Catuscia Palamidessi
Mark D. Ryan (Eds.)

LNCS 8191

Trustworthy Global Computing

7th International Symposium, TGC 2012
Newcastle upon Tyne, UK, September 2012
Revised Selected Papers

Lecture Notes in Computer Science 8191

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
TU Dortmund University, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Catuscia Palamidessi Mark D. Ryan (Eds.)

Trustworthy Global Computing

7th International Symposium, TGC 2012
Newcastle upon Tyne, UK, September 7-8, 2012
Revised Selected Papers

Springer

Volume Editors

Catuscia Palamidessi
Inria, Campus de l'École Polytechnique, Paris, France
E-mail: catuscia@lix.polytechnique.fr

Mark D. Ryan
University of Birmingham, UK
E-mail: m.d.ryan@cs.bham.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-41156-4 e-ISBN 978-3-642-41157-1
DOI 10.1007/978-3-642-41157-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013948498

CR Subject Classification (1998): K.6.5, D.4.6, C.2, F.4, E.3, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in ist current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of TGC 2012, the 7th International Symposium on Trustworthy Global Computing held during September 7–8, 2012, in Newcastle Upon Tyne, UK, co-located with CONCUR and PATMOS.

TGC is an international annual venue dedicated to safe and reliable computation in the so-called global computers, i.e., those computational abstractions emerging in large-scale infrastructures such as service-oriented architectures, autonomic systems and cloud computing.

The related models of computation incorporate code and data mobility over distributed networks that connect heterogeneous devices and have dynamically changing topologies.

The TGC series focuses on providing frameworks, tools, algorithms and protocols for designing open-ended, large-scale applications and for reasoning about their behaviour and properties in a rigorous way.

The first TGC event took place in Edinburgh during April 7–9, 2005, with the co-sponsorship of IFIP TC-2, as part of ETAPS 2005. TGC 2005 was the evolution of the previous Global Computing I workshops held in Rovereto in 2003 and 2004 (see LNCS vol. 2874) and the workshops on Foundation of Global Computing held as satellite events of ICALP and CONCUR (see ENTCS vol. 85). The last four editions of TGC were co-located with the reviews of the EU-funded projects AEOLUS, MOBIUS and SENSORIA within the FP6 initiative. They were held in Lucca, Italy (TGC 2006); in Sophia-Antipolis, France (TGC 2007); in Barcelona, Spain (TGC 2008); in Munich, Germany (TGC 2010); and in Aachen, Germany (TGC 2011); see, respectively, LNCS vol. 4661, LNCS vol. 4912, LNCS vol. 5474, LNCS vol. 6084, LNCS vol. 7173.

The main themes investigated by the TGC community are concerned with theories, languages, models and algorithms for global computing; abstraction mechanisms, models of interaction and dynamic components management; trust, access control and security enforcement mechanisms; privacy, reliability and business integrity; resource usage and information flow policies; contract-oriented software development; game-theoretic approaches to collaborative and competitive behaviour; self configuration, adaptation, and dynamic components management; software principles and tools to support debugging and verification; model checkers, theorem provers and static analyzers.

TGC 2012 received 14 submissions out of which the Program Committee selected nine regular papers to be included in this volume and be presented at the symposium. To guarantee the fairness and quality of the selection, each paper received at least three reviews. Additionally, the program included four invited speakers:

- Dan Ghica (University of Birmingham, UK)
- Cosimo Laneve (Università di Bologna, Italy)

– David Naccache (ENS Cachan, France)
– Stefan Saroiu (Microsoft Research, USA)

All the invited speakers were invited to contribute to the proceedings with a paper related to their talk, and Dan Ghica, Cosimo Laneve, and David Naccache accepted. In addition, this volume contains a paper by Michele Bugliesi, who was an invited speaker at TGC 2011, and whose paper, for logistical reasons, could not appear in the proceedings.

We thank the Steering Committee of TGC for inviting us to chair the conference and all PC members and external referees for their detailed reports and the stimulating discussions that emerged in the review phase. We thank all the authors of submitted papers and all the invited speakers for their interest in TGC. We want to thank the providers of the EasyChair system, which was used to manage the submissions, for reviewing (including the electronic PC meeting), and to assemble the proceedings. We thank Joshua Phillips for administrating the website, Maciej Koutny, Irek Ulidowski and the local organization of CONCUR and PATMOS for their help.

September 2012

Catuscia Palamidessi
Mark D. Ryan

Organization

Program Committee

Myrto Arapinis	University of Birmingham, UK
Roberto Bruni	Università di Pisa, Italy
Kostas Chatzikokolakis	Ecole Polytechnique of Paris, France
Thomas Jensen	Inria, France
Steve Kremer	Inria Nancy - Grand Est, France
Boris Köpf	IMDEA Software Institute, Spain
Zhiming Liu	United Nations University - International Institute for Software Technology, Macao
Alberto Lluch Lafuente	IMT Institute for Advanced Studies Lucca, Italy
Catuscia Palamidessi	Inria and LIX, Ecole Polytechnique
Dusko Pavlovic	Royal Holloway, UK
Alfredo Pironti	Inria, France
Mark D. Ryan	University of Birmingham, UK
Vladimiro Sassone	University of Southampton, UK
Ben Smyth	Toshiba
Martin Wirsing	Ludwig-Maximilians-Universität München, Germany
Nobuko Yoshida	Imperial College London, UK

Additional Reviewers

Bielova, Nataliia
Fossati, Luca
Hamadou, Sardaouna
Mancini, Loretta
Montesi, Fabrizio
Qamar, Nafees
Sebastio, Stefano
Tiezzi, Francesco
Wang, Hao
Zunino, Roberto

Table of Contents

From Rational Number Reconstruction to Set Reconciliation and File Synchronization

Antoine Amarilli, Fabrice Ben Hamouda, Florian Bourse,
Robin Morisset, David Naccache, and Pablo Rauzy

École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France
`firstname.lastname@ens.fr` (except for `fabrice.ben.hamouda@ens.fr`)

Abstract. This work revisits *set reconciliation*, the problem of synchronizing two multisets of fixed-size values while minimizing transmission complexity. We propose a new number-theoretic reconciliation protocol called *Divide and Factor* (D&F) that achieves optimal asymptotic transmission complexity – as do previously known alternative algorithms. We analyze the computational complexities of various D&F variants, study the problem of synchronizing sets of variable-size files using hash functions and apply D&F to synchronize file hierarchies taking file locations into account.

We describe `btrsync`, our open-source D&F implementation, and benchmark it against the popular software `rsync`. It appears that `btrsync` transmits much less data than `rsync`, at the expense of a relatively modest computational overhead.

1 Introduction

File synchronization is the important practical problem of retrieving a file hierarchy from a remote host given an outdated version of the retrieved files. In many cases, the bottleneck is network bandwidth. Hence, transmission must be optimized using the information given by the outdated files to the fullest possible extent. Popular file synchronization programs such as `rsync` use rolling checksums to skip remote file parts matching local file parts; however, such programs are usually unable to use the outdated files in more subtle ways, *e.g.*, detect that information is already present on the local machine but at a different location or under a different name.

File synchronization is closely linked to the theoretical *Set Reconciliation Problem*: given two sets of fixed-size data items on different machines, determine the sets' symmetric difference while minimizing transmission complexity. The size of the symmetric difference (*i.e.*, the difference's cardinality times the elements' size) is a clear information-theoretic lower bound on the quantity of information to transfer, and several known algorithms already achieve this bound [11].

This paper considers set reconciliation and file synchronization from both a theoretical and practical perspective:

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 1–18, 2013.
© Springer-Verlag Berlin Heidelberg 2013

- Section 2 introduces *Divide and Factor* (D&F), a new number-theoretic set reconciliation algorithm. D&F represents the items to synchronize as prime numbers, accumulates information during a series of rounds and computes the sets' difference using Chinese remaindering and rational number reconstruction.
- Section 3 shows that D&F's transmission complexity is linear in the size of the symmetric difference of the multisets to reconcile.
- Section 4 extends D&F to perform *file reconciliation*, *i.e.*, reconcile sets of variable-size files. We show how to choose the hash functions to optimally trade transmission for success probability. Several elements in this analysis are generic and apply to all set reconciliation algorithms.
- Section 5 studies D&F's time complexity and presents constant-factor tradeoffs between transmission and computation.
- Section 6 compares D&F with existing set reconciliation algorithms.
- Section 7 extends D&F to *file synchronization*, taking into account file locations and dealing intelligently (*i.e.*, in-place) with file moves. We describe an algorithm applying a sequence of file moves while avoiding the excessive use of temporary files.
- Section 8 presents `btrsync`, our D&F implementation, and benchmarks it against `rsync`. Experiments reveal that `btrsync` requires more computation than `rsync` but transmits less data in most cases.

