset E \times E$ , called the conflict relation. The relations satisfy*

$$[e] \stackrel{(def)}{=} \{e' \in E \mid e' \leq e\} \text{ is finite; and}$$

$$\text{if } e \# e' \text{ and } e' \leq e'' \text{ then } e \# e''$$

for all  $e, e', e'' \in E$ . We say that two events are independent if they are not in either of the two relations.

As mentioned, event structures are used in SECURE to model interaction protocols. For example, the event structure in Figure 4.1 could model a small scenario where a principal may ask a bank for the transfer of electronic cash from its bank account to an electronic wallet. After making the request, the principal observes that the request is either rejected or granted. After a successful transaction, the principal could observe that the cash sent in the transaction is forged or perhaps run an authentication algorithm to establish that it is authentic. Also, the principal could observe a withdrawal from its bank account with the present transaction's id, and this withdrawal may or may not be of the correct amount. The two basic relations on event structures have an intuitive meaning in our set up. An event may *exclude* the possibility of the occurrence of a number of other events. In our example the occurrence of the event 'transaction rejected' clearly excludes the event 'transaction granted'. The necessity relation is also natural: some events are *only possible* when others have already occurred. In the example structure, 'money forged' only makes sense in a transaction where the transfer of money actually did occur. Whether the e-cash is forged and whether the correct

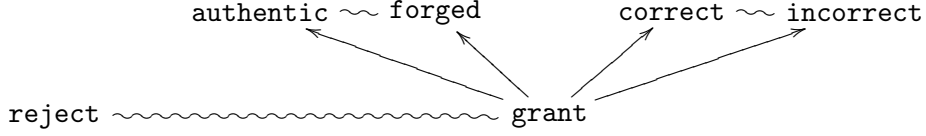


Figure 4.1: An event structure describing our example. The curly lines  $\sim$  describe the immediate conflict relation and pointed arrows, the causality relation.

amount is charged are two independent observations that may be observed, in any order, which is modelled as independence in the event structure.

The event structure models the set of events that can occur in a protocol; however, due to the relations on event structures, not all sets of events can occur in a particular run. The notion of configurations formalises this.

#### Definition 4.2 (Configurations of an Event Structure)

Let  $ES = (E, \leq, \#)$  be an event structure. Say that a set of events  $x \subseteq E$  is a configuration (of  $ES$ ) if it satisfies the following two properties:

1. *Conflict free:* for any  $e, e' \in x : e \not\# e'$  (i.e.  $(e, e') \notin \#$ ).
2. *Necessity downwards closed:* for any  $e \in x, e' \in E : e' \leq e \Rightarrow e' \in x$ .

We say that two configurations  $x$  and  $y$  are compatible if  $x \cup y$  is a configuration. Write  $\mathcal{C}_{ES}$  for the set of configurations of  $ES$ ;  $\max(\mathcal{C}_{ES})$  for the set of maximal configurations; and,  $\mathcal{C}_{ES}^0$  for the set of finite configurations. Finally, we define relation  $\rightarrow \subseteq \mathcal{C}_{ES} \times E \times \mathcal{C}_{ES}$  by

$$x \xrightarrow{e} x' \iff e \notin x \text{ and } x' = x \cup \{e\}$$

The configurations of our example is given in Figure 4.2. A (finite) configuration models information regarding the result of a single interaction: A maximal configuration models complete knowledge of the outcome, whereas a non-maximal configuration models partial knowledge of the outcome.

Note that the set of all maximal configurations defines a set of mutually exclusive and exhaustive outcomes. For the SECURE model, our goal is to derive a probability distribution  $P(x | t)$  on the set of maximal configurations from a “trust value”  $t$ .

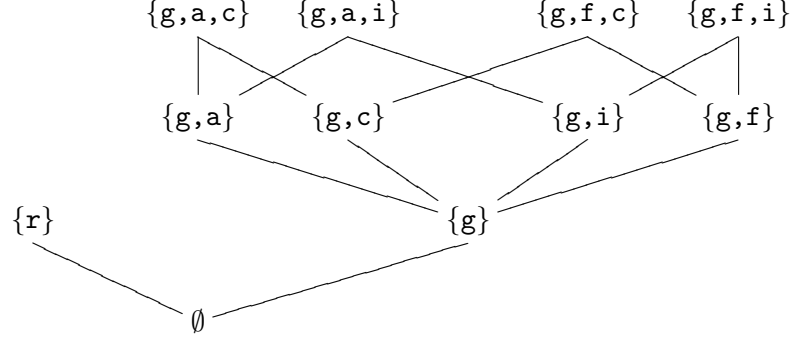


Figure 4.2: Configurations of the event structure in Figure 4.1. The lines indicate inclusion and the events are abbreviated.

**Histories.** With our formalisation of protocols and outcomes, we can now be more precise about the role of the collaboration monitor in the SECURE framework. Informally, the collaboration monitor of principal  $p$  monitors the behaviour of other principals that  $p$  interacts with. A finite configuration models information regarding *a single* interaction, i.e., a single run of a protocol. In general, the information that one principal possesses about another will consist of information about *several* protocol runs; the information about each individual run being represented by a configuration in the corresponding event structure. The concept of a local interaction history models this.

**Definition 4.3 (Interaction History)** Let  $ES = (E, \leq, \#)$  be an event structure. Define an interaction history in  $ES$  to be a finite ordered sequence of configurations,  $h = x_1x_2 \cdots x_n \in \mathcal{C}_{ES}^*$ . The entries  $x_i$  (for  $1 \leq i \leq n$ ) are called the sessions (of  $h$ ).

An interaction history in the event structure from Figure 4.1 could be the sequence  $\{g, a, c\}\{g, c\}\{g\}\{r\}$ .

When the collaboration monitor learns about the occurrence of an event in a particular instance of a protocol, the interaction history changes. We define an interface to the collaboration monitor.

**Definition 4.4 (Interface)** Define an operation  $\mathbf{new} : \mathcal{C}_{ES}^* \rightarrow \mathcal{C}_{ES}^*$  by  $\mathbf{new}(h) = h\emptyset$ . Define also a partial operation  $\mathbf{update} : \mathcal{C}_{ES}^* \times E \times \mathbb{N} \rightarrow \mathcal{C}_{ES}^*$  as follows. For any  $h = x_1x_2 \cdots x_i \cdots x_n \in \mathcal{C}_{ES}^*$ ,  $e \in E$ ,  $i \in \mathbb{N}$ ,

**update**( $h, e, i$ ) is undefined if  $i \notin \{1, 2, \dots, n\}$  or  $x_i \not\stackrel{e}{\rightarrow} x_i \cup \{e\}$ . Otherwise

$$\mathbf{update}(h, e, i) = x_1 x_2 \cdots (x_i \cup \{e\}) \cdots x_n$$

Let  $h \Rightarrow k$  denote that either  $\mathbf{new}(h) = k$  or there exists  $e \in E$ ,  $i \in \mathbb{N}$  so that  $\mathbf{update}(h, e, i) = k$ , and  $\Rightarrow^*$  the reflexive and transitive closure of  $\Rightarrow$ .

**Remarks.** The notion of *time* in the model is based on when sessions are *started*. More precisely, in our interaction histories,  $h = x_1 x_2 \cdots x_n$  where  $x_i \in \mathcal{C}_{ES}$ , the order of the sessions reflects *the order in which the corresponding interaction-protocols are initiated*, i.e.,  $x_i$  refers to the observed events in the  $i$ th-initiated session. Different notions of time could just as well be considered, e.g., if  $x_i$  precedes  $x_j$  in sequence  $h$  then  $x_j$  was updated more recently than  $x_i$ .

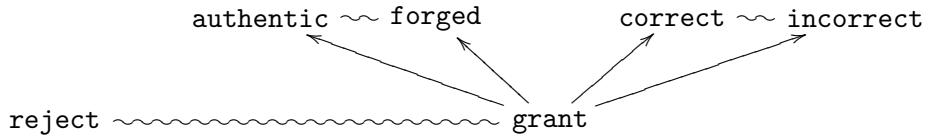
Note, while the order of sessions is recorded (histories are *sequences*), in contrast, the order of *independent* events within a *single session* is not. For example, for Figure 4.1

$$\begin{aligned} \mathbf{update}(\mathbf{update}(\{\text{grant}\}, \text{correct}, 1), \text{forged}, 1) = \\ \mathbf{update}(\mathbf{update}(\{\text{grant}\}, \text{forged}, 1), \text{correct}, 1) \end{aligned}$$

Hence independence of events is a choice of abstraction one may make when designing an event-structure model (because one is not interested in the particular order of events, or because the exact recording of the order of events is not feasible). However, note that this is not a limitation of event structures: in a scenario where this order of events is relevant (and observable), one can always use a “serialised” event structure in which this order of occurrences is recorded. A serialisation of events consists of splitting the events in question into different events depending on the order of occurrence.

### 4.2.1 Confusion-free Event Structures

We consider a special type of event structures, the *confusion free* event structures with independence, for which it is especially simple to adjoin probabilities [99]. Consider again the following event structure.



The events **authentic** (**a**) and **correct** (**c**) are *independent*; as are **a** and **incorrect** (**i**); **forged** (**f**) and **c**; and finally, **f** and **i**. However, in terms of the relations of event structures, *independent* simply means that both events can occur in the same configuration and in any order. Later we shall consider a probabilistic model where *independence* means also *probabilistic independence*. To do this we first introduce a notion of *cells* and *immediate conflict* [99]. In the following  $ES = (E, \leq, \#)$  is a fixed event structure.

Write  $[e]$  for  $[e] \setminus \{e\}$ , and say that events  $e, e' \in E$  are in *immediate conflict*, writing  $e \#_{\mu} e'$ , if  $e \# e'$  and both  $[e] \cup [e']$  and  $[e'] \cup [e]$  are configurations. It is easy to see that a conflict  $e \# e'$  is immediate if-and-only-if there exists a configuration  $x$  where both  $e$  and  $e'$  are enabled (i.e., can occur in  $x$ ). For example **reject** and **grant** are in immediate conflict (as are **a** and **f**), whereas **reject** (**r**) and **authentic** (**a**) are not.

A *partial cell* is a non-empty set of events  $c \subseteq E$  such that  $e, e' \in c$  implies  $e \#_{\mu} e'$  and  $[e] = [e']$ . A maximal partial cell is called a *cell*. There are three cells in the above event structure:  $\{\mathbf{r}, \mathbf{g}\}$ ,  $\{\mathbf{a}, \mathbf{f}\}$  and  $\{\mathbf{c}, \mathbf{i}\}$ . Cells represent choices; in probabilistic event structures, *probabilistic choices*. A *confusion free* event structure is an event structure where (the reflexive closure of) immediate conflict is an equivalence relation and *within cells* (i.e., that  $e \#_{\mu} e'$  implies  $[e] = [e']$ ). We suspect that most event structures for simple interaction protocols are confusion free (as is the case with our example above).

In confusion free event structures, if an event  $e$  of a cell  $c$  is enabled, then all events  $e' \in c$  are enabled too. If the event structure is also finite, a maximal configuration is obtained by starting with the empty configuration and then repeating the following: Let  $C$  be the set of cells that are enabled in the current configuration if  $C$  is empty then stop: the current configuration is maximal; otherwise, non-deterministically select a cell  $c \in C$ , and then non-deterministically select, or probabilistically sample, an event  $e \in c$ ; finally, update the current configuration by adding  $e$ .

The following notion of cell-valuation formalises probabilistic sampling in cells.

**Definition 4.5 (Cell valuation [99])** *When  $f : X \rightarrow [0, +\infty]$  is a function, for every  $Y \subseteq X$ , we define  $f[Y] = \sum_{y \in Y} f(y)$ . A cell valuation on a confusion free event structure  $ES = (E, \leq, \#)$  is a function  $p : E \rightarrow [0, 1]$  such that for every cell  $c$ , we have  $p[c] = 1$ .*

If cell choices are probabilistic, say given by a cell-valuation  $p$ , and if we assume independence between cells, then one can obtain the probability of any configuration  $x$  (i.e., any outcome) as the product of the probabilities

of each event in  $x$  given  $p$ . The following proposition (Proposition 4.1) formalises this.

Given two configurations  $x, x' \in \mathcal{C}_{ES}$  we say that  $x'$  covers  $x$ , written  $x \triangleleft x'$ , if there exists  $e \notin x$  so that  $x \xrightarrow{e} x'$ .

**Proposition 4.1 ([99])** *Let  $p$  be a cell valuation and let  $v_p : \mathcal{C}_{ES}^0 \rightarrow [0, 1]$  be defined by  $v_p(x) = \prod_{e \in x} p(e)$ . Then we have*

- $v_p(\emptyset) = 1$  (Normality);
- If  $C$  is a non-empty, maximal set of pairwise incompatible configurations covering  $x \in \mathcal{C}_{ES}$ , then  $v_p[C] = v_p(x)$  (Conservation);
- If  $x, y \in \mathcal{C}_{ES}$  are compatible then  $v_p(x)v_p(y) = v_p(x \cup y)v_p(x \cap y)$  (Independence).

A function  $v$  satisfying the first two properties is called a configuration valuation; if  $v$  satisfies also the Independence property then  $v$  is called a configuration valuation with independence. A configuration valuation assigns a probability to each (finite) configuration, i.e., to each outcome. Normality means that the empty configuration always occurs in a run; conservation means that if  $c$  is a cell which is enabled at  $x$  then  $v$  defines a probability distribution on the configurations  $C = \{x \cup \{e\} \mid e \in c\}$  given  $x$  ( $v(x) \neq 0$ ):

$$P(C \mid x, v) = P(C \mid v) \frac{P(x \mid Cv)}{P(x \mid v)} = P(C \mid v) \frac{1}{v(x)} = \frac{v[C]}{v(x)} = 1;$$

Finally, independence means that if cell  $c$  is enabled at  $x$  and at  $y$  (hence also at  $x \cap y$ ), then then for any event  $e \in c$  we have

$$P(x \cup \{e\} \mid x, v) = P(y \cup \{e\} \mid y, v) = P(x \cap y \cup \{e\} \mid x \cap y, v)$$

i.e., the “probability of event  $e$ ” is independent of the configuration we are at when sampling from  $c$ . This can be shown as follows: Define  $z^e$  as  $z \cup \{e\}$  for any  $z$ , then use Independence on  $((x \cap y)^e$  and  $y$ ) and then on  $((x \cap y)^e$  and  $x$ ) to obtain

$$v((x \cap y)^e)v(y) = v(y^e)v(x \cap y) \quad \text{and} \quad v((x \cap y)^e)v(x) = v(x^e)v(x \cap y)$$

which implies (for non-zero  $v(x), v(y), v(x \cap y)$ )

$$\frac{v(x^e)}{v(x)} = \frac{v(y^e)}{v(y)} = \frac{v((x \cap y)^e)}{v(x \cap y)}$$

which, in turn, implies what we wanted.

### 4.3 A Probabilistic Framework

We will be concerned with adjoining probabilities to the configurations of a finite confusion-free event structure  $ES$ , i.e., to define a configuration valuation with independence. By Proposition 4.1 we can do this by finding a cell valuation  $p : E \rightarrow [0, 1]$ , or, equivalently, for each cell  $c$ , a function  $p_c : c \rightarrow [0, 1]$  with  $p_c[c] = 1$ . The functions  $p_c$  should be derived from the past experience obtained from interacting with an entity in  $ES$ . In the following paragraph, we state the assumptions about the behaviour of entities, in our model. We then proceed to (i) find abstractions that preserve sufficient information under the model; and (ii) derive equations for the predictive probabilities, i.e., answering “what is the probability of outcome  $x$  in the next interaction with entity  $q$  (in the model)?”

**The model.** Let us consider a finite and confusion-free event structure  $ES$ . Let us write  $C(ES)$  for the set of cells (which are then the equivalence classes of immediate conflict). Write  $C(ES) = \{c_1, c_2, \dots, c_k\}$ , and let  $N_i = |c_i|$  for each  $i$ . Let us make the following assumptions about principal behaviour, and write  $\lambda_{DES}$  the assumptions of this model:

- Each principal’s behaviour is so that there are a fixed parameters such that at each interaction we have, *independently of anything we know about other interactions*, the probability  $\theta_{c,e}$  for event  $e$  at cell  $c$ .

Each  $\theta_{c_i}$  for  $c_i \in C(ES)$  is a vector of size  $N_i$  such that  $\sum_{e \in c_i} \theta_{c_i,e} = 1$ . Hence, the collection  $(\theta_c \mid c \in C(ES))$  defines a cell valuation on  $ES$ . For each configuration  $x \in \mathcal{C}_{ES}$  the probability of obtaining  $x$  in any run of  $ES$  with a principal parameterised by  $\theta$  is

$$P(x \mid \theta \lambda_{DES}) = \prod_{e \in x} \theta_e \quad (4.1)$$

where  $\theta_e$  is given by  $\theta_{c,e}$  where  $c$  is the unique cell with  $e \in c$ .

The goal of our probabilistic framework is to estimate the parameters  $\theta$  given a prior distribution and data regarding past interactions. By the  $\lambda_{DES}$  model, we need only estimate the parameters of each cell  $c$ , i.e.,  $\theta_c$ , to obtain a probability distribution on configurations (equation 4.1). Furthermore, given a sequence  $h = x_1 x_2 \dots x_n \in \mathcal{C}_{ES}^*$  of observed data, we need only keep track of event counts of  $h$  to estimate the parameters  $\theta_c$ .

To estimate the the parameters  $\theta$ , we shall use Bayesian analysis. Hence, we need prior distributions. It turns out that the family of Dirichlet distributions are a family of conjugate prior distributions to the family of multinomial trials (see Chapter 2 on the notion of ‘conjugate priors’). Since each

sampling from a cell is a multinomial trial, we can use Dirichlet distributions as our prior distributions. We explain Dirichlets in the following; notice the close correspondence to the presentation of the beta model in Chapter 2.

### 4.3.1 The Dirichlet Distribution

The Dirichlet family  $\mathcal{D}(\cdot)$  of order  $K$ , where  $2 \leq K \in \mathbb{N}$ , is a parameterised collection of continuous probability density functions defined on  $[0, 1]^K$ . There are  $K$  parameters, given by a vector  $\alpha$  of positive reals,  $\alpha = (\alpha_i)_{i=1}^K$ , that select a specific Dirichlet distribution from the family. The pdf for  $\mathcal{D}(\alpha)$  is given by the following: For variable  $\theta = (\theta_i)_{i=1}^K \in [0, 1]^K$  we have

$$f(\theta | \alpha) = \frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma(\alpha_i)} \prod_i \theta_i^{\alpha_i - 1}$$

where  $\Gamma$  is the Gamma function,  $\Gamma(z) = \int_0^\infty dt t^{z-1} e^{-t}$ , for  $z > 0$  (notice how this generalises the beta pdfs). Define  $|\alpha| = \sum_j \alpha_j$ ; the expected value and variance are given by

$$\mathbf{E}_{f(\theta|\alpha)}(\theta_i) = \frac{\alpha_i}{|\alpha|}, \quad \sigma_{f(\theta|\alpha)}^2(\theta_i) = \frac{\alpha_i(|\alpha| - \alpha_i)}{|\alpha|^2(|\alpha| + 1)}$$

**A conjugate prior.** Consider again sequences of independent experiments with  $K$ 'ary outcomes, each yielding outcome  $i$  with some fixed probability; let us call such experiments multinomial trials. Let  $p, q \in \mathcal{P}$  be principals, and assume that  $p$  and  $q$  have interacted  $n$  times and each interaction is a multinomial trial. Let  $X_i^{pq} \in \{1, 2, \dots, K\}$ , for  $i = 1, 2, \dots, n$ , be the results of the  $i$ th interaction between  $p$  and  $q$ .

Let us *assume* that principal  $q$ 's behaviour is so that there are a fixed parameters such that at each interaction we have, *independently of anything we know about other interactions*, the probability  $\theta_i$  for outcome  $i$ . This gives us a probabilistic model, and let us call it the *Dirichlet model*. Let  $\lambda_{\mathcal{D}}$  denote a formal proposition representing the Dirichlet model, and let  $\theta \in [0, 1]^K$  be the parameter determining the probabilities of the outcomes at each trial. Finally, let  $\mathbf{X}$  be the conjunction of  $n$  statements of the form

$$Z_i \equiv (X_i^{pq} = j_i) \quad \text{for } i \in \{1, \dots, n\} \text{ and each } j_i \in \{1, \dots, K\}$$

i.e.,  $\mathbf{X} = \bigwedge_{i=1}^n Z_i$ , and let there be  $m_j$  statements of the form  $X_i^{pq} = j$ . Then, by definition of our model  $\lambda_{\mathcal{D}}$ , we have the following likelihood.

$$P(\mathbf{X} | \theta \lambda_{\mathcal{D}}) = \prod_{i=1}^n P(Z_i | \theta \lambda_{\mathcal{D}}) = \prod_{i=1}^K \theta_i^{m_i}$$

If we choose the prior pdf  $g(\theta | \lambda_{\mathcal{D}})$  to be  $\mathcal{D}(\alpha)$ , for  $\alpha = (\alpha_i)_{i=1}^K$  (again,  $\alpha_i = 1$  (for all  $i$ ) given the uniform distribution), then we can compute the posterior as follows. By the rules of probability theory (Jaynes [43]), we obtain the following expressions for the joint distribution  $g(\mathbf{X}\theta | \alpha\lambda_{\mathcal{D}})$ :

$$g(\mathbf{X}\theta | \alpha\lambda_{\mathcal{D}}) = P(\mathbf{X} | \theta\lambda_{\mathcal{D}})g(\theta | \alpha) = g(\theta | \mathbf{X}\lambda_{\mathcal{D}})P(\mathbf{X} | \lambda_{\mathcal{D}})$$

or equivalently

$$g(\mathbf{X}\theta | \alpha\lambda_{\mathcal{D}}) = \prod_{i=1}^K \theta_i^{m_i} \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K \theta_i^{\alpha_i-1} = g(\theta | \mathbf{X}\lambda_{\mathcal{D}})P(\mathbf{X} | \lambda_{\mathcal{D}})$$

Hence,

$$g(\theta | \mathbf{X}\lambda_{\mathcal{D}}) = \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{P(\mathbf{X} | \lambda_{\mathcal{D}}) \prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K \theta_i^{\alpha_i+m_i-1}$$

Since the fraction is independent of  $\theta$  and  $g(\theta | \mathbf{X}\lambda_{\mathcal{D}})$  is a pdf, the fraction must be the normalising constant of the Dirichlet distribution with the following parameters:  $\alpha' = (\alpha_1 + m_1, \alpha_2 + m_2, \dots, \alpha_K + m_K)$ , which means that  $g(\theta | \mathbf{X}\lambda_{\mathcal{D}})$  is  $\mathcal{D}(\alpha')$ . In other words, the Dirichlet family is conjugate prior to the multinomial likelihoods.

**The predictive probability ( $\lambda_{\mathcal{D}}$ ).** Now let  $Z_{n+1} \equiv (X_{n+1}^{pq} = i)$ , i.e., the statement that the  $(n+1)$ 'st outcome is of type  $i$ , then  $P(Z_{n+1} | \mathbf{X}\lambda_{\mathcal{D}})$  is a predictive probability: Given no direct knowledge of  $\theta$ , but only past evidence ( $\mathbf{X}$ ) and the model ( $\lambda_{\mathcal{D}}$ ), then  $P(Z_{n+1} | \mathbf{X}\lambda_{\mathcal{D}})$  is the probability that the next interaction will result in a type  $i$  outcome. By the rules of probability theory:

$$\begin{aligned} P(Z_{n+1} | \mathbf{X}\lambda_{\mathcal{D}}) &= \int_{\theta} d\theta P(Z_{n+1} | \mathbf{X}\lambda_{\mathcal{D}}\theta)g(\theta | \mathbf{X}\lambda_{\mathcal{D}}) \\ &= \int_{\theta} d\theta \theta_i g(\theta | \mathbf{X}\lambda_{\mathcal{D}}) \\ &= \mathbf{E}_{g(\theta|\mathbf{X}\lambda_{\mathcal{D}})}(\theta_i) \end{aligned}$$

Now recall the expectation of Dirichlet distributions; then we can compute the predictive probability:

$$P(Z_{n+1} | \mathbf{X}\lambda_{\mathcal{D}}) = \mathbf{E}_{g(\theta|\mathbf{X}\lambda_{\mathcal{D}})}(\theta) = \frac{\alpha_i + m_i}{|\alpha| + n} \quad (4.2)$$

(since  $g(\theta | \mathbf{X}\lambda_{\mathcal{D}})$  is  $\mathcal{D}(\theta | (\alpha_1 + m_1, \alpha_2 + m_2, \dots, \alpha_K + m_K))$  and  $\sum_i m_i = n$ ).

To summarise, given the assumptions of the Dirichlet model, one can compute the probability of outcome  $i$  in the next interaction as the expectation of the Dirichlet pdf  $g(\theta \mid \mathbf{X}\lambda_{\mathbf{B}})$  which results via Bayesian updating given the past history  $\mathbf{X}$ .

### 4.3.2 Dirichlets on Cells

Let us return to our probabilistic model. For each cell  $c \in C(ES)$  we will associate a prior distribution on the parameters  $\theta_c$  determining the behaviour of a fixed principal. As we interact, we obtain data about these parameters, and the distribution on each cell is updated via Bayes' Theorem. Each cell  $c \in C(ES)$  presents a choice between the mutually exclusive and exhaustive events of  $c$ , and by the assumptions of  $\lambda_{DES}$  a sequence of such choices from  $c$  is a sequence multinomial trials. We use Dirichlet priors on each cell so that the posterior distributions are also Dirichlets. At any time, we obtain the predictive probability of the next interaction resulting in a configuration by multiplying the expectations (according to the current cell distributions) of each event in the configuration.

Let us be precise: Let  $f_c(\theta_c \mid \lambda_{DES})$  denote the prior distribution on the parameters for each cell  $c \in C(ES)$  (when interacting with a fixed principal). Let  $\alpha_c$  be a vector of positive real numbers of size  $N_c = |c|$ ; we take,

$$f_c(\theta_c \mid \lambda_{DES}) = \mathcal{D}(\theta_c \mid \alpha_c) = \frac{\Gamma(\sum_{i=1}^{N_c} \alpha_{c,i})}{\prod_{i=1}^{N_c} \Gamma(\alpha_{c,i})} \prod_{i=1}^{N_c} \theta_{c,i}^{\alpha_{c,i}-1}$$

For example, taking  $\alpha_{cj} = 1$  (for all  $j$ ) gives the uniform distribution. Let  $\mathbf{X} : E \rightarrow \mathbb{N}$  be an event count modelling data about past runs with a specific principal. Let  $\mathbf{X}_c = \mathbf{X}|_c$  (i.e., the restriction of  $\mathbf{X}$  to cell  $c$ ), then the posterior pdf is given by the following: Assume that  $c = \{e_1, e_2, \dots, e_{N_c}\}$  then (assuming that  $\mathbf{X}$  is also a proposition representing observed counts),

$$\begin{aligned} f_c(\theta_c \mid \mathbf{X}\lambda_{DES}) &= \frac{\Gamma(\sum_{i=1}^{N_c} \alpha_{c,i} + \mathbf{X}(e_i))}{\prod_{i=1}^{N_c} \Gamma(\alpha_{c,i} + \mathbf{X}(e_i))} \prod_{i=1}^{N_c} \theta_{c,i}^{\alpha_{c,i} + \mathbf{X}(e_i) - 1} \\ &= \mathcal{D}(\theta_c \mid \alpha_c + \mathbf{X}_c) \end{aligned}$$

Hence, each event count  $\mathbf{X} : E \rightarrow \mathbb{N}$  can be used to do Bayesian updating of the distribution at each cell.

**The predictive probability ( $\lambda_{DES}$ ).** By Bayesian updating, we obtain a Dirichlet distribution for each cell  $c$  of  $ES$ . Hence, the distribution on

the maximal configurations is a product of Dirichlets. Let  $\mathbf{X}$  be an event count corresponding to  $n$  previously observed configurations, and let  $Z$  be the proposition that “the  $(n+1)$ ’st interaction results in outcome  $i$ ” (where  $1 \leq i \leq M$  and  $M$  is the number of maximal configurations in  $ES$ ). Let  $x_i$  be the  $i$ ’th maximal configuration, and for  $e \in x_i$  let  $c(e)$  denote the unique cell  $c$  with  $e \in c$ ; then the predictive probability is the product of the expectations of each of the cell parameters.

$$P(Z \mid \mathbf{X}\lambda_{DES}) = \prod_{e \in x_i} \mathbf{E}_{f_{c(e)}(\theta_{c(e)} \mid \mathbf{X}\lambda_{DES})}(\theta_{c(e),e}) = \prod_{e \in x_i} \frac{\alpha_{c(e),e} + \mathbf{X}(e)}{|\alpha_{c(e)}| + \mathbf{X}[c]}$$

#### 4.4 Trust Values and Orderings

Recall that the SECURE model is based on the trust structure framework; hence, we need a trust structure  $T = (D, \sqsubseteq, \preceq)$ . The set of trust values,  $D$ , will be event counts in a finite confusion-free event structure  $ES = (E, \leq, \#)$ , i.e., functions from  $E$  into  $\mathbb{N}$ . The following definition shows how one can move from (observed) sequences of configurations to event counts.

**Definition 4.6** *Let  $ES = (E, \leq, \#)$  be an event structure, define the function **count** :  $\mathcal{C}_{ES}^* \rightarrow (\mathcal{C}_{ES} \rightarrow \mathbb{N})$  by: For  $h = x_1x_2 \cdots x_n \in \mathcal{C}_{ES}^*$  and for  $x \in \mathcal{C}_{ES}$ ,*

$$\mathbf{count}_h(x) = N_x(h) \stackrel{(def)}{=} |\{j \in \mathbb{N} \mid 1 \leq j \leq n, x_j = x\}|$$

*We define also a function **events** :  $\mathbb{N}^{\mathcal{C}_{ES}} \rightarrow \mathbb{N}^E$  by: For each  $t \in \mathbb{N}^{\mathcal{C}_{ES}}$  and  $e \in E$ ,*

$$\mathbf{events}_t(e) = \sum_{x \in \{y \in \mathcal{C}_{ES} \mid e \in y\}} t(x)$$

*Finally for  $h \in \mathcal{C}_{ES}^*$  write  $\hat{h}$  for  $(\mathbf{events} \circ \mathbf{count})(h)$ .*

Notice how each step throws away information (which is irrelevant according to the model  $\lambda_{DES}$ ): **count** forgets the order of sessions, and **events** forgets which and how many sessions have occurred, remembering only the number of occurrences of each event.

**An information ordering.** To be compatible with the trust structure framework, we must define an information ordering  $\sqsubseteq$  on our trust values. A trust value  $t : E \rightarrow \mathbb{N}$  represents event counts; one may refine the value  $t$  by adding counts, i.e.,  $t'(e) = t(e) + k_e$  for natural numbers  $k_e > 0$ .

**Definition 4.7 (Information Ordering)** Let  $ES = (E, \leq, \#)$  be a finite confusion-free event structure. Define a binary relation  $\sqsubseteq \subseteq \mathbb{N}^E \times \mathbb{N}^E$  by

$$t \sqsubseteq s \iff (\forall e \in E. t(e) \leq s(e))$$

**Lemma 4.1** The relation  $\sqsubseteq$  partially orders  $\mathbb{N}^E$ .

*Proof.*  $\sqsubseteq$  is simply the pointwise extension of  $\leq$  to  $\mathbb{N}^E$ .  $\square$

The following proposition shows that updates of histories are monotonic.

**Proposition 4.2** Let  $ES$  be a finite confusion-free event structure and  $h, k \in \mathcal{C}_{ES}^*$  interaction histories. Then **events**  $\circ$  **count** is monotonic in the sense that if  $h \Rightarrow^* k$  then also  $\hat{h} \sqsubseteq \hat{k}$ .