2 Divide and Factor Set Reconciliation

This section introduces *Divide and Factor*. After introducing the problem and notations, we present a *basic* D&F version assuming that the number of differences between the multisets to reconcile is bounded by some constant t known to the parties. We then extend this basic protocol to a *complete* algorithm dealing with any number of differences.

2.1 Problem Definition and Notations

Oscar possesses an old version of a multiset $\mathcal{H} = \{h_1, \ldots, h_n\}$ that he wishes to update. Neil has a newer, up-to-date multiset $\mathcal{H}' = \{h'_1, \ldots, h'_{n'}\}$. The h_i, h'_i are u-bit primes. Note that Neil does not need to learn $\mathcal{H}$[1].

Let $\mathcal{D} = \mathcal{H} \setminus \mathcal{H}'$ be the multiset of values owned by Neil but not by Oscar, with adequate multiplicities. Likewise, let $\mathcal{D}' = \mathcal{H}' \setminus \mathcal{H}$ be the values owned by Oscar and not by Neil.

Let $T = \#\mathcal{D} + \#\mathcal{D}' = \#(\mathcal{D} \Delta \mathcal{D}')$ be the number of differences between $\mathcal{H}$ and $\mathcal{H}'$, where $\mathcal{D} \Delta \mathcal{D} = (\mathcal{D} \setminus \mathcal{D}') \cup (\mathcal{D}' \setminus \mathcal{D})$.

[1] The protocol can be easily transformed to do so without changing asymptotic transmission complexities.

Oscar		**Neil**
$c \leftarrow \prod_{i=1}^{n} h_i \bmod p$	$\xrightarrow{\quad c \quad}$	$c' \leftarrow \prod_{i=1}^{n'} h'_i \bmod p$ $s \leftarrow c'/c \bmod p$ reconstruct a, b from s
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i = 0\}$	$\xleftarrow{\quad \mathcal{D}', b \quad}$	$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$

Fig. 1. Basic D&F protocol (assuming that $T \leq t$)

2.2 Basic Protocol with Bounded T

Assume that $T \leq t$ for some fixed t known by Neil and Oscar. The initial phases of the protocol are as follows:

- Generate a prime p such that $2^{2ut} \leq p < 2^{2ut+1}$.
- Oscar computes the *redundancy* $c = \prod_{i=1}^{n} h_i \bmod p$ and sends it to Neil.
- Neil computes $c' = \prod_{i=1}^{n} \bmod p$ and $s = \frac{c'}{c} \bmod p$.

Because $T \leq t$, $\mathcal{H}$ and $\mathcal{H}'$ differ by at most t elements and s can be written as follows:

$$s = \frac{a}{b} \bmod p \text{ where } \begin{cases} a = \prod_{h'_i \in \mathcal{H}' \setminus \mathcal{H}} h'_i \leq 2^{ut} - 1 \\ b = \prod_{h_i \in \mathcal{H} \setminus \mathcal{H}'} h_i \leq 2^{ut} - 1 \end{cases}.$$

The problem of efficiently recovering a and b from s is called *Rational Number Reconstruction* (RNR) [12,16]. The following theorem (*cf.* Theorem 1 of [8]) guarantees that RNR can be solved efficiently in the present setting:

Theorem 1. *Let a, $b \in \mathbb{Z}$ be two co-prime integers such that $0 \leq a \leq A$ and $0 < b \leq B$. Let $p > 2AB$ be a prime and $s = ab^{-1} \bmod p$. Then a and b are uniquely defined given s and p, and can be recovered from A, B, s, and p in polynomial time.*

Taking $A = B = 2^{ut} - 1$, we have $AB < p$, since $2^{2ut} \leq p < 2^{2ut+1}$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Thus Oscar can recover a and b from s in polynomial time *e.g.*, using Gauß's algorithm for finding the shortest vector in a bi-dimensional lattice [15].

Oscar and Neil can then test, respectively, the divisibility of a and b by elements of the sets $\mathcal{H}$ and $\mathcal{H}'$ to identify the differences between $\mathcal{H}$ and $\mathcal{H}'$ and settle them[2]. This basic protocol is depicted in Figure 1.

2.3 Full Protocol with Unbounded T

In practice, we cannot assume that we have an upper bound t on the number of differences T. This section extends the protocol to any T. We do this in two

[2] Actually, this only works if $\mathcal{H}$ and $\mathcal{H}'$ are sets. In the case of multisets, if the multiplicity of h'_i in $\mathcal{H}'$ is j', then we would need to check the divisibility of b by $h_i, h_i^2, \ldots, h_i^{j'}$. For the sake of clarity we will assume that $\mathcal{H}$ and $\mathcal{H}'$ are sets. Adaptation to the general case is straightforward.

steps. We first show that we can slightly change the protocol to detect whether a choice of t was large enough for a successful reconciliation (which is guaranteed to be true if t was $\geq T$). We then construct a protocol that works with any T.

Detecting Reconciliation Failures. If $t < T$, with high probability, a will not factor completely over the set of primes $\mathcal{H}'$[3]. We will (improperly) consider that in such a case a is a random (tu)-bit number. For each i, the probability that h'_i divides a is at most $1/2^u$. The probability that $\prod h'_i \mod a = 0$ is roughly the probability that exactly t h'_i's amongst n' divide a, *i.e.*, $\binom{n'}{t} 2^{-ut}(1-2^{-u})^{n'-t} \leq n2^{-ut}$ which is very small if $2^u \gg n'$, a condition that we assume hereafter.

Thus, Neil can check very quickly that $\prod h'_i \mod a = 0$ without sending any data to Oscar. We call $\perp_1$ the event where this test failed (which implies that reconciliation failed), and $\perp_2$ the event where this test succeeds but reconciliation failed (which is very unlikely according to the previous discussion).

To handle $\perp_2$, we will use a collision-resistant hash function `Hash`, such as SHA: Before any exchanges take place, Neil will send to Oscar $H = \texttt{Hash}(\mathcal{H}')$. After computing $\mathcal{D}$, Oscar will compute a candidate $\mathcal{H}'$ from $\{\mathcal{H}, \mathcal{D}', \mathcal{D}\}$ and check that this candidate $\mathcal{H}'$ hashes into H. As `Hash` is collision-resistant, we can detect event $\perp_2$ in this fashion.

Complete D&F Protocol. To extend the protocol to an arbitrary T, assume that Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as $\perp_1$ or $\perp_2$ occurs, Neil and Oscar will repeat the protocol with a new p_ℓ to learn more information on $\mathcal{H}'$. Oscar will keep accumulating information about the difference between $\mathcal{H}$ and $\mathcal{H}'$ during these protocol runs (called *rounds*).

Formally, assume that $2^{2ut_k} \leq p_k < 2^{2ut_k+1}$. Let $P_k = \prod_{i=1}^{k} p_i$ and $T_k = \sum_{i=1}^{k} t_i$. After receiving the redundancies $c_1, \ldots, c_k$ corresponding to $p_1, \ldots, p_k$, Neil has as much information as if Oscar would have transmitted a redundancy C_k modulo P_k. Oscar can indeed compute $S_k = C'_k/C_k$ from $s_k = c'_k/c_k$ and S_{k-1} using the Chinese Remainder Theorem (CRT):

$$\begin{aligned} S_k &= \mathrm{CRT}(S_{k-1}, P_{k-1}, s_k, p_k) \\ &= S_{k-1}(p_k^{-1} \bmod P_{k-1})p_k + s_k(P_{k-1}^{-1} \bmod p_k)P_{k-1} \bmod P_k. \end{aligned}$$

The full protocol is given in Figure 2 page 5. Note that no information is lost and that the transmitted modular knowledge about the difference adds up until it becomes sufficiently large to reconcile $\mathcal{H}$ and $\mathcal{H}'$. Therefore, the worst-case number of necessary rounds κ is the smallest integer k such that $T_k \geq T$.

In what follows, we will focus on two interesting choices of t_k:

- **Fixed** t**:** $\forall k,\ t_k = t$ for some fixed t, in which case $\kappa = \left\lceil \frac{T}{t} \right\rceil$;
- **Exponential** t_k**:** $\forall k,\ t_k = 2^k t$ for some fixed t, in which case $\kappa = \left\lceil \log_2 \frac{T}{t} \right\rceil$.

[3] *i.e.*, $\prod h'_i \mod a \neq 0$.

Oscar		**Neil**
Initial phase: Neil sends a global hash to detect $\perp_2$		
		$H \leftarrow \texttt{Hash}(\mathcal{H}')$
	$\xleftarrow{H}$	
Main phase: Neil amasses modular information on the difference between $\mathcal{H}$ *and* $\mathcal{H}'$		
	Round 1	
$c_1 \leftarrow \prod_{i=1}^{n} h_i \bmod p_1$	$\xrightarrow{c_1}$	$c_1' \leftarrow \prod_{i=1}^{n'} h_i' \bmod p_1$ $s_1 \leftarrow c_1'/c_1 \bmod p_1$ $S_1 \leftarrow s_1$ reconstruct a, b from $S_1 \pmod{P_1}$ $\mathcal{D}' \leftarrow \{h_i' \in \mathcal{H}' \mid a \bmod h_i' = 0\}$ if $\prod_{h \in \mathcal{D}'} h \bmod P_1 = a$ then go to final phase else continue (event $\perp_1$)
	Round 2	
$c_2 \leftarrow \prod_{i=1}^{n} h_i \bmod p_2$	$\xrightarrow{c_2}$	$c_2' \leftarrow \prod_{i=1}^{n'} h_i' \bmod p_2$ $s_2 \leftarrow c_2'/c_2 \bmod p_2$ $S_2 \leftarrow \mathrm{CRT}(S_1, P_1, s_2, p_2)$ reconstruct a, b from $S_2 \pmod{P_2}$ $\mathcal{D}' \leftarrow \{h_i' \in \mathcal{H}' \mid a \bmod h_i' = 0\}$ if $\prod_{h \in \mathcal{D}'} h \bmod P_2 = a$ then go to final phase else continue (event $\perp_1$)
	$\vdots$	
Final phase: Oscar obtains a candidate $\mathcal{H}'$ *and hash-checks it*		
	$\xleftarrow{\mathcal{D}', b}$	
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i\}$ compute $\mathcal{H}'$ from $\{\mathcal{H}, \mathcal{D}, \mathcal{D}'\}$ if $\texttt{Hash}(\mathcal{H}') \neq H$ then go back to Main phase (event $\perp_2$) else the protocol is complete.		

Fig. 2. Complete D&F Protocol (for any T)

3 Transmission Complexity

This section proves that D&F achieves optimal asymptotic transmission complexity.

Assuming that no $\perp_2$ occurred (since $\perp_2$'s happen with negligible probability), D&F's transmission complexity is:

$$\sum_{k=1}^{\kappa} \log c_k + \log b + \log |\mathcal{D}'| \leq \sum_{k=1}^{\kappa} (ut_k + 1) + \frac{1}{2}(uT_\kappa) + uT \leq \frac{5}{2} uT_\kappa + u\kappa,$$

where κ is the required number of rounds.

For the two choices of t_k that we mentioned, transmission complexity is:

- **Fixed t:** $\kappa = \lceil T/t \rceil$, $T_\kappa = \kappa t < T + t$ and transmission is $\leq \frac{5}{2}u(T+t) + \lceil T/t \rceil = O(uT)$;
- **Exponential t:** $\kappa = \lceil \log(T/t') \rceil$, $T_\kappa < 2T$ and transmission is $\leq \frac{5}{2} \times 2uT + \lceil \log(T/t') \rceil = O(uT)$.

While asymptotic transmission complexities are identical for both choices, we note that the fixed t option is slightly better in terms of constant factors and halves transmission with respect to the exponential option. However, as we will see in Section 5.2, an exponential t results in a lower computational complexity.

Note that in both cases asymptotic transmission complexity is proportional to the size of the symmetric difference (*i.e.*, the number of differences times the size of an individual element). This is also the information-theoretic lower bound on the quantity of data needed to perform reconciliation. Hence, the protocol is asymptotically optimal from a transmission complexity standpoint.

Probabilistic Decoding: Reducing p. We now describe an improvement that reduces transmission by a constant factor at the expense of higher RNR failure rates. For simplicity, we will focus on one round D&F and denote by p the current P_k. We will generate a p about twice smaller than the p recommended in Section 2.2, namely $2^{ut-1} \leq p < 2^{ut}$.

Unlike Section 2.2, we do not have a fixed bound for a and b anymore; we only have a bound for the *product* ab, namely $ab \leq 2^{ut}$.

Therefore, we define $t+1$ couples of possible bounds: $(A_j, B_j)_{0 \leq j \leq t} = \left(2^{uj}, 2^{u(t-j)}\right)$.

Because $2^{ut-1} \leq p < 2^{ut}$ and $ab \leq 2^{ut}$, there must exist at least one index j such that $0 \leq a \leq A_j$ and $0 < b \leq B_j$. We can therefore apply Theorem 1 with $A = A_j$ and $B = B_j$: since $A_j B_j = 2^{ut} < p$, given (A_j, B_j, p, s), one can recover (a, b), and hence Oscar can compute $\mathcal{H}'$.

This variant will roughly halve transmission with respect to Section 2.2. The drawback is that, unlike Section 2.2, we have no guarantee that such an (a, b) is unique. Namely, we could in theory stumble over an $(a', b') \neq (a, b)$ satisfying the equation $a'b' \leq 2^{ut}$ for some index $j' \neq j$. We conjecture that, when u is large enough, such failures happen with a negligible probability (that we do not try to estimate here). This should lower the expected transmission complexity of this variant. In any case, thanks to hashing ($H = \texttt{Hash}(\mathcal{H}')$), if a failure occurs, it will be detected.

4 From Set Reconciliation to File Reconciliation

We now show how to perform *file reconciliation* using hashing and D&F. We then devise methods to reduce the size of hashes and thus improve transmission

by constant factors. The presented methods are generic and can be applied to any set reconciliation protocol.

4.1 File Reconciliation Protocol

So far Oscar and Neil know how to synchronize sets of u-bit primes. They now want to reconcile files modeled as arbitrary length binary strings. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be Oscar's set of files and let $\mathcal{F}' = \{F'_1, \ldots, F'_n\}$ be Neil's. Let $\eta = |\mathcal{F} \cup \mathcal{F}'| \leq n + n'$ be the total number of files.

A naïve way to reconcile $\mathcal{F}$ and $\mathcal{F}'$ is to simply hash the content of each file into a prime and proceed as before. Upon D&F's completion, Neil can send to Oscar the actual content of the files matching the hashes in $\mathcal{D}'$, *i.e.*, the files that Oscar does not have.

More formally, define for $1 \leq i \leq n$, $h_i = \texttt{HashPrime}(F_i)$ and for $1 \leq i \leq n'$, $h'_i = \texttt{HashPrime}(F'_i)$ where `HashPrime` is a collision-resistant hash function into primes so that the mapping from $\mathcal{F} \cup \mathcal{F}'$ to $\mathcal{H} \cup \mathcal{H}'$ is injective for all practical purposes. Section 5.1 shows how to construct such a hash function from usual hash functions.

4.2 The File Laundry: Reducing u

What happens if we brutally shorten u in the basic D&F protocol? As expected by the birthday paradox, we should start seeing collisions. [3] analyzes the statistics governing the appearance of collisions. The average number of colliding files is $\sim \eta(\eta - 1)2^{-u'}$ where $u' = u - \ln(u)$. For instance, the expected number of collisions for $\eta = 10^6$ and 42-bit digests, the average number of colliding files is < 4. We remark that a collision can only yield a *false positive*, and never a *false negative*. In other words, while a collision may make Oscar and Neil miss a real difference, it will never create a nonexistent difference *ex nihilo*. Thus, it suffices to replace $\texttt{HashPrime}(F)$ by a diversified $\hbar_k(F) = \texttt{HashPrime}(k|F)$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \ldots$ We call each complete D&F protocol repetition (which usually involves several basic protocol *rounds*) an *iteration*. At each iteration, the parties will detect files in $\mathcal{F} \Delta \mathcal{F}'$ whose hash $\hbar_{i,\ell}$ or $\hbar'_{i,\ell}$ does not collide, reconciliate these differences, remove these files from $\mathcal{F}$ and $\mathcal{F}'$ to avoid further collisions, and "launder" again the remaining files in the updated versions of $\mathcal{F}$ and $\mathcal{F}'$.