*Proof.* By definition,  $h \Rightarrow^* k$  if and only if (i) an event is added; or (ii) a new configuration is started. In each case the proposition holds.  $\square$

Let  $\mathbf{V}$  denote the completion of  $(\mathbb{N}^E, \sqsubseteq)$  by a greatest element  $\top_{\sqsubseteq}$ .

**Theorem 4.1** The structure  $(\mathbf{V}, \sqsubseteq)$  is a complete lattice. The binary join is given by  $t \sqcup s = \max(t, s) = \lambda e. \max(t(e), s(e))$ . The join with respect to  $\top_{\sqsubseteq}$  is as expected, and the join of any infinite set is  $\top_{\sqsubseteq}$ .

*Proof.* Simple inspection.  $\square$

**Trust orderings.** In the trust structure framework, the set of trust values is also ordered by a trust-ordering,  $\preceq$ . However, given an arbitrary (finite confusion-free) event structure  $ES = (E, \leq, \#)$  what should  $t \preceq s$  mean for  $t, s \in \mathbb{N}^E$ ? If  $E = \{\text{good}, \text{bad}\}$  with  $\text{good} \# \text{bad}$ , then one might take  $t \preceq s$  only if

$$t(\text{good}) \leq s(\text{good}) \quad \text{and} \quad t(\text{bad}) \geq s(\text{bad})$$

But in arbitrary event structures, such an ordering is not possible; which configurations are ‘good’ and which are ‘bad’ depends on the principals’ subjective assessments. Instead, there are really orderings for each “choice of what good means,” i.e., if configuration  $x$  is considered to be “good” then a trust value which assigns more evidence in favour of  $x$  is “more trust” than a trust value which assigns less evidence in favour of  $x$ .

**Definition 4.8 (Trust orderings)** Let  $ES = (E, \leq, \#)$  be a finite confusion-free event structure, and let  $x \in \mathcal{C}_{ES}$ . For each event  $e \in E$ , write  $\text{Cell}(e)$  for the cell  $c$  with  $e \in c$ . Define a binary relation  $\preceq_x$  on  $\mathbb{N}^E$  as follows:

$$t \preceq_x s \iff \forall e \in E. (\text{let } c = \text{Cell}(e) \text{ in } t[c] \leq s[c] \wedge t(e)s[c] \leq s(e)t[c])$$

The definition says that, first of all, at each relevant cell of  $x$ , the weight of  $s$  is greater than the weight of  $t$  ( $t[c] \leq s[c]$ ); secondly,  $t(e)s[c] \leq s(e)t[c]$  means that the relative weight of event  $e$  in its cell is greater in  $s$  than in  $t$ .

**Lemma 4.2** *For each  $x \in \mathcal{C}_{ES}$  the relation  $\preceq_x$  pre-orders  $\mathbb{N}^E$ .*

*Proof.* Simple inspection. □

The relation  $\preceq_x$  is not a partial order because for any event  $e \notin x$  which is still compatible with  $x$  (i.e., independent events), there is no restriction on  $t(e)$  compared to  $s(e)$ .

Hence, we propose to generalise the trust structure framework:

**Definition 4.9 (Trust structure)** *A (generalised) trust structure is a triple  $T = (V, \sqsubseteq, (\preceq)_{i \in I})$ , where  $V$  be a set and  $\sqsubseteq$  is a binary relation on  $V$  so that  $(V, \sqsubseteq)$  is a cpo. Further,  $I$  is an index set, and  $(\preceq_i \mid i \in I)$  is a collection of pre-orderings on  $V$ .*

Notice that all the theory of trust policies works in the exact same manner as in the normal trust structure framework since there are no assumptions about trust policies and the trust ordering.

One would expect that  $P(x \mid t) \leq P(x \mid s)$  when-ever  $t \preceq_x s$ ; indeed, this is true:

**Proposition 4.3** *Let  $ES = (E, \leq, \#)$  be a finite confusion-free event structure, and let  $x \in \mathcal{C}_{ES}$ . Let  $t, s \in \mathbb{N}^E$  and assume that  $t \preceq_x s$ . Then,*

$$P(x \mid t\lambda_{DES}) \leq P(x \mid s\lambda_{DES}).$$

*Proof.* Let  $e \in x$  be arbitrary but fixed, and let  $e \in c = \{e, e_1, e_2, \dots, e_k\}$ , where  $c$  is the cell of  $e$ . Since  $t[c] \leq s[c]$  and  $t(e)s[c] \leq s(e)t[c]$ , we get

$$\mathbf{E}_{f_c(\theta_c \mid t\lambda_{DES})}(\theta_{c,e}) \leq \mathbf{E}_{f_c(\theta_c \mid s\lambda_{DES})}(\theta_{c,e}).$$

Since this holds for all  $e \in x$ , also

$$P(x \mid t\lambda_{DES}) = \prod_{e \in x} \mathbf{E}_{f_c(\theta_c \mid t\lambda_{DES})}(\theta_{c,e}) \leq \prod_{e \in x} \mathbf{E}_{f_c(\theta_c \mid s\lambda_{DES})}(\theta_{c,e}) = P(x \mid s\lambda_{DES})$$

□

## 4.5 Transferring Information

It has often been mentioned by researchers that trust in one context does not necessarily transfer to trust in another. However, clearly, sometimes contexts are related so that information about the trustworthiness of an entity in one context may be useful in another. For a crude example, suppose  $q$  is a supplier of books and movies, and that  $p$  cares about quality and shipping-time for both products. Suppose also that  $p$  has information which indicates that the shipping time for  $q$ 's books is short. Clearly, the relation between the contexts means that the same information would make it probable that the shipping-time for  $q$ 's movies is also short; however, the information would not make it more or less likely that the quality of neither  $q$ 's books or movies is high.

### 4.5.1 Morphisms

In the following, we present a notion of composable morphisms of event structures which serve as customisable information transfer functions between contexts in SECURE.

**Definition 4.10 (Morphism of event structures)** *Let  $ES = (E, \leq, \#)$  and  $ES' = (E', \leq', \#')$  be event structures. A morphism of event structure,  $\eta : ES \rightarrow ES'$  is a function  $\eta : E' \rightarrow \mathbf{2}^E$  which has the following two properties:*

1. *Monotonic: For any  $e', e'' \in E'$  if  $e' \leq' e''$  then we have*

$$\forall e_2 \in \eta(e'') \exists e_1 \in \eta(e') : e_1 \leq e_2$$

2. *Preserves conflict: For any  $e', e'' \in E'$  if  $e' \# e''$  then*

$$\forall e_1 \in \eta(e') \forall e_2 \in \eta(e'') : e_1 \# e_2$$

A morphism  $\eta : ES \rightarrow ES'$  can be thought of as a way of transferring evidence from  $ES$  to  $ES'$ . The idea is that  $e \in \eta(e')$  means that an occurrence of  $e$  in  $ES$  is an indication of the event  $e'$  occurring in  $ES'$ . We will think of the set  $\eta(e')$  as a disjunction of conditions in the sense that  $e'$  occurs if there is some  $e \in \eta(e')$  which has occurred in  $ES$ . If  $\eta(e') = \emptyset$  then we say that  $e'$  has no enabling condition under  $\eta$ . A morphism,  $\eta : ES \rightarrow ES'$  can then be used to map configurations of  $ES$  to configurations of  $ES'$  by the mapping,  $\bar{\eta} : \mathcal{C}_{ES} \rightarrow \mathcal{C}_{ES'}$

$$\bar{\eta}(x) = \{e' \in E' \mid \eta(e') \cap x \neq \emptyset\}$$

The axioms of morphisms imply that  $\bar{\eta}(x)$  is a configuration.

The event structures with event structure morphisms constitute a category. This means that one can compose information transfer functions.

**Definition 4.11 (Category of Event Structures,  $\mathbf{E}$ )** Consider the following categorical data, which we will call the category of event structures, and denote  $\mathbf{E}$ .

- Objects are event structures  $ES = (E, \leq, \#)$
- Morphisms  $\eta : ES \rightarrow ES'$ , are the morphisms of Definition 4.10.
- Identities  $1_{ES} : ES \rightarrow ES$  are the functions  $1_{ES} : E \rightarrow \mathbf{2}^E$  given by

$$1_{ES}(e) = \{e\}$$

- For  $\eta : ES \rightarrow ES'$  and  $\epsilon : ES' \rightarrow ES''$  composition,  $\epsilon \circ \eta : ES \rightarrow ES''$  is given by the following function  $\epsilon \circ \eta : E'' \rightarrow \mathbf{2}^E$

$$\epsilon \circ \eta(e'') = \bigcup_{e' \in \epsilon(e'')} \eta(e')$$

**Proposition 4.4 ( $\mathbf{E}$  is a category)** The definition of  $\mathbf{E}$  yields a category.

*Proof.* Simple inspection. □

**Transferring trust values.** The morphisms of event structures lift naturally to mappings from trust values in one event structure to trust values in another, hence, providing a means to transfer SECURE trust values. Let  $ES$  and  $ES'$  be event structures, and suppose that  $t : E \rightarrow \mathbb{N}$ , define a trust value  $t_\eta : E' \rightarrow \mathbb{N}$  by

$$t_\eta(e') = \sum_{e \in \eta(e')} t(e)$$

**A very simple example.** Consider the following (finite confusion-free) event structures,  $ES = (\{\text{quality, trash, ontimebook, latebook}\}, \leq, \#)$  and  $ES' = (\{\text{ontimemovie, latemovie}\}, \leq, \#)$ . Suppose the first event structure represents a protocol where a supplier can send quality books and trash books, and that the books can arrive on time or late. Similarly, the second event structure represents a protocol where the supplier can send movies which arrive on time or late.

Now consider the mapping

$$\eta = [\text{latemovie} \mapsto \{\text{latebook}\}, \text{ontimemovie} \mapsto \{\text{ontimebook}\}].$$

This represents that the late (on-time) arrival of a book transfers to a late (on-time) arrival of a movie. Hence a trust value  $t = [\text{latebook} \mapsto 3, \text{ontimemovie} \mapsto 13]$  would map to a trust value in  $ES'$ :  $t_\eta = [\text{latebook} \mapsto 3, \text{ontimemovie} \mapsto 13]$ .



# Chapter 5

## Operational Techniques for Trust Structures

The material presented in this chapter is based on a number of papers, published in the form of two conference papers and two journal papers (also as BRICS technical reports which are not otherwise mentioned here).

- [59] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. In *Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 805–814. IEEE, 2005.
- [53] K. Krukow. An operational semantics for trust policies. Presented at Workshop on Issues in the Theory of Security, 2006 (WITS'06), March 25–26 2006, Vienna, Austria, co-located with ETAPS 2006. No published proceedings yet.
- [55] K. Krukow and M. Nielsen. Trust Structures: Denotational and operational semantics. Journal version. Accepted for publication in the *International Journal of Information Security*, special edition on Formal Aspects of Security and Trust. Available online <http://www.brics.dk/~krukow>, 2006.
- [61] K. Krukow and A. Twigg. The complexity of fixed point models of trust in distributed networks. To be published in *TCS – Special issue on Semantic and Logical Foundations of Global Computing*, Elsevier, 2006.

The publications are related and should be viewed as one single contribution. The ICDCS paper [59] and the TCS paper [61] develops algorithmic techniques for sound approximation of the least fixed point in the trust structure framework. The TCS paper additionally develops complexity bounds on the problems considered. However, the algorithms and protocols are described in an informal notation, and the proofs of correctness are sketches. The WITS paper [53] and the Information Security paper [55] formalise and prove correct these algorithms and protocols. The WITS paper is a short

workshop paper, which does not contain proofs, and does not contain a formalisation of the approximation protocols. The version presented here is a combination of papers [59] and [55]. There is no concluding section, as this material is covered in Chapter 3.

## Operational Techniques for Trust Structures

### 5.1 Introduction

The need for flexible security mechanisms in emerging distributed-systems is evident. However, the diversity and scale of such systems, combined with the lack of centralised authority, means that traditional mechanisms for security decision-making, e.g. access-control lists, are often too restrictive and complex to deploy [8]. The concept of trust management, introduced by Blaze *et al.* [10], was presented as a solution to the problems with authorisation in large-scale distributed systems. Traditional trust-management systems make security decisions based on policies, dealing with authorisation 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 contrast, reputation-based and experience-based trust management systems focus on the *past behaviour* of principals instead of credentials [47, 92, 89]. This 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.* [22, 68, 69], and, to a large extent, captured concisely in a mathematical framework of Weeks [100]; 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 [47]). This lack prompted the development of a mathematical framework for trust [19], inspired by that of Weeks, but departing from Weeks by emphasising the concept of *information* in contrast to *authorisation*. The framework, which was introduced by Carbone *et al.* [19], discussed also by Nielsen *et al.* [81] and Krukow [54],

is the focus of this paper. Our motivation and contributions are to complement this denotational semantic model with operational techniques, and, consequently, we recall the model now. Readers already familiar with the framework may skip the next section.

### 5.1.1 The Trust-Structure Framework

A *trust model* is a mathematical model which gives precise meaning to the (otherwise overloaded) concept of *trust* within a system or a class of systems. A trust model should be generic enough to be instantiated to support authorisation in a variety of distributed computing systems. For example, in a P2P file-sharing system, appropriate authorisations may include using resources ‘download’ and ‘upload’, while in the PGP system [85], a principal may be authorised to introduce new signed (*key, name*) pairs.

The trust-structure framework [19] is a generic model, parameterised by a set  $D$  of possible *trust values* representing distinct levels or degrees of trust, relevant for a particular application. For example, in the P2P file-sharing application, one might identify a trust-level with an authorisation, say  $D_{P2P} = \{\text{upload, download, no, both, unknown}\}$ . Here a principal  $p$  might assign trust-value ‘upload’ to  $q$ , while, since it doesn’t know  $r$ ,  $p$  assigns value ‘unknown’ to  $r$ ; at the same time,  $c$  is known never to be trusted and hence is assigned value ‘no’. In the trust-structure framework, trust levels are not always identified with *authorizations*, e.g., in the P2P scenario one could instead use more “dynamic” trust values, related to the *past behaviour* of principals; for instance  $D'_{P2P} = \{(ul, dl) \mid ul, dl \in \mathbb{N}\}$ , where  $(m, n) \in D'_{P2P}$  represents the past history of a principal that has performed  $m$  uploads and  $n$  downloads.

The trust-structure framework simply assumes that  $D$  is a set for which it makes sense to (partially) order its values in two ways: with respect to *more trust* and with respect to *more information*. For example, in  $D_{P2P}$  the value ‘no’ clearly denotes a lower degree of trust than ‘download’, which is reflected by the *trust ordering*  $\preceq$ , e.g. we have  $\text{no} \preceq \text{download}$ . The ordering  $\preceq$  is *partial*, meaning that it may not make sense to relate all pairs of trust values, e.g. relating *download* and *upload* is not meaningful, so neither  $\text{download} \preceq \text{upload}$  nor  $\text{upload} \preceq \text{download}$ . An important characteristic of the trust structure framework is the requirement that not only does it make sense to order values according to the trust ordering ( $\preceq$ ), but values are also partially ordered with respect to *information*. Since we are allowing various degrees of precision (or information) in the trust values, it makes sense to compare some values with respect to their information content,

e.g. **unknown** is clearly less information than **upload** or **no**. In general, the framework assumes that the set  $D$  can be partially ordered by  $\sqsubseteq$ , called the information ordering. One may think of assertion  $x \sqsubseteq y$  as the statement that  $x$  can be refined into  $y$ , or that  $x$  approximates  $y$ , e.g. ‘unknown’ could be refined into ‘no’ if more (trust-wise negative) information was provided.

We provide now the formal definitions of the trust-structure framework. A complete formal understanding of the framework requires an understanding of the theory of partial orders as can be obtained, e.g., in Winskel’s book on programming language semantics [104]. The casual reader should be able to understand the model at a more intuitive level from the following description.

**Trust structures.** Formally, trust is something which exists between *pairs of principals*; it is *quantified* and *asymmetric* in that we care of “how much” or “to what degree” principal  $p$  trusts principal  $q$  (which may not be to the same degree that  $q$  trusts  $p$ ). Each application instance of the framework defines a so-called *trust structure*,  $T = (D, \preceq, \sqsubseteq)$ , which consists of a set  $D$  of *trust values*, together with two partial orderings of  $D$ , the trust ordering ( $\preceq$ ) and the information ordering ( $\sqsubseteq$ ). The elements  $s, t \in X$  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 refinement. As a simple example of a trust structure, consider the so-called “ $MN$ ” trust-structure  $T_{MN}$  [54]. In this structure, trust values are pairs  $(m, n)$  of natural numbers (as in the set  $X'_{P_2P}$ ), 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.* [54, 82, 19], have considered several additional examples of trust structures.

**Global trust-states.** Given a fixed trust structure  $T = (D, \preceq, \sqsubseteq)$ , and a set  $\mathcal{P}$  of principal identities; a *global trust-state* of the system is a function  $\mathbf{gts} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . The interpretation is that  $\mathbf{gts}$  represents the trust state where  $p$ ’s trust in  $q$  (formalized as an element of  $D$ ) is given by  $\mathbf{gts}(p)(q)$ . A good way of thinking about  $\mathbf{gts}$  is to consider it a large matrix, indexed by

pairs of principal identities, in which the row indexed by principal  $p$  (denoted  $\mathbf{gts}(p)$ ) contains principal  $p$ 's trust in any other principal. For example, in the row  $\mathbf{gts}(p)$ , column  $q$  represents  $p$ 's trust in  $q$ , given as an element in the set  $D$ ; this entry is denoted  $\mathbf{gts}(p)(q)$  (“row vectors” like  $\mathbf{gts}(p)$  are also called *local* trust-states). Thus, the matrix  $\mathbf{gts}$  gives a complete (system global) description of how everyone trusts everyone else. We shall write  $\mathbf{GTS}$  for the set of global trust-states  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . Similarly we write  $\mathbf{LTS}$  for the set  $\mathcal{P} \rightarrow D$  of *local* trust-states (corresponding to rows of  $\mathbf{gts}$  matrices).

**Trust policies.** The goal of the trust-structure framework is to define, at any time, a global trust state  $\overline{\mathbf{gts}}$ , thus giving a precise meaning to “ $p$ 's trust in  $q$ ” at all times (i.e., as value  $\overline{\mathbf{gts}}(p)(q)$ ). In order to uniquely define the global trust state  $\overline{\mathbf{gts}}$ , an approach similar to that of Weeks [100] is adopted. Each principal  $p \in \mathcal{P}$  defines a *trust policy* which is a function  $\pi_p$  of type  $\mathbf{GTS} \rightarrow \mathbf{LTS}$ , i.e. taking a global matrix as input, and providing a local row-vector as output. This function then determines  $p$ 's trust-row within the unique global trust-matrix, i.e. determines row  $\overline{\mathbf{gts}}(p)$ , as follows. In the simplest case,  $\pi_p$  could be a constant function, ignoring its first argument  $\mathbf{gts} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . As an example,  $\pi_p(\mathbf{gts}) = \lambda q. t_0$  (for some  $t_0 \in D$ ) defines  $p$ 's trust in any  $q \in \mathcal{P}$  as the constant  $t_0$ . In general we allow a form of *delegation* called *policy reference*: policy  $\pi_p$  may refer to other policies ( $\pi_z$ ,  $z \in \mathcal{P}$ ), e.g.,  $p$  might trust  $q$  to download if  $A$  or  $B$  trusts  $q$  to download. The general interpretation of  $\pi_p$  is the following. *Given that all principals assign trust-values as specified in the global trust-state  $\mathbf{gts}$ , then  $p$  assigns trust values as specified in vector  $\pi_p(\mathbf{gts}) : \mathcal{P} \rightarrow D$ .* For example, in the  $D_{P^2P}$  trust structure, function  $\pi_p(\mathbf{gts}) = \lambda q \in \mathcal{P}. (\mathbf{gts}(A)(q) \vee_{\succeq} \mathbf{gts}(B)(q)) \wedge_{\succeq} \text{download}$ , represents a policy saying “for any  $q \in \mathcal{P}$ , the trust in  $q$  is the least upper-bound in  $(D_{P^2P}, \preceq)$  of what  $A$  and  $B$  say, but no more than the constant  $\text{download} \in D_{P^2P}$ .”<sup>1</sup>

**Unique trust-state.** The collection of all trust policies,  $\Pi = (\pi_p | p \in \mathcal{P})$ , thus “spins a global web-of-trust” in which the trust policies mutually refer to each other. Since trust policies  $\Pi$  may give rise to cyclic policy-references, it is not *a priori* clear how to define the unique global trust-state  $\overline{\mathbf{gts}}$  for a given collection of trust policies  $\Pi$ . One may consider the unique function  $\Pi_\lambda = \langle \pi_p | p \in \mathcal{P} \rangle$ , of type  $\mathbf{GTS} \rightarrow \mathbf{GTS}$  with the property that  $\text{Proj}_p \circ \Pi_\lambda = \pi_p$

<sup>1</sup>Assuming that  $(D, \preceq)$  is a lattice. We always denote information ( $\sqsubseteq$ ) least-upper-bounds by “square” symbols  $\sqcup$ , and trust ( $\preceq$ ) least-upper-bounds/greatest-lower-bounds by  $\vee/\wedge$ .

for all  $p \in \mathcal{P}$ , where  $\text{Proj}_p$  is the  $p$ 'th projection.<sup>2</sup> Intuitively, the function  $\Pi_\lambda$  is easy to understand: each  $\pi_p$  maps a matrix  $\mathbf{gts} \in \mathbf{GTS}$  to a “row-vector”  $\pi_p(\mathbf{gts})$  in  $\mathbf{LTS}$ ; on input  $\mathbf{gts}$ , function  $\Pi_\lambda$  builds the output matrix from all these rows by taking the  $p$ 'th row of the output matrix to be  $\pi_p(\mathbf{gts})$ . We can now state a minimal requirement that the unique trust state,  $\overline{\mathbf{gts}}$ , should satisfy:  $\overline{\mathbf{gts}}$  *should be consistent with all policies*  $\pi_p$ . This amounts to requiring that it should satisfy the following fixed-point equation:  $\mathbf{gts}(p) = \pi_p(\mathbf{gts})$  for all  $p \in \mathcal{P}$ ; or equivalently:

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

Any matrix  $\mathbf{gts} : \mathbf{GTS}$  satisfying this equation is *consistent* with the policies  $(\pi_p | p \in \mathcal{P})$ , i.e. row  $p$  of  $\mathbf{gts}$  is consistent with  $\pi_p$  in that, if all principals trust as specified in  $\mathbf{gts}$ , then  $p$  trusts as specified in  $\pi_p(\mathbf{gts})$  which (by the fixed-point equation) can be read-off as the  $p$ th row of  $\mathbf{gts}$ .

This means that *any* fixed point of  $\Pi_\lambda$  is consistent with all policies  $\pi_p$ . But arbitrary functions  $\Pi_\lambda$ , may have multiple or even no fixed points.

Here we appeal to the power of the mathematical theory of complete partial orders and continuous functions (domain theory), known from formal programming language semantics [104]. A crucial requirement in the trust-structure framework is that the information ordering  $\sqsubseteq$  makes  $(D, \sqsubseteq)$  a complete partial order (cpo) with a least element (this element is denoted  $\perp_\sqsubseteq$ , and can be thought of as a value representing “unknown”). We require also that all policies  $\pi_p : \mathbf{GTS} \rightarrow \mathbf{LTS}$  are *information continuous*, i.e. continuous with respect to  $\sqsubseteq$ .<sup>3</sup> Since this implies that  $\Pi_\lambda$  is also information-continuous, and since  $(\mathbf{GTS}, \sqsubseteq)$  is a cpo with bottom, standard theory [104] tells us that  $\Pi_\lambda$  has a (unique) least fixed-point which we denote  $\text{lfp}_\sqsubseteq \Pi_\lambda$  (or simply  $\text{lfp} \Pi_\lambda$ ):

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

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

<sup>2</sup> $\text{Proj}_p$  is given by: for all  $\mathbf{gts} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ .  $\text{Proj}_p(\mathbf{gts}) = \mathbf{gts}(p)$ .

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

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

### 5.1.2 The Operational Problem

Many interesting systems are instances of the trust-structure framework [19, 54], 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  $\overline{\mathbf{gts}}(p)$ . While the framework does ensure the existence of a unique (theoretically well-founded) global trust-state, it is not “operational” in the sense of providing a way for principals to actually *compute* the trust values. Furthermore, as we shall argue in the following, the standard way of computing least fixed-points is inadequate in our scenario.

When the cpo  $(D, \sqsubseteq)$  is of finite height  $h$ , the cpo  $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D, \sqsubseteq)$  has height  $|\mathcal{P}|^2 \cdot h$ .<sup>4</sup> In this case, the least fixed-point of  $\Pi_\lambda$  can, *in principle*, be computed by finding the first identity in the chain of approximants  $(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \Pi_\lambda(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \Pi_\lambda^2(\lambda p. \lambda q. \perp_{\sqsubseteq}) \sqsubseteq \dots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p. \lambda q. \perp_{\sqsubseteq})$  [104]. However, in the environment envisioned, such a computation is infeasible. The functions  $(\pi_p : p \in \mathcal{P})$  defining  $\Pi_\lambda$  are distributed throughout the network, and, more importantly, even if the height  $h$  is finite, the number of principals  $|\mathcal{P}|$ , though finite, will be *very* large. Furthermore, even if resources were available to make this computation, we can not assume that any central authority is present to perform it. Finally, since each principal  $p$  defines its trust policy  $\pi_p$  autonomously, an inherent problem with trying to compute the fixed point is the fact that  $p$  might decide to change its policy  $\pi_p$  to  $\pi'_p$  at any time. Such a policy update would be likely to invalidate data obtained from a fixed-point computation done with global function  $\Pi_\lambda$ , i.e., one might not have time to compute  $\text{lfp } \Pi_\lambda$  before the policies have changed to  $\Pi'$ .

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

<sup>4</sup>The height of a cpo is the size of its longest chain.

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

**Contributions.** Our work essentially deals with the operational problems left as “future work” by Carbone *et al.* [19].

Firstly, techniques for distributed computation of approximations to the idealised trust-values over a global, highly dynamic, decentralised network. We start by showing that although it may be infeasible to compute the global trust-state,  $\overline{\text{gts}} : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ , one can instead compute so-called *local* fixed-point values. We take the practical point-of-view of a specific principal  $p$ , wanting to reason about its trust value for a fixed principal  $q$ . The basic idea is that instead of computing the entire state  $\overline{\text{gts}}$ , and *then* “looking up” value  $\overline{\text{gts}}(p)(q)$  to learn  $p$ ’s trust in  $q$ , one may instead compute this value directly. In Sections 5.2 and 5.3, we describe a general language for specifying trust policies, and provide a compositional operational semantics. The semantics of a collection of policies will be defined by translation into an I/O automaton [73, 72], providing a formal operational foundation for trust policies. Our main theorem (Theorem 5.2) proves in this *formal* model, that even in infinite height cpos the I/O automata will converge towards the local least fixed-point, as intended.

Secondly, often it is infeasible and even unnecessary to compute the *exact* denotation of a set of policies. In many cases it is sufficient (in order to make a trust-based decision) to know that a certain property of this value is satisfied. In Section 5.6, we take very mild assumptions on the relation between the two orderings in trust structures. This enables us to develop two efficient protocols for safe approximation of the least fixed-point. Often this allows principals to take security-decisions without having to compute the exact fixed-point. For example, suppose we know a function  $I : \mathcal{P} \rightarrow \mathcal{P} \rightarrow D$  with the property that  $I \preceq \overline{\text{gts}}$ . In many trust structures it is the case that if  $I$  is sufficient to authorise a given request, so is the actual fixed-point. Furthermore, we provide a formal description of the two approximation protocols, and sketch a proof of their correctness, also in terms of I/O automata.

## 5.2 A Basic Language for Trust Policies

In this section we present a simple language for writing trust policies. The language is similar to that of Carbone *et al.* [19], but simplified slightly. We

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

Figure 5.1: Syntax

provide a denotational semantics for the language which is similar to the denotational semantics of Carbone et al. Throughout this paper we let  $\mathcal{P}$  be a finite set of principal identities, and  $(D, \sqsubseteq, \preceq)$  be a trust structure.

**Syntax.** We assume a countable collection of  $n$ 'ary function symbols  $\text{op}_n^i$  for each  $n > 0$ . These are meant to denote functions  $\llbracket \text{op}_n^i \rrbracket^{\text{den}} : D^n \rightarrow D$ , continuous with respect to  $\sqsubseteq$ . The syntax is given in Figure 5.1. A policy  $\pi$  is essentially a list of pairs  $p : \tau$ , where  $p$  is a principal identity, and  $\tau$  is an expression defining the policy's trust-specification for  $p$ . Since we cannot assume that the writer of the policy knows all principals, we include a generic construct  $\star : \tau$ , which intuitively means “for everyone not mentioned explicitly in this policy, the trust specification is  $\tau$ .” Note this could easily be extended to more practical constructs, say  $G : \tau$  meaning that  $\tau$  is the trust-specification for any member of the group (or role)  $G$ .

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

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

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

Figure 5.2: Denotational Semantics,  $\pi$ 

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

Figure 5.3: Denotational Semantics,  $\tau$ 

**Denotational Semantics.** The denotational semantics of the basic policy language is given in Figure 5.2, 5.3 and 5.4. We assume that for each of the function symbols  $\text{op}_n^i$ ,  $\llbracket \text{op}_n^i \rrbracket^{\text{den}}$  is a  $\sqsubseteq$ -continuous function of type  $D^n \rightarrow D$ . The semantics follows closely the ideas presented in the introduction, and a more detailed explanation is given by Carbone et al. [19]. For a collection  $\Pi = (\pi_p \mid p \in \mathcal{P})$ , the semantics of each  $\pi_p$  is an information-continuous function  $\llbracket \pi_p \rrbracket^{\text{den}}$  of type  $\text{GTS} \rightarrow \text{LTS}$ .

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

**An example.** We present a small and very simple example, intended only to illustrate the denotational semantics presented above. Let us consider an example with 3 principals, named R, A and B. The example is meant to illustrate the situation where R wants to compute its trust value in a certain fixed subject  $S$  (which won't be involved in the computation). We will use the so-called “MN trust structure”  $T_{MN} = (D, \preceq, \sqsubseteq)$ , where trust values are

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

Figure 5.4: Denotational Semantics,  $\Pi$

pairs of (extended) natural numbers, i.e.,  $D = (\mathbb{N} \cup \{\infty\})^2$ , and  $(m, n) \in D$  intuitively represents a history of  $m + n$  interactions with a principal with  $m$  interactions classified as “good” and  $n$  as “bad.” 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  $n \geq n'$ .

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

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

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

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

and

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

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

Let us illustrate the *synchronous* or *central* least fixed-point computation. This is described by Table 5.1 containing the “synchronous” entries of  $\Pi_{\lambda}^i(\perp_{\sqsubseteq})$  (i.e.,  $\perp_{\sqsubseteq}, \Pi_{\lambda}(\perp_{\sqsubseteq}), \Pi_{\lambda}(\Pi_{\lambda}(\perp_{\sqsubseteq})), \dots$ ). In the table, column  $x$  of row  $i + 1$  is obtained by applying policy  $\pi_x$  to row  $i$ , e.g., the value  $(4, 0)$  in column  $R$  of row 2 is obtained by  $\pi_R$  and row 1, as illustrated by the following informal “calculation:”

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

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

Table 5.1: Example centralised fixed-point computation.