Let $\epsilon_{\eta,u,k}$ be the probability that at least one file will persist colliding during k rounds. Assuming that η is invariant between iterations, we find that $\epsilon_{\eta,u,k} \leq n((\eta - 1)2^{-u'})^k$ *i.e.*, $\epsilon_{\eta,u,k}$ decreases exponentially in k (*e.g.*, $\epsilon_{10^6,42,2} \leq 10^{-3}\%$, see [3]).

We still need a condition to stop "laundering", *i.e.*, a condition ensuring that there are no more differences hidden by collisions. Before we describe this condition, let us first spell out the three kinds of collisions that can appear during iteration ℓ:

1. Collisions in $\mathcal{F} \cap \mathcal{F}'$ (*i.e.*, between common files). These are never a problem because they cannot hide any differences.
2. Collisions between between $\mathcal{F} \cap \mathcal{F}'$ and $\mathcal{F} \Delta \mathcal{F}'$ (*i.e.*, between a common file of Oscar and Neil, and a file not in common). These collisions can be easily detected by Oscar or Neil, at the end of iteration ℓ. However, if there is a collision of this kind involving an $h \in \mathcal{H} \Delta \mathcal{H}'$, we will not be able to find the file in $\mathcal{F} \Delta \mathcal{F}'$ matching h. For this reason, another D&F iteration will be necessary to reconcile this file.
3. Collisions in $\mathcal{F} \Delta \mathcal{F}'$ (*i.e.*, between files not in common). Such collisions hide real differences between $\mathcal{F}$ and $\mathcal{F}'$ and cannot be detected without a further iteration. This is why we need a condition to detect that no more collisions of this kind exist and stop laundering.

We propose the following method to decide termination. Before the first *iteration*, Neil sends a global hash $H' = \texttt{Hash}(\texttt{Hash}(F'_1), \ldots, \texttt{Hash}(F'_{n'}))$ to Oscar. This H' should not be confused with the H sent at the beginning of *each* iteration[4]. Now, if iteration ℓ is successful, Neil sends the list of $\texttt{Hash}(F'_i)$ for the new files $F'_i \in \mathcal{F}' \setminus \mathcal{F}$ whose hash $\hbar'_{i,\ell}$ does not collide with files in $\mathcal{F}' \cap \mathcal{F}$ (*i.e.*, type-2 collisions). Oscar can then use H' to check whether a type-3 collision remains, in which case a new iteration is performed. If no type-2 or type-3 collisions remain, then reconciliation is complete.

5 Computational Complexity

We now analyze D&F's computational complexity. We first describe the time complexity of a straightforward implementation (Section 5.1), and then present four independent optimizations (Section 5.2). A summary of all costs is given in Table 1. To simplify analysis, we assume that there are no collisions, and that $n = n'$.

5.1 Basic Complexity and Hashing into Primes

Let $\mu(\ell)$ be the time required to multiply two ℓ-bit numbers, with the assumption that $\forall \ell, \ell', \mu(\ell + \ell') \geq \mu(\ell) + \mu(\ell')$. For naïve (*i.e.*, convolutional) algorithms, $\mu(\ell) = O(\ell^2)$, but using FFT multiplication [13], $\mu(\ell) = \tilde{O}(\ell)$ (where $\tilde{O}(f(\ell))$ is a shorthand for $O(f(\ell) \log^k f(\ell))$ for some k). FFT is experimentally faster than convolutional methods from $\ell \sim 10^6$ and on. The modular division of a 2ℓ-bit number by an ℓ-bit number and the reduction of a 2ℓ-bit number modulo an ℓ-bit number are also known to cost $\tilde{O}(\mu(\ell))$ [4]. Indeed, in packages such as GMP, division and modular reduction run in $\tilde{O}(\ell)$ for sufficiently large ℓ.

The naïve complexity of `HashPrime` is $u^2 \mu(u)$, as per [9,2].

– A recommended implementation of $\texttt{HashPrime}(F)$ consists in defining the digest as $h = 2 \cdot \texttt{Hash}(F|i) + 1$ and increasing i until h is prime. Because there are roughly $\frac{2^u}{u}$ u-bit primes we need to perform (on average) u primality

[4] H is a hash of the (potentially colliding) diversified hashes $\hbar(\ell|F)$.

Table 1. Protocol Complexity Synopsis

Entity	Computation	Complexity in $\tilde{O}$ of		Optimization
		Basic algo.[a]	**Opt. algo.**[b]	
Both	computation of h_i and h'_i	$nu^2 \cdot \mu(u)$	$\frac{\phi(\alpha)}{\alpha} nu^2 \cdot \mu(u)$	fast hashing
	Round k			
Both	compute redundancies c_k and c'_k	$n \cdot \mu(ut_k)$	$\frac{n}{t_k} \cdot \mu(ut_k)$	prod. trees
Neil	compute $s_k = c'_k/c_k$[c] or $S_k = C'_k/C_k$[d]	$\mu(uT_k)$	$\mu(uT_k)$	
Neil	compute S_k from S_{k-1} and s_k (CRT)[c]	$\mu(uT_k)$	none	$p_k = 2^{ut_k}$
Neil	find a_k, b_k such that $S_k = a_k/b_k \bmod P_k$[e]	$\mu(uT_k)$	$\mu(uT_k)$	
Neil	factor a_k	$n \cdot \mu(uT_k)$	$\frac{n}{T_k} \cdot \mu(uT_k)$	prod. trees
	Last round			
Oscar	factor b_k	$n \cdot \mu(ut_k)$	$\frac{n}{t_k} \cdot \mu(ut_k)$	prod. trees
	global complexity			
	... with naïve mult.	$nu^2T^3/t + nu^4$	$nu^2T + \frac{\phi(\alpha)}{\alpha} nu^4$	doubling[f]
	... with FFT	$nuT + nu^3$	$nu + \frac{\phi(\alpha)}{\alpha} nu^3$	doubling[f]

[a] using the basic algorithms of Section 5.1, and taking $t_1 = t_2 = \cdots = t$
[b] using all the optimizations of Sections 5.1, 5.2 ($p_k = 2^{ut_k}$ also yields substantial constant factor accelerations not shown in this table), the product trees and the doubling as described in the full version of this paper
[c] only for prime p_i (or variant 1 or 2 in Section 5.2)
[d] only for $p_i = 2^{ut_i}$
[e] using advanced algorithms in [12,16] — naïve extended GCD gives $(uT_i)^2$
[f] in addition to the previous optimizations

tests before finding a suitable h. The cost of a Miller-Rabin primality test is $\tilde{O}(u\mu(u))$. Hence, the total cost of this implementation is $\tilde{O}(u^2\mu(u))$. A more precise analysis can be found in [2].

- If u is large enough (*e.g.*, 160) one might sacrifice uniformity to avoid repeated file hashings using $\texttt{HashPrime}(F) = \texttt{NextPrime}(\texttt{Hash}(F))$.
- Yet another acceleration option consists in computing $h = \alpha \lfloor \texttt{Hash}(F)/\alpha \rfloor + 1$, where $\alpha = 2 \times 3 \times 5 \times \cdots \times \texttt{Prime}[d]$ is the product of the first primes until some rank d, and then subtract α from h until h becomes prime. Denote by ϕ Euler's totient function and assume that this algorithm randomly samples u-bit numbers congruent to 1 mod α until it finds a prime. There are about $\frac{2^u}{u\phi(\alpha)}$ u-bit primes congruent to 1 mod α, and there are $\frac{2^u}{\alpha}$ u-bit numbers congruent to 1 mod α. Thus, the algorithm is expected to do about $\frac{2^u}{\alpha} / \frac{2^u}{u\phi(\alpha)} = \frac{\phi(\alpha)}{\alpha} u$ primality tests before finding a prime, which improves

over the u tests required by the naïve algorithm. The main drawback of this algorithm is that, even if `Hash` is uniformly random, `HashPrime` isn't. This slightly increases `HashPrime`'s collision-rate and u has to be increased subsequently.