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

### 5.3 An Operational Semantics

Clearly, the *synchronous* algorithm illustrated in the previous section is not feasible as a general algorithm for computing the meaning of a collection of policies large, open distributed systems. In this section, we develop an *asynchronous* counterpart to the synchronous algorithm. This is done in terms of I/O automata, a natural model of asynchronous distributed computation. We then proceed to give the operational semantics of the trust policy language. More specifically, in the operational semantics, a principal-indexed collection of policies  $\Pi$  is translated into an I/O Automaton, denoted  $\llbracket \Pi \rrbracket^{\text{op}}$ . I/O Automata are a form of labelled transition systems, suitable for modelling and reasoning about distributed discrete event systems [72, 73]. Later, we shall define also another translation  $\llbracket \Pi \rrbracket^{\text{op-abs}}$  into what we call a Bertsekas Abstract Asynchronous System (BAAS). There will be a tight correspondence between the “abstract” operational semantics  $\llbracket \Pi \rrbracket^{\text{op-abs}}$  and the actual operational semantics  $\llbracket \Pi \rrbracket^{\text{op}}$ . The reason for introducing  $\llbracket \cdot \rrbracket^{\text{op-abs}}$  is to make reasoning about the actual operational semantics easier. More specifically, we will make use of a general convergence result of Bertsekas for BAAS’s. By virtue of the connection between the semantics, this result translates into a result about the runs of the concrete I/O automaton  $\llbracket \Pi \rrbracket^{\text{op}}$ .

#### 5.3.1 The I/O Automaton Model

We review the basic definitions of I/O automata. For a more in-depth treatment, we refer to Lynch’s book [72]. An I/O automaton is a (possibly infinite) state automaton, where transitions are labelled with so-called actions. An *action signature*  $S$  is given by a set  $\text{acts}(S)$  of actions, and a partition of this set into three sets  $\text{in}(S)$ ,  $\text{out}(S)$  and  $\text{int}(S)$  of *input*, *output* and *internal* actions, respectively. We denote by  $\text{local}(S) = \text{out}(S) \cup \text{int}(S)$  the set of locally controlled actions.

**Definition 5.1 (I/O Automaton)** An *input/output automaton*  $A$ , consists of five components:

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

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

An important feature of I/O automata is that input-actions are always enabled. This property means that while the automaton can put restrictions on when output and internal actions are performed, it cannot control when input actions are performed. Instead, this is controlled by the environment.

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

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

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

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

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

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

Figure 5.5: Interface of the  $\mathbf{Channel}(p, q)$  I/O automaton.

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

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

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

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

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

**Example 5.1 (Channel)** The  $\mathbf{Channel}$  automaton is meant to model a reliable asynchronous communication channel in a network. Suppose  $\mathcal{P}$  is a set of principal identities, and  $V$  is a countable set of values. The channel is a one-way communication channel between two identities, transmitting values from  $V$ . The automaton is *parametric* in two principal identities, meaning that for any  $p, q \in \mathcal{P}$ ,  $\mathbf{Channel}(p, q)$  is an I/O automaton (intend to model a FIFO communication channel that can  $p$  can use to send  $V$ -values to  $q$ ). Fix any two  $p, q \in \mathcal{P}$ , and consider the following data.

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

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

and the output actions are

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

The signature is illustrated graphically in Figure 5.5.

- $states(\mathbf{Channel}(p, q)) = V^*$ , the set of finite sequences of elements from  $V$ . A state  $s = v_1 \cdot v_2 \cdots v_n$  represents  $n$  messages in transit from  $p$  to  $q$  (sent in that particular order).
- $start(\mathbf{Channel}(p, q)) = \epsilon$  (the empty sequence).
- $steps(\mathbf{Channel}(p, q))$  is given by the following. For any state  $s \in V^*$  and any  $v \in V$ , we have

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

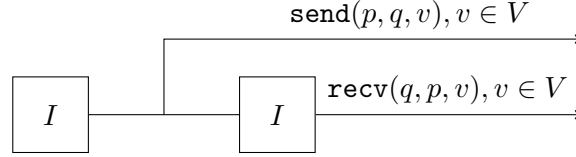
For any  $v_0 \in V$  and any sequence  $s = v_0 \cdot s' \in V^+$ , we have

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

- $part(\mathbf{Channel}(p, q))$  is the trivial partition where all  $\mathbf{recv}(q, p, v)$  actions are in the same equivalence class.

We will often use a pseudo-language for specifying I/O automata. The language is similar to IOA [39, 40], and its semantics should be clear. In the language, an automaton is given by specifying its signature, state, actions, transitions and partition. The state is given in terms of a collection of variables, for example,  $\mathbf{buffer} : Seq[V] := \{\}$  declares a variable “**buffer**” of type “sequences of values from the set  $V$ ,” and initialises this variable to the empty sequence. The transitions are given in a precondition/effect-style, where the precondition represents the set of states in which the action is enabled. The effect is an imperative program, executed atomically, manipulating the state variables.

The syntactic representation of the  $\mathbf{Channel}(p, q)$  automaton is the following.

Figure 5.6: The interface of  $A \times \text{Channel}(p, q)$ .

```

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

```

**Example 5.2 (Composition)** Continuing from the previous example, consider now an automaton  $A$  which will represent principal  $p$ . Suppose  $A$  has the following signature  $\text{sig}(A)$ :  $\text{acts}(A) = \{\text{send}(p, q, v) \mid v \in V\} \cup I$ ,  $\text{in}(A) = \emptyset$ ,  $\text{int}(A) = I$ , and  $\text{out}(A) = \{\text{send}(p, q, v) \mid v \in V\}$ , where  $I$  is some set of internal actions (disjoint from any other set of actions in this example). Then  $A$  and  $\text{Channel}(p, q)$  are compatible automata, and their composition  $A \times \text{Channel}(p, q)$  is illustrated in Figure 5.6. Notice that the composition has no input actions, but output actions  $\{\text{send}(p, q, v) \mid v \in V\} \cup \{\text{recv}(q, p, v) \mid v \in V\}$ .

### 5.3.2 $\llbracket \cdot \rrbracket^{\text{op}}$ Translation: An Operational Semantics

The concrete operational semantics can be seen as an asynchronous distributed algorithm in which the principals  $\mathcal{P}$  perform a computation of  $\llbracket \Pi \rrbracket^{\text{den}}$ . Each principal  $p \in \mathcal{P}$  will be computing its local values (i.e.,  $\llbracket \Pi \rrbracket^{\text{den}}(p)(q)$  for each  $q \in \mathcal{P}$ ). We present now such an algorithm, inspired by the work of Bertsekas [6]. We then give an I/O automata version of this algorithm, which is used for formal proofs. We use the following “dot notation:” if  $A$  is an I/O-automaton  $A.v$  refers to variable  $v$  of  $A$ .

**Algorithm.** The asynchronous algorithm is executed in a network of nodes, each denoted  $pq$  for  $p, q \in \mathcal{P}$ . A node  $pq$  represent a component of principal  $p$ , which is responsible for computing the value  $\llbracket \Pi \rrbracket^{\text{den}}(p)(q)$ . Each node  $pq$  allocates variables  $pq.t_{\text{cur}}$  and  $pq.t_{\text{old}}$  of type  $D$ , which will later record the “current” value and the last computed value. Each node  $pq$  has also a matrix, denoted by  $pq.gts$ , of type  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ . Initially,  $pq.t_{\text{cur}} = pq.t_{\text{old}} = \perp_{\square}$ , and the matrix is also initialised with  $\perp_{\square}$ . For all nodes  $pq$  and  $rs$ , when  $pq$  receives a message from  $rs$ , it stores this message in  $pq.gts(r)(s)$  (messages are always values in  $D$ , i.e., ‘updates’). Any node is always in one of two states: *sleep* or *wake*. All nodes start in the *wake* state, and if a node is in the *sleep* state, the reception of a message triggers a transition to the *wake* state. In the *wake* state any node  $pq$  repeats the following: it starts by assigning to variable  $pq.t_{\text{cur}}$  the result of applying its function  $f_{pq}$  to the values in  $pq.gts$  (function  $f_{pq}$  corresponds to  $p$ ’s policy entry for  $q$ ). If there is no change in the resulting value (compared to  $pq.t_{\text{old}}$ ), it will go to the *sleep* state (unless a new message was received since the computation). Otherwise, if a new value resulted from the computation, an update is sent to all nodes.

Although we have presented the algorithm so that each principal has an approximation for *every* other principal (i.e., the arrays  $pq.gts$ ), it is only necessary for principal  $p$  to store approximations for the principals that  $p$  depends on (in its policy). Similarly,  $pq$  only needs to send updates to nodes  $xy$  that depend on  $p$ ’s value for  $q$ . While the semantics of trust policies are functions of type  $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow \mathcal{P} \rightarrow D$  which (due to policy referencing) in general may depend on the trust values of *all* principals, we expect that in practice, policies will not be written in this way. Instead, policies are likely to refer to a few known (and usually “trusted”) principals. For fixed  $p$  and  $q$ , the set of principals that  $p$ ’s policy *actually* depends on in its entry for  $q$ , is often a significantly smaller subset of  $\mathcal{P}$ . For example, consider the policy:

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

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

It is easy to convert the proposed algorithm into one that takes dependencies into account (for details, including a distributed algorithm for computing a dependency graph from a set of policies, consult the technical report [60]).

**Communication model.** We use an asynchronous communication-model, assuming no known bound on the time it takes for a sent message to arrive. We assume that communication is reliable in the sense that any message sent eventually arrives, exactly once, unchanged, to the right node, and that messages arrive in the order in which they are sent. We assume (in the spirit of the global-computing vision) an underlying physical communication-network allowing any node to send messages to any other node. Furthermore, we assume that all nodes are willing to participate, and that they do not fail (however, we believe that the fixed-point algorithm we apply is highly robust [6]).

### An example

In this subsection, we give a small example of a run of the asynchronous algorithm. Let us consider an extended version of the simple example from Section 5.2. Our example has 5 principals, named R, A, B, C and D. The example is meant to illustrate the situation where R wants to compute its trust value in a certain fixed subject  $S$  (which won't be involved in the computation). We will use the  $MN$  trust-structure  $T_{MN}$  (Section 5.1.1) in the example.

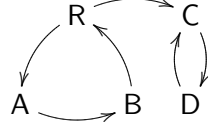
**Policies.** The policies of the principals are the following:

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

The constructs  $\text{Loc}_p$  (for  $p \in \mathcal{P}$ ) are special unary operators:  $\text{Loc}_p(q)$  refers to the trust-value derived from the local observations made by principal  $p$  about the subject  $q \in \mathcal{P}$  (this construct is discussed also by Nielsen and Krukow [82]). For example, R's policy for the subject  $S$  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.

The dependency graph derived from the policies is given in Figure 5.7.

Figure 5.7: Example dependency-graph.



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

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

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

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

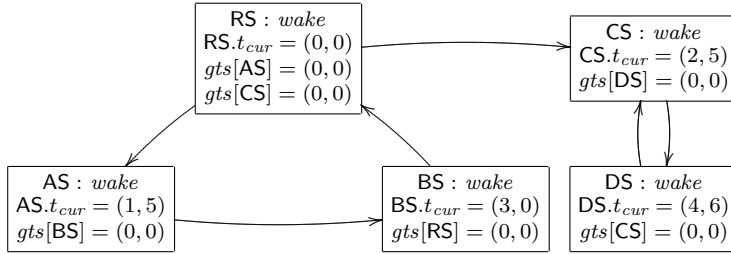
It is easy to verify that the last row in the table below is the least fixed-point of the policies (i.e., iterating round 6 will give the same row as iteration 5).

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

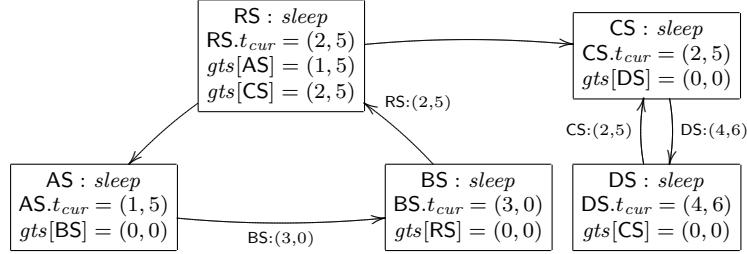
**An asynchronous run.** We now show a possible run of the asynchronous algorithm for the same set of policies as above. We illustrate the algorithm by showing the local-states of the nodes in the network at various points in time. The nodes are denoted by boxes describing the local state in terms of values of arrays  $xy.gts$ , and the values of  $xy.t_{cur}$ . 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. Note also, only nodes  $xy$  of form  $xS$  are considered (since these are the only relevant nodes for the computation).

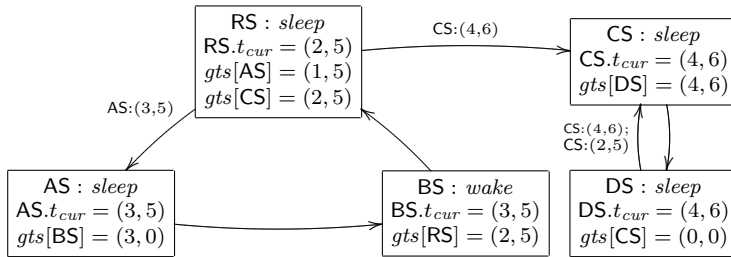
**Network Snapshot 1.** We assume that the initial states of the nodes are given by the following. All nodes are *wake*, the arrays are initialized to  $\perp_{\square} = (0, 0)$ . Each node  $xy$  has  $xy.t_{cur} = \llbracket \pi_x \rrbracket^{\text{den}}(xy.gts)(y)$ .

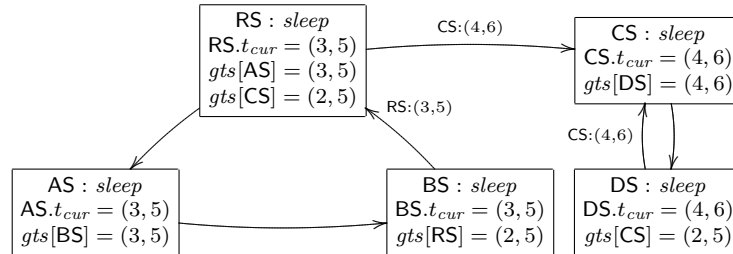


**Network Snapshot 2.** Here RS has received value  $(1, 5)$  from AS and  $(2, 5)$  from CS. Further values are in transit, e.g. value  $RS : (2, 5)$  “on” edge  $BS \rightarrow RS$  represents a message in transit from RS to BS.

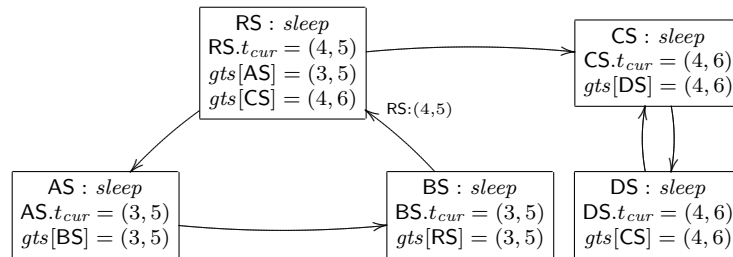
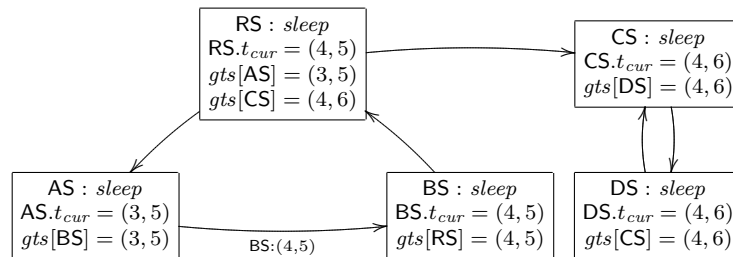


**Network Snapshot 3.** Two messages are in transit on the (presumably slow) path from CS to DS (we are assuming a reliable network, so the first sent will also arrive first). BS has just finished computing  $\llbracket \pi_B \rrbracket^{\text{den}}(BS.gts)(S) = (3, 5)$ , but has not yet sent this value.

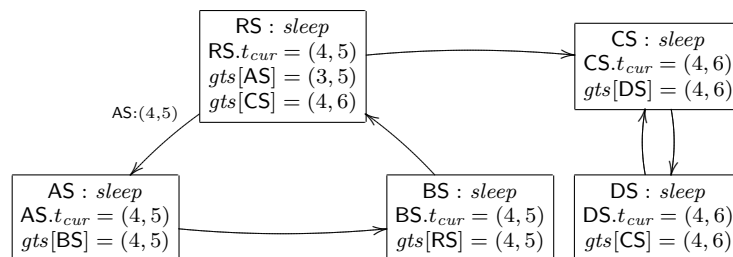


**Network Snapshot 4.**

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

**Network Snapshot 6.**

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



### Formalisation via I/O Automata

In the I/O-automaton version of this algorithm, each principal  $p$  is modelled as a collection of nodes,  $(pq \mid q \in \mathcal{P})$ , where component  $pq$  of  $p$  is responsible for computing  $p$ 's value for  $q$  (technically, principal  $p$  is modelled as the automaton-composition of each of the component-automata  $pq$  for  $q \in \mathcal{P}$ ). The sending of a message  $d$  from component  $pq$  to, say  $rs$ , is represented by the I/O-automaton action  $\text{send}(p, r, q, d)$  (note this is independent of  $s$  because all components  $rs$  will receive this update simultaneously). Note that the formal version of the algorithm does not consider policy-dependencies. While it is not too hard to change the algorithm to one which does [60], we choose to leave this out here as it is not too interesting (technically), and it adds complexity to the presentation.

The semantic function  $\llbracket \cdot \rrbracket^{\text{op}}$  maps a collection of trust policies from the basic language to an I/O automaton. The semantics uses two parameterised I/O automata:  $\text{Channel}(p, q, r)$  (modelling a communication medium for sending values from  $pq$  to  $r$ ), and  $\text{IOTemplate}(p, q, f_{pq})$ , where  $p, q, r \in \mathcal{P}$  and  $f : (\mathcal{P} \rightarrow \mathcal{P} \rightarrow D) \rightarrow D$  is a continuous function. The latter  $\text{IOTemplate}(p, q, f_{pq})$  is the component we denoted “ $pq$ ” when  $f_{pq}$  is principal  $p$ 's policy for principal  $q$ , i.e, the entry for  $q$  in  $\pi_p$ .

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

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

The Channel automaton is described syntactically below.

```

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

```

A principal  $p$  is represented as the automaton  $\llbracket \pi_p \rrbracket_{p,\emptyset}^{\text{op}}$  which is the composition of the collection of automata  $\text{IOTemplate}(p, q, f_{pq})$  for  $q \in \mathcal{P}$  and where  $f_{pq}(\text{gts}) = \llbracket \pi_p \rrbracket^{\text{den}} \text{gts } q$ . The component  $\text{IOTemplate}(p, q, f_{pq})$ , denoted simply as “ $pq$ ”, is responsible for computing principal  $p$ ’s trust value for principal  $q$ , i.e., value  $\overline{\text{gts}}(p)(q)$ . The semantics of each policy  $\pi$  is given by the following.

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

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

```

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

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

internal eval(p,q)

```

```

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

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

```

**Lemma 5.1 (Composability)** *If all policies of  $\Pi$  are well-formed, then all the automata occurring in the definition of  $\llbracket \Pi \rrbracket^{op}$  have compatible signatures, and, hence, are composable.*

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

### 5.3.3 Reasoning about the Semantics: Cause and Effect

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

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

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

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

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

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

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

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

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

Define

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

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

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

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

The following lemma establishes some simple properties of the  $\mathit{cause}_{r_c}$  function.

**Lemma 5.2 (Simple properties of  $\mathit{cause}$ )** *For any run  $r_c$ , function  $\mathit{cause}_{r_c}$  satisfies the following.*

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

*Proof.* See Section 5.7. □

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

**Lemma 5.3 (FIFO)** *Let  $r_c = s_0a_1s_2\cdots$  be a finite or infinite fair run of  $\text{Channel}(p, r, q)$  for  $p, r, q \in \mathcal{P}$ . Suppose that  $a_k = \text{send}(p, r, q, d)$  and  $a_{k'} = \text{send}(p, r, q, d')$  for some  $d, d' \in D$ . If  $k \leq k'$  then there exists unique  $j, j'$  with  $k < j$  and  $k' < j'$ , so that  $j \leq j'$ ,  $a_j = \text{recv}(r, p, q, d)$ ,  $a_{j'} = \text{recv}(r, p, q, d')$ ,  $\text{cause}_{r_c}(j) = k$  and  $\text{cause}_{r_c}(j') = k'$ .*

*Proof.* See Section 5.7 □

Notice that, by the above, if  $a_k = \text{send}(p, r, q, d)$  then uniqueness of  $j$  with  $\text{cause}_{r_c}(j) = k$  implies that  $\text{effect}_{r_c}(k) = \{j\}$ . By abuse of notation, we write

$$\text{effect}_{r_c}(k) = j.$$

Hence,  $\text{cause}_{r_c}(\text{effect}_{r_c}(k)) = k$ . This implies also that if  $a_m = \text{recv}(r, p, q, d)$  then  $\text{effect}_{r_c}(\text{cause}_{r_c}(m)) = m$ .

**Lemma 5.4 (Cause and Effect)** *Let  $\Pi = (\pi_p \mid p \in \mathcal{P})$  be a collection of policies, and let  $r_c = s_0a_1s_1a_2\cdots$  be a finite or infinite fair run of  $\llbracket \Pi \rrbracket^{\text{op}}$ . The following properties hold of  $r_c$ :*

1. *Assume that for some  $k \geq 0$ , we have  $s_k.pq.\text{wake} = \text{true}$ , then there exists a  $k' > k$  so  $a_{k'} = \text{eval}(p, q)$ .*
2. *Assume that  $a_{k_0} = \text{eval}(p, q)$  and that  $s_{k_0-1}.pq.t_{\text{cur}} \neq s_{k_0}.pq.t_{\text{cur}} = d$ . Let  $k_1 > k_0$  be least with  $a_{k_1} = \text{eval}(p, q)$  (note, such an index must exist by the above). Then, for every  $r \in \mathcal{P}$  there exists a unique  $k_r$  with  $k_0 < k_r < k_1$  so that  $a_{k_r} = \text{send}(p, r, q, -)$ . Furthermore,  $a_{k_r} = \text{send}(p, r, q, d)$  and  $\text{cause}_{r_c}(k_r) = k_0$ .*
3. *Assume that  $a_k = \text{send}(p, r, q, d)$  and also that  $a_{k'} = \text{send}(p, r, q, d')$ . Then,  $k < k'$  implies  $\text{cause}_{r_c}(k) < \text{cause}_{r_c}(k')$ .*
4. *Assume that  $a_k = \text{recv}(r, p, q, d)$  and also that  $a_{k'} = \text{recv}(r, p, q, d')$ . Then,  $k < k'$  implies  $\text{cause}_{r_c}(k) < \text{cause}_{r_c}(k')$ .*

*Proof.* See Section 5.7 □

## 5.4 Bertsekas Abstract Asynchronous Systems

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

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

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

**BAAS's.** A *Bertsekas Abstract Asynchronous System* (BAAS) is a pair  $B = ((X_i)_{i=1}^n, (f_i)_{i=1}^n)$  consisting of  $n$  sets  $X_1, X_2, \dots, X_n$ , and  $n$  functions  $f_1, f_2, \dots, f_n$ , where for each  $i$ ,  $f_i : \prod_{j=1}^n X_j \rightarrow X_i$ . Let  $X = \prod_{i=1}^n X_i$  and  $[n] = \{1, 2, \dots, n\}$ . We assume that there is a (partial) notion of convergence on  $X$ , so that some sequences  $(x^i)_{i=1}^\infty, x^i \in X$  have a unique limit point,  $\lim_i x_i \in X$ . We let  $f$  denote the product function  $f = \langle f_1, f_2, \dots, f_n \rangle : X \rightarrow X$ . The objective of a BAAS is to find a fixed point  $x^*$  of  $f$ .

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

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

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

Examples of BAAS's include distributed optimisation, numerical and dynamic programming algorithms [6].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Definition 5.2 (Fairness)** *We say that a run is finite if all the sets  $T^i$  are finite. If a run is not finite, it is infinite. An infinite run  $r$  of a BAAS is fair if for each  $i \in [n]$ :*

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

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

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

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

**Lemma 5.5** *If  $r$  is a finite fair run of a BAAS  $B$ , then  $x(t^*)$  is a fixed point of the product function of  $B$ .*

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

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

□

### 5.4.1 The Asynchronous Convergence Theorem

The Bertsekas abstract asynchronous systems are models of asynchronous distributed algorithms. The so-called Asynchronous Convergence Theorem (ACT) (Proposition 6.2.1 of Bertsekas' book [6]) is a general theorem which gives sufficient conditions for BAAS runs to converge to a fixed point of the product function  $f$ . The ACT applies in any scenario in which the so-called “Synchronous Convergence Condition” and the “Box Condition” are satisfied. Intuitively, the synchronous convergence condition states that if the algorithm is executed synchronously, then one obtains the desired result. In our case, this amounts to requiring that the “synchronous” sequence  $\perp_{\square}^n \sqsubseteq f(\perp_{\square}^n) \sqsubseteq \dots$  converges to the least fixed-point, which is true for continuous  $f$ . Intuitively, the box condition requires that one can split the set of possible values appearing during synchronous computation into a product (“box”) of sets of values that appear locally at each node in the asynchronous computation.

We now recall the definition of the Synchronous Convergence Condition (SCC) and the Box Condition (BC) (Section 6.2 [6]). Consider a BAAS with  $X = \prod_{i=1}^n X_i$ , and  $f : X \rightarrow X$  any function with  $f = \langle f_1, f_2, \dots, f_n \rangle$ .

**Definition 5.3 (SCC and BC)** Let  $\{X(k)\}_{k=0}^{\infty}$  be a sequence of subsets  $X(k) \subseteq X$  satisfying  $X(k+1) \subseteq X(k)$  for all  $k \geq 0$ .

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

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

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

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

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

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

**Theorem 5.1 (ACT, Bertsekas [6])** Let  $B = ((X_i)_{i=1}^n, (f_i)_{i=1}^n)$  be a Bertsekas Abstract Asynchronous System,  $X = \prod_{i=1}^n X_i$ , and  $f = \langle f_i : i \in [n] \rangle$ . Let  $\{X(k)\}_{k=0}^{\infty}$  be a sequence of sets with  $X(k) \subseteq X$  and  $X(k+1) \subseteq X(k)$  for all  $k \geq 0$ . Assume that  $\{X(k)\}_{k=0}^{\infty}$  satisfies the SCC and the BC. Let  $r$  be any infinite fair run of  $B$ , with initial solution estimate  $x(0) \in X(0)$ . Then, if  $\{x(t)\}_{t \in \mathbb{N}}$  has a limit point, this limit point is a fixed point of  $f$ .

#### 5.4.2 $\llbracket \cdot \rrbracket^{\text{op-abs}}$ Translation: An Abstract Operational Semantics

We map a collection of policies  $\Pi = (\pi_p \mid p \in \mathcal{P})$  to a BAAS, in a way similar to the concrete operational semantics. The BAAS  $\llbracket \Pi \rrbracket^{\text{abs-op}}$  consists of the set  $D$  of trust values, a collection of  $n = |\mathcal{P}|^2$  functions  $f_{pq} : D^n \rightarrow D$  (the functions are indexed by pairs  $pq$  where  $p, q \in \mathcal{P}$ ). The functions  $f_{pq}$  are given by the policies,

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

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

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

Note that  $\Pi_\lambda = \langle \langle f_{pq} \mid q \in \mathcal{P} \rangle \mid p \in \mathcal{P} \rangle = f$  (hence a value  $\hat{d} \in D^n$  is a fixed point of  $f$  if-and-only-if it is a fixed point of  $\Pi_\lambda$ ).

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

### 5.4.3 Correspondence of Abstract and Concrete Operational Semantics

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

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

1. For any  $p, q \in \mathcal{P}$ ,  $T^{pq}$  is defined as  $\{k - 1 \mid k \in \mathbb{N}, a_k = \text{eval}(p, q)\}$ . That is, the update-times of  $pq$  are the indexes of pre-states of  $\text{eval}(p, q)$  actions in  $r_c$ . Note that for  $(p, q) \neq (r, s)$  we have an empty intersection,  $T^{pq} \cap T^{rs} = \emptyset$ .
2. For each  $p, q \in \mathcal{P}$ , the function  $\tau_{pq}^{pq} : \mathbb{N} \rightarrow \mathbb{N}$  is given by the identity  $\tau_{pq}^{pq}(t) = t$ . This reflects the fact that node  $pq$  always has an exact “estimate” of its own current value, i.e.,  $x^{pq}(t)_{pq} = x_{pq}(\tau_{pq}^{pq}(t)) = x_{pq}(t)$ .
3. For each  $p, q \in \mathcal{P}$  and each  $r, s \in \mathcal{P}$  with  $(r, s) \neq (p, q)$ , the function  $\tau_{rs}^{pq} : \mathbb{N} \rightarrow \mathbb{N}$  is given by the following. Let  $t \in \mathbb{N}$  be arbitrary but fixed.
  - (a) Let  $k \leq t$  be the *largest*, with the property that  $a_k = \text{recv}(p, r, s, d)$  for some  $d \in D$ . If no such index exists, then  $\tau_{rs}^{pq}(t)$  is defined as the largest  $j \leq t$  with the property that for all  $j'$  with  $0 \leq j' \leq j$  we have  $s_{j'}.rs.t_{cur} = \perp_{\square}$ . If such  $k$  exists, let  $k' = \text{cause}_{r_c}(k)$ . Note that  $a_{k'} = \text{send}(r, p, s, d)$ .
  - (b) We then define  $k'' = \text{cause}_{r_c}(k')$ . Note that  $a_{k''} = \text{eval}(r, s)$ , and that we must have  $s_{k''}.rs.t_{cur} = d$ .
  - (c) Finally, define  $\tau_{rs}^{pq}(t)$  to be the largest index  $j \leq t$  with the property that for all  $j'$  with  $k'' \leq j' \leq j$ , also  $s_{j'}.rs.t_{cur} = d$ . Note, in particular  $s_j.rs.t_{cur} = d$ .

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

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

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

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

**Lemma 5.6** *For any  $p, q, r, s \in \mathcal{P}$ , function  $\tau_{rs}^{pq}$  is monotonically increasing.*

*Proof.* See Section 5.7 □

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

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

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

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

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

**Lemma 5.7 (Corresponding runs)** *Let  $\Pi = (\pi_p \mid p \in \mathcal{P})$  be a collection of policies. Let  $r_c$  be a run of  $[[\Pi]]^{\text{op}}$ , and let  $r_a$  be the corresponding run. Then,*

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

*Proof.* See Section 5.7 □

**Lemma 5.8** *For any fair run  $r_c$  of  $\llbracket \Pi \rrbracket^{op}$ , let  $r_a$  denote its corresponding run. Then,*

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

*Proof.* See Section 5.7 □

## 5.5 Correspondence of Denotational and Operational Semantics

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

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

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

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

*Proof.* See Section 5.7 □

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

**Definition 5.4 (Information Approximation)** *Let  $(X, \sqsubseteq)$  be a CPO with bottom  $\perp_{\sqsubseteq}$ . Let  $f : X \rightarrow X$  be any continuous function. An element  $x \in X$  is an information approximation for  $f$  if*

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

**Lemma 5.9** *Let  $(X, \sqsubseteq)$  be a CPO with bottom  $\perp_{\sqsubseteq}$ . Let  $f : X \rightarrow X$  be any continuous function, and  $\hat{x} \in X$  an information approximation for  $f$ . Then the set,  $\{f^k(\hat{x}) \mid k \in \mathbb{N}\}$  is a chain and*

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

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

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

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

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

□

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

*Proof.* First we must show that the sequence  $\{x(t)\}_t$  actually has a limit. We show that  $x(0) \sqsubseteq x(1) \sqsubseteq \dots \sqsubseteq x(t) \sqsubseteq \dots$ , i.e.,  $\{x(t)\}_t$  is an increasing omega chain. This follows from Proposition 5.1 since

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

and we have  $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$  for all  $t$ . Now to show that  $\lim_t x(t)$  (which is actually  $\bigsqcup_t x(t)$ ) is a fixed point of  $\Pi_\lambda$ , we shall invoke the Asynchronous Convergence Theorem of Bertsekas. Define a sequence of subsets of  $X^n$ ,  $X(0) \supseteq X(1) \supseteq \dots \supseteq X(k) \supseteq X(k+1) \supseteq \dots$  by

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

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

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

The box condition is easy:

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

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

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

**Theorem 5.2 (Correspondence of semantics)** *Let  $\Pi$  be any collection of policies, indexed by a finite set  $\mathcal{P}$  of principal identities. Let  $r = s_0 \pi_1 s_1 \pi_2 s_2 \dots$  be any fair run of the operational semantics of  $\Pi$ ,  $\llbracket \Pi \rrbracket^{\text{op}}$ . Let  $\text{state}_{\text{con}}(r, k) = (E^k, V^k)$ , then  $\{V^k \mid k \in \mathbb{N}\}$  is a chain in  $(\mathcal{P} \rightarrow \mathcal{P} \rightarrow X, \sqsubseteq)$ , and*

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

*Proof.* First, map  $r_c$  to its corresponding run  $r_a$ . This is a fair run of  $\llbracket \Pi \rrbracket^{\text{op-abs}}$  by Lemma 5.8. By Lemma 5.7,  $\{x(t)\}_{t \in \mathbb{N}} = \{V^k\}_{k \in \mathbb{N}}$ , and by the Proposition 5.2  $\{x(t)\}_{t \in \mathbb{N}}$  has a limit which is  $\text{lfp } \Pi_\lambda = \llbracket \Pi \rrbracket^{\text{den}}$ .  $\square$

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

*Proof.* Let  $r'$  denote the corresponding run of the concrete run  $r = s_0 a_1 s_1 \dots$ . Proposition 5.1 implies that  $x(t)$  is an increasing chain. When  $(D, \sqsubseteq)$  has finite height, there exists some  $t_0$  so that for all  $t \geq t_0$ , we have  $x(t) = x(t_0)$ . But by Lemma 5.7 then for all  $t$  with  $t_0 \leq t < |r_c|$  we have  $s_t.pq.t_{\text{cur}} = s_{t+1}.pq.t_{\text{cur}}$  for all  $p, q \in \mathcal{P}$ . Hence there can only be finitely

many `send` actions after  $t_0$ , and hence only finitely many `recv` actions after  $t_0$ . But then there can only be finitely many `eval` actions in  $r$ , and, hence  $r$  must be finite. Since  $\sqcup x(t) = x(t_0)$  the correspondence theorem implies that  $V = \llbracket \Pi \rrbracket^{\text{den}}$ .  $\square$

### 5.5.1 Practical remarks about the operational semantics

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

Regarding complexity, we have the following results. Assume that trust structure  $(D, \sqsubseteq, \preceq)$  is finite with  $(D, \sqsubseteq)$  of height,  $h \in \mathbb{N}$ . Since any node sends values only when a change in its current value occurs, by monotonicity of policies, each node  $pq$  will send at most  $h \cdot M_{pq}$  messages, where  $M_{pq}$  denotes the number of principals that depend on  $p$ 's entry for  $q$ . Each message is of size  $O(\log |D|)$  bits.<sup>5</sup> Node  $pq$  will receive at most  $h \cdot N_{pq}$  messages, where  $N_{pq}$  denotes the number of principals that  $p$  depends on in its entry for  $q$ . Each message (possibly) triggers an evaluation of  $\pi_p$ .

Another practical issue concerns dynamics. Suppose that a policy  $\pi_p$  changes to  $\pi'_p$  at some point in time (during computation or not). Mathematically, this is not an issue: The result is a new well-defined global trust-state given by the fixed point of the collection  $(\pi_x \mid x \in \mathcal{P} \setminus \{p\}) \cup (\pi'_p)$ . However, in practice, how would principals compute the new local values given the information about the old fixed-point? When  $\pi_p$  and  $\pi'_p$  satisfies  $\pi_p \sqsubseteq \pi'_p$  (i.e.,  $\pi'_p$  is more precise than  $\pi_p$ ), this presents no problems: Principal  $p$  can change its policy locally, and continue the algorithm without notifying anyone; the correct fixed-point will be computed. For more general scenarios, the technical report of Krukow and Twigg [60] suggest some simple algorithms to reuse as much of the old information as possible. However, in the worst-case a complete recomputation is necessary. For this reason we do not present these minor results here.

---

<sup>5</sup>In fact, there will be *only*  $O(h)$  different messages, each sent to all of the nodes that depend on  $pq$ . Consequently, a broadcast mechanism could implement the message delivery efficiently.

## 5.6 Approximation Techniques

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

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

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

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

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

### 5.6.1 Proof-carrying Requests

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

**Proposition 5.3 (Proof-carrying requests)** *Assume that  $(D, \preceq, \sqsubseteq)$  is a trust structure where  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. If  $I \preceq \perp_{\sqsubseteq}^n$  and  $I \preceq f(I)$ , then  $I \preceq \text{lfp}_{\sqsubseteq} f$ .*

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

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

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

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

□

As in Section 5.4.2, we assume that  $f$  is given by coordinate functions  $f_{pq}$  for  $pq \in [n]$  (which are the policies of principals). Returning to the approximation protocol, the prover  $p$  first sends  $I$  of type  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$  to  $v$ . If  $v$  can verify that  $I$  satisfies the properties  $I(x)(y) \preceq \perp_{\sqsubseteq}$  and  $I(x)(y) \preceq f_{xy}(I)$  (for all  $x, y \in \mathcal{P}$ ), then we can invoke the proposition to obtain  $I \preceq \text{lfp}_{\sqsubseteq} f$ . This means that if  $\sigma(p, I(v)(p)) = \top$ , also  $\sigma(p, \overline{\text{gts}}(v)(p)) = \top$ , and  $v$  can make a safe decision based on  $I$ .

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

### An Example

Consider for simplicity the “ $MN$ ” trust-structure  $T_{MN}$  from Section 5.1, which satisfies the information-continuity requirement. Recall that, in this structure, trust values are pairs  $(m, n)$  of natural numbers, representing a

number,  $m + n$ , of past interactions;  $m$  of which were classified ‘good’, and  $n$ , classified as ‘bad’.<sup>6</sup> 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  $\pi_v$  is monotonic, also with respect to  $\preceq$ , and that it depends on a large set  $S$  of principals. Assume also that it is sufficient that principals  $a$  and  $b$  in  $S$  have a reasonably “good” trust-value for  $p$ , to ensure that  $v$ ’s trust-value for  $p$  is not too “bad”. An example policy with this property could be:

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

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

**Protocol.** Principal  $p$  sends to  $v$  the information:

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

which can be thought of as a “proof” (analogous to a ‘proof-of-compliance’) or a “claim” made by  $p$ , stating that  $(0, N) \preceq \Pi_\lambda(v)(p)$  (and similarly for  $a$  and  $b$ ). Upon reception,  $v$  first extends  $I$  to a global trust state, which is the extension of  $I$  to a function of type  $\mathcal{P} \rightarrow \mathcal{P} \rightarrow D$ , given by

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

To check the proof, principal  $v$  must verify that  $\bar{I}$  satisfies the conditions of Proposition 5.3. First,  $v$  must check that  $\bar{I}(x)(y) \preceq \perp_{\sqsubseteq} = (0, 0)$  for all

<sup>6</sup>To be precise, the set  $\mathbb{N}^2$  is completed by allowing also value  $\infty$  as “ $m$ ” or “ $n$ ” or both.

$x, y$ . But this holds trivially if  $y \neq p$  or  $x \neq v, a, b$  because then  $\bar{I}(x)(y) = (0, \infty) = \perp_{\preceq}$ . For the other few entries it is simply an order-theoretic comparison  $\bar{I}(x)(y) \preceq (0, 0)$ . Now  $v$  tries to verify that  $\bar{I} \preceq \Pi_{\lambda}(\bar{I})$ . To do this,  $v$  verifies that  $(0, N) \preceq \llbracket \pi_v \rrbracket^{\text{den}}(\bar{I})(p)$ . If this holds then  $v$  sends the information  $I$  to  $a$  and  $b$ , and ask  $a$  and  $b$  to perform a similar verification (e.g.  $(0, N_a) \preceq \llbracket \pi_a \rrbracket^{\text{den}}(\bar{I})(p)$ ). Then  $a$  and  $b$  reply with ‘yes’ if this holds and ‘no’ otherwise. If both  $a$  and  $b$  reply ‘yes’, then  $p$  is sure that  $\bar{I} \preceq \Pi_{\lambda}(\bar{I})$ : By the checks made by  $v, a$  and  $b$ , we have that  $\bar{I}(x)(y) \preceq \Pi_{\lambda}(\bar{I})(x)(y)$  holds for pairs  $(x, y) = (v, p), (a, p), (b, p)$ , but for all other pairs it holds trivially since  $\bar{I}$  is the  $\preceq$ -bottom on these. By Proposition 5.3, we have  $\bar{I} \preceq \llbracket \Pi \rrbracket^{\text{den}}$ , and so,  $v$  is *ensured* that its trust value for  $p$  is  $\preceq$ -greater than  $(0, N)$ .

### 5.6.2 I/O-Automaton Version

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

```

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

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

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

```

```

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

```

**Verify** takes an identity  $v \in \mathcal{P}$  and  $v$ 's policy  $\llbracket \pi_v \rrbracket^{\text{den}}$  as parameters. The verifier has an input-action  $\text{proof}(I)$ , which models the reception of information  $I \subseteq \mathcal{P} \times \mathcal{P} \times D$ . We can view  $I$  as a claim made by a prover: for all  $(x, y, d) \in I$  we have  $d \preceq \overline{\text{gts}}(x)(y)$ . It is now the job of the verifier to check this by checking that  $I$  satisfies the assumptions of Proposition 5.3. First all  $(x, y, d) \in I$  with  $x = v$  are checked locally. For all  $(x, y, d)$  with  $x \neq v$  we add  $x$  to a set *pending* of non-local checks that are to be performed. Eventually, an output action  $\text{check}(x, I)$  is performed for each such  $x \in \text{pending}$ . This models the verifier sending a request to principal  $x$  to check that all “ $x$ -entries” of  $I$  satisfy the assumptions of the proposition. For each such message sent, the verifier expects to receive one of two possible replies from  $x$ :  $\text{reply-ok}(x)$  or  $\text{reply-fail}(x)$ . If all  $x \in \text{pending}$  send  $\text{reply-ok}(x)$  and the local checks succeeded, then the verifier performs the  $\text{accept}$  action to model the acceptance of the proof; otherwise  $\text{reject}$  is performed.

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

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

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

**Lemma 5.10 (PCR Protocol)** Let  $I \subseteq \mathcal{P} \times \mathcal{P} \times D$ , and let  $r = s_0 a_1 s_1 \dots$  be any fair run of the composite automaton  $\text{Verify}(v, \llbracket \pi_v \rrbracket^{\text{den}}) \times \prod_{x \in \mathcal{P} \setminus v} A_x$ . Assume that  $a_1 = \text{proof}(I)$ . Then there exists an index  $j$  so that  $a_j = \text{accept}$  or  $a_j = \text{reject}$ . Further  $a_j = \text{accept}$  if-and-only-if  $I$  satisfies: for

all  $(x, y, d) \in I$  we have  $d \preceq \llbracket \pi_x \rrbracket^{\text{den}}(I)(y)$  and  $d \preceq \perp_{\square}$ . Hence if  $a_j = \text{accept}$  then  $I \preceq \overline{\text{gts}}$ .

*Proof. (sketch)* Since,  $a_1 = \text{proof}(I)$  the state  $s_1$  satisfies  $\text{abort} = \text{false}$  if-and-only-if  $\forall (x, y, d) \in I : x = v \Rightarrow d \preceq \perp_{\square}, f_v(I)(y)$ . If  $\text{abort} = \text{true}$  in this state we are done, as  $\text{reject}$  is scheduled. Otherwise, for each  $(x_0, y, d) \in I$  with  $x_0 \neq v$  action  $\text{check}(x_0, I)$  is scheduled, and by fairness, eventually performed. By the assumptions about the  $A_x$ , action  $\text{reply-ok}(x_0)$  occurs if  $\forall (x, y, d) \in I : x = x_0 \Rightarrow d \preceq \perp_{\square}, f_{x_0}(I)(y)$ ; otherwise  $\text{reply-fail}(x_0)$  occurs. Assuming all  $x$  reply with “ok,” we obtain  $\text{ok} = \text{pending}$  and  $\text{accept}$  is eventually performed, which is correct because then we have  $\forall (x, y, d) \in I : d \preceq \perp_{\square}, f_x(I)(y)$ . Otherwise, some  $\text{reply-fail}(x)$  action was performed, and  $\text{reject}$  is eventually performed.  $\square$

As noted previously, the PCR protocol gives an efficient way of asserting  $\sigma(p, \overline{\text{gts}}(v)(p)) = \top$  without computing  $\overline{\text{gts}}(v)(p)$ . However, two essential practical problems arise with this “proof-carrying” approach. First, to construct proof  $I$ , the prover needs information about the verifiers trust policy and of the policies of those whom the verifier depends on. If policies are not public, it is not clear how the verifier would construct  $I$ . Second, because of the requirement in Proposition 5.3 that  $I \preceq \perp_{\square}^n$ , the protocol can usually only be used to prove properties limiting the amount of previous “bad behaviour,” and *not* properties guaranteeing sufficient previous “good behaviour.” For example, in the  $T_{MN}$  trust structure  $I \preceq \perp_{\square}^n = (0, 0)^n$  means that we can only allow information  $I$  with entries of the form  $(x, y, (0, N))$ , i.e., stating that  $x$ ’s trust in  $y$  has no more than  $N$  “bad” interactions.

### 5.6.3 A Snapshot-based Approximation Technique

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

**Proposition 5.4 (Snapshot [59])** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. Assume that  $I$  is an information approximation for  $f$ . If  $I \preceq f(I)$  then  $I \preceq \text{lfp}_{\sqsubseteq} f$ .*

*Proof.* Since  $I \preceq f(I)$ , monotonicity gives  $I \preceq f^k(I)$  for all  $k \geq 0$ .  $I$  is an information approximation for  $f$ , so Lemma 5.9 implies  $\bigsqcup_{k \in \mathbb{N}} f^k(I) = \text{lfp}_{\sqsubseteq} f$ . Finally, information continuity of  $\preceq$  implies  $I \preceq \text{lfp}_{\sqsubseteq} f$ .  $\square$

The above proposition requires that  $I$  be an information approximation. Fortunately, the invariance property of the operational semantics (Proposition 5.1) implies that we can, indeed, compute a snapshot  $I$  which is an information approximation. Once  $I$  is computed, each principal  $pq$  checks that  $I(p)(q) \preceq f_{pq}(I)$ . If all principals succeed with the check, Proposition 5.4 implies  $I \preceq \text{lfp} f$  as desired.

Imagine that during a run of the operational semantics there is a point in time in which no messages are in transit, all nodes  $pq$  have computed their function  $f_{pq}$ , and sent the value to all nodes. Thus we have a “consistent” state in the sense that for any node  $pq$  and any node  $rs$  we have  $pq.gts(r)(s) = rs.t_{cur}$ . In particular any  $pq$  and  $xy$  agree on  $rs$ ’s value:  $pq.gts(r)(s) = rs.t_{cur} = xy.gts(r)(s)$ . In this ideal state, there is a consistent vector  $I$  which, by Proposition 5.1 (and Lemma 5.7), is an information approximation for  $f$ . If all nodes  $pq$ , can make the local check  $I(p)(q) = pq.t_{cur} \preceq f_{pq}(I)$ , then  $I$  satisfies  $I \preceq f(I)$ , and hence  $I \preceq \overline{\text{gts}}$ .

The approximation algorithm computes a consistent view that corresponds to such an ideal situation, and incorporates the local checks. In so-called snapshot algorithms (see Lynch [72] or Bertsekas [6]), the (local views of the) global state of the system is recorded during execution of an algorithm. Our problem is slightly less complicated since we are not interested in the status of communication links, but slightly more complicated since each snapshot-value must be propagated to a specific set of nodes.

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

```

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

```

```

        y : P, d : D)
    reply-par(const p, const q, x : P,
              y : P, d : D, b : Boolean)
state
  val : D;
  parent : (P x P) + {undef} := undef;
  Iapp: P -> P -> D;
  ok : Boolean;
  reply_to: Set[PxP] := {};
  sent: Set[PxP] := {};
  recvd: Set[PxP] := {};
  initially
    \forall p, q \in P (Iapp(p)(q) = bot)
transitions
  input ping(x,y,p,q)
    eff if (parent = undef)
      then parent := (x,y);
        val := t_cur;
        Iapp(p)(q) := val;
        ok := true
      else reply_to.add(x,y)
    fi
  input reply(x,y,p,q,d)
    eff
      Iapp(x)(y) := d;
      recvd.add(x,y)
  input reply-par(x,y,p,q,d,b)
    eff
      Iapp(x)(y) := d;
      recvd.add(x,y);
      ok := ok /\ b
  output ping(p,q,x,y)
    pre parent != undef /\
      (x,y) \in Dep /\ not ((x,y) \in sent)
    eff sent.add(x,y)
  output reply(p,q,x,y,d)
    pre (x,y) \in reply_to /\ (x,y) != parent
      /\ d = val
    eff reply_to.remove(x,y)
  output reply-par(p,q,x,y,d,b)
    pre (x,y) = par /\ recvd = Dep /\ d = val
      /\ b = (ok /\ (val <= f_pq(Iapp)))
    eff par := undef
partition {ping(p,q,x,y) | x,y : P};
         {reply(p,q,x,y,d) | x,y : P,d:D};
         {reply-par(p,q,x,y,d,b) | x,y : P,
                               d:D,
                               b:Boolean};

```

The *Snapshot*( $p, q, \dots$ ) automaton is supposed to be viewed as an extension of the parametric *IOTemplate*( $p, q, f_{pq}$ ) automaton in the sense that it

must have access to its local state of  $pq$  and policy  $f_{pq}$ . Our I/O automaton implementation differs from the algorithm sketched by Krukow and Twigg [59] in that it does not assume a known spanning tree on the nodes. Instead, we assume simply that each node  $pq$  knows the set  $Dep \subseteq \mathcal{P}$  of principals that its policy  $f_{pq}$  depends on.

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

We assume that there is a special initiating automaton, **Init**( $p, v, Dep_{pv}$ ) (denoted simply  $pv$ ), which is like the snapshot automaton, but doesn’t have a parent (it is, in fact, the parent of the spanning tree to be formed). Automaton **Init**( $p, v, Dep_{pv}$ ) initiates the computation by performing output action **ping**( $p, v, x, y$ ) for each  $(x, y) \in Dep_{pv}$ , the dependencies of  $f_{pv}$ . Further,  $pv$  performs output action **accept** if it receives a “message” **reply-par**( $\dots, \mathbf{true}$ ) from each of its children in the spanning tree, and its local check succeeds; otherwise output action **reject** is performed. We have the following lemma, which establishes the correctness of the protocol.

**Lemma 5.11 (Snapshot Protocol)** Let  $r = s_0 a_1 s_1 \dots$  be any fair run of

$$\mathbf{Init}(p, v, Dep_{pv}) \times \prod_{x, y \in \mathcal{P}, (x, y) \neq (p, v)} \mathbf{Snapshot}(x, y, Dep_{xy}).$$

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

*Proof. (sketch)* We assume for simplicity that  $Dep_{pv} = \{xy\}$ , i.e., that  $pv$  only depends on a single node  $xy$  (the proof idea works also in the general case). Let us assume that  $pv$  initiates the protocol by performing action  $\text{ping}(p, v, x, y)$  and records its value in the operational semantics,  $pv.t_{cur}$ , at time 1 in  $r$ . Hence  $xy.parent = (p, v)$  and  $pv.val = pv.t_{cur}$ . Eventually, the set of *parent* “pointers” will form a spanning tree, rooted at  $pv$ . Further, each node  $rs$  in this tree is associated a value  $v_{rs}$  at the time it performs the input action  $\text{ping}(r, s, z, w)$  for some  $zw$ . For each such  $v_{rs}$  we have  $v_{rs} = x_{rs}(t_{rs})$  (using the notation of Section 5.3.2), for some time  $t_{rs}$ . Hence there is a consistent vector  $I = (\dots, v_{rs}, \dots) \in D^n$ , consisting of all these values; the vector is consistent because each node  $zw$ ’s view of the vector will agree. It follows from Proposition 5.1 that  $I$  is an information approximation (this is actually non-trivial). Each node  $zw$  in the spanning tree will make the check  $v_{zw} \preceq f_{zw}(I)$ , and if all the children of  $zw$  reply with  $\text{reply-par}(\dots, \text{true})$ ,  $zw$  will perform action  $\text{reply-par}(z, w, zw.parent, zw.val, \text{true})$ ; otherwise, action  $\text{reply-par}(z, w, zw.parent, zw.val, \text{false})$  is performed. Hence, if  $pv$  receives “true” from each of its children and its check succeeded, action  $\text{accept}$  is performed correctly. Otherwise, if the check fails or some child replies with “true,”  $\text{reject}$  is performed.  $\square$

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

**Proposition 5.5** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. If  $\perp_{\sqsubseteq}^n \preceq I$  and  $f(I) \preceq I$  then  $\text{lfp } f \preceq I$ .*

**Proposition 5.6** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $I \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. Assume that  $I$  is an information approximation for  $f$ . If  $f(I) \preceq I$  then  $\text{lfp } f \preceq I$ .*

Proposition 5.5 may not seem as useful as its dual. The conclusion  $\text{lfp } f \preceq I$  can usually only be used to *deny* a request, and a prover in the protocol for Proposition 5.5 would probably not be interested in supplying information which would help refuting its request! However, this is not always so. For example, suppose one is using trust structures conveying probabilistic information (e.g., [16, 82]), and that  $I \preceq J$  expresses (informally) that, when interacting with a certain principal, the probability of a specific outcome given  $I$ , is lower than the probability of that outcome given  $J$ . In this case, an assertion of the form  $\text{lfp } f \preceq I$ , can convince the verifier

that when interacting with the prover, the probability of a “bad” outcome is *below* a certain threshold.

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

**Proposition 5.7 (Generalised Approximation)** *Let  $(D, \preceq, \sqsubseteq)$  be a trust structure in which  $\preceq$  is  $\sqsubseteq$ -continuous. Let  $J \in D^n$ , and  $f : D^n \rightarrow D^n$  be any function that is  $\sqsubseteq$ -continuous and  $\preceq$ -monotonic. Assume that  $J$  satisfies  $J \preceq f(J)$ . If there exists an information approximation  $I \in D^n$  for  $f$ , with the property that  $J \preceq I$ , then  $J \preceq \text{lfp } f$ .*

*Proof.* The proof of Proposition 5.7 is similar to that of Proposition 5.3. Follow the path from  $J$  via  $\preceq$  to  $f^i(I)$  in the following diagram.

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

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

One obtains Proposition 5.3 with the trivial information approximation  $I = \perp_{\sqsubseteq}$ , and Proposition 5.4 by taking the proof to be the approximation, i.e.  $J = I$ .

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

We note finally that the  $\sqsubseteq$ -continuity property required of  $\preceq$  in our propositions is satisfied for all interesting trust-structures we are aware of: Theorem 3 of Carbone *et al.* [19] implies that the information-continuity condition is satisfied for all interval-constructed structures. Furthermore, their Theorem 1 ensures that interval-constructed structures are complete lattices with respect to  $\preceq$  (thus ensuring existence of  $\perp_{\preceq}$ ). Several natural examples of non-interval domains can also be seen to have the required properties [82]. Also, the requirement that all policies  $\pi_p$  are monotonic

also with respect to  $\preceq$  is sensible. It amounts to saying that if everyone raises their trust-levels in everyone, then policies should not assign lower trust levels to anyone.

## 5.7 Proofs

**Lemma 5.3 (Simple properties of cause)** *For any run  $r_c$ , function  $cause_{r_c}$  satisfies the following.*

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

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

$$cause_{r_c}(k) = \min(S_0 \setminus cause_{r_c}(R_0))$$

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

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

**Lemma 5.4 (FIFO)** *Let  $r_c = s_0 a_1 s_2 \dots$  be a finite or infinite fair run of  $\text{Channel}(p, r, q)$  for  $p, r, q \in \mathcal{P}$ . Suppose that  $a_k = \text{send}(p, r, q, d)$  and  $a_{k'} = \text{send}(p, r, q, d')$  for some  $d, d' \in D$ . If  $k \leq k'$  then there exists unique  $j, j'$  with  $k < j$  and  $k' < j'$ , so that  $j \leq j'$ ,  $a_j = \text{recv}(r, p, q, d)$ ,  $a_{j'} = \text{recv}(r, p, q, d')$ ,  $cause_{r_c}(j) = k$  and  $cause_{r_c}(j') = k'$ .*

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

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

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

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

$$S_0 \setminus cause_{r_c}(R_0) = S_0 \setminus S_0^k$$

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

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

**Lemma 5.5 (Cause and Effect)** *Let  $\Pi = (\pi_p \mid p \in \mathcal{P})$  be a collection of policies, and let  $r_c = s_0 a_1 s_1 a_2 \cdots$  be a finite or infinite fair run of  $[\Pi]^{op}$ . The following properties hold of  $r_c$ :*

1. Assume that for some  $k \geq 0$ , we have  $s_k.pq.wake = \mathbf{true}$ , then there exists a  $k' > k$  so that  $a_{k'} = \mathbf{eval}(p, q)$ .
2. Assume that  $a_{k_0} = \mathbf{eval}(p, q)$  and that  $s_{k_0-1}.pq.t_{cur} \neq s_{k_0}.pq.t_{cur} = d$ . Let  $k_1 > k$  be least with  $a_{k_1} = \mathbf{eval}(p, q)$  (note, such an index must exist by the above). Then, for every  $r \in \mathcal{P}$  there exists a unique  $k_r$  with  $k_0 < k_r < k_1$  so that  $a_{k_r} = \mathbf{send}(p, r, q, -)$ . Furthermore,  $a_{k_r} = \mathbf{send}(p, r, q, d)$  and  $\mathit{cause}_{r_c}(k_r) = k_0$ .
3. Assume that  $a_k = \mathbf{send}(p, r, q, d)$  and also that  $a_{k'} = \mathbf{send}(p, r, q, d')$ . Then,  $k < k'$  implies  $\mathit{cause}_{r_c}(k) < \mathit{cause}_{r_c}(k')$ .
4. Assume that  $a_k = \mathbf{recv}(r, p, q, d)$  and also that  $a_{k'} = \mathbf{recv}(r, p, q, d')$ . Then,  $k < k'$  implies  $\mathit{cause}_{r_c}(k) < \mathit{cause}_{r_c}(k')$ .

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

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

$$s_{k_0-1}.pq.t_{cur} \neq s_{k_0}.pq.t_{cur} = d.$$

Notice that  $s_{k_0}.pq.wake = \mathbf{true}$ , and let  $k_1 > k_0$  be the (index of the) first occurrence of an  $\mathbf{eval}(p, q)$  event after time  $k_0$ . Notice that since no  $\mathbf{eval}(p, q)$  event occurs in the interval  $(k_0, k_1)$  we have  $s_l.pq.t_{cur} = d$

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

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

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

is contained in

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

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

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

Then  $cause_{r_c}(k) < cause_{r_c}(k')$  follows because if we have  $cause_{r_c}(k') \leq cause_{r_c}(k)$ . By FIFO Lemma,

$$k' = effect_{r_c}(cause_{r_c}(k')) \leq effect_{r_c}(cause_{r_c}(k)) = k$$

□

**Lemma 5.6** *For any  $p, q, r, s \in \mathcal{P}$ , function  $\tau_{rs}^{pq}$  is monotonically increasing.*

*Proof.* We must show for all  $t, u$  if  $t \leq u$  then  $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$ . If no  $\mathbf{recv}(p, r, s, d)$  exists before  $u$ , or no  $\mathbf{recv}(p, r, s, d)$  exists before  $t$  but  $\mathbf{recv}(p, r, s, d)$  exists before  $u$ , then it is simple to verify that  $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$ . Let  $t \leq u$ , and let  $k, l$  denote the “ $k$ ’s”, corresponding to  $t$  and  $u$  respectively, in the definition of  $\tau_{rs}^{pq}$ , i.e.,  $a_k = \mathbf{recv}(p, r, s, d)$ ,  $k \leq t$ ,  $a_l = \mathbf{recv}(p, r, s, d')$ ,  $l \leq u$ . Because  $t \leq u$ , clearly  $k \leq l$ . By Lemma 5.4 (4),  $k' = \mathbf{cause}_{r_c}(k) \leq \mathbf{cause}_{r_c}(l) = l'$ . Similarly, by Lemma 5.4 (3),  $k'' = \mathbf{cause}_{r_c}(k') \leq \mathbf{cause}_{r_c}(l') = l''$ . If  $d = d'$  then  $k'' \leq l''$  and  $t \leq u$  implies that the largest index  $j \leq t$  with the property that for all  $j'$  with  $k'' \leq j' \leq j$ , is less than the similar largest index  $j \leq u$  with the property that for all  $j'$  with  $l'' \leq j' \leq j$ , hence  $\tau_{rs}^{pq}(t) \leq \tau_{rs}^{pq}(u)$ . If  $d \neq d'$  then for all  $j'$  with  $k'' \leq j' \leq \tau_{rs}^{pq}(t)$ ,  $s_k.rs.t_{cur} = d$ , means that  $k'' \leq l''$  implies  $\tau_{rs}^{pq}(t) < l'' \leq \tau_{rs}^{pq}(u)$ . □

**Lemma 5.7 (Corresponding runs)** *Let  $\Pi = (\pi_p \mid p \in \mathcal{P})$  be a collection of policies. Let  $r_c$  be a run of  $[\Pi]^{op}$ , and let  $r_a$  be the corresponding run. Then,*

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

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

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

- Case  $a_{k+1} = \mathbf{eval}(p, q)$  for some  $p, q \in \mathcal{P}$ . Since by the induction hypothesis  $\mathit{state}_{con}(r_c, k) = \mathit{state}_{abs}(r_a, k)$ , we get  $x^{pq}(k) = s_k.pq.gts$ . Hence, since  $k \in T^{pq}$ , we get  $x_{pq}(k+1) = f_{pq}(x^{pq}(k)) = f_{pq}(s_k.pq.gts) = s_{k+1}.pq.t_{cur}$ . Further, we have  $x^{pq}(k+1)_{pq} = x_{pq}(k+1) = s_{k+1}.pq.t_{cur} = s_{k+1}.pq.gts(p)(q)$ . For all  $rs \neq pq$ ,  $x^{pq}(k+1)_{rs} = x^{pq}(k)_{rs} = s_k.pq.gts(r)(s) = s_{k+1}.pq.gts(r)(s)$ .
- Case  $a_{k+1} = \mathbf{send}(p, q, s, v)$ . Notice that send-actions don't affect the abstract state:  $\mathit{state}_{con}(r, k+1) = \mathit{state}_{con}(r, k) = \mathit{state}_{abs}(r, k) = \mathit{state}_{abs}(r, k+1)$ .
- Case  $a_{k+1} = \mathbf{recv}(p, r, s, v)$ . Note first that for all  $u, v \in \mathcal{P}$ , we have  $x_{uv}(k+1) = x_{uv}(k) = s_k.uv.t_{cur} = s_{k+1}.uv.t_{cur}$ . For the estimate-part, let  $q \in \mathcal{P}$  be arbitrary, and notice first that for all  $u, v, w \in \mathcal{P}$  if

$u \neq p$  or  $(v, w) \neq (r, s)$ ,

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

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

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

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

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

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

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

□

**Lemma 5.8** *For any fair run  $r_c$  of  $[[\Pi]]^{op}$ , let  $r_a$  denote its corresponding run. Then,*

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

*Proof.* We prove each point separately. In the following ‘‘The Lemma’’ refers to Lemma 5.4.