5.2 Optimizations

Adapting p_k. Taking the p_k's to be ut_k-bit primes is inefficient, because large prime generation is slow. In this section, we study alternative p_k choices yielding constant factor improvements.

Let $\texttt{Prime}[i]$ denote the i-th prime, with $\texttt{Prime}[1] = 2$. Besides conditions on size, the *only* property required from a p_k is to be co-prime with the $\{h_i, h'_i\}$. We can hence consider the following variants, all which will imply a few conditions on $\{h_i, h'_i\}$ to ensure this co-primality:

- *Variant 1.* $p_k = \prod_{j=r_k}^{r_{k+1}-1} \texttt{Prime}[j]$ where the bounds r_k are chosen to ensure that each p_k has the proper size. Generating such smooth numbers is much faster than generating large primes.
- *Variant 2.* $p_k = \texttt{Prime}[k]^{r_k}$ where the exponents r_k are chosen to ensure that each p_k has the proper size. This is faster than Variant 1 and requires that $\min\{h_i, h'_i\} > \max(\texttt{Prime}[k])$.
- *Variant 3.* $P_k = 2^{ut_k}$. In this case $C_k = \prod_{i=1}^{n} h_i \bmod P_k$, $c_1 = C_1$ and $c_k = (C_k - C_{k-1})/P_{k-1}$. *i.e.*, c_k is the slice of bits $ut_{k-1}, \ldots, ut_k - 1$ of C_k denoted $c_k = C_k[ut_{k-1}, \ldots, ut_k]$. Variant 3 is modular-reduction-free and CRT-free: C_k is just the binary concatenation of c_k and C_{k-1}. Computations are thus much faster. Algorithm 1 (justified hereafter) computes c_k efficiently. Note that we only need to store $D_{k,i}$ and $D_{k+1,i}$ during round k (for all i). So space overhead is $O(nu)$.

  Let $X_i = \prod_{j=1}^{i} h_j$ (with $X_0 = 1$), $X_{i,k} = X_i[ut_{k-1} \cdots ut_k]$ and let $D_{i,k}$ be the u most significant bits of the product of $Y_{i,k} = X_i[0 \cdots ut_k]$ and h_i, *i.e.*, $D_{i,k} = (Y_{i,k} \times h_i)[ut_k \cdots u(t_k + 1)]$ (with $D_{i,0} = 0$ and $D_{0,k} = 0$). Since $X_{i+1} = X_i \times h_{i+1}$, we have, for $k \geq 0$, $i \geq 0$:

$$\begin{aligned} D_{i+1,k+1} \times 2^{ut_{k+1}} + Y_{i+1,k+1} &= Y_{i,k+1} \times h_{i+1} \\ &= (X_{i,k+1} \times 2^{ut_k} + Y_{i,k}) \times h_{i+1} \\ &= X_{i,k+1} \times 2^{ut_k} \times h_{i+1} + Y_{i,k} \times h_{i+1} \\ &= X_{i,k+1} \times h_{i+1} \times 2^{ut_k} + (D_{i,k} \times 2^{ut_k} + \cdots). \end{aligned}$$

  Therefore, if we only consider bits $[ut_k \cdots u(t_{k+1} + 1)]$, for $k \geq 1$, $i \geq 0$:

$$D_{i+1,k+1} \times 2^{u(t_{k+1} - t_k)} + X_{i+1,k+1} = X_{i,k+1} \times h_{i+1} + D_{i,k}.$$

  Since $c_k = X_{n,k}$, Algorithm 1 is correct.

Algorithm 1. Computation of c_k for $p_k = 2^{ut_k}$

Require: k, the set h_i, $(D_{k,i})$ as explained in [3]
Ensure: $c_{k+1} = \prod_{i=1}^{n} h_i \bmod p_{k+1}$, (D_{k+1}) as explained in [3]
1: **if** $k = 0$ **then** $X \leftarrow 1$ **else** $X \leftarrow 0$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad Z \leftarrow X \times h_i$
4: $\quad D_{i,k+1} \leftarrow Z[u(t_{k+1} - t_k) \cdots u(t_{k+1} - t_k + 1)]$
5: $\quad X \leftarrow Z[0 \cdots u(t_{k+1} - t_k)]$
6: $c_{k+1} \leftarrow X$

Algorithmic Optimizations Using Product Trees. The non-overwhelming (but nonetheless important) complexities of the computations of (c, c') and of the factorizations can be even reduced to $\tilde{O}(\frac{n}{t_k}\mu(ut_k))$ and $\tilde{O}(\frac{n}{T_k}\mu(uT_k))$ as explained in [3].

Doubling. As seen at the end of Section 2.3, using the exponential t variant (*i.e.*, doubling t_k at each iteration) doubles transmission (at most) with respect to the fixed t option, but drastically reduces the amount of computation to perform.

6 Related Work on Set Reconciliation

This section compares D&F with the set reconciliation algorithm of Minsky *et alii* [11] (hereafter MTZ). We do not analyze here reconciliation algorithms achieving better computational performances at the cost of supra-linear transmission complexity (*e.g.*, [7] or [5]).

Unlike MTZ which is based on polynomials, D&F is based on integers. D&F and MTZ both achieve an optimal (*i.e.*, linear) transmission complexity, but D&F only deals with fixed-size primes, whereas MTZ deals with any fixed-size bit strings.

MTZ is mostly designed for "incremental" settings where $\mathcal{H}$ and $\mathcal{H}'$ are often updated[5] and re-synchronized. This differs from our setting and there seems to be no straightforward manner to extend D&F in that fashion while maintaining a low time complexity. For that reason, our analysis of MTZ's time complexity will take into account the cost of computing redundancies, as we did for D&F. The main differences between MTZ and D&F are the following:

- MTZ synchronizes monic polynomials $X - h_i$ and $X - h'_i$ over a field $\mathbb{F}_q$ (where q is a $(u+1)$-bit prime), instead of u-bit primes $\{h_i, h'_i\}$;
- In MTZ, p_k are square-free, mutually co-prime polynomials which are also co-prime with all $X - h_i$ and $X - h'_i$. In D&F this role is played by mutually co-prime integers that are also co-prime with respect to the $\{h_i, h'_i\}$ (for all the variants in Section 5.2 except the last).

[5] *e.g.*, by adding or removing a few values to $\mathcal{H}$ or $\mathcal{H}'$.

Indeed, in the basic one-round version:

- If we write $p_k = (X - \rho_1) \cdots (X - \rho_t)$, then $c = \prod_{i=1}^{n}(X - h_i) \bmod p_k$ and $\chi_{\mathcal{H}} = \prod_{i=1}^{n}(X - h_i)$, $\chi_{\mathcal{H}}(\rho_j) = c(\rho_j)$. Thus, thanks to Lagrange interpolation, sending evaluations of $\chi_{\mathcal{H}}$ in t points $\rho_1, \ldots, \rho_j$, as Oscar does in [11], is equivalent to sending c.
- The rational function interpolation of [11] can also be seen as an RNR version of Theorem 1 for polynomials: we try to recover two polynomials a, b (with a correct bound on degrees) such that $ab^{-1} \bmod p = c'c^{-1} \bmod p$. Note that this implies that the Gaussian elimination of cost $O(t^3\mu(u))$ (used for this step by MTZ) can be replaced by an extended GCD computation that costs only $O(t^2\mu(u))$ (and $\tilde{O}(\mu(ut))$ using the advanced algorithms of [12,16]);

We will compare the computational complexities of MTZ and D&F without taking into account the cost of hashing the files that has to be incurred by both algorithms. MTZ's time complexity is thus $O(nu^2T + u^2T^3)$ when doubling is used, which is not as good as our $\tilde{O}(nu)$ with FFT, and also not as good as our non-FFT complexity $\tilde{O}(nu^2T)$ when $n \ll T^2$. However, our better complexity bounds stems from optimizations that are all equally applicable to MTZ (except, of course, the optimizations concerning the choice of p_k).

An improved way to perform set reconciliation is presented in [10]. This algorithm uses MTZ as a black box and requires at least about $24e \cong 65.23$ times more bandwidth (with a bipartition) but substantially improves MTZ's computational complexity. However, this construction is generic with respect to the underlying reconciliation algorithm and can hence be applied to D&F to yield identical complexity gains.

7 From File Reconciliation to File Synchronization

In Section 4, we reconciled file sets by looking only at their contents. However, in practice, users synchronize file *sets*, and not just *hierarchies*. In other words, we are not just interested in file *contents* but also in their *metadata*. The most important metadata is the file's path (*i.e.*, its name and location in the filesystem), though other kinds of metadata exist (*e.g.*, modification time, owner, permissions). In many cases, file metadata change while the file contents do not: *e.g.*, files can be *moved* to a different directory. When performing reconciliation, we must be aware of this fact, and reflect file moves without re-transferring the moved files' contents. (This is an important improvement over popular synchronization tools such as `rsync`).

We will call this task *file synchronization*. This section achieves *file synchronization* using D&F as a black-box. The described algorithms are hence generic and can leverage any reconciliation algorithm.

7.1 General Principle

To perform file synchronization, Oscar and Neil will hash the contents of each of their files using a collision-resistant hash function `Hash`: we will call this the file's

content hash and denote it by C_i or C'_i for the i-th file in $\mathcal{F}$ or $\mathcal{F}'$. Likewise, we denote by M_i or M'_i the files' metadata. We let F_i or F'_i denote the pair (C_i, M_i) or (C'_i, M'_i). Oscar and Neil will reconciliate those sets as in Section 4.

Once the reconciliation has completed, Oscar is aware of the metadata and the content hash of all of Neil's files that do not exist in his disk with the same content and metadata (we will call these the *missing files*).

Oscar now looks at the list of the missing files' content hashes. For some of these hashes, Oscar may already have a file with the same content hash, but only with a wrong metadata. For others, Oscar may not have any file with the same content hash. In the first case, Oscar can recreate Neil's file by altering the metadata, without retransferring the file's contents. This is presented in Section 7.2. In the second case, Oscar needs to retrieve the full file contents from Neil. This is presented in Section 7.3.

7.2 Moving Existing Files

Adjusting the metadata of existing files is trivial, except for file location which is the focus of this section: Oscar needs to perform a sequence of file moves on his copy to reproduce the structure of Neil's copy. Sadly, it is not straightforward to apply the moves, because, if we take a file to move, its destination might be blocked, either because a file already exists (we want to move a to b, but b already exists), or because a folder cannot be created (we want to move a to b/c, but b already exists as a file and not as a folder). Note that for a move operation $a \to b$, there is at most one file blocking the location b: we will call it the *blocker*.

If the blocker is absent on Neil, then we can just delete the blocker. However, if a blocker exists and is a file which appears in Neil with different metadata, then we might need to move this blocker somewhere else before we apply the move we are interested in. Moving the blocker might be impossible because of another blocker that we need to keep, and so on, possibly ending in a cycle (*e.g.*, move a to b and b to a) in which case we need to use an intermediate temporary location.

How should we perform the moves? A simple way would be to move each file to a unique temporary location and then move them to their final location: however, this performs many unnecessary moves and could lead to problems if the process is interrupted. We can do something more clever by performing a decomposition into Strongly Connected Components (SCC) of the *move graph* (with one vertex per file and one edge per move operation going from to the file to its blocker or to its destination if no blocker exists).

Once the SCC decomposition is known, moves can be applied by performing them in each SCC in a bottom-up fashion, an SCC's moves being solved either trivially (for single files) or using one intermediate location (for cycles).

The detailed algorithm is implemented as two mutually recursive functions and presented as Algorithm 2.

Algorithm 2. Perform moves

Require: $\mathfrak{M}$ is a dictionary where $\mathfrak{M}[f]$ denotes the intended destinations of f

```
1: C ← []                                   ▷ Stores the color of a file (initially "not_done")
2: T ← []                                   ▷ Stores the temporary location assigned for a file
3: function UNBLOCK_COPY(f, d)
4:     if d is blocked by some b then
5:         if b is not in 𝔐's keys then delete(b)          ▷ We don't need b
6:         else RESOLVE(b)                  ▷ Take care of b and make it go away
7:     if T[f] was set then f ← T[f]
8:     copy(f, d)
9: function RESOLVE(f)
10:    if C[f] = done then
11:        return                           ▷ Already managed by another in-edge
12:    if C[f] = doing then
13:        if T[f] was not set then
14:            T[f] ← mktemp()              ▷ Use a new temporary location
15:            move(f, T[f])
16:        return                           ▷ We found a loop, moved f out of the way
17:    C[f] ← doing
18:    for d ∈ 𝔐[f] with d ≠ f do
19:        UNBLOCK_COPY(f, d)               ▷ Perform all the moves
20:    if f ∉ 𝔐[f] and T[f] was not set then delete(f)
21:    if T[f] was set then delete(T[f])
22:    C[f] ← done
23: for f in 𝔐's keys do
24:    RESOLVE(f)
```

7.3 Transferring Missing Files

Once all moves have been applied, Oscar's hierarchy contains all of its files which also exist on Neil. These have been put at the correct location and have the right metadata. The only thing that remains is to transfer the contents of Neil's files that do not exist in Oscar's hierarchy and create those files at the right position. To do so, we can just use `rsync` to synchronize explicitly the correct files on Neil to the matching locations in Oscar's hierarchy, using the fact that Oscar is now aware of all of Neil's files and their locations. In so doing, we have to ensure that multiple files on Neil that have the same content are only transferred once and then copied to all their locations without being retransferred.

It is interesting to notice that if a file's contents has been changed slightly on Neil but its location hasn't changed, then in most cases the `rsync` invocation will reuse the existing copy of the file on Oscar when transferring this file from Neil to Oscar. Because `rsync` uses rolling checksums to retransfer only relevant file parts, this may actually reduce the transmission complexity. If a file's content is slightly changed and the file is moved, however, then this gain will not occur.

8 Implementation

We implemented D&F, extended it to perform file synchronization, and benchmarked it against `rsync`. The implementation is called `btrsync`, its source code is available from [1]. `btrsync` was written in Python (using GMP to perform the number theoretic operations), and uses a bash script (invoking SSH) to create a secure communication channel between Oscar and Neil.

8.1 Implementation Choices

Our implementation does not take into account all the possible optimizations described in Section 5: it implements doubling (Section 5.2) and uses powers of small primes for the p_k (variant 2 of Section 5.2), but does not implement product trees (Section 5.2) nor does it use the prime hashing scheme (Section 5.1). Besides, we did not implement the proposed improvement in transmission complexity for file reconciliation (Section 4.2).

As for file synchronization (Section 7), the only metadata managed by `btrsync` is the file's path (name and location). Other metadata types (modification date, owner, permissions) are not implemented, although it would be very easy to do so. An optimization implemented by `btrsync` over the move resolution algorithm described in Section 7.2 is to avoid doing a copy of a file F and then removing F: the implementation replaces such operations by moves, which are faster than copies on most file systems because the OS does not need to copy the actual file contents.

8.2 Experimental Comparison to `rsync`

We compared `rsync`[6] and our implementation `btrsync`. The directories used for the benchmark are described in Table 2. Experiments were performed without any network transfer, by synchronizing two folders on the same host. Hence, time measurements mostly represent the synchronization's CPU cost.

Results are given in Table 3. In general, `btrsync` spent more time than `rsync` on computation (especially when the number of files is large, which is typically seen in the experiments involving `syn`). Transmission results, however, are favorable to `btrsync`.

In the trivial experiments where either Oscar or Neil have no data at all, `rsync` outperforms `btrsync`. This is especially visible when Neil has no data: `rsync`, unlike `btrsync`, immediately notices that there is nothing to transfer.

[6] `rsync` version 3.0.9, used both as a competitor to benchmark against and as an underlying call in our own code. `rsync` was passed the following options: `--delete` to delete Oscar's files that were deleted on Neil like `btrsync` does, `-I` to disable heuristics based on file modification times that `btrsync` does not use, `--chmod="a=rx,u+w"` to make it unnecessary to transfer file permission that `btrsync` does not transfer (though verbose logging suggest that `rsync` wastes a few bytes per file because it transmits them anyway), and `-v` to count the number of sent and received bytes.