- Let  $r_c = s_0 a_1 s_1 \dots$  be an infinite fair run of  $[[\Pi]]^{op}$ . We let  $p, q \in \mathcal{P}$  be arbitrary, and prove that there are infinitely many  $\mathbf{eval}(p, q)$  events in  $r_c$ , which implies that  $T^{pq}$  is infinite. Note that it suffices to prove that there are infinitely many **send**-events in  $r_c$ , by the following. Assume there are infinitely many **send**-events in  $r_c$ , and note that since  $\mathcal{P}$  is

finite, there must exist  $r, s, t \in \mathcal{P}$  so that  $\mathbf{send}(r, t, s, -)$  events occur infinitely often (i.o.) in  $r_c$ . By The Lemma (2,3),  $\mathit{cause}_{r_c}$  maps the indexes of these  $\mathbf{send}(r, t, s, -)$  events, injectively, to indexes  $k$  with  $a_k = \mathbf{eval}(r, s)$  and  $s_{k-1}.rs.t_{cur} \neq s_k.rs.t_{cur}$ , hence, such indexes occur i.o. in  $r_c$ . By The Lemma (2), also,  $\mathbf{send}(r, p, s, -)$  events occur i.o. in  $r_c$ , hence, by the FIFO Lemma,  $\mathbf{recv}(p, \dots)$  events occur i.o. in  $r_c$ . But this implies that  $pq.wake = \mathbf{true}$  infinitely often, and hence by The Lemma (1), we have  $\mathbf{eval}(p, q)$  infinitely often.

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

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

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

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

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

In the final case, there are infinitely many  $k$  with  $a_k = \mathbf{eval}(r, s)$  and  $s_k.rs.t_{cur} \neq s_{k+1}.rs.t_{cur}$ . Hence by The Lemma (2), there are infinitely many  $\mathbf{send}(r, p, s, -)$  events. By The Lemma (3), the injectivity of  $\mathit{cause}_{r_c}$  on these events implies that for some  $v \in D$ , there exist an

event,  $\mathbf{send}(r, p, s, v)$ , with index  $k' > K$  so that  $k = \mathit{cause}_{r_c}(k')$  and  $k \geq K$ . Hence by the FIFO Lemma, there exists a  $\mathbf{recv}(p, r, s, v)$  event with index  $k'' > k'$  so that  $\mathit{cause}(k'') = k'$ . Now let  $t^{j_0} > k''$ , then by monotonicity of  $\tau_{rs}^{pq}$ , we obtain  $\tau_{rs}^{pq}(t^{j_0}) \geq \tau_{rs}^{pq}(k'') \geq k > K$ .

- Now assume that  $r_c$  is finite fair. Clearly  $r_a$  is finite. We must show it is also fair. So let  $p, q, r, s \in \mathcal{P}$  and consider  $x_{rs}(\tau_{rs}^{pq}(t_{pq}^*))$ . If  $(p, q) = (r, s)$  then  $\tau_{rs}^{pq}$  is the identity, and hence, by definition of  $t_{pq}^*$ ,  $x_{pq}(t^*) = x_{pq}(t_{pq}^*) = x_{pq}(\tau_{rs}^{pq}(t_{pq}^*))$ . So assume that  $(p, q) \neq (r, s)$ . Assume, for the sake of contradiction, that  $x_{rs}(\tau_{rs}^{pq}(t_{pq}^*)) \neq x_{rs}(t^*)$ . By Lemma 5.7, this implies that there must exist an index  $k$  with  $a_k = \mathbf{eval}(r, s)$  and  $s_{k-1}.rs.t_{cur} \neq s_k.rs.t_{cur}$ . Let  $K$  be the greatest index with this property (such a greatest index must exist since  $r_c$  is finite), and note that for all  $j$  with  $K \leq j \leq t^*$ ,  $s_j.rs.t_{cur} = x_{rs}(t^*) \stackrel{\text{(def)}}{=} d$ . By Lemma 5.4 (2) and the FIFO Lemma, there exists  $j_p > k_p > K$  so that  $a_{j_p} = \mathbf{recv}(p, r, s, d)$ ,  $a_{k_p} = \mathbf{send}(r, p, s, d)$ ,  $\mathit{cause}_{r_c}(k_p) = K$  and  $\mathit{cause}_{r_c}(j_p) = k_p$ .

Note, there cannot be a later index  $j' > k_p$  with  $a_{j'} = \mathbf{recv}(p, r, s, -)$ , because then

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

which contradicts maximality of  $K$ . Hence, if  $k_p \leq t_{pq}^*$  then  $s_{t_{pq}^*}.pq.t_{cur} = d$ , and by Lemma 5.7,  $x_{pq}(\tau_{rs}^{pq}(t_{pq}^*)) = d = x_{rs}(t^*)$ : a contradiction. But, if  $k_p > t_{pq}^*$  then we must have  $s_{k_p}.pq.wake = \mathbf{true}$  and by Lemma 5.4 there must be a later  $\mathbf{eval}(p, q)$  event, contradicting maximality of  $t_{pq}^*$ .

□

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

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

*Proof.* By induction in  $t$ .

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

**Inductive step.**

1. Show that  $x^{pq}(t+1) \sqsubseteq \llbracket \Pi \rrbracket^{\text{den}} = \text{lfp } \Pi_{\lambda}$ . Let  $r, s \in \mathcal{P}$  be arbitrary but fixed. By definition,  $x^{pq}(t+1)_{rs} = x_{rs}(\tau_{rs}^{pq}(t+1))$ . Now there are two cases.
  - (a)  $\exists t' \leq t. x_{rs}(\tau_{rs}^{pq}(t+1)) = f_{rs}(x^{rs}(t'))$ . Since  $t' \leq t$  the induction hypothesis (i.h.) implies that  $x^{rs}(t') \sqsubseteq \text{lfp } \Pi_{\lambda}$ , hence  $f_{rs}(x^{rs}(t')) \sqsubseteq f_{rs}(\text{lfp } \Pi_{\lambda}) = (\text{lfp } \Pi_{\lambda})_{rs}$ .
  - (b) No such  $t'$  exists. Then  $x_{rs}(\tau_{rs}^{pq}(t+1)) = \perp_{\square}$ .
2. Show that  $x_{pq}(t+1) \sqsubseteq f_{pq}(x^{pq}(t+1))$ . Again there are two cases. In both cases we will assume that  $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$ , which we prove later (note that this is essentially the ‘monotonic’-property).
  - (a) If  $t \notin T^{pq}$  then  $x_{pq}(t+1) = x_{pq}(t)$ . Now the i.h. implies that  $x_{pq}(t) \sqsubseteq f_{pq}(x^{pq}(t))$ . Now, monotonicity of  $f_{pq}$  together with  $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$  implies  $f_{pq}(x^{pq}(t)) \sqsubseteq f_{pq}(x^{pq}(t+1))$ .
  - (b) If  $t \in T^{pq}$  then  $x_{pq}(t+1) = f_{pq}(x^{pq}(t))$ . Now since we assume  $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$ , monotonicity of  $f_{pq}$  implies  $f_{pq}(x^{pq}(t)) \sqsubseteq f_{pq}(x^{pq}(t+1))$ .
3. Show that  $\forall t' \leq t+1. x^{pq}(t') \sqsubseteq x^{pq}(t+1)$ . Note that by the i.h., it suffices proving  $x^{pq}(t) \sqsubseteq x^{pq}(t+1)$ . So let  $r, s \in \mathcal{P}$  be arbitrary but fixed. Show that  $x^{pq}(t)_{rs} \sqsubseteq x^{pq}(t+1)_{rs}$ . We have

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

and

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

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

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

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

□

# Chapter 6

## A Logical Framework for Reputation Systems

The material presented in this chapter is based on number of papers, published in the form of a technical report, a conference paper and a journal paper.

- [56] K. Krukow, M. Nielsen, and V. Sassone. A formal framework for concrete reputation-systems with applications to history-based access control. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 260–269, New York, NY, USA, 2005. ACM Press.
- [57] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems. Technical Report RS-05-23, BRICS, University of Aarhus, July 2005.
- [58] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for reputation systems. Submitted. Available online [www.brics.dk/~krukow](http://www.brics.dk/~krukow), 2006.

The technical report [57] is an extended version of the conference paper [56], containing proofs, and an additional section about possible language extensions. The journal version [58] further extends the technical report with a section describing an application of the framework to history-based access control. Except for minor typographical changes the content of this chapter is equal to the journal version [58]. There is no concluding section, as this material is covered in Chapter 3.

An implementation of the Java prototype framework for history-based access control (JavaHBAC) is available online at Sourceforge (<https://sourceforge.net/projects/javahbac>).



## A Logical Framework for Reputation Systems

### 6.1 Introduction

Rich opportunities for fraud exist on the Internet. Still, risky interactions like electronic commerce, involving disclosure of private informations to semi-trusted parties, are every-day activities in our Internet lives. It seems that in practice, for most people, the utility of the Internet outweighs its risks. When one tries to understand better these facts, mathematical models from economic theory are very appealing. Online interaction can often be seen as a ‘repeated game’ played between selfish (semi) rational principals. Such interaction may result in utility gains for the involved principals, but often, with interaction comes also an associated inherent risk; a potential utility-loss. For risk-adverse principals, the fear of loss may outweigh the expectation of gain, leading to an unwillingness to participate. For example, one might have expected that an online auctioning system such as eBay, “a market ripe with the possibility of large-scale fraud and deceit” [51], would never have reached the more than one million transactions per day that are presently processed. The liveness on eBay is often attributed to its so-called Feedback Forum, a simple example of a reputation system. When principals have transacted, each party may leave feedback on the eBay web-site, consisting of a rating of ‘positive’, ‘neutral’ or ‘negative’. A principal’s aggregated rating is then visible to potential buyers or sellers before deciding whether to interact or not. In general, reputation systems record, aggregate and (sometimes) distribute information about the past behaviour of principals. Hence reputation systems may serve as a trust-enabling, or perhaps, more generally, trust-*informing* technology. Resnick et al. argue that reputation systems foster an incentive for principals to well-behave because of “the expectation of reciprocity or retaliation in future interactions” [90], and reputation itself has previously been formalized and analysed by economists in simple game-theoretic models, leading to similar conclusions (e.g., [52, 102, 27, 28]); it seems that reputation systems are well established, and their usefulness generally accepted.

Many reputation systems have been proposed in the literature [47], and often the recorded behavioural information is heavily abstracted. This has the effect that several quite different concrete behaviours are collapsed in to the same “equivalence class” of recorded behaviours. For example, the eBay-rating of ‘negative’ may be the (subjective) result of several distinct seller behaviours: the seller may never ship the auctioned item, the item may be in a poor condition, a certain timeliness is expected, credit cards may be over-charged because of, say, shipping fees, etc. Different users will be interested in the actual meaning of the rating ‘negative’; the *concrete behaviour* of the seller. There are other examples: In the EigenTrust system [49], behavioural information is obtained by counting the number of ‘satisfactory’ and ‘unsatisfactory’ interactions with a principal. Besides lacking a precise semantics, this information has abstracted away any notion of time, and is further reduced (by normalisation) to a number in the interval  $[0, 1]$ . In the Beta reputation system [46], similar abstractions are performed, obtaining a numerical value in  $[-1, 1]$  (with a statistical interpretation). The only non-example of such crude information abstraction (that we are aware of) is the framework of Shmatikov and Talcott [94] which we discuss further in the concluding section.

Abstract representations of behavioural information have their advantages (e.g., numerical values are often easily comparable, and require little space to store), but clearly, information is lost in the abstraction process. For example, in EigenTrust, value ‘0’ may represent both “no previous interaction” and “many unsatisfactory previous interactions” [49]. Consequently, one cannot verify exact properties of past behaviour given only the reputation information.

In this paper, the concept of ‘reputation system’ is to be understood very broadly, simply meaning *any system in which principals record and use information about past behaviour of principals, when assessing the risk of future interaction*. A *principal* is simply an identity; it may be the identity of a human users, a public key, a software program (e.g., an identifiable instance), etc. We present a formal framework for a class of simple reputation systems in which, as opposed to most “traditional” systems, behavioural information is represented in a very concrete form. The advantage of our concrete representation is that sufficient information is present to check precise properties of past behaviour. In our framework, such requirements on past behaviour are specified in a declarative policy-language, and the basis for making decisions regarding future interaction becomes the verification of a behavioural history with respect to a policy. This enables us to define reputation systems that provide a form of provable “security” guarantees,

intuitively, of the form: “If principal  $p$  gains access to resource  $r$  at time  $t$ , then the past behaviour of  $p$  up *until* time  $t$  satisfies requirement  $\psi_r$ .”

To get the flavour of such requirements, we preview an example policy from a declarative language formalized in the following sections. Edjlali *et al.* [33] consider a notion of history-based access control in which unknown programs, in the form of mobile code, are dynamically classified into equivalence classes of programs according to their behaviour (e.g. “browser-like” or “shell-like”). This dynamic classification falls within the scope of our very broad understanding of reputation systems. The following is an example of a policy written in our language, which specifies a property similar to that of Edjlali *et al.*, used to classify “browser-like” applications:

$$\begin{aligned} \psi \equiv & \neg F^{-1}(\text{modify-file}) \wedge \\ & \neg F^{-1}(\text{create-subprocess}) \wedge \\ & G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))]) \end{aligned}$$

Informally, the atoms `modify-file`, `create-subprocess`, `open(x)` and `create(x)` are *events* which are observable by monitoring an entity’s behaviour. The latter two are *parameterised* events, and the quantification ‘ $\forall x$ ’ ranges over the possible parameters of these. Operator  $F^{-1}$  means “at some point in the past,”  $G^{-1}$  means “always in the past,” and constructs  $\wedge$  and  $\neg$  are conjunction and negation, respectively. Thus, clauses  $\neg F^{-1}(\text{modify-file})$  and  $\neg F^{-1}(\text{create-subprocess})$  require that the application has never modified a file, and has never created a sub-process. The final, quantified clause  $G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))])$  requires that whenever the application opens a file, it must previously have created that file. For example, if the application has opened the local system-file `"/etc/passwd"` (i.e. a file which it has not created) then it cannot access the network (a right assigned to the “browser-like” class). If, instead, the application has previously only read files it has created, then it will be allowed network access.

### 6.1.1 Contributions and Outline

We present a formal model of the behavioural information that principals obtain in our class of reputation systems. This model is based on previous work using event structures [103] for modelling observations [82], but our treatment of behavioural information departs from the previous work in that we perform (almost) no information abstraction. The event-structure model is presented in Section 6.2.

We describe our formal declarative language for interaction policies. In the framework of event structures, behavioural information is modelled as

sequences of sets of events. Such linear structures can be thought of as (finite) models of linear temporal logic (LTL) [86]. Indeed, our basic policy language is based on a (pure-past) variant of LTL. We give the formal syntax and semantics of our language, and provide several examples illustrating its naturality and expressiveness. We are able to encode several existing approaches to history-based access control, e.g. the Chinese Wall security policy [14] and a restricted version of so-called ‘one-out-of- $k$ ’ access control [33]. The formal description of our language, as well as examples and encodings, is presented in Section 6.3.

An interesting new problem is how to re-evaluate policies efficiently when interaction histories change as new information becomes available. It turns out that this problem, which can be described as dynamic model-checking, can be solved very efficiently using an algorithm adapted from that of Havelund and Roşu, based on the technique of dynamic programming, used for runtime verification [42]. Interestingly, although one is verifying properties of an *entire* interaction history, one needs not store this complete history in order to verify a policy: old interaction can be efficiently summarised relative to the policy. In Section 6.4, two dynamic algorithms for policy checking is described, analysed and compared.

Our simple policy language can be extended to encompass policies that are more realistic and practical (e.g., for history-based access control [33, 37, 5, 96], and within the traditional domain of reputation systems: peer-to-peer- and online feedback systems [49, 90]). More specifically, we present two extensions. The first is quantification (as is used in the example policy in the introductory section). We extend the basic language, allowing parameterised events and quantification over the parameters. An algorithm for checking the extended language along with complexity analyses is provided. The second extension covers the two aspects of *information sharing*, and *quantitative properties*. We introduce constructs that allow principals to state properties, not only of their personally-observed behaviour, but also of the behaviour observed by others (in the terminology of Mui *et al.* [78], the first is *direct* and *encounter driven*, and the latter, *indirect* and *propagated*). Such information sharing is characteristic of most existing reputation systems. Another common characteristic is focus on conveying quantitative information. In contrast, standard temporal logic is qualitative: it deals with concepts such as *before*, *after*, *always* and *eventually*. We show that we can extend our language to include a range of quantitative aspects, intuitively, operators like ‘almost always,’ ‘more than  $N$ ,’ etc. Section 6.5 illustrates these two extensions, and briefly discusses policy-checking for the extended languages.

Throughout the paper, we have small examples illustrating the applicability of our framework within the area of history-based access control. We have taken this one step further by developing a prototype security manager for Java, based on our logical framework. The security manager is parameterised by a policy in our language, and monitors a Java program with respect to this policy, throwing an exception if a violation is about to happen. In Section 6.6, we describe this application of our framework to history-based access control for Java programs. Related work is discussed in the concluding section.

## 6.2 Observations as Events

Agents in a distributed system obtain information by observing events which are typically generated by the reception or sending of messages. The structure of these message exchanges are given in the form of protocols known to both parties before interaction begins. By *behavioural observations*, we mean observations that the parties can make about specific runs of such protocols. These include information about the contents of messages, diversion from protocols, failure to receive a message within a certain time-frame, etc.

Our goal in this section, is to give precise meaning to the notion of behavioural observations. Note that, in the setting of large-scale distributed environments, often, a particular agent will (concurrently) be involved in several instances of protocols; each instance generating events that are logically connected. One way to model the observation of events is using a process algebra with “state”, recording input/output reactions, as is done in the calculus for trust management, *ctm* [20]. Here we are not interested in modelling interaction protocols in such detail, but merely assume some system responsible for generating events.

We will use the event-structure framework of Nielsen and Krukow [82] as our model of behavioural information. The framework is suitable for our purpose as it provides a *generic* model for observations that is independent of any specific programming language. In the framework, the information that an agent has about the behaviour of another agent  $p$ , is information about a number of (possibly active) protocol-runs with  $p$ , represented as a sequence of *sets of events*,  $x_1x_2 \cdots x_n$ , where event-set  $x_i$  represents information about the  $i$ th initiated protocol-instance. Note, in frameworks for history-based access control (e.g., [33, 5, 37]), histories are always sequences of *single* events. Our approach generalises this to allow sequences of (finite) *sets* of events; a generalisation useful for modelling information about protocol runs

in distributed systems.

We present the event-structure framework as an abstract interface providing two operations, **new** and **update**, which respectively records the initiation of a new protocol run, and updates the information recorded about an older run (i.e. updates an event-set  $x_i$ ). A specific implementation then uses this interface to notify our framework about events.

### 6.2.1 The Event Structure Framework

In order to illustrate the event-structure framework, we use an example complementing its formal definitions. We will use a scenario inspired by the eBay online auction-house [32], but deliberately over-simplified to illustrate the framework.

On the eBay website, a seller starts an auction by announcing, via the website, the item to be auctioned. Once the auction has started the highest bid is always visible, and bidders can place bids. A typical auction runs for 7 days, after which the bidder with the highest bid wins the auction. Once the auction has ended, the typical protocol is the following. The buyer (winning bidder) sends payment of the amount of the winning bid. When payment has been received, the seller confirms the reception of payment, and ships the auctioned item. Optionally, both buyer and seller may leave feedback on the eBay site, expressing their opinion about the transaction. Feedback consist of a choice between ratings ‘positive’, ‘neutral’ and ‘negative’, and, optionally, a comment.

We will model behavioural information in the eBay scenario from the buyers point of view. We focus on the interaction following a winning bid, i.e. the protocol described above. After winning the auction, buyer ( $B$ ) has the option to send payment, or ignore the auction (possibly risking to upset the seller). If  $B$  chooses to send payment, he may observe confirmation of payment, and later the reception of the auctioned item. However, it may also be the case that  $B$  doesn’t observe the confirmation within a certain time-frame (the likely scenario being that the seller is a fraud). At any time during this process, each party may choose to leave feedback about the other, expressing their degree of satisfaction with the transaction. In the following, we will model an abstraction of this scenario where we focus on the following events: buyer pays for auction, buyer ignores auction, buyer receives confirmation, buyer receives no confirmation within a fixed time-limit, and *seller* leaves positive, neutral or negative feedback (note that we do not model the *buyer* leaving feedback).

The basis of the event-structure framework is the fact that the obser-

vations about protocol runs, such as an eBay transaction, have structure. Observations may be in *conflict* in the sense that one observation may exclude the occurrence of others, e.g. if the seller leaves positive feedback about the transaction, he can not leave negative or neutral feedback. An observation may *depend* on another in the sense that the first may only occur if the second has already occurred, e.g. the buyer cannot receive a confirmation of received payment if he has not made a payment. Finally, if two observations are neither in conflict nor dependent, they are said to be *independent*, and both may occur (in any order), e.g. feedback-events and receiving confirmation are independent. Note that ‘independent’ just means that the events are not in conflict nor dependent (e.g., it does *not* mean that the events are independent in any statistical sense). These relations between observations are directly reflected in the definition of an event structure. (For a general account of event structures [103], traditionally used in semantics of concurrent languages, consult the handbook chapter of Winskel and Nielsen [105]).

**Definition 6.1 (Event Structure)** An *event structure* is a triple  $ES = (E, \leq, \#)$  consisting of a set  $E$ , and two binary relations on  $E$ :  $\leq$  and  $\#$ . The elements  $e \in E$  are called *events*, and the relation  $\#$ , called the *conflict relation*, is symmetric and irreflexive. The relation  $\leq$  is called the (*causal*) *dependency relation*, and partially orders  $E$ . The dependency relation satisfies the following axiom, for any  $e \in E$ :

$$\text{the set } [e] \stackrel{\text{(def)}}{=} \{e' \in E \mid e' \leq e\} \text{ is finite.}$$

The conflict- and dependency-relations satisfy the following “transitivity” axiom for any  $e, e', e'' \in E$

$$(e \# e' \text{ and } e' \leq e'') \text{ implies } e \# e''$$

Two events are *independent* if they are not in either of the two relations.

We use event structures to model the possible observations of a single agent in a protocol, e.g. the event structure in Figure 6.1 models the events observable by the buyer in our eBay scenario.

The two relations on event structures imply that not all subsets of events can be observed in a protocol run. The following definition formalises exactly what sets of observations are observable.

**Definition 6.2 (Configuration)** Let  $ES = (E, \leq, \#)$  be an event structure. We say that a subset of events  $x \subseteq E$  is a *configuration* if it is *conflict*

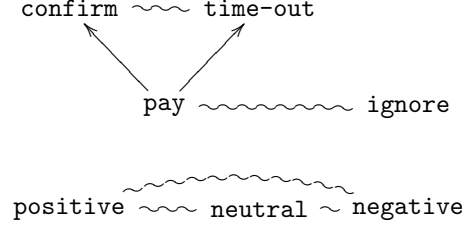


Figure 6.1: An event structure modelling the buyer's observations in the eBay scenario. (Immediate) Conflict is represented by  $\sim$ , and dependency by  $\rightarrow$ .

*free* (C.F.), and *causally closed* (C.C.). That is, it satisfies the following two properties, for any  $d, d' \in x$  and  $e \in E$

$$(C.F.) d \# d'; \text{ and } (C.C.) e \leq d \Rightarrow e \in x$$

**Notation 6.1**  $\mathcal{C}_{ES}$  denotes the set of configurations of  $ES$ , and  $\mathcal{C}_{ES}^0 \subseteq \mathcal{C}_{ES}$  the set of *finite* configurations. A configuration is said to be *maximal* if it is maximal in the partial order  $(\mathcal{C}_{ES}, \subseteq)$ . Also, if  $e \in E$  and  $x \in \mathcal{C}_{ES}$ , we write  $e \# x$ , meaning that  $\exists e' \in x. e \# e'$ . Finally, for  $x, x' \in \mathcal{C}_{ES}, e \in E$ , define a relation  $\rightarrow$  by  $x \xrightarrow{e} x'$  iff  $e \notin x$  and  $x' = x \cup \{e\}$ . If  $y \subseteq E$  and  $x \in \mathcal{C}_{ES}, e \in E$  we write  $x \not\xrightarrow{e} y$  to mean that either  $y \notin \mathcal{C}_{ES}$  or it is not the case that  $x \xrightarrow{e} y$ .

A finite configuration models information regarding a *single* interaction, i.e. a single run of a protocol. A maximal configuration represents complete information about a single interaction. In our eBay example, sets  $\emptyset$ ,  $\{\text{pay}, \text{positive}\}$  and  $\{\text{pay}, \text{confirm}, \text{positive}\}$  are examples of configurations (the last configuration being maximal), whereas

$$\{\text{pay}, \text{confirm}, \text{positive}, \text{negative}\}$$

and  $\{\text{confirm}\}$  are non-examples.

In general, the information that one agent possesses about another will consist of information about *several* protocol runs; the information about each individual run being represented by a configuration in the corresponding event structure. The concept of a local interaction history models this.

**Definition 6.3 (Local Interaction History)** Let  $ES$  be an event structure, and define a *local interaction history* in  $ES$  to be a sequence of finite

configurations,  $h = x_1x_2 \cdots x_n \in \mathcal{C}_{ES}^0$ . The individual components  $x_i$  in the history  $h$  will be called *sessions*.

In our eBay example, a local interaction history could be the following:

$$\{\text{pay}, \text{confirm}, \text{pos}\} \{\text{pay}, \text{confirm}, \text{neu}\} \{\text{pay}\}$$

Here **pos** and **neu** are abbreviations for the events **positive** and **neutral**. The example history represents that the buyer has won three auctions with the particular seller, e.g. in the third session the buyer has (so-far) observed only event **pay**.

We assume that the actual system responsible for notification of events will use the following interface to the model.

**Definition 6.4 (Interface)** Define an operation **new** :  $\mathcal{C}_{ES}^0 \rightarrow \mathcal{C}_{ES}^0$  by **new**( $h$ ) =  $h\emptyset$ . Define also a partial operation **update** :  $\mathcal{C}_{ES}^0 \times E \times \mathbb{N} \rightarrow \mathcal{C}_{ES}^0$  as follows. For any  $h = x_1x_2 \cdots x_i \cdots x_n \in \mathcal{C}_{ES}^0$ ,  $e \in E$ ,  $i \in \mathbb{N}$ , **update**( $h, e, i$ ) is undefined if  $i \notin \{1, 2, \dots, n\}$  or  $x_i \not\stackrel{e}{\rightarrow} x_i \cup \{e\}$ . Otherwise

$$\mathbf{update}(h, e, i) = x_1x_2 \cdots (x_i \cup \{e\}) \cdots x_n$$

**Remarks.** The notion of *time* in the model is based on when sessions are *started*. More precisely, in our local interaction histories,  $h = x_1x_2 \cdots x_n$  where  $x_i \in \mathcal{C}_{ES}$ , the order of the sessions reflects *the order in which the corresponding interaction-protocols are initiated*, i.e.  $x_i$  refers to the observed events in the  $i$ th-initiated session. Different notions of time could just as well be considered, e.g. if  $x_i$  precedes  $x_j$  in sequence  $h$ , then it means that  $x_j$  was updated more recently than  $x_i$ .

Note, while the order of sessions is recorded (a local history is a *sequence*), in contrast, the order of *independent* events within *a single session* is not. For example, in our eBay scenario we have

$$\begin{aligned} & \mathbf{update}(\mathbf{update}(\{\text{pay}\}, \text{neutral}, 1), \text{confirm}, 1) = \\ & \mathbf{update}(\mathbf{update}(\{\text{pay}\}, \text{confirm}, 1), \text{neutral}, 1) \end{aligned}$$

Hence independence of events is a choice of abstraction one may make when designing an event-structure model (because one is not interested in the particular order of events, or because the exact recording of the order of events is not feasible). However, note that this is not a limitation of event structures: in a scenario where this order of events is relevant (and observable), one can always use a “serialised” event structure in which this order of occurrences is

recorded. A serialisation of events consists of splitting the events in question into different events depending on the order of occurrence, e.g., supposing in the example one wants to record the order of `pay` and `pos`, one replaces these events with events `pay-before-pos`, `pos-before-pay`, `pay-after-pos` and `pos-after-pay` with the obvious causal- and conflict-relations.

When applying our logic (described in the next section) to express policies for history-based access control (HBAC), we use a special type of event structure in which the conflict relation is the maximal irreflexive relation on a set  $E$  of events. The reason is that *histories* in many frameworks for HBAC, are sequences of single events for a set  $E$ . When the conflict relation is maximal on  $E$ , the configurations of the corresponding event structure are exactly singleton event-sets, hence we obtain a useful specialisation of our model, compatible with the tradition of HBAC.

### 6.3 A Language for Policies

The reason for recording behavioural information is that it can be used to guide future decisions about interaction. We are interested in binary decisions, e.g., access-control and deciding whether to interact or not. In our proposed system, such decisions will be made according to interaction policies that specify exact requirements on local interaction histories. For example, in the eBay scenario from last section, the bidder may adopt a policy stating: “only bid on auctions run by a seller which has never failed to send goods for won auctions in the past.” Notice, by the way, that users would have a hard time implementing such a policy using the current eBay feedback forum.

In this section, we propose a declarative language which is suitable for specifying interaction policies. In fact, we shall use a pure-past variant of linear-time temporal logic, a logic introduced by Pnueli for reasoning about parallel programs [86]. Pure-past temporal logic turns out to be a natural and expressive language for stating properties of past behaviour. Furthermore, linear-temporal-logic models are linear Kripke-structures, which resemble our local interaction histories. We define a satisfaction relation  $\models$ , between such histories and policies, where judgement  $h \models \psi$  means that the history  $h$  satisfies the requirements of policy  $\psi$ .

### 6.3.1 Formal Description

#### Syntax.

The syntax of the logic is parametric in an event structure  $ES = (E, \leq, \#)$ . There are constant symbols for each  $e \in E$  (ranged over by meta-variables  $e, e', e_i, \dots$ ). The syntax of our language, which we denote  $\mathcal{L}(ES)$ , is given by the following BNF.

$$\psi ::= e \mid \diamond e \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid \mathbf{X}^{-1}\psi \mid \psi_0 \mathbf{S} \psi_1$$

The constructs  $e$  and  $\diamond e$  are both *atomic* propositions. In particular,  $\diamond e$  is *not* the application of the usual modal operator  $\diamond$  (with the “temporal” semantics) to formula  $e$ . Informally, the formula  $e$  is true in a session if the event  $e$  has been observed in that session, whereas  $\diamond e$ , pronounced “ $e$  is possible”, is true if event  $e$  *may still occur* as a future observation in that session. The operators  $\mathbf{X}^{-1}$  (‘last time’) and  $\mathbf{S}$  (‘since’) are the usual past-time operators.