Table 2. Test Directories

Directory	Description
`syn`	a directory containing 1000 very small files
`syn_shuf`	`syn` changed by 10 deletions, renames and modifications
`source`	a snapshot of `btrsync`'s own source tree
`source_moved`	`source` with one big folder (a few megabits) renamed
`ff-13.0`	the source archive of Mozilla Firefox 13.0
`ff-13.0.1`	the source archive of Mozilla Firefox 13.0.1
`empty`	an empty folder

Table 3. Experimental results. Synchronization is performed *from* Neil *to* Oscar. `RX` and `TX` denote the quantity of received and sent bytes, `rs` and `bt` denote `rsync` and `btrsync`, and $\delta_\square = \texttt{TX}_\square + \texttt{RX}_\square$. $\delta_{\texttt{rs}} - \delta_{\texttt{bt}}$ and $\delta_{\texttt{bt}}/\delta_{\texttt{rs}}$ express the absolute and the relative differences in transmission between `rsync` and `btrsync`. The last two columns show timing results on an Intel Core i3-2310M CPU clocked at 2.10 Ghz.

Entities and Datasets		Transmission (Bytes)						Time (s)	
Neil's $\mathcal{F}'$	Oscar's $\mathcal{F}$	$\texttt{TX}_{\texttt{rs}}$	$\texttt{RX}_{\texttt{rs}}$	$\texttt{TX}_{\texttt{bt}}$	$\texttt{RX}_{\texttt{bt}}$	$\delta_{\texttt{bt}} - \delta_{\texttt{rs}}$	$\frac{\delta_{\texttt{bt}}}{\delta_{\texttt{rs}}}$	$\texttt{t}_{\texttt{rs}}$	$\texttt{t}_{\texttt{bt}}$
`source`	`empty`	778k	2k	780k	10k	10k	1.0	0.1	0.7
`empty`	`source`	24	12	12k	6k	18k	496.6	0.0	0.4
`empty`	`empty`	24	12	19	30	13	1.4	0.0	0.3
`syn`	`syn_shuf`	55k	19k	7k	3k	-63k	0.1	0.5	0.8
`syn_shuf`	`syn`	54k	19k	7k	3k	-63k	0.1	0.2	0.8
`syn`	`syn`	55k	19k	327	30	-73k	0.0	0.5	0.7
`ff-13.0.1`	`ff-13.0`	41M	1k	40M	3k	-1M	1.0	1.6	8.1
`source_moved`	`source`	778k	1k	3k	2k	-775k	0.0	0.1	0.4

In non-trivial tasks, however, `btrsync` outperforms `rsync`. This is the case of the `syn` datasets, where `btrsync` does not have to transfer information about all unmodified files, and even more so in the case where there are no modifications at all. For Firefox source code datasets, `btrsync` saves a very small amount of bandwidth, presumably because of unmodified files. For the `btrsync` source code dataset, we notice that `btrsync`, unlike `rsync`, was able to detect the move and avoid retransferring the moved folder.

9 Conclusion and Further Improvements

This paper introduced the new number-theoretic set reconciliation protocol called *Divide and Factor* (D&F). We analyzed D&F's transmission and time complexities and describing several optimizations and parameter choices. We have shown how D&F can be applied to file reconciliation using hashing, and to solve the practical file synchronization problem. D&F was benchmarked against

`rsync`. The comparison reveals that D&F transmits less data than `rsync` but performs more computation.

Many interesting problems are left open. These problems are both theoretical and practical. A first theoretical challenge consists in eliminating the costly hashing into primes. *e.g.*, if the p_k are powers of two then hashing into odd integers might suffice. This would make reconciliation harder because multiple factorizations of a and b as products of h_i and h'_i could exist while only one of them would be the correct one. A careful probabilistic analysis would be required to determine the probability of multiple factorizations and bound the cost of recovering the correct factorization. This phenomenon is tightly linked to the cryptographic notion of *collision-division* [6]. As for other aspects of our construction, many bounds on transmission and computational complexities could be refined and improved.

Other theoretical questions are left open by our study of move resolution: The algorithm that we propose is *suboptimal* because there should never be any need to use two different temporary file locations: one location is always sufficient to break cycles, and a more careful exploration of the move graph could proceed in that fashion. It is also interesting to find out if there is a way to perform a minimal number of temporary moves (or if this problem is NP-complete), or if we can reduce the total number of moves by moving folders in addition to files.

From a practical standpoint, our `btrsync` implementation could be improved in several ways. First, the numerous possible improvements described in the paper could be implemented and benchmarked. Then, heuristics could be added to work around the situations in which `btrsync` is outperformed by `rsync`, such as the ones identified during our experimental comparison of the two programs. For instance, whenever the product of Neil's hashes becomes smaller than P_k, then Neil should send its hashes immediately to Oscar and terminate the protocol: this would avoid transmitting a lot of data in situations where Neil's copy is empty or very small. Last but not least, the development of our `btrsync` prototype could be continued to make it suitable for real-world users, including proper management of all metadata, using the file modification time as a heuristic to detect changes, and caching of file content hashes to avoid recomputing them.

A possible additional feature that could be added to `btrsync` is to detect files that have been both moved and altered slightly. A related improvement would be to use a variant of `rsync`'s algorithm to transfer Neil's new files to Oscar by considering simultaneously several related files on Oscar's copy and computing rolling checksums.

Finally, we could study how additional information could be used to speed up set reconciliation. An interesting possibility is to give to Neil and Oscar, in addition to their files, a value for each file indicating the probability that the other party does not have this file. To what extent could this prior knowledge be exploited to perform reconciliation more efficiently?

Acknowledgment. The authors acknowledge Guillain Potron for his early involvement in this research work.

References

1. https://github.com/RobinMorisset/Btrsync
2. Abdalla, M., Ben Hamouda, F., Pointcheval, D.: Tighter Reductions for Forward-Secure Signature Schemes. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 292–311. Springer, Heidelberg (2013)
3. Amarilli, A., Ben Hamouda, F., Bourse, F., Morisset, R., Naccache, D., Rauzy, P.: From Rational Number Reconstruction to Set Reconciliation and File Synchronization. Full version available from the authors' webpage
4. Burnikel, C., Ziegler, J., Stadtwald, I.: Fast Recursive Division, Tech. Rep., MPI-I-98-1-022, MPI Informatik Saarbrucken (1998)
5. Byers, J., Considine, J., Mitzenmacher, M., Rost, S.: Informed Content Delivery Across Adaptive Overlay Networks. ACM SIGCOMM Computer Communication Review 32(4), 47–60 (2002)
6. Coron, J.-S., Naccache, D.: Security Analysis of the Gennaro-Halevi-Rabin Signature Scheme. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 91–101. Springer, Heidelberg (2000)
7. Eppstein, D., Goodrich, M., Uyeda, F., Varghese, G.: What's the Difference?: Efficient Set Reconciliation Without Prior Context. ACM SIGCOMM Computer Communication Review 41, 218–229 (2011)
8. Fouque, P.A., Stern, J., Wackers, J.G.: Cryptocomputing With Rationals. In: Blaze, M. (ed.) FC 2002. LNCS, vol. 2357, pp. 136–146. Springer, Heidelberg (2003)
9. Hohenberger, S., Waters, B.: Short and Stateless Signatures from the RSA Assumption. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 654–670. Springer, Heidelberg (2009)
10. Minsky, Y., Trachtenberg, A.: Practical Set Reconciliation, Tech. Rep., Department of Electrical and Computer Engineering, Boston University, Technical Report BU-ECE-2002-01, 2002, a full version can be, downloaded from http://ipsit.bu.edu/documents/BUTR2002-01.ps
11. Minsky, Y., Trachtenberg, A., Zippel, R.: Set Reconciliation With Nearly Optimal Communication Complexity. IEEE Transactions on Information Theory 49(9), 2213–2218 (2003)
12. Pan, V., Wang, X.: On Rational Number Reconstruction and Approximation. SIAM Journal on Computing 33(2), 502–503 (2004)
13. Schönhage, A., Strassen, V.: Schnelle Multiplikation großer Zahlen. Computing 7(3), 281–292 (1971)
14. Tridgell, A.: Efficient Algorithms for Sorting and Synchronization, Ph.D. thesis, The Australian National University (1999)
15. Vallée, B.: Gauss' Algorithm Revisited. Journal of Algorithms 12(4), 556–572 (1991)
16. Wang, X., Pan, V.: Acceleration of Euclidean Algorithm and Rational Number Reconstruction. SIAM Journal on Computing 32(2), 548–556 (2003)