#### Semantics.

A *structure* for  $\mathcal{L}(ES)$ , where  $ES = (E, \leq, \#)$  is an event structure, is a local interaction history in  $ES$ ,  $h \in \mathcal{C}_{ES}^0$ . We define the satisfaction relation  $\models$  between structures and policies, i.e.  $h \models \psi$  means that the history  $h$  satisfies the requirements of policy  $\psi$ . We will use a variation of the semantics in linear Kripke structures: satisfaction is defined from the *end* of the sequence “towards” the beginning, i.e.  $h \models \psi$  iff  $(h, |h|) \models \psi$ . To define the semantics of  $(h, i) \models \psi$ , let  $h = x_1x_2 \cdots x_N \in \mathcal{C}_{ES}^0$ , and  $i \in \mathbb{N}$ . Define  $(h, i) \models \psi$  by structural induction in  $\psi$ .

$$\begin{aligned} (h, i) \models e & \quad \text{iff } 1 \leq i \leq N \text{ and } e \in x_i \\ (h, i) \models \diamond e & \quad \text{iff } 1 \leq i \leq N \Rightarrow e \notin x_i \\ (h, i) \models \psi_0 \wedge \psi_1 & \quad \text{iff } (h, i) \models \psi_0 \text{ and } (h, i) \models \psi_1 \\ (h, i) \models \neg\psi & \quad \text{iff } (h, i) \not\models \psi \\ (h, i) \models \mathbf{X}^{-1}\psi & \quad \text{iff } i > 1 \text{ and } (h, i-1) \models \psi \\ (h, i) \models \psi_0 \mathbf{S} \psi_1 & \quad \text{iff } \exists j \leq i. [(h, j) \models \psi_1 \text{ and} \\ & \quad \forall k. (j < k \leq i \Rightarrow (h, k) \models \psi_0)] \end{aligned}$$

**Remarks.** There are two main reasons for restricting ourselves to the *pure-past* fragment of temporal logic (PPLTL). Most importantly, PPLTL is an expressive and *natural* language for stating requirements over *past* behaviour, e.g. history-based access control. Hence in our application one

wants to speak about the past, not the future. We justify this claim further by providing (natural) encodings of several existing approaches for checking requirements of past behaviour (c.f. Example 6.2 and 6.3 in the next section). Secondly, although one could add future operators to obtain a seemingly more expressive language, a result of Laroussinie *et al.* quantifies exactly what is lost by this restriction [63]. Their result states that LTL can be *exponentially more succinct* than the pure-future fragment of LTL. It follows from the duality between the pure-future and pure-past operators, that when restricting to finite linear Kripke structures, and interpreting  $h \models \psi$  as  $(h, |h|) \models \psi$ , then our pure-past fragment can express *any* LTL formula (up to initial equivalence), though possibly at the cost of an exponential increase in the size of the formula. Another advantage of PPLTL is that, while Sistla and Clarke proved that the model-checking problem for linear temporal logic with future- and past-operators (LTL) is PSPACE-complete [95], there are very efficient algorithms for (finite-path) model-checking pure-past fragments of LTL, and (as we shall see in Section 6.4) also for the dynamic policy-checking problem.

Note that the logic cannot distinguish the empty structure  $\epsilon \in \mathcal{C}_{ES}^*$  from a structure consisting of any number of empty configurations, e.g.,  $\emptyset\emptyset\emptyset$ . More generally, one way of looking at our structures is as *infinite* sequences  $x_1x_2 \cdots x_n\emptyset\emptyset \cdots$ , having only finitely many non-empty configurations.

We define standard abbreviations using syntactic equality:  $\text{false} \equiv e \wedge \neg e$  for some fixed  $e \in E$ ,  $\text{true} \equiv \neg \text{false}$ ,  $\psi_0 \vee \psi_1 \equiv \neg(\neg\psi_0 \wedge \neg\psi_1)$ ,  $\psi_0 \rightarrow \psi_1 \equiv \neg\psi_0 \vee \psi_1$ ,  $\mathbf{F}^{-1}(\psi) \equiv \text{true S } \psi$ ,  $\mathbf{G}^{-1}(\psi) \equiv \neg\mathbf{F}^{-1}(\neg\psi)$ . Note that,  $\mathbf{F}^{-1}(\psi)$  means “formula  $\psi$  is true at *some time* in the past,” whereas  $\mathbf{G}^{-1}(\psi)$  means “ $\psi$  is true at *all times* in the past.” We also define a non-standard abbreviation  $\sim e \equiv \neg\Diamond e$  (pronounced ‘conflict  $e$ ’ or ‘ $e$  is impossible’).

### 6.3.2 Example Policies

To illustrate the expressive power of our language, we consider a number of example policies.

**Example 6.1 (eBay)** Recall the eBay scenario from Section 6.2, in which a buyer has to decide whether to bid on an electronic auction issued by a seller. We express a policy for decision ‘bid’, stating “only bid on auctions run by a seller that has never failed to send goods for won auctions in the past.”

$$\psi^{\text{bid}} \equiv \neg\mathbf{F}^{-1}(\text{time-out})$$

Furthermore, the buyer might require that “the seller has never provided negative feedback in auctions where payment was made.” We can express this by

$$\psi^{\text{bid}} \equiv \neg F^{-1}(\text{time-out}) \wedge G^{-1}(\text{negative} \rightarrow \text{ignore})$$

**Example 6.2 (Chinese Wall)** The Chinese Wall policy is an important commercial security-policy [14], but has also found applications within computer science. In particular, Edjlali *et al.* [33] use an instance of the Chinese Wall policy to restrict program accesses to database relations. The Chinese Wall security-policy deals with subjects (e.g. users) and objects (e.g. resources). The objects are organized into *datasets* which, in turn, are organized in so-called *conflict-of-interest classes*. There is a hierarchical structure on objects, datasets and classes, so that each object has a unique dataset which, in turn, has a unique class. In the Chinese-Wall policy, any subject initially has freedom to access any object. After accessing an object, the set of future accessible objects is restricted: the subject can no longer access an object in the same conflict-of-interest class unless it is in a dataset already accessed. Non-conflicting classes may still be accessed.

We now show how our logic can encode any instance of the Chinese Wall policy. Following the model of Brewer *et al.* [14], we let  $S$  denote a set of *subjects*,  $O$  a set of *objects*, and  $L$  a labelling function  $L : O \rightarrow C \times D$ , where  $C$  is a set of *conflict-of-interest classes* and  $D$  a set of *datasets*. The interpretation is that if  $L(o) = (c_o, d_o)$  for an object  $o \in O$ , then  $o$  is in dataset  $d_o$ , and this dataset belongs to the conflict-of-interest class  $c_o$ . The hierarchical structure on objects, datasets and classes amounts to requiring that for any  $o, o' \in O$  if  $L(o) = (c, d)$  and  $L(o') = (c', d)$  then  $c = c'$ . The following ‘simple security rule’ defines when access is granted to an object  $o$ : “either it has the same dataset as an object already accessed by that subject, or, the object belongs to a different conflict-of-interest class.” [14] We can encode this rule in our logic. Consider an event structure  $ES = (E, \leq, \#)$  where the events are  $C \cup D$ , with  $(c, c') \in \#$  for  $c \neq c' \in C$ ,  $(d, d') \in \#$  for  $d \neq d' \in D$ , and  $(c, d) \in \#$  if  $(c, d)$  is not in the image of  $L$  (denoted  $\text{Img}(L)$ ). We take  $\leq$  to be discrete. Then a maximal configuration is a set  $\{c, d\}$  so that the pair  $(c, d) \in \text{Img}(L)$ , corresponding to an object access. A history is then a sequence of object accesses. Now stating the simple security rule as a policy is easy: to access object  $o$  with  $L(o) = (c_o, d_o)$ , the history must satisfy the following policy:

$$\psi^o \equiv F^{-1}d_o \vee G^{-1}\neg c_o$$

In this encoding we have one policy per object  $o$ . One may argue that the policy  $\psi^o$  only captures Chinese Wall for a single object ( $o$ ), whereas the

“real” Chinese Wall policy is a *single policy* stating that “for every object  $o$ , the simple security rule applies.” However, in practical terms this is inessential. Even if there are infinitely many objects, a system implementing Chinese Wall one could easily be obtained using our policies as follows. Say that our proposed security mechanism (intended to implement “real” Chinese Wall) gets as input the object  $o$  and the subject  $s$  for which it has to decide access. Assuming that our mechanism knows function  $L$ , it does the following. If object  $o$  has never been queried before in the run of our system, the mechanism generates “on-the-fly” a new policy  $\psi^o$  according to the scheme above; it then checks  $\psi^o$  with respect to the current history of  $s$ .<sup>1</sup> If  $o$  has been queried before it simply checks  $\psi^o$  with respect to the history of  $s$ . Since only finitely many objects can be accessed in any finite run, only finitely many different policies are generated. Hence, the described mechanism is operationally equivalent to Chinese Wall.

**Example 6.3 (Shallow One-Out-of- $k$ )** The ‘one-out-of- $k$ ’ (OOok) access-control policy was introduced informally by Edjlali *et al.* [33]. Set in the area of access control for mobile code, the OOok scheme dynamically classifies programs into equivalence classes, e.g. “browser-like applications,” depending on their past behaviour. In the following we show that, if one takes the *set-based* formalisation of OOok by Fong [37], we can encode all OOok policies. Since our model is sequence-based, it is richer than Fong’s shallow histories which are sets. An encoding of Fong’s OOok-model thus provides a good sanity-check as well as a *declarative* means of specifying OOok policies (as opposed to the more implementation-oriented security automata).

In Fong’s model of OOok, a finite number of application classes are considered, say,  $1, 2, \dots, k$ . Fong identifies an application class,  $i$ , with a *set of allowed actions*  $C_i$ . To encode OOok policies, we consider an event structure  $ES = (E, \leq, \#)$  with events  $E$  being the set of all access-controlled actions. As in the last example, we take  $\leq$  to be discrete, and the conflict relation to be the maximal irreflexive relation, i.e. a local interaction history in  $ES$  is simply a sequence of single events. Initially, a monitored entity (originally, a piece of mobile code [33]) has taken no actions, and its history (which is a set in Fong’s formalisation) is  $\emptyset$ . If  $S$  is the current history, then action  $a \in E$  is allowed if there exists  $1 \leq i \leq k$  so that  $S \cup \{a\} \subseteq C_i$ , and the history is updated to  $S \cup \{a\}$ . For each action  $a \in E$  we define a policy  $\psi^a$  for  $a$ , expressing Fong’s requirement. Assume, without loss of generality, that the sets  $C_j$  that contain  $a$  are named  $1, 2, \dots, i$  for some  $i \leq k$ . We will assume that each set  $C_j$  is either finite or co-finite.

<sup>1</sup>This check can be done in time linear in the history of subject  $s$ .

Fix a  $j \leq i$ . If the set  $C_j$  is co-finite (i.e., its complement  $E \setminus C_j$  is finite), the following formula  $\psi_j^a$  encodes the requirement that  $S \cup \{a\} \subseteq C_j$ .

$$\psi_j^a \equiv \neg F^{-1} \left( \bigvee_{e \in E \setminus C_j} e \right)$$

If instead  $C_j$  is itself finite, we encode

$$\psi_j^a \equiv G^{-1} \left( \bigvee_{e \in C_j} e \right)$$

Now we can encode the policy for allowing action  $a$  as  $\psi^a \equiv \bigvee_{j=1}^i \psi_j^a$ .

## 6.4 Dynamic Model Checking

The problem of verifying a policy with respect to a given observed history is the model-checking problem: given  $h \in \mathcal{C}_{ES}^+$  and  $\psi$ , does  $h \models \psi$  hold? However, our intended scenario requires a more dynamic view. Each entity will make many decisions, and each decision requires a model check. Furthermore, since the model  $h$  changes as new observations are made, it is not sufficient simply to cache the answers. This leads us to consider the following *dynamic* problem. Devise an implementation of the following interface, ‘*DMC*’. *DMC* is initially given an event structure  $ES = (E, \leq, \#)$  and a policy  $\psi$  written in the basic policy language. Interface *DMC* supports three *operations*: *DMC.new*(), *DMC.update*( $e, i$ ), and *DMC.check*(). A sequence of non-‘check’ operations gives rise to a local interaction history  $h$ , and we shall call this the *actual history*. Internally, an implementation of *DMC* must maintain information about the actual history  $h$ , and operations **new** and **update** are those of Section 6.2, performed on  $h$ . At any time, operation *DMC.check*() must return the truth of  $h \models \psi$ .

In this section, we describe two implementations of interface *DMC*. The first has a cheap precomputation, but higher complexity of operations **update** and **new**, whereas the second implementation has a higher time- and space-complexity for its precomputation, but gains in the long run with a better complexity of the interface operations. Both implementations are inspired by the very efficient algorithm of Havelund and Roşu for model checking past-time LTL [42]. Their idea is essentially this: because of the recursive semantics, model-checking  $\psi$  in  $(h, m)$ , i.e. deciding  $(h, m) \models \psi$ , can be done easily if one knows (1) the truth of  $(h, m - 1) \models \psi_j$  for all sub-formulas  $\psi_j$  of  $\psi$ , and (2) the truth of  $(h, m) \models \psi_i$  for all proper sub-formulas  $\psi_i$  of  $\psi$  (a sub-formula of  $\psi$  is proper if it is not  $\psi$  itself). The truth

of the atomic sub-formulas of  $\psi$  in  $(h, m)$  can be computed directly from the state  $h_m$ , where  $h_m$  is the  $m$ th configuration in sequence  $h$ . For example, if  $\psi_3 = \mathbf{X}^{-1}\psi_4 \wedge e$ , then  $(h, m) \models \psi_3$  iff  $(h, m-1) \models \psi_4$ , and  $e \in h_m$ . This information needed to decide  $(h, m) \models \psi$  can be stored efficiently as two boolean arrays  $B_{last}$  and  $B_{cur}$ , indexed by the sub-formulas of  $\psi$ , so that  $B_{last}[j]$  is true iff  $(h, m-1) \models \psi_j$ , and  $B_{cur}[i]$  is true iff  $(h, m) \models \psi_i$ . Given array  $B_{last}$  and the current state  $h_m$ , one then constructs array  $B_{cur}$  starting from the atomic formulas (which have the largest indices), and working in a ‘bottom-up’ manner towards index 0, for which entry  $B_{cur}[0]$  represents  $(h, m) \models \psi$ . We shall generalise this idea of Havelund and Roşu to obtain an algorithm for the dynamic problem.

We need some preliminary terminology. Initially, the actual interaction history  $h$  is empty, but after some time, as observations are made, the history can be written  $h = x_1 \cdot x_2 \cdots x_M \cdot y_{M+1} \cdots y_{M+K}$ , consisting of a *longest prefix*  $x_1 \cdots x_M$  of *maximal* configurations, followed by a suffix of  $K$  possibly non-maximal configurations  $y_{M+1} \cdots y_{M+K}$ , called the *active sessions* (since we consider the longest prefix,  $y_{M+1}$  must be non-maximal). A maximal configuration represents complete information about a protocol-run, and has the property that it will never change in the future, i.e. cannot be changed by operation **update**. This property will be essential to our dynamic algorithms as it implies that the maximal prefix needs not be stored to check  $h \models \psi$  dynamically.

In the following, the number  $M$  will always refer to the size of the maximal prefix, and  $K$  to the size of the suffix.

### 6.4.1 An Array-based Implementation

We describe an implementation of the *DMC* interface based on a data structure  $DS$  maintaining the active sessions and a collection of boolean arrays. Understanding the data structure is understanding the invariant it maintains, and we will describe this in the following.

The data structure  $DS$  has a vector, accessed by variable  $DS.h$ , storing configurations of  $ES$ , which we denote  $DS.h = (y_1, y_2, \dots, y_K)$ . Part of the invariant is that  $DS.h$  stores only the suffix of active configurations, i.e. the *actual history*  $h$  can be written  $h = x_1 \cdot x_2 \cdots x_M \cdot (DS.h)$ , where the  $x_i$  are all maximal.

**Initialisation.** The data structure is initialised with (a representation of) an event structure  $ES = (E, \leq, \#)$  and a policy  $\psi$ . We assume that the

representation of the configurations of  $ES$ ,  $x \in \mathcal{C}_{ES}$ , is so that the membership  $e \in x$ , conflict  $e \# x$ , singleton union  $x \cup \{e\}$  and maximality (i.e. is  $x \in \mathcal{C}_{ES}$  maximal?) can be computed in constant time. Initialisation starts by enumerating the sub-formulas of  $\psi$ , denoted  $Sub(\psi)$ , such that the following property holds. Let there be  $n + 1$  sub-formulas of  $\psi$ , and let  $\psi_0 = \psi$ . The sub-formula enumeration  $\psi_0, \psi_1, \psi_2, \dots, \psi_n$  satisfies that if  $\psi_i$  is a proper sub-formula of  $\psi_j$  then  $i > j$ .

**Invariance.** As mentioned, part of the invariant is that  $DS.h$  stores exactly the active configurations of the actual history  $h$ . In particular, this means that  $DS.h_1$  is non-maximal, since otherwise there was a larger longest prefix of  $h$ .<sup>2</sup> In addition to  $DS.h$ , the data structure maintains a boolean array  $DS.B_j$  for each entry  $y_j$  in the vector  $DS.h$ . The boolean arrays are indexed by the sub-formulas of  $\psi$  (more precisely, by the integers  $0, 1, \dots, n$ , corresponding to the sub-formula enumeration). The following invariant will be maintained:  $DS.B_k[j]$  is true iff  $(h, M + k) \models \psi_j$ , that is, if-and-only-if the *actual history*  $h = x_1 \cdots x_M \cdot DS.h$  is a model of sub-formula  $\psi_j$  at time  $M + k$ . Additionally, once the longest prefix of maximal configurations becomes non-empty, we allocate a special array  $B_0$ , which maintains a “summary” of the entire maximal prefix of  $h$  with respect to  $\psi$ , meaning that it will satisfy the invariant:  $B_0[j]$  is true iff  $(h, M) \models \psi_j$ .

**Operations.** The invariants above imply that the model-checking problem  $h \models \psi$  can be computed simply by looking at entry 0 of array  $DS.B_K$ , i.e.  $DS.B_K[0]$  is true iff  $(h, M + K) \models \psi_0$  iff  $h \models \psi$ . This means that operation  $DS.check()$  can be implemented in constant time  $O(1)$ . Operation  $DS.new$  is also easy: the vector  $DS.h$  is extended by adding a new entry consisting of the empty configuration. We must also allocate a new boolean array  $DS.B_{K+1}$ , which is initialised using the recursive semantics, consulting the array  $DS.B_K$ , and the current state  $\emptyset$ . This can be done in linear time in the number of sub-formulas of  $\psi$ ,  $O(|\psi|)$ .

The final and most interesting operation, is  $DS.update(e, i)$ . It is assumed as a pre-condition, that  $1 \leq i \leq K$ , and that  $e$  is not in conflict with  $DS.h_i$ . First we must add event  $e$  to configuration  $DS.h_i$ , i.e.  $DS.h_i$  becomes  $DS.h_i \cup \{e\}$ . This is simple, but it may break the invariant. In particular, arrays  $DS.B_k$  (for  $k \geq i$ ) may no longer satisfy  $(h, M + k) \models \psi_j \iff DS.B_k[j] = \mathbf{true}$ . Note, however, that for any  $0 \leq k < i$ , the array  $DS.B_k$  still maintains its invariant. This is due to the

<sup>2</sup>We do not consider, here, the case where  $DS.h$  is empty.

fact that all (sub) formulas are pure-past, and so their truth in  $h$  at time  $k$  does not depend on configurations *later than*  $k$ . In particular, since  $i \geq 1$ , the special array  $DS.B_0$  always maintains its invariant. This means that we can always assume that  $DS.B_{i-1}[j]$  is true iff  $(h, M + i - 1) \models \psi_j$ . This information can be used to correctly fill-in array  $i$ , in time linear in  $|\psi|$ , using the recursive semantics. In turn, this can be used to update array  $i + 1$ , and so forth until we have correctly updated array  $K$ , and the invariants are restored. Finally, in the case that  $i = 1$  and the updated session  $DS.h_1$  has become maximal, the updated actual history  $h$  now has a larger longest prefix of maximal configurations. We must now find the largest  $k \leq K$  so that for all  $1 \leq k' \leq k$ ,  $DS.h_{k'}$  is maximal. All arrays  $DS.B_{k'}$  and configurations  $DS.h_{k'}$  for  $k' < k$  may then be deallocated (configuration  $DS.h_k$  may also be deallocated), and the new “summarising” array  $DS.B_0$  becomes  $DS.B_k$ . We summarise the result of this section as a theorem.

**Theorem 6.1 (Array-based DMC)** *The array-based data structure ( $DS$ ) implements the DMC interface correctly. More specifically, assume that  $DS$  is initialized with a policy  $\psi$  and an event structure  $ES$ , then initialization of  $DS$  is  $O(|\psi|)$ . At any time during execution, the complexity of the interface operations is:*

- $DMC.check()$  is  $O(1)$ .
- $DMC.new()$  is  $O(|\psi|)$ .
- $DMC.update(e, i)$  is  $O((K - i + 1) \cdot |\psi|)$  where  $K$  is the current number of active configurations in  $h$  ( $h$  is the current actual history).

Furthermore, the space requirement of  $DS$  is  $O(K + |E| \cdot |\mathcal{C}_{ES}|)$ .

### 6.4.2 An Automata-based Implementation

In this section, we describe an alternative implementation of the *DMC* interface. The implementation uses a finite automaton to improve the *dynamic* complexity of the algorithm at the cost of a one-time computation, constructing the automaton.

We consider the problem of model-checking  $\psi$  in a history  $h = x_1x_2 \cdots x_{M+K}$  as the string-acceptance problem for an automaton,  $A_\psi$ , reading symbols from an alphabet consisting of the finite configurations of  $ES$ . The language  $\{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$  turns out to be regular for all  $\psi$  in our policy language.

The states of the automaton  $A_\psi$  will be boolean arrays of size  $|\psi|$ , i.e. indexed by the sub-formulas of  $\psi$ . Thinking slightly more abstractly about the Havelund-Roşu algorithm, filling the array  $B_{cur}$  using  $B_{last}$  and the current configuration  $x \in \mathcal{C}_{ES}$  can be seen as an automaton transition from state  $B_{last}$  to state  $B_{cur}$  performed when reading symbol  $x$ . We need some preliminary notation.

Let us identify a boolean array  $B$  indexed by the sub-formulas of  $\psi$  with a set  $s \in \mathbf{2}^{Sub(\psi)}$ , i.e.  $B[j] = \mathbf{true}$  iff  $\psi_j \in s$ . The recursive semantics for a fixed formula  $\psi$ , can be seen as an algorithm, denoted  $RecSem$ , taking as input the array  $B_{last} \in \mathbf{2}^{Sub(\psi)}$  and the current configuration  $x \in \mathcal{C}_{ES}$ , and giving as output  $B_{cur} \in \mathbf{2}^{Sub(\psi)}$ . Furthermore, the base-case of the recursive semantics can be seen as an algorithm taking only a configuration as input and giving a subset  $s \in \mathbf{2}^{Sub(\psi)}$  as output. The input-output behaviour of the recursive-semantics algorithm is exactly the transition function of our automaton.

**Definition 6.5 (Automaton  $A_\psi$ )** Let  $\psi$  be a formula in the pure-past policy language  $\mathcal{L}(ES)$ , where  $ES$  is an event structure. Define a deterministic finite automaton  $A_\psi = (S, \Sigma, s_0, F, \delta_\psi)$ , where  $S = \mathbf{2}^{Sub(\psi)} \cup \{s_0\}$  is the set of states,  $s_0 \notin \mathbf{2}^{Sub(\psi)}$  being a special initial state, and  $\Sigma = \mathcal{C}_{ES}$  is the alphabet. The final states  $F$  consist of the set  $\{s \in S \mid \psi \in s\}$ , and if  $\epsilon \models \psi$  then the initial state is also final, i.e.  $s_0 \in F$  iff  $\emptyset \models \psi$ . The transition function restricted to the non-initial states,  $\delta_\psi : \mathbf{2}^{Sub(\psi)} \times \mathcal{C}_{ES} \rightarrow \mathbf{2}^{Sub(\psi)}$ , is given by the recursive semantics, i.e.  $\delta_\psi(s, x) = RecSem(s, x)$  for all  $s \in \mathbf{2}^{Sub(\psi)}, x \in \Sigma$ . The transition function on the initial state,  $\delta_\psi(s_0, -)$ , is given by the base-case of the recursive semantics.

Since we have identified the empty structure  $\epsilon \in \mathcal{C}_{ES}^*$  with the singleton sequence  $\emptyset$ , we take the initial state to be a final state if-and-only-if  $\emptyset \models \psi$ . The additional accepting states are those that contain formula  $\psi$ .

Let  $\hat{\delta}_\psi$  denote the canonical extension of function  $\delta_\psi$  to strings  $h \in \mathcal{C}_{ES}^*$ .

**Lemma 6.1 (Automaton Invariant)** *Let  $h \in \mathcal{C}_{ES}^+$  be any non-empty history, and  $\psi_j$  be any sub-formula of  $\psi$ . Then  $\hat{\delta}_\psi(s_0, h) \neq s_0$  and furthermore,  $\psi_j \in \hat{\delta}_\psi(s_0, h)$  if-and-only-if  $h \models \psi_j$ .*

*Proof.* Simple induction in  $h$ . □

**Theorem 6.2**  $\mathcal{L}(A_\psi) = \{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$

*Proof.* Immediate from Lemma 6.1 and the definition of  $s_0$  and  $F$ .  $\square$

the abstract setting of automaton  $A_\psi$ , we can now give a very simple and concise description of an alternative data structure  $DS'$  for implementing the interface for dynamic model checking,  $DMC$ . The basic idea is to pre-construct the automaton during initialisation, and basically replacing the dynamic filling of the arrays  $DS.B_j$  of  $DS$  with automaton-transitions.

**Initialisation.** Just as with  $DS$ , the data structure  $DS'$  is initialised with an event structure  $ES$  and formula  $\psi$ . Initialisation now simply consists of constructing the automaton  $A_\psi$ . More specifically, we construct the transition-matrix of  $\delta_\psi$  so that  $\delta_\psi(s, x)$  can be computed in time  $O(1)$  by a matrix-lookup.<sup>3</sup>  $DS'$  maintains a variable  $DS'.s_{\text{summ}}$  of type  $S$  (the automaton states) which is initialised to  $s_0$ . In addition to  $s_{\text{summ}}$ ,  $DS'$  will store a vector of pairs  $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$ , where the  $y_i$ 's are configurations representing active sessions, and the  $s_i$ 's are corresponding automaton-states where  $s_i$  is the state that  $A_\psi$  is in after reading  $y_i$ . Initially this vector is empty.

**Invariance.** Let  $h = x_1x_2 \cdots x_M \cdot y_{M+1} \cdots y_{M+K}$  be the actual interaction history, i.e.  $(x_i)_{i=1}^M$  is the longest prefix of maximal configurations. The data-structure invariant of  $DS'$  is that, if  $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$  then  $(y_1, \dots, y_K)$  are the active configurations of  $h$ , and  $s_i$  is the state of the automaton after reading the string  $x_1x_2 \cdots x_M \cdot y_1 \cdots y_i$ , when started in state  $s_0$ . The invariant regarding the special variable  $DS'.s_{\text{summ}}$  is simply that  $DS'.s_{\text{summ}} = \hat{\delta}_\psi(s_0, x_1x_2 \cdots x_M)$ , i.e.  $DS'.s_{\text{summ}}$  “summarises” the history up to time  $M$  with respect to formula  $\psi$ . Notice that the invariant is satisfied after initialisation.

**Operations.** Let  $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$ . Then operation  $DMC.\text{check}()$  returns true iff  $s_K \in F$ . By the invariant and Lemma 6.1 this is equivalent to  $h \models \psi$ . For operation  $DMC.\text{new}()$ , extend  $DS'.h$  with the pair  $(\emptyset, \delta_\psi(s_K, \emptyset))$ . Finally, for operation  $DMC.\text{update}(e, i)$ , add  $e$  to configuration  $y_i$  of  $DS'.h$ , then update the table  $DS'.h$  by starting the automaton in state  $s_{i-1}$  (or  $s_{\text{summ}}$  if  $i = 1$ ), and setting  $s_i := \delta_\psi(s_{i-1}, y_i)$ , and then  $s_{i+1} := \delta_\psi(s_i, y_{i+1})$ , and so on until the entire table  $DS'.h$  satisfies the invariant. If  $i = 1$  and  $y_1 \cup \{e\}$  is maximal, we must, as in  $DS$ , recompute

<sup>3</sup>We choose a transition-matrix representation of  $\delta_\psi$  for simplicity. In practice, any representation allowing efficient computations of  $\delta_\psi(s, x)$  could be used.

the largest longest prefix, and we may deallocate the corresponding part of the table  $DS'.h$  (taking care to update  $DS'.s_{\text{summ}}$  appropriately).

Since  $\delta_\psi$  can be evaluated in time  $O(1)$ , we get the following theorem.

**Theorem 6.3 (Automata-based DMC)** *The automata-based data structure ( $DS'$ ) implements the DMC interface correctly. More specifically, assume that  $DS'$  is initialised with a policy  $\psi$  and an event structure  $ES = (E, \leq, \#)$ , then initialisation of  $DS'$  is  $O(2^{|\psi|} \cdot |C_{ES}| \cdot |\psi|)$ . At any time during execution, the complexity of the interface operations is:*

- $DMC.\text{check}()$  is  $O(1)$ .
- $DMC.\text{new}()$  is  $O(1)$ .
- $DMC.\text{update}(e, i)$  is  $O(K - i + 1)$  where  $K$  is the current number of active configurations in  $h$  ( $h$  is the current actual history).

Furthermore, the space requirement of  $DS'$  is  $O(K + |E| \cdot |C_{ES}| + 2^{|\psi|} \cdot |C_{ES}|)$ .

### 6.4.3 Remarks

The array- and automata-based implementations are very similar. The automata-based implementation simply precomputes a matrix of transitions  $B \xrightarrow{x} B'$  instead of recomputing from scratch the array  $B'$  from  $B$  and  $x$ , every time it is needed. This reduces the complexity of operations  $DMC.\text{update}(e, i)$  and  $DMC.\text{new}()$  by a factor of  $|\psi|$ . The cost of this is in terms of storage and time for pre-computation, where, in the worst case, the transition matrix is exponential in  $\psi$  (of size  $2^{|\psi|} \times |C_{ES}|$ ). One important advantage with the automata-based implementation (besides being conceptually simpler) is that we can apply the standard technique for constructing the minimal finite automaton equivalent to  $A_\psi$ . We believe that, in practice, this minimisation will give significant time and space reductions. Note that minimisation can be run several times, and not just during initialisation. In particular, one could run minimisation each time state  $s_{\text{summ}}$  is updated in order to obtain optimisations, e.g. removing states that are unreachable in the future.

Recall, that one might be interested in different notions of “time” in our temporal logic. Consider the following. Redefine **update**( $\cdot$ ): for  $h = x_1 \cdots x_n$ ,  $1 \leq i \leq N$ ,  $e \in E$ , define

$$\mathbf{update}(h, e, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} x_{i+2} \cdots x_N (x_i \cup \{e\})$$

This definition implements the idea that  $x_i$  precedes  $x_j$  in sequence  $h$  if  $x_j$  was updated more recently than  $x_i$ . Notice that our algorithms (as well as complexity analyses) can be easily adapted to this time concept: the update operation simply swaps the indexes of configurations  $i$  and  $N$  in the vector of configurations before updating the boolean arrays (or automata-states in case of the automata-based algorithm).

## 6.5 Language Extensions

In this section, we consider two extensions of the basic policy language to include more realistic and practical policies. The first is parameters and quantification. For example, consider the OOk policy for classifying “browser-like” applications (Section 6.3). We could use a clause like  $G^{-1}(\text{open-f} \rightarrow F^{-1}\text{create-f})$  for two events `open-f` and `create-f`, representing respectively the opening and creation of a file with name  $f$ . However, this only encodes the requirement that for a *fixed*  $f$ , file  $f$  must be created before it is opened. Ideally, one would want to encode that for *any* file, this property holds, i.e., a formula similar to

$$G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))])$$

where  $x$  is a variable, and the universal quantification ranges over all possible file-names. The first language extension allows this sort of quantification, and considers an accompanying notion of parameterised events.

The second language extension covers two aspects: quantitative properties and referencing. Pure-past temporal logic is very useful for specifying qualitative properties. For instance, in the eBay example, “the seller has never provided negative feedback in auctions where payment was made,” is directly expressible as  $G^{-1}(\text{negative} \rightarrow \text{ignore})$ . However, sometimes such qualitative properties are too strict to be useful in practice. For example, in the policy above, a single erroneous negative feedback provided by the seller will lead to the property being irrevocably unsatisfiable. For this reason, our first extension to the usual past-time temporal-logic is the ability to express also *quantitative* properties, e.g. “in at least 98% of the previous interactions, seller has not provided negative feedback in auctions where payment was made.” The second extension is the ability, to not only refer to the locally observed behaviour, but also to require properties of the behaviour observed by others. As a simple example of this, suppose that  $b_1$  and  $b_2$  are two branches of the same network of banks. When a client  $c$  wants to obtain a loan in  $b_1$ , the policy of  $b_1$  might require not only that

$c$ 's history in  $b_1$  satisfy some appropriate criteria, but also that  $c$  has always paid his mortgage on time in his previous loans with  $b_2$ . Thus we allow local policies, like that of  $b_1$ , to refer to the *global* behaviour of an entity.

### 6.5.1 Quantification

We introduce a notion of parameterised event structure, and proceed with an extension of the basic policy language to include quantification over parameters. A parameterised event structure is like an ordinary event structure, but where events occur with certain parameters (e.g. `open("/etc/passwd")` or `open("./tmp.txt")`).

#### Parameterised Event Structures

We define parameterised event structures and an appropriate notion of configuration.

**Definition 6.6 (Parameterised Event Structure)** A *parameterised event structure* is a tuple  $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$  where  $(E, \leq, \#)$  is an (ordinary) event structure, component  $\mathcal{P}$ , called the *parameters*, is a set of countable *parameter sets*,  $\mathcal{P} = \{P_e \mid e \in E\}$ , and  $\rho : E \rightarrow \mathcal{P}$  is a function, called the *parameter-set assignment*.

**Definition 6.7 (Configuration)** Let  $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$  be a parameterised event structure. A *configuration* of  $\rho ES$  is a partial function  $x : E \rightarrow \bigcup_{e \in E} \rho(e)$  satisfying the following two properties. Let  $dom(x) \subseteq E$  be the set of events on which  $x$  is defined. Then

$$\begin{aligned} dom(x) &\in \mathcal{C}_{ES} \\ \forall e \in dom(x). x(e) &\in \rho(e) \end{aligned}$$

When  $x$  is a configuration, and  $e \in dom(x)$ , then we say that  $e$  has occurred in  $x$ . Further, when  $x(e) = p \in \rho(e)$ , we say that  $e$  has occurred with parameter  $p$  in  $x$ . So a configuration is a set of event occurrences, each occurred event having exactly one parameter.

**Notation 6.2** We write  $\mathcal{C}_{\rho ES}$  for the set of configurations of  $\rho ES$ , and  $\mathcal{C}_{\rho ES}^0$  for the set of *finite* configurations of  $\rho ES$  (a configuration  $x$  is finite if  $dom(x)$  is finite). If  $x, y$  are two partial functions  $x : A \rightarrow B$  and  $y : C \rightarrow D$  we write  $(x/y)$  (pronounced  $x$  over  $y$ ) for the partial function  $(x/y) : A \cup B \rightarrow C \cup D$  given by  $dom(x/y) = dom(x) \cup dom(y)$ , and for all  $e \in dom(x/y)$  we

have  $(x/y)(e) = x(e)$  if  $e \in \text{dom}(x)$  and otherwise  $(x/y)(e) = y(e)$ . Finally we write  $\emptyset$  for the totally undefined configuration (when the meaning is clear from the context).

Here we are not interested in the theory of parameterised event structures, but mention only that they can be explained in terms of ordinary event structures by expanding a parameterised event  $e$  of type  $\rho(e)$  in to a set of conflicting events  $\{(e, p) \mid p \in \rho(e)\}$ . However, the parameters give a convenient way of saying that the *same* event can occur with different parameters (in different runs).

**Definition 6.8 (Histories)** A local (interaction) history  $h$  in a parameterised event structure  $\rho ES$  is a finite sequence  $h \in \mathcal{C}_{\rho ES}^0$ .

**Definition 6.9 (Extended Interface)** Overload operation  $\mathbf{new} : \mathcal{C}_{\rho ES}^0 \rightarrow \mathcal{C}_{\rho ES}^0$  by  $\mathbf{new}(h) = h\emptyset$ . Overload also partial operation  $\mathbf{update} : \mathcal{C}_{\rho ES}^0 \times E \times (\bigcup_{e \in E} \rho(e)) \times \mathbb{N} \rightarrow \mathcal{C}_{\rho ES}^0$  as follows. For any  $h = x_1 x_2 \cdots x_i \cdots x_n \in \mathcal{C}_{\rho ES}^0$ ,  $e \in E$ ,  $p \in \bigcup_{e \in E} \rho(e)$ , and  $i \in \mathbb{N}$ ,  $\mathbf{update}(h, e, p, i)$  is undefined if  $i \notin \{1, 2, \dots, n\}$ ,  $\text{dom}(x_i) \not\supseteq \text{dom}(x_i) \cup \{e\}$  or  $p \notin \rho(e)$ . Otherwise

$$\mathbf{update}(h, e, p, i) = x_1 x_2 \cdots ((e \mapsto p)/x_i) \cdots x_n$$

Throughout the following sections, we let  $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$  be a parameterised event structure, where  $\mathcal{P} = \{P_i \mid i \in \mathbb{N}\}$ .

### Quantified Policies

We extend the basic language from Section 6.3 to parameterised event structures, allowing quantification over parameters.

**Syntax.** Let  $Var$  denote a countable set of variables (ranged over by  $x, y, \dots$ ). Let the meta-variables  $v, u$  range over  $Val \stackrel{(def)}{=} Var \cup \bigcup_{i=1}^{\infty} P_i$ , and metavariable  $p$  range over  $\bigcup_{i=1}^{\infty} P_i$ .

The quantified policy language is given by the following BNF.

$$\begin{aligned} \psi \quad ::= \quad & e(v) \mid \diamond e(v) \mid \psi_0 \wedge \psi_1 \mid \neg \psi \mid \\ & \mathbf{X}^{-1} \psi \mid \psi_0 \mathbf{S} \psi_1 \mid \forall x : P_i. \psi \end{aligned}$$

We need some terminology.

- Write  $fv(\psi)$  for the set of free variables in  $\psi$  (defined in the usual way).
- A *policy* of the quantified language is a closed formula.

- Let  $\psi$  be any formula. Say that a variable  $x$  has type  $P_i$  in  $\psi$  if it occurs in a sub-formula  $e(x)$  of  $\psi$  and  $\rho(e) = P_i$ .
- We use the syntactic abbreviations of Section 6.3, and additionally the existential quantification  $\exists x : P_i.\psi \equiv \neg\forall x : P_i.\neg\psi$ .

We impose the following static well-formedness requirement on formulas  $\psi$ . All free variables have unique type, and, if  $x$  is a bound variable of type  $P_i$  in  $\psi$ , then  $x$  is bound by a quantifier of the correct type (e.g., by  $\forall x : P_i.\psi$ ). Further, for each occurrence of  $e(p)$ ,  $p$  is of the correct type:  $p \in \rho(e)$ .

**Semantics.** A (generalised) substitution is a function  $\sigma : Val \rightarrow \bigcup_{i=1}^{\infty} P_i$  so that  $\sigma$  is the identity on each of the parameter sets  $P_i$ . Let  $h = x_1 \cdots x_n \in \mathcal{C}_{\rho ES}^0$  be a history, and  $i \in \mathbb{N}$ . We define a satisfaction relation  $(h, i) \models^\sigma \psi$  by structural induction on  $\psi$ .

$$\begin{aligned}
(h, i) \models^\sigma e(v) & \quad \text{iff } 1 \leq i \leq N \text{ and } e \in \text{dom}(x_i) \text{ and } x_i(e) = \sigma(v) \\
(h, i) \models^\sigma \diamond e(v) & \quad \text{iff } 1 \leq i \leq N \Rightarrow (e \notin \text{dom}(x_i) \text{ and} \\
& \quad (e \in \text{dom}(x_i) \Rightarrow x_i(e) = \sigma(v))) \\
(h, i) \models^\sigma \psi_0 \wedge \psi_1 & \quad \text{iff } (h, i) \models^\sigma \psi_0 \text{ and } (h, i) \models^\sigma \psi_1 \\
(h, i) \models^\sigma \neg\psi & \quad \text{iff } (h, i) \not\models^\sigma \psi \\
(h, i) \models^\sigma \mathbf{X}^{-1}\psi & \quad \text{iff } i > 1 \text{ and } (h, i-1) \models^\sigma \psi \\
(h, i) \models^\sigma \psi_0 \mathbf{S} \psi_1 & \quad \text{iff } \exists j \leq i. ((h, j) \models^\sigma \psi_1) \text{ and} \\
& \quad [\forall j < j' \leq i. (h, j') \models^\sigma \psi_0] \\
(h, i) \models^\sigma \forall x : P_j.\psi & \quad \text{iff } \forall p \in P_j. (h, i) \models^{((x \mapsto p)/\sigma)} \psi
\end{aligned}$$

**Example 6.4 (True OOk)** Recall the ‘one-out-of- $k$ ’ policy (Example 6.3). Edjlali *et al.* give, among others, the following example of an OOk policy classifying “browser-like” applications: “allow a program to connect to a remote site if and only if it has neither tried to open a local file that it has not created, nor tried to modify a file it has created, nor tried to create a sub-process.” Since this example implicitly quantifies over all possible files (for *any* file  $f$ , if the application tries to open  $f$  then it must have previously have created  $f$ ), it cannot be expressed directly in our basic language. Note also that this policy cannot be expressed in Fong’s set-based model [37]. This follows since the above policy essentially depends on the *order* in which events occur (i.e. **create** before **open**).

Now consider a parameterised event structure with two conflicting events: **create** and **open**, each of type *String* (representing file-names). Consider the following quantified policy:

$$\mathbf{G}^{-1}(\forall x : \text{String}.(\text{open}(x) \rightarrow \mathbf{F}^{-1}\text{create}(x)))$$

This faithfully expresses the idea of Edjlali *et al.* that the application “can only open files it has previously created.”

### Model Checking the Quantified Language

We can extend the array-based algorithm to handle the quantified language. The key idea is the following. Instead of having *boolean* arrays  $B_k[j]$ , we associate with each sub-formula  $\psi_j$  of a formula  $\psi$ , a *constraint*  $C_k[j]$  on the free variables of  $\psi_j$ . The invariant will be that the sub-formula  $\psi_j$  is true for a substitution  $\sigma$  at time  $(h, k)$  if-and-only-if  $\sigma$  “satisfies” the constraint  $C_k[j]$ , i.e.,  $C_k[j]$  represents the set of substitutions  $\sigma$  so  $(h, k) \models^\sigma \psi_j$ .

**Constraints.** Fix a quantified formula  $\psi$  and a history  $h = x_1x_2 \cdots x_n \in \mathcal{C}_{\rho ES}^0$ . We assume for simplicity that all  $m$  variables of  $\psi$ , say  $\text{vars}(\psi) = \{y_1, y_2, \dots, y_m\}$ , have the same type  $P$  (this restriction is inessential). Let  $P_h \subset P$  denote the set of distinct parameter occurrences in  $h$  (i.e.,  $P_h = \{q \in P \mid \exists e \in E \exists i \leq |h|. e \in \text{dom}(x_i) \text{ and } x_i(e) = q\}$ ). For a finite set  $V$  of variables, let  $\Sigma_V$  denote the set of substitutions for the variables  $V$ , i.e.,  $\Sigma_V = V \rightarrow P$ . Let us define an equivalence  $\equiv_{P_h}$  on substitutions  $\Sigma_V$ , by

$$\sigma \equiv_{P_h} \sigma' \text{ iff } \forall x \in V. \begin{cases} \sigma(x) = \sigma'(x) & \text{if } \sigma(x) \in P_h \\ \sigma'(x) \notin P_h & \text{if } \sigma(x) \notin P_h \end{cases}$$

Let  $\Sigma_V^{P_h} = \Sigma_V / \equiv_{P_h}$  be the set of equivalence classes for  $\equiv_{P_h}$ . Let  $\star \notin P$  be arbitrary but fixed. Note that an equivalence class  $[\sigma]$  can be uniquely represented as a function  $s : V \rightarrow P_h \cup \{\star\}$ , i.e., by  $s(x) = \sigma(x)$  if  $\sigma(x) \in P_h$  and  $s(x) = \star$  otherwise. This is clearly independent of the class representative  $\sigma$ . For the rest of this paper we shall identify  $\Sigma_V^{P_h}$  with  $V \rightarrow P_h \cup \{\star\}$ . The following lemma establishes that with respect to model checking, substitutions are only distinguished up to  $\equiv_{P_h}$ -equivalence.

**Lemma 6.2** *For all quantified formulas  $\psi$ , all histories  $h$ , and all substitutions  $\sigma, \sigma' \in \Sigma_{fv(\psi)}$*

$$\text{if } \sigma \equiv_{P_h} \sigma' \text{ then } h \models^\sigma \psi \iff h \models^{\sigma'} \psi$$

*Proof.* Let  $h = x_1 \cdots x_n$  be fixed, and recall  $P_h = \{q \in P \mid \exists e \in E \exists i \leq |h|. e \in \text{dom}(x_i) \text{ and } x_i(e) = q\}$ . Let  $\sigma \equiv_{P_h} \sigma'$ . Our proof is by structural induction in  $\psi$ . For the base case we need only consider the atomic formulas of form  $e(x)$  or  $\diamond e(x)$  (if  $\psi$  doesn't have a free variable then its truth is independent of the substitution). If  $\sigma(x) \in P_h$  then since  $\sigma \equiv_{P_h} \sigma'$ , we have  $\sigma'(x) = \sigma(x)$  and the result is obvious. If  $\sigma(x) \notin P_h$  then since  $\sigma \equiv_{P_h} \sigma'$ ,

we also have  $\sigma'(x) \notin P_h$ . Hence  $h \not\models^\sigma e(x)$  and  $h \not\models^{\sigma'} e(x)$ . If  $e$  is in conflict with  $x_n$  or  $e \in x_n$  then  $h \not\models^\sigma \diamond e(x)$  and  $h \not\models^{\sigma'} \diamond e(x)$ , otherwise  $h \models^\sigma \diamond e(x)$  and  $h \models^{\sigma'} \diamond e(x)$ .

For the inductive step, all cases follow trivially from the inductive hypothesis. For example, for  $\psi = \forall x : P_j.\psi$  then since  $h \models^\sigma \psi \iff h \models^{\sigma'} \psi$ , clearly  $h \models^\sigma \forall x : P_j.\psi$  iff for all  $p \in P_j.h \models^{(x \mapsto p)/\sigma} \psi_j$  iff  $p \in P_j.h \models^{(x \mapsto p)/\sigma'} \psi_j$  (because for any fixed  $p \in P_j$  we have  $(x \mapsto p)/\sigma \equiv_{P_h} (x \mapsto p)/\sigma'$ ).  $\square$

function  $c : \Sigma_V^{P_h} \rightarrow \{\top, \perp\}$  is called a ( $V$ -) *constraint* (in  $h$ ). A substitution  $\sigma \in \Sigma_V$  satisfies constraint  $c$  if  $c([\sigma]) = \top$ . In this case we write  $\sigma \models c$ . We write  $\text{Constraint}_V$  for the set of  $V$ -constraints (in some fixed history  $h$  which is clear from the context), and if  $c : \Sigma_V^{P_h} \rightarrow \{\top, \perp\}$  is a constraint, then  $\text{vars}(c) \stackrel{(\text{def})}{=} V$ . Notice that when  $\text{fv}(\psi) = \emptyset$  then  $\Sigma_{\text{fv}(\psi)} \simeq \mathbf{1}$  (i.e., a singleton set), hence a constraint is simply a boolean. In this sense, constraints generalise booleans.

In the array-based algorithm, sub-formula  $\psi_j$  will be associated with a  $\psi_j$ -constraint  $C_k[j]$  in  $h$ , i.e., on the free variables of  $\psi_j$  (where  $C_k$  will correspond to time  $k$  in a history  $h$ ). Notice that replacing the boolean arrays  $B_k[j]$  with constraint arrays  $C_k[j]$  can be seen as a proper generalisation of the array-based algorithm. We generalise the (main) invariant of the algorithm from

$$h, k \models \psi_j \iff B_k[j] = \text{true}$$

to

$$\forall \sigma \in \Sigma_{\text{fv}(\psi_j)}. [h, k \models^\sigma \psi_j \iff \sigma \models C_k[j]]$$

Notice that for closed  $\psi_j$ , the invariants are equivalent. It is also important to notice that constraints can be viewed as functions taking as input an  $m$ 'ary vector of  $(P_h \cup \{\star\})$ -values (where  $m$  is the number of variables) and giving a boolean value as output. Hence constraints are finite objects. Notice also that since constraints are boolean valued, it makes sense to consider logical operators on constraints, e.g., the conjunction  $(c \wedge c')([\sigma]) = c([\sigma]) \wedge c'([\sigma])$  of two constraints  $c$  and  $c'$  (even if they are not on the same variables).<sup>4</sup> For a variable  $x$  and a parameter  $p \in P_h$  we will use notation  $x \in \{p\}$  to denote the constraint given by  $(x \in \{p\})([\sigma]) = \top \iff \sigma(x) = p$ . Further  $\top$  and  $\perp$  denote respectively the two constant constraints.

**Constructing constraints.** Let  $h = x_1 \cdots x_n$  be a history and  $1 < k \leq n$ . Define a translation  $\llbracket \cdot \rrbracket_h^k$  from the quantified language to constraints,

<sup>4</sup>If  $A \subseteq B$  then an  $A$ -constraint can be seen as a  $B$ -constraint by imposing no additional requirements on the extra variables.

associating with each formula in the quantified language  $\psi$ , a constraint  $\llbracket \psi \rrbracket_h^k$  on the free variables of  $\psi$ . The function  $\llbracket \cdot \rrbracket_h^k$  is defined relative to index  $k$  and history  $h$ , and we assume (inductively) that when defining  $\llbracket \psi \rrbracket_h^k$ , we have access to  $\llbracket \psi' \rrbracket_h^k$  for all proper sub-formulas  $\psi'$  of  $\psi$ , and also  $\llbracket \psi' \rrbracket_h^{k-1}$  for all sub-formulas  $\psi'$  of  $\psi$ . In the model-checking algorithm, the constraint  $\llbracket \psi_j \rrbracket_h^k$  will correspond to entry  $j$  in array  $C_k$ . Recall that the invariant we aim to maintain is the following.

$$\forall \sigma \in \Sigma_{fv(\psi_j)}. [h, k \models^\sigma \psi_j \iff \sigma \models C_k[j]]$$

We define function  $\llbracket \cdot \rrbracket_h^k$  as follows.

$$\llbracket e(v) \rrbracket_h^k = \begin{cases} y \in \{p\} & \text{if } v = y \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \top & \text{if } v = p \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \perp & \text{otherwise} \end{cases}$$

$$\llbracket \diamond e(v) \rrbracket_h^k = \begin{cases} y \in \{p\} & \text{if } v = y \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \top & \text{if } (v = p \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p) \text{ or if} \\ & e \notin \text{dom}(x_k) \text{ and } e \notin \# \text{dom}(x_k) \\ \perp & \text{otherwise} \end{cases}$$

We proceed inductively in  $\psi$ .

$$\begin{aligned} \llbracket \psi_0 \wedge \psi_1 \rrbracket_h^k &= \llbracket \psi_0 \rrbracket_h^k \wedge \llbracket \psi_1 \rrbracket_h^k \\ \llbracket \neg \psi \rrbracket_h^k &= \neg \llbracket \psi \rrbracket_h^k \\ \llbracket X^{-1} \psi \rrbracket_h^k &= \llbracket \psi \rrbracket_h^{k-1} \\ \llbracket \psi_0 \text{ S } \psi_1 \rrbracket_h^k &= \llbracket \psi_1 \rrbracket_h^k \vee (\llbracket \psi_0 \text{ S } \psi_1 \rrbracket_h^{k-1} \wedge \llbracket \psi_0 \rrbracket_h^k) \\ \llbracket \forall x : P. \psi \rrbracket_h^k &= \text{elim}_x(\llbracket \psi \rrbracket_h^k) \end{aligned}$$

All the clauses are straightforward except for  $\forall x : P. \psi$ , which is handled by auxiliary function  $\text{elim}_x$ . We define this function now. Assuming we have access to  $c = \llbracket \psi \rrbracket_h^k$  so that  $\sigma \models c \iff (h, k) \models^\sigma \psi$ , we must produce a new constraint  $c'$  of type  $\text{Constraint}_{fv(\psi) \setminus \{x\}}$ , so that

$$\sigma \models c' \iff [\forall p \in P. ((x \mapsto p) / \sigma) \models c] \quad (\text{for all } \sigma)$$

The function  $\text{elim}_x$  does this; it transforms a constraint  $c$  into a constraint  $c' = \text{elim}_x(c)$  with  $\text{vars}(c') = \text{vars}(c) \setminus \{x\}$ , satisfying the above equivalence.

Since  $P_h$  is finite we can build  $c'$  as one large conjunction: for all  $\sigma \in \Sigma_{fv(\psi) \setminus \{x\}}$

$$c'([\sigma]) = \left( \bigwedge_{q \in P_h} c([(x \mapsto q)/\sigma]) \right) \wedge c((x \mapsto \star)/[\sigma])$$

Notice that we would obtain a function for existential quantification by taking a disjunction instead of a conjunction.

**Array-based Model Checking.** In the light of function  $[\cdot]$  there is a straightforward extension of data-structure  $DS$  into a similar data-structure  $DS^\forall$  for array-based dynamic model-checking of the quantified language. Structure  $DS^\forall$  will maintain a history  $DS^\forall.h = x_1x_2 \cdots x_n$ , and a collection of  $n + 1$  constraint-arrays  $DS^\forall.C_k[j]$  (for  $0 \leq k \leq n$ ), each array indexed by the sub-formulas of  $\psi$ . The constraint in  $C_k[j]$  will be  $C_k[j] = \llbracket \psi_j \rrbracket_k^h$  for  $k > 0$  ( $C_0$  is the special summary constraint). The invariant implies that for any closed  $\psi$ ,

$$(h, n) \models \psi \iff \models DS^\forall.C_n[0]$$

(we write  $\models c$ , and say that  $c$  is *valid*, if  $c = \top$ ). Hence operation **check** is a validity check, which is easy since  $vars(DS^\forall.C_n[0]) = \emptyset$  when  $\psi$  is closed. Operation **new** is essentially as in  $DS$  (with the generalisation from booleans to constraints).

For operation **update**( $e, p, i$ ) there are two cases. In the first case  $p \in P_h$ , and update works as usual (again generalising to constraints). In the case where  $p \notin P_h$ , we update history  $h$  to  $h'$  appropriately, and thus obtain a new, larger  $P_{h'} = P_h \cup \{p\}$ . Notice that constraints in  $h$  can be easily extended to constraints in  $h'$ : if  $c : \Sigma_V^{P_h} \rightarrow \{\top, \perp\}$  then we can think of  $c$  as a constraint in  $h'$  by the following. For all  $\sigma \in \Sigma_V$ , let  $[\sigma]_{h'}$  be the  $\equiv_{P_{h'}}$ -equivalence class for  $\sigma$ , and let  $[\sigma]_h$  be the  $\equiv_{P_h}$ -equivalence class for  $\sigma$ , then

$$c([\sigma]_{P_{h'}}) = c([\sigma]_{P_h})$$

This means that we can use the logical operators on constraints  $c$  in the history  $h$  and constraints  $c'$  in the history  $h'$ , by first extending  $c$  to a constraint in  $h'$ , and then performing the logical operation. Hence **update**( $e, p, i$ ) can be implemented as usual, except that we may need to dynamically extend some constraints in  $h$  to constraints in  $h'$ .

**Complexity.** The above paragraphs show that dynamic model-checking for the quantified language is decidable in spite of the fact that we allow quantification over infinite parameter sets. This is essentially due to the fact that in any history, only a finite portion of the parameters can actually occur. However, we do have the following hardness result.

**Proposition 6.1 (PSPACE Hardness)** Even for single element models, the model-checking problem for the quantified policy language is PSPACE hard.

*Proof.* Fix a parameterised event structure  $ES$ . A quantified model-checking (QMC) instance (for  $ES$ ) consists of a history  $h = x_1 \cdots x_n$  and a closed formula  $\psi$  of the quantified language (over  $ES$ ). Say that a QMC instance  $(h, \psi)$  is in QMC if  $h \models \psi$ . A single element model is a model,  $h \in \mathcal{C}_{\rho ES}^0$ , with  $h = x$ , where  $x \in \mathcal{C}_{\rho ES}^0$ .

The quantified boolean formula (QBF) problem is the problem of deciding the truth of quantified formulas of the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. \phi(x_1, \dots, x_n)$$

where each  $Q_i$  is a quantifier ( $\forall$  or  $\exists$ ), and  $\phi$  is a quantifier-free boolean formula (i.e., a propositional formula) with  $fv(\phi) \subseteq \{x_1, \dots, x_n\}$ . The QBF problem is known to be PSPACE complete [97]. Given a QBF  $f = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. \phi$ , construct an MC-instance as follows. Use a parameterised event structure with a single event  $\star$  having two possible parameters  $\perp$  and  $\top$ . Let  $h = [\star \mapsto \top]$  be a single element history. Construct formula  $\psi$  as  $\psi \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. \psi'$ , where  $x_1, \dots, x_n$  are the variables of  $f$ , and  $\psi'$  is  $\phi$  with each variable  $x_j$  replaced by  $\star(x_j)$ . Then  $f$  is satisfiable if-and-only-if  $(h, |h|) \models \psi$ .  $\square$

While the general problem is PSPACE hard, we are able to obtain the following quantitative result which bounds the complexity of our algorithm. Suppose we are to check a formula  $\psi' \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. \psi$ , where the  $Q_i$  are quantifiers and  $x_i$  variables. We can obtain a bound on the running time of our proposed algorithm in terms of the number of quantifiers  $n$ . This is of practical relevance since many useful policies have few quantifiers. Clearly the complexity depends on the representation of constraints  $c : \Sigma_V^{P_h} \rightarrow \{\top, \perp\}$ . One efficient representation of constraints is using multiple-valued decision diagrams [48]. With this representation, constraints  $c$  can be efficiently stored in space  $O((|P_h| + 1)^n)$  and the logical operations can all be computed in linear time  $O((|P_h| + 1)^n)$ . Further a constraint in  $h$  can be extended to a constraint in  $h' = \mathbf{update}(h, e, p, i)$  in linear time  $O((|P_{h'}| + 1)^n)$ .

**Theorem 6.4 (Complexity Bound)** Let formula  $\psi \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n. \psi'$  where the  $Q_i$  are quantifiers,  $x_i$  variables all of type  $P$ , and  $\psi'$  is a quantifier-free formula from the quantified language with  $fv(\psi') \subseteq \{x_1, \dots, x_n\}$ . Let

$h \in \mathcal{C}_{\rho ES}^0$  and  $|P_h|$  be the number of parameter occurrences in history  $h$ . The constraint-based algorithm for dynamic model checking has the following complexity.

- $DMC.\mathbf{check}()$  is  $O(1)$ .
- $DMC.\mathbf{new}()$  is  $O(|\psi| \cdot (|P_h| + 1)^n)$ .
- $DMC.\mathbf{update}(e, p, i)$  when  $p \in P_h$  and  $K$  is the current number of active configurations in  $h$ , is  $O((K - i + 1) \cdot |\psi| \cdot (|P_h| + 1)^n)$
- $DMC.\mathbf{update}(e, p, i)$  when  $p \notin P_h$  and  $K$  is the current number of active configurations in  $h$ , is  $O((K - i + 1) \cdot |\psi| \cdot (|P_h| + 2)^n)$

Furthermore, the space requirement of  $DS'$  is  $O(K \cdot (|E| + |\psi| \cdot (|P_h| + 1)^n))$ .

### 6.5.2 References and Quantitative Properties

In this section, we briefly illustrate another way to extend the core policy-language to a more practical one. As mentioned, we consider two aspects: referencing and quantitative properties. For referencing we introduce a construct  $p : \psi$ , where  $p$  is a principal-identity and  $\psi$  is a basic policy. The construct is intended to mean that principal  $p$ 's observations (about a subject) must satisfy past-time  $\psi$ . For quantitative properties, we introduce a counting operator  $\overline{\#}$ , used e.g. in formula  $p : \overline{\#}\psi$  which counts the number of  $p$ -observed sessions satisfying  $\psi$  (we use  $\overline{\#}$  to avoid confusion with the conflict relation, often denoted by  $\#$ ).

To express referencing, we extend the basic syntax to include a new syntactic category  $\pi$  (for policy). Let  $\mathcal{P}$  be a collection of principal identities.

$$\pi ::= p : \psi \mid \pi_0 \wedge \pi_1 \mid \neg\pi \quad p \in \mathcal{P}$$

The policy  $p : \psi$  means that the observations that  $p$  has made should satisfy  $\psi$ . Note that in this extended language, models are no longer local interaction histories, but, instead, global interaction histories, represented as a principal-indexed collection of local histories (i.e., functions of type  $\mathcal{P} \rightarrow \mathcal{C}_{ES}^*$ ).

The quantitative extension is given by extending the category  $\psi$ . Let  $(\mathcal{R}_j)_{j=1}^\infty$  be a countable collection of  $k$ 'ary relation-symbols for each  $k \in \mathbb{N}$ , representing computable relations  $\llbracket \mathcal{R}_j \rrbracket \subseteq \mathbb{N}^k$ .

$$\psi ::= \dots \mid \mathcal{R}_j(\overline{\#}\psi_1, \overline{\#}\psi_2, \dots, \overline{\#}\psi_k)$$

The denotation of the construct  $\overline{\#}\psi$  is the number of sessions in the past which satisfy formula  $\psi$ , e.g.,  $\overline{\#}\mathbf{negative}$  counts the number of states in the past satisfying  $\mathbf{negative}$ . So the denotation of  $\overline{\#}\psi$  is a number, and the semantics of  $\mathcal{R}_j(\overline{\#}\psi_1, \overline{\#}\psi_2, \dots, \overline{\#}\psi_k)$  is true iff  $(n_1, n_2, \dots, n_k) \in \llbracket \mathcal{R}_j \rrbracket$ , where  $n_i$  is the denotation of  $\overline{\#}\psi_i$ . Finally, we extend also category  $\pi$ :

$$\pi ::= \dots \mid \mathcal{R}_j(p_1 : \overline{\#}\psi_1, \dots, p_k : \overline{\#}\psi_k) \quad p_i \in \mathcal{P}$$

The construct  $\mathcal{R}_j(p_1 : \overline{\#}\psi_1, \dots, p_k : \overline{\#}\psi_k)$  means that, letting  $n_i$  denote the number of sessions observed by principal  $p_i$  satisfying  $\psi_i$ , then the relation  $\llbracket \mathcal{R}_j \rrbracket$  on numbers must have  $(n_1, \dots, n_k) \in \llbracket \mathcal{R}_j \rrbracket$ .

We do not provide a formal semantics as the meaning of our constructs should be intuitively clear, and our purpose is simply to illustrate how the core language can be extended to encompass more realistic policies. To further illustrate the constructs, we consider a number of example policies. In the following examples,  $p, p_1, p_2, \dots, p_n \in \mathcal{P}$  are principal identities.

**Example 6.5 (eBay revisited)** Consider the eBay scenario again. The policy of Example 6.1 could be extended with referencing, e.g. principal  $p$  might use policy:

$$\pi_p^{\text{bid}} \equiv p : \mathbf{G}^{-1}(\mathbf{negative} \rightarrow \mathbf{ignore}) \wedge \bigwedge_{q \in \{p, p_1, \dots, p_n\}} q : \neg \mathbf{F}^{-1}(\mathbf{time-out})$$

Intuitively, this policy represents a requirement by principal  $p$ : “seller has never provided negative feedback about me, regarding auctions where I made payment, and, furthermore, seller has never cheated me or any of my friends.”

**Example 6.6 (P2P File-sharing)** This example is inspired by the example used in the license-based system of Shmatikov and Talcott [94]. Consider a scenario where a P2P file-server has two resources,  $\mathbf{dl}$  (download), and  $\mathbf{ul}$  (upload). Suppose this is modelled by an event structure with two independent events  $\mathbf{dl}$  and  $\mathbf{ul}$ , so that in each session, a peer-client either uploads, downloads or both. We express a policy used by server  $p$  for granting download, stating that “the number of uploads should be at least a third of the number of downloads.”

$$\pi_p^{\text{client-dl}} \equiv p : (\overline{\#}\mathbf{dl} \leq 3 \cdot \overline{\#}\mathbf{ul})$$

This refers only to the local history with  $p$ . Supposing we instead want to express a more “global” policy on the behaviour, stating that globally,  $p$  has uploaded at least a third of its downloads (e.g. locally this may be violated).

$$\pi_p^{\text{client-dl}} \equiv \frac{(p : \overline{\#}\mathbf{dl}) + (\sum_{i=1}^n p_i : \overline{\#}\mathbf{dl})}{3 \cdot (p : \overline{\#}\mathbf{ul} + (\sum_{i=1}^n p_i : \overline{\#}\mathbf{ul}))} \leq$$

**Example 6.7 (“Probabilistic” policy)** Consider an arbitrary event structure  $ES = (E, \leq, \#)$ . We express a policy ensuring that “statistically, event  $ev \in E$  occurs with frequency at least 75%.”

$$\pi_p^{\text{probab}} \equiv p : \frac{\overline{\#ev}}{\#ev + \overline{\# \sim ev} + 1} \geq \frac{3}{4}$$

Here  $\overline{\# \sim ev}$  counts the number of sessions in which  $ev$  has not occurred and cannot occur in the future.

### Implementation remarks.

Dynamic model checking for the extended policy language can be done by extending the array-based algorithm from the previous section. Note that the value of  $\overline{\# \psi}$  can easily be defined in the style of the recursive semantics. To handle the construct  $\mathcal{R}(\overline{\# \psi})$ , one maintains a number of integer variables which denote the values of sub-formula  $\overline{\# \psi}$  at each active session. The integers are then updated using the recursive semantics in a way similar to the array-updates in Section 6.4. We have the following result, assuming that the relations can be evaluated in constant time, and that numbers can be stored/manipulated in constant space/time.

**Theorem 6.5** Let formula  $\psi$  be from the basic language extended with the quantitative constructs. Let  $h \in \mathcal{C}_{ES}^0$  be a history. The dynamic model checking can be implemented with the following complexity.

- $DMC.\text{check}()$  is  $O(1)$ .
- $DMC.\text{new}()$  is  $O(|\psi|)$ .
- $DMC.\text{update}(e, i)$  is  $O((K-i+1) \cdot |\psi|)$  where  $K$  is the current number of active configurations in  $h$ .

Note, that the automata-based algorithm does not easily extend: the (semantics of the) extended language is no-longer regular, e.g. illustrated by formula  $\psi_p \equiv p : (\#d1 \leq \overline{\#u1})$ .

The construct  $p : \psi$ , where  $p$  is a principal identity, requires that  $p$ 's interaction history (with the subject in question) satisfies  $\psi$ . This is handled simply by “sending formula  $\psi$ ” to  $p$ . Principal  $p$  maintains the truth of  $\psi$  with respect to its interaction history using the algorithms of last section, and

sends the required value to the requesting principal when needed.<sup>5</sup> Another approach is for  $p$  to send its entire interaction history so that the verification can be performed locally, e.g., as is done with method `exportEvents` in the license-based framework of Shmatikov and Talcott [94]. It does not make sense to consider the algorithmic complexity of referencing. The message complexity of referencing, however, is linear in the number of principals to be contacted (one query and one reply).

## 6.6 A Java Security Manager

In this section we describe an application of our logical framework to the area of history-based access control for untrusted code. We have designed and implemented a prototype Java Security Manager which is able to monitor a Java program with respect to a “history-based” policy, written in our logic. If the security manager detects a violation of policy, a Java security exception is thrown and the violating action is aborted. We describe briefly the Java security model, and proceed with a more detail description of the design and implementation of our history-based security manager.

The Java Programming Language supports the concept of a *security manager*: an object that supervises another Java application with respect to security sensitive operations, e.g., file or network access.<sup>6</sup> Java programs that run other Java programs, e.g., a browser running a Java applet, can install a security manager that mediates the untrusted program’s security sensitive operations. Operations, like connecting to a socket on a remote site, are performed by Java applications via the Java API, e.g., class `Socket` of the `java.net` library provides an appropriate abstraction for “sockets.” API classes make calls to the security manager’s `checkPermission(java.lang.Permission)` method whenever a security sensitive operation is requested, e.g., the `Socket` class calls `checkPermssion` with an appropriate instance of the `java.net.SocketPermission` (containing information about which remote site and port is being accessed). The security manager then inspects the `java.lang.Permission` object (possibly consulting a

---

<sup>5</sup>One might argue that this leads to problems of timing: at what point in time is  $\psi$  then to be evaluated? But such timing-issues are inherent in distributed systems. Formula  $p : \psi$  is a relative temporal specification that is interpreted by the sender as referring to the current history of  $p$ , *when  $p$  decides to evaluate it*. The sender of  $\psi$  thus knows that received valuation (true or false) reflects an evaluation of  $\psi$  with respect to some *recent view* of  $p$ ’s history.

<sup>6</sup>More information about the Java security architecture, and security managers can be found at <http://java.sun.com/security/index.jsp>.

user-specified security policy) and throws a `java.lang.SecurityException` if access should not be granted.

The Java security architecture allows users to write their own security managers by extending the `java.lang.SecurityManager` class. We have written a security manager that decides access by checking conformance to policies from our history-based framework. The application is a simple prototype, used only for testing the validity of our approach to history-based access control, and consequently, the current version supports only a small subset of the security relevant operations available in Java.<sup>7</sup>

### 6.6.1 Design

We have designed two versions of the HBAC application, a basic and a parameterized version, corresponding to the basic and parameterized languages described previously. The basic events in our event structure correspond to the Java security events, e.g., `java.io.FilePermission` for representing file-access. For simplicity, the current version supports only the events corresponding to file and network access, corresponding to the Java classes `java.io.FilePermission` and `java.net.SocketPermission`, however it would be simple to extend this to all the security relevant events. The basic event structure thus consists of conflicting events `FilePermission(read)`, `FilePermission(write)`, `FilePermission(delete)`, `FilePermission(execute)`; and `SocketPermission(connect)`, `SocketPermission(listen)`, `SocketPermission(accept)`, `SocketPermission(resolve)`. Note that since there are only four types of operations for each event-type (e.g. ‘read’ for the ‘FilePermission’) these “finitely parameterized” events can be represented in the basic model. In the parameterized model, the parameterized events include also information about filenames/hostnames, e.g., event `SocketPermission(connect)` has further string-type parameters specifying a port and hostname, and `FilePermission(read)` has a parameter specifying the filename.

We have provided DSD2.0 [80, 50] descriptions of XML languages for both the basic and parameterized policies. A policy consists of a list of actions, e.g., `java.net.SocketPermission(connect)`, followed by a formula from one of the two logics. An example policy is provided in Figure 6.6.1; it describes the policy requiring that for the application to perform the actions of connecting a socket or accepting a socket connection, the history must

---

<sup>7</sup>The prototype source code is available as an open-source project, hosted at SourceForge, <https://sourceforge.net/projects/javahbac>.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<?dsd href="http://www.brics.dk/~krukow/dsd/quantifiedjavapolicies.dsd"?>

<policy xmlns:xi="http://www.w3.org/2001/XInclude"
        xmlns="http://www.brics.dk/~krukow/dsd/quantifiedjavapolicies">

  <actions>
    <java.net.SocketPermission host="*">
      connect
    </java.net.SocketPermission>
    <java.net.SocketPermission host="*">
      accept
    </java.net.SocketPermission>
  </actions>
  <behaviour>
    <always>
      <forall var="x">
        <implication>
          <premise>
            <event>
              <java.io.FilePermission path="x">
                read
              </java.io.FilePermission>
            </event>
          </premise>
          <conclusion>
            <sometime>
              <event>
                <java.io.FilePermission path="x">
                  write
                </java.io.FilePermission>
              </event>
            </sometime>
          </conclusion>
        </implication>
      </forall>
    </always>
  </behaviour>
</policy>

```

Figure 6.2: Example XML quantified HBAC policy.

satisfy the property

$$G^{-1}(\forall x : \text{String}.(\text{java.lang.FilePermission}(\text{read})(x) \rightarrow F^{-1}\text{java.lang.FilePermission}(\text{write})(x)))$$

We have implemented a SAX parser which reads a policy file from the disk and generates an internal data-structure representing the policy. This parser can be used by an application that wishes to install a security manager implementing a policy.

We have defined two security managers: an automata-based security manager (`SecMan.java`) for the basic language, using the efficient Java package `dk.brics.automaton` [79]; and an array-based security manager for the quantified language (`QSecMan.java`), using the JavaBDD binary decision diagram package for implementing constraints [101]. The input for both security managers is an XML representation of a policy, and both override the method `checkPermission` of the `SecurityManager` class to check whether a specific action is allowed. The basic security manager uses the automata-based algorithm from Section 6.4, whereas the quantified security manager uses the array-based algorithm from the same section, but extending the booleans to the constraints of Section 6.5.

We illustrate, by means of example, how one might use our security managers. Consider the following Java program which tries to read the file `"secret.txt"` and then send the contents to a location on the Internet.

```
import java.io.*;
import java.net.*;
public class Evil {
    public static void main(String[] args) throws Exception {
        System.out.println("begin");
        BufferedReader buf = new BufferedReader(new FileReader("secret.txt"));
        String line = null;
        StringBuffer sbuf = new StringBuffer();
        System.out.println("reading password");
        while ((line = buf.readLine())!=null) {
            System.out.println(line);
            sbuf.append(line);
        }
        System.out.println("opening connection");
        Socket s = new Socket("www.microsoft.com",80);
        Writer out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
        out.write(sbuf.toString(),0,sbuf.toString().length());

        System.out.println("done!");
    }
}
```

Suppose we want to run this program under the above example security policy. An application class for installing a security manager and running the program could be the following.

```
import java.util.*;
import java.security.Permission;

public class TestQSecMan {
    public static void main(String[] args) throws Exception {
        QPolicyParser pp = new QPolicyParser();
        pp.parse(args[0]);

        QSecMan sec = setupSecurityManager(pp);
        System.out.println("Setting Security Manager");
    }
}
```

```

        System.setSecurityManager(sec);
        System.out.println("Starting Program");
        Evil.main(null);
    }
    private static QSecMan setupSecurityManager(QPolicyParser pp) {
        // initialise security manager
        ...
    }
}

```

First, the policy is specified as an argument to the program. The program parses the policy using our SAX parser. Then it constructs a quantified security manager object, using the setup-method (the specifics are simple and not relevant here). Once the security manager is constructed, it uses the `System.setSecurityManager` Java method, to install the monitor. After this, the program can be run, and the security manager ensures that the policy is not violated.

Specifically, the output of running the above looks as follows.

```

[krukow@smeagol:...HBAC-Security/test]$ java TestQSecMan file-quant-example.xml
Setting Security Manager
...
Starting Program
...
check: (java.util.PropertyPermission user.dir read)
check: (java.io.FilePermission secret.txt read)
reading password
thesecretpassWord
opening connection
check: (java.util.PropertyPermission java.net.preferIPv6Addresses read)
check: (java.lang.RuntimePermission loadLibrary.net)
check: (java.io.FilePermission /home/java/Linux-jdk1.5.0_04/jre/lib/i386/libnet.so read)
check: (java.util.PropertyPermission java.net.preferIPv4Stack read)
check: (java.util.PropertyPermission impl.prefix read)
check: (java.lang.reflect.ReflectPermission suppressAccessChecks)
check: (java.util.PropertyPermission sun.net.spi.nameservice.provider.1 read)
check: (java.net.SocketPermission www.microsoft.com resolve)
Exception in thread "main" java.lang.SecurityException: Execution History Exception:
Neg(QSSince(QTrue, Neg(QForall x.(Neg(Conj(Event((java.io.FilePermission x read)(x)),
Neg(QSSince(QTrue, Event((java.io.FilePermission x write)(x))))))))))
    at QSecMan.checkPermission(QSecMan.java:37)
    at java.lang.SecurityManager.checkConnect(SecurityManager.java:1031)
    at java.net.InetAddress.getAllByName0(InetAddress.java:1117)
    at java.net.InetAddress.getAllByName0(InetAddress.java:1098)
    at java.net.InetAddress.getAllByName(InetAddress.java:1061)
    at java.net.InetAddress.getByName(InetAddress.java:958)
    at java.net.InetSocketAddress.<init>(InetSocketAddress.java:124)
    at java.net.Socket.<init>(Socket.java:178)
    at Evil.main(Evil.java:15)
    at TestQSecMan.main(TestQSecMan.java:16)

```

We see that a security exception is thrown, not when the programs accesses the password, but when it tries to open a socket connection. We see also

that there are a number of additional operations that are necessary for opening sockets, e.g., (`java.io.FilePermission,/home/java/Linux-jdk1.5.0_04/jre/lib/i386/libnet.so,read`).

We have not yet done further experimentation with the framework, but our initial impression is good. Finally, we would like to compare our proposed framework to the similar system *Deeds*, of Edjlali et al. [33]. The *Deeds* system is similar to our prototype system in that *Deeds* also seeks to do history-based access control for Java (in fact, *Deeds* was the main source of inspiration for this application). First, *Deeds* is more general than our system because “the set of security events is not fixed.” In our system, the set of security-relevant events is restricted to what Java considers security events (this may change with future releases of Java). Secondly, *Deeds* is more *low-level* than our system: in *Deeds*, the programmer explicitly must maintain the event history (performing optimizations as he sees fit), and the programmer explicitly programs the security monitor (using full Java). This has the advantage that it is more flexible, but the disadvantage that such programming is error-prone, and highly security sensitive. In contrast, specifying an XML policy which is automatically monitored is less error-prone as the policy is declarative, and domain-specific. Furthermore, we’re using standard algorithms that can efficiently handle all policies in the XML language, and which performs optimizations automatically, e.g., event history maintenance (and deallocation) and automata minimization. Finally, *Deeds* is much more fully developed while our approach is still at the prototype and evaluation level. We encourage interested readers to download the source code at <https://sourceforge.net/projects/javahbac>, and develop it further.

## 6.7 Related Work

Many reputation-based systems have been proposed in the literature (Jøsang *et al.* [47] provide many references), so we choose to mention only a few typical examples and closely related systems. Kamvar *et al.* present EigenTrust [49], Shmatikov and Talcott propose a license-based framework [94], and the EU project ‘SECURE’ [16, 17] (which also uses event structures for modelling observations) can be viewed as a reputation-based system, to name a notable few.

The framework of Shmatikov and Talcott is the most closely related in that they deploy also a very concrete representation of behavioural information (“evidence” [94]). This representation is not as sophisticated as in the

event-structure framework (e.g., as histories are sets of time-stamped events there is no concept of a session, i.e., a logically connected set of events), and their notion of reputation is based on an entity's past ability to fulfil so-called licenses. A license is a contract between an issuer and a licensee. Licenses are more general than interaction policies since they are *mutual* contracts between issuer and licensee, which may *permit* the licensee to perform certain actions, but may also *require* that certain actions are performed. The framework does not have a domain-specific language for specifying licenses (i.e. for specifying license-methods **permits** and **violated**), and the *use* of reputation information is not part of their formal framework (i.e. it is up to each application programmer to write method **useOk** for protecting a resource). We do not see our framework as competing, but, rather, *compatible* with theirs. We imagine using a policy language, like ours, as a domain-specific language for specifying licenses as well as use-policies. We believe that because of the simplicity of our declarative policy language and its formal semantics, this would facilitate verification and other reasoning about instances of their framework.

Pucella and Weissman use a variant of pure-future linear temporal logic for reasoning about licenses [87]. They are not interested in the specific details of licenses, but merely require that licenses can be given a trace-based semantics; in particular, their logic is illustrated for licenses that are regular languages. As our basic policies can be seen (semantically) as regular languages (Theorem 6.2), and policies can be seen as a type of license, one could imagine using their logic to reason about our policies.

Roger and Goubault-Larreq [91] have used linear temporal logic and associated model-checking algorithms for log auditing. The work is related although their application is quite different. While their logic is first-order in the sense of having variables, they have no explicit quantification. Our quantified language differs (besides being pure-past instead of pure-future) in that we allow explicit quantification (over different parameter types)  $\forall x : P_i.\psi$  and  $\exists x : P_i.\psi$ , while their language is implicitly universally quantified.

The notion of security automata, introduced by Schneider [93], is related to our policy language. A security automaton runs in parallel with a program, monitoring its execution with respect to a security policy. If the automata detects that the program is about to violate the policy, it terminates the program. A policy is given in terms of an automata, and a (non-declarative) domain-specific language for defining security automata (SAL) is supported but has been found awkward for policy specification [35]. One can view the finite automaton in our automata-based algorithm as a kind of security automaton, *declaratively* specified by a temporal-logic

formula.

Security automata are also related, in a technical sense [37], to the notion of history-based access control (HBAC). HBAC has been the subject of a considerable amount of research (e.g., papers [96, 5, 33, 37, 93, 38]). There is a distinction between *dynamic* HBAC in which programs are monitored as they execute, and terminated if about to violate policy [37, 33, 93, 38]; and *static* HBAC in which some preliminary static analysis of the program (written in a predetermined language) extracts a safe approximation of the programs' runtime behaviour, and then (statically) checks that this approximation will always conform to policy (using, e.g., type systems or model checking) [96, 5]. Clearly, our approach has applications to dynamic HBAC. It is noteworthy to mention that many ad-hoc optimisations in dynamic HBAC (e.g., history summaries relative to a policy in the system of Edjlali [33]) are captured in a *general* and optimal way by using the automata-based algorithm, and exploiting the finite-automata minimisation-theorem. Thus in the automata based algorithm, one gets “for free,” optimisations that would otherwise have to be discovered manually.



# Bibliography

- [1] A. Abdul-Rahman. *A Framework for Decentralised Trust Reasoning*. PhD thesis, University of London, Department of Computer Science, University College London, England, 2005.
- [2] A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, page 9. IEEE, 2000.
- [3] S. Abramsky, P. Buneman, P. Gardner, A. Gordon, M. Kwiatkowska, R. Milner, V. Sassone, and S. Stepney. Science for global ubiquitous computing – a fifteen-year grand challenge for computing research. Available from: [http://www.nesc.ac.uk/esi/events/Grand\\_Challenges/proposals/Ubiq.pdf](http://www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/Ubiq.pdf). Draft. Unpublished manuscript., 2003.
- [4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM conference on Computer and communications security (CCS'99)*, pages 52–62, New York, NY, USA, 1999. ACM Press.
- [5] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, pages 316–332. Springer, 2005.
- [6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall International Editions. Prentice-Hall, Inc., 1989.

- [7] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust management system, version 2. RFC-2704, ftp-site: <ftp://ftp.rfc-editor.org/in-notes/rfc2704.txt>, 1999.
- [8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. 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.
- [9] M. Blaze, J. Feigenbaum, and A. 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.
- [10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings from the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.
- [11] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Proceedings from Financial Cryptography: Second International Conference (FC'98), Anguilla, British West Indies, February 1998*, pages 254–274, 1998.
- [12] M. Blaze, J. Ioannidis, and A. D. Keromytis. Trust management for ipsec. In *Network and Distributed System Security Symposium (NDSS'01) Conference Proceedings*, 2001. <http://www.isoc.org/isoc/conferences/ndss/01/>.
- [13] M. Blaze, J. Ioannidis, and A. D. Keromytis. Offline micropayments without trusted hardware. In P. F. Syverson, editor, *Financial Cryptography, 5th International Conference (FC'01), Grand Cayman, British West Indies, February 19-22, 2002, Proceedings*, volume 2339 of *Lecture Notes in Computer Science*. Springer, 2002.
- [14] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings from the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, 1989.
- [15] S. Buchegger and J.-Y. Le Boudec. A Robust Reputation System for Peer-to-Peer and Mobile Ad-hoc Networks. In *P2PEcon 2004*, 2004.

- [16] V. Cahill and E. Gray *et al.* Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.
- [17] V. Cahill and J.-M. Seigneur. The SECURE website. <http://secure.dsg.cs.tcd.ie>, 2004.
- [18] M. Carbone. *Trust and Mobility*. PhD thesis, Department of Computer Science, BRICS, University of Aarhus, 2005.
- [19] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *Proceedings from Software Engineering and Formal Methods (SEFM'03)*. IEEE Computer Society Press, 2003.
- [20] M. Carbone, M. Nielsen, and V. Sassone. A calculus for trust management. In *Proceedings from Foundations of Software Technology and Theoretical Computer Science: 24th International Conference (FSTTCS'04)*, pages 161–173. Springer, December 2004.
- [21] D. Chalmers, M. Chalmers, J. Crowcroft, M. Kwiatkowska, R. Milner, E. O'Neill, T. Rodden, V. Sassone, and M. Sloman. Ubiquitous computing: Experience, design and science. Version 4. Available from: <http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/Manifesto/manifesto.pdf>. Draft. Unpublished manuscript. Website <http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/index.html>, 2006.
- [22] A. Chander, D. Dean, and J. Mitchell. Reconstructing trust management. *Journal of Computer Security*, 12(1):131–164, 2004.
- [23] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.
- [24] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285, 2001.
- [25] M. Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. den Hartog. Nonmonotonic trust management for p2p applications. In *1st Int. Workshop on Security and Trust Management (STM)*, pages 101–116. Elsevier Science, 2005.

- [26] R. K. Dash, S. D. Ramchurn, and N. R. Jennings. Trust-based mechanism design. In *Proceedings from the 3rd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, pages 748–755. ACM Press, 2004.
- [27] C. Dellarocas. The digitization of word of mouth: Promise and challenges of online feedback mechanisms. *Management Science*, 49(10):1407–1424, October 2003.
- [28] C. Dellarocas. Sanctioning reputation mechanisms in online trading environments with moral hazard. Working paper. Available online: <http://ccs.mit.edu/dell>, July 2004.
- [29] Z. Despotovic and K. Aberer. A probabilistic approach to predict peers' performance in P2P networks. In *Proceedings from the Eighth International Workshop on Cooperative Information Agents (CIA 2004)*, volume 3191 of *Springer Lecture Notes in Computer Science*, pages 62–76. Springer, 2004.
- [30] Z. Despotovic and K. Aberer. P2P reputation management: Probabilistic estimation vs. social networks. *Computer Networks*, 60(4):485–500, Mar. 2006.
- [31] J. DeTreville. Binder, a logic-based security language. In *Proceedings from the 2002 IEEE Symposium on Security and Privacy (S&P 2002)*, pages 105–113. IEEE Computer Society Press, 2002.
- [32] eBay Inc. The eBay website. <http://www.ebay.com>.
- [33] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings from the 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 38–48. ACM Press, 1998.
- [34] 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.
- [35] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings from the 2000 DARPA Information Survivability Conference and Exposition*, pages 1287–1295. IEEE Computer Society Press, 2000.

- [36] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proceedings from the 6th International Workshop on Discrete Algorithms and methods for mobile computing and communications (Dial-M'02)*, pages 1–13. ACM Press, 2002.
- [37] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings from the 2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society Press, 2004.
- [38] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
- [39] S. J. Garland and N. A. Lynch. Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, NY, 2000.
- [40] S. J. Garland, N. A. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical Report MIT-LCS-TR-961, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), Cambridge, MA, Dec. 2004.
- [41] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys & Tutorials*, 3(4), 2000.
- [42] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.
- [43] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, The Edinburgh Building, Cambridge, CB2 2RU, United Kingdom, 2003.
- [44] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings from the 2001 IEEE Symposium on Security and Privacy, Oakland, California*, pages 106–115. IEEE Computer Society Press, 2001.
- [45] A. Jøsang. A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, 2001.

- [46] A. Jøsang and R. Ismail. The beta reputation system. In *Proceedings from the 15th Bled Conference on Electronic Commerce, Bled*, 2002.
- [47] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, (to appear, preprint available online: <http://sky.fit.qut.edu.au/~josang/>), 2006.
- [48] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4(1-2):9–62, 1998.
- [49] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM Press.
- [50] N. Klarlund, A. Møller, and M. I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3):285–319, August 2002.
- [51] P. Kollock. The production of trust in online markets. *Advances in Group Processes*, 16, 1999.
- [52] D. Kreps, R. Milgrom, J. Roberts, and R. Wilson. Reputation and imperfect information. *Journal of Economic Theory*, 27(2):253–279, August 1982.
- [53] K. Krukow. An operational semantics for trust policies. Presented at Workshop on Issues in the Theory of Security, 2006 (WITS'06). No published proceedings yet.
- [54] K. Krukow. On foundations for dynamic trust management. Unpublished PhD Progress Report, available online: <http://www.brics.dk/~krukow>, 2004.
- [55] K. Krukow and M. Nielsen. Distributed trust management: Denotational and operational semantics. Submitted. Available online <http://www.brics.dk/~krukow>, 2006.
- [56] K. Krukow, M. Nielsen, and V. Sassone. A formal framework for concrete reputation-systems with applications to history-based access control. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 260–269, New York, NY, USA, 2005. ACM Press.

- [57] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems. Technical Report RS-05-23, BRICS, University of Aarhus, July 2005.
- [58] K. Krukow, M. Nielsen, and V. Sassone. A logical framework for reputation systems. Submitted. Available online [www.brics.dk/~krukow](http://www.brics.dk/~krukow), 2006.
- [59] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. In *Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 805–814. IEEE, 2005.
- [60] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. Technical Report RS-05-6, BRICS, University of Aarhus, Feb. 2005. Available online: <http://www.brics.dk/RS/05/6>.
- [61] K. Krukow and A. Twigg. The complexity of fixed point models of trust in distributed networks. To be published in TCS – Special issue on Semantic and Logical Foundations of Global Computing, Elsevier, 2006.
- [62] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, March 1951.
- [63] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings from the 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392. IEEE Computer Society Press, 2002.
- [64] N. Li, B. Grosf, and J. Feigenbaum. A logic-based knowledge representation for authorization with delegation. In *Proceedings of the 9th Computer Security Foundations Workshop (CSFW'99)*, pages 162–174. IEEE Computer Society, 1999.
- [65] N. Li, B. Grosf, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*, pages 27–42, Washington, DC, USA, 2000. IEEE Computer Society.
- [66] N. Li, B. N. Grosf, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

- [67] N. Li and J. Mitchell. Datalog with constraints: A foundation for trust-management languages. In *Proceedings from the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, volume 2562 of *Springer Lecture Notes in Computer Science*, pages 58–73. Springer, 2003.
- [68] N. Li and J. 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).
- [69] N. Li and J. 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.
- [70] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proceedings from the 2002 IEEE Symposium on Security and Privacy (S&P 2002)*, pages 114–130. IEEE Computer Society Press, 2002.
- [71] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005.
- [72] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- [73] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–151. ACM Press, 1987.
- [74] S. P. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, Department of Computer Science and Mathematics, University of Stirling, 1994.
- [75] R. Milner, A. Gordon, V. Sassone, P. Buneman, P. Gardner, S. Abramsky, and M. Kwiatkowska. Theories for ubiquitous processes and data. Unpublished manuscript. Available from the homepage of Robin Milner: <http://www.cl.cam.ac.uk/users/rm135/plat.pdf>.

- [76] R. Milner and T. Hoare. Grand Challenges in Computing — Research. Available from: <http://www.ukcrc.org.uk/gcresearch.pdf>, also at <http://www.bcs.org/server.php?show=conWebDoc.1510>. Website: [http://www.ukcrc.org.uk/grand\\_challenges/index.cfm](http://www.ukcrc.org.uk/grand_challenges/index.cfm). Published by The British Computer Society (BCS), 1 Sanford Street, Swindon Wiltshire, SN1 1HJ, UK, [www.bcs.org](http://www.bcs.org), 2004.
- [77] L. Mui, M. Mohtashemi, and A. Halberstadt. A computational model of trust and reputation (for ebusinesses). In *Proceedings from 5th Annual Hawaii International Conference on System Sciences (HICSS'02)*, page 188. IEEE, 2002.
- [78] L. Mui, M. Mohtashemi, and A. Halberstadt. Notions of reputation in multi-agents systems: a review. In *Proceedings from The First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS'02)*, pages 280–287. ACM Press, 2002.
- [79] A. Møller. The `dk.brics.automaton` project website. <http://www.brics.dk/automaton/>, 2005.
- [80] A. Møller. The DSD2.0 project website. <http://www.brics.dk/DSD/>, 2005.
- [81] M. Nielsen and K. 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.
- [82] M. Nielsen and K. 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 on the Occasion of His 70th Birthday*, volume 3113 of *Lecture Notes in Computer Science*, pages 192–204. Springer Verlag, 2004.
- [83] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13:85–108, 1981.
- [84] C. H. Papadimitriou. Algorithms, games and the internet. In *Proceedings of the thirty-third annual ACM Symposium on Theory of Computing (STOC'01)*, pages 749–753. ACM Press, 2001.

- [85] PGPi website. An introduction to cryptography, in pgp user's guide 7.0. ftp: <ftp://ftp.pgpi.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf> website: <http://www.pgpi.org/doc>, 2000.
- [86] A. Pnueli. The temporal logic of programs. In *Proceedings from the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, New York, 1977.
- [87] R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *Proceedings from 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.
- [88] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.
- [89] S. D. Ramchurn, D. Huynh, and N. R. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(1):1–25, Mar. 2004.
- [90] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. Reputation systems. *Communications of the ACM*, 43(12):45–48, Dec. 2000.
- [91] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *Proceedings from the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Computer Society Press, 2001.
- [92] J. Sabater and C. Sierra. Review on computational trust and reputation models. *Artificial Intelligence Review*, 24(1):33–60, 2005.
- [93] F. B. Schneider. Enforceable security policies. *Journal of the ACM*, 3(1):30–50, 2000.
- [94] V. Shmatikov and C. Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005.
- [95] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
- [96] C. Skalka and S. Smith. History effects and verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, pages 107–128. Springer, 2005.

- [97] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the fifth annual ACM symposium on Theory of computing (STOC'73)*, pages 1–9, New York, NY, USA, 1973. ACM Press.
- [98] W. T. L. Teacy, J. Patel, N. R. Jennings, and M. Luck. Coping with inaccurate reputation sources: experimental analysis of a probabilistic trust model. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 997–1004, New York, NY, USA, 2005. ACM Press.
- [99] D. Varacca, H. Völzer, and G. Winskel. Probabilistic event structures and domains. In P. Gardner and N. Yoshida, editors, *Proceedings from 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 2004.
- [100] S. Weeks. Understanding trust management systems. In *Proceedings from the 2001 IEEE Symposium on Security and Privacy*, pages 94–106. IEEE Computer Society Press, 2001.
- [101] J. Whaley. The javabdd project website. <http://javabdd.sourceforge.net/>, 2005.
- [102] R. Wilson. Reputations in games and markets. In A. Roth, editor, *Game-Theoretic Models of Bargaining*, pages 27–62. Cambridge University Press, 1985.
- [103] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings from ICALP 1982*, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer Verlag, 1982.
- [104] G. Winskel. *Formal Semantics of Programming Languages : an introduction*. Foundations of computing. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.
- [105] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995.
- [106] L. Xiong and L. Liu. PeerTrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on knowledge and data engineering*, 16(7):843–857, 2004.

- [107] B. Yu and M. P. Singh. An evidential model of distributed reputation management. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 294–301, New York, NY, USA, 2002. ACM Press.