Affine Refinement Types for Authentication and Authorization

Michele Bugliesi[1], Stefano Calzavara[1], Fabienne Eigner[2], and Matteo Maffei[2]

[1] Università Ca' Foscari Venezia
{bugliesi,calzavara}@dais.unive.it
[2] Saarland University
{eigner,maffei}@cs.uni-saarland.de

Abstract. Refinement type systems have proved very effective for security policy verification in distributed authorization systems. In earlier work [12], we have proposed an extension of existing refinement typing techniques to exploit sub-structural logics and affine typing in the analysis of *resource aware* authorization, with policies predicating over access counts, usage bounds and resource consumption. In the present paper, we show that the invariants that we enforced by means of ad-hoc typing mechanisms in our initial proposal can be internalized, and expressed directly as proof obligations for the underlying affine logical system. The new characterization leads to a more general, modular design of the system, and is effective in the analysis of interesting classes of authentication protocols and authorization systems.

1 Introduction

Authorization policies constitute an effective device for security specification in distributed protocols and systems [3,5]. In language-based security, such policies are specified by means of code annotations marking authorization-sensitive program points with logical formulas that serve as *assumptions* and *assertions*: the former express the credentials available at the clients, and track the clients' authorization requests; the latter are employed as resource guards at the server, and express the conditions required to accept the authorization requests. The annotations have no semantic import, and only serve the verification process: to show a system *safe*, i.e., to prove that it complies with a given policy, one must prove that all the active (unguarded) assertions at a given execution step are entailed by the active assumptions at that step, for every possible system run. Proving a system *robustly safe* amounts to prove that the same property holds in the presence of other, possibly malicious agents [6].

Safety (and robust safety) proofs for annotated specifications such as those of our present interests can be carried out statically, and effectively, with refinement typing systems [15,4,6,7]. Refinement types [16] are dependent types of the form $\{x : T \mid F(x)\}$: a value M of this type is a value of type T such that the formula $F\{M/x\}$ holds. In type-based authorization systems, the refinement formulas are employed to capture the dynamic exchange of credentials

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 19–33, 2013.
© Springer-Verlag Berlin Heidelberg 2013

required for authorization: this is accomplished by encoding such credentials as formulas that refine the payload types of the cryptographic keys involved in the authorization protocols.

Depending on the authorization properties expected, different logical frameworks may be appealed to for specification and verification. Our focus in the present paper is on *resource conscious* policies such as those governing large classes of modern authorization frameworks, based on consumable credentials, access counts and/or usage bounds. For such policies, and for the strong authentication protocols supporting them, one may resort to sub-structural (e.g., linear or affine) logics [17,25] for specification. Correspondingly, typing systems with linear (or affine) refinements may be employed to achieve static accounts of the desired safety proofs.

In our earlier work in [12], we made a first step towards the design of a sound system for resource-sensitive authorization, drawing on techniques from typing systems for authentication and an affine extension of existing refinement typing systems. Here we make a step further, and show that the invariants that were enforced by means of ad-hoc mechanisms in our original proposal can be internalized into the underlying affine logical system, and expressed directly as proof obligations for the logic. Besides shedding new light on the logical foundations of the cryptographic patterns for authentication and distributed authorization, the new characterization is interesting, and promising, as it leads to a more modular and more powerful typing system.

Plan of the paper. Section 2 reviews the background material. Section 3 gives an overview of our approach. Section 4 provides a detailed description of the type system and its main properties. Section 5 demonstrates the effectiveness of the type system with two small, yet significant, examples. Section 6 concludes.

2 Background

We give a brief review of the relevant components of our approach: affine logics for policy specification, applied pi-calculus for protocol description, and refinement typing systems for analysis and verification.

Affine logic. We focus on the following fragment of intuitionistic affine logic [25]:

$$F \ ::= \ A \ \mid \ F \otimes F \ \mid \ F \multimap F \ \mid \ \forall x.F \ \mid \ !F$$

This is the multiplicative fragment of affine logic, extended with the exponential modality to express persistent truths. We presuppose an underlying signature of predicate symbols which includes the binary equality predicate, and a countably infinite set of terms. Atomic formulas, noted A in the above productions, are built around predicates applied to terms, as in $p(M_1, \ldots, M_n)$; term equality uses infix notation, as in $M = N$. We assume familiarity with the resource interpretation of linear logic, by which each formula denotes a resource which is consumed once used in a derivation. In sequent calculus presentations of linear

logic, that is achieved by dispensing with the structural rules of weakening and contraction, and by a careful manipulation of the environment, as exemplified in two representative rules, below:

$$\frac{\Gamma_1 \vdash F \qquad \Gamma_2 \vdash G}{\Gamma_1, \Gamma_2 \vdash F \otimes G}\ (\otimes\text{-Right}) \qquad \frac{\Gamma_1 \vdash F \qquad \Gamma_2, G \vdash H}{\Gamma_1, F \multimap G, \Gamma_2 \vdash H}\ (\multimap\text{-Left})$$

Affine logic is a variant of linear logic which admits the weakening rule, whereby $\Gamma, F \vdash G$ is derivable when so is $\Gamma \vdash G$. As a result, proofs in affine logics must use each formula *at most* once (as to opposed to *exactly* once as in linear logic).

Applied pi-calculus. We specify protocols in a dialect of the applied pi-calculus [2], in which destructors are only used in let-expressions and may not occur in arbitrary terms [8]. We presuppose an underlying set of constructors and two countable sets of names (a, b, c, k, m, n) and variables (w, x, y, z), and let u range over names or variables uniformly. The syntax of terms M, N is as follows:

$$\begin{array}{rcl} M, N & ::= & a \mid x \mid ek(M) \mid vk(M) \mid inl(M) \mid inr(M) \mid (M, N) \\ & \mid & enc(M, N) \mid sign(M, N) \mid senc(M, N). \end{array}$$

Unary constructors include *ek* and *vk* to form encryption/verification keys from the corresponding decryption/signing keys, and *inl* and *inr* to construct tagged unions (see Section 4). Binary constructors comprise pairs and *senc*, *enc* and *sign* for symmetric, asymmetric encryption and digital signature, respectively. Destructors, ranged over by g, are partial functions to decompose terms. They include the unary *casel* and *caser* to deconstruct tagged unions, *sdec*, *dec* and *ver* for symmetric, asymmetric decryption and signature verification, respectively. For technical reasons, pairs are not decomposed by destructors, but with pattern matching within a specific process form (discussed below). Destructor evaluation may succeed and return a term, noted $g(\widetilde{M}) \Downarrow N$, or fail. The semantics of destructors is as expected, e.g., we have $dec(enc(M, ek(K)), K) \Downarrow M$ and $ver(sign(N, K'), vk(K')) \Downarrow N$.

The syntax of processes P, Q is defined as follows, in terms of two syntactic categories of *actions* A, and proper *processes* P, Q (the distinction is technically convenient in the definition of the typing rules):

$$\begin{array}{rcl} A & ::= & \mathbf{0} \mid in(M, x).P \mid *in(M, x).P \mid out(M, N).P \mid A|A \\ & \mid & if\ (M = N)\ then\ P\ else\ Q \mid let\ (x, y) = M\ in\ P\ else\ Q \\ & \mid & let\ x = g(\widetilde{M})\ in\ P\ else\ Q \mid \ll F. \\ P, Q & ::= & A \mid P|Q \mid new\ a : T.P \mid \gg F. \end{array}$$

The scope of names and variables is delimited by restrictions, inputs and let expressions. The notions of free names and variables, denoted by *fn* and *fv* respectively, arise as expected. A process P is *closed* when $fv(P) = \emptyset$. Processes evolve according to the reduction relation $P \to Q$ ($\to^*$ denotes the reflexive and transitive closure of $\to$). The definition of reduction is standard: $\mathbf{0}$ is the

stuck process; $new\ a : T.P$ creates a fresh name a and behaves as P; $in(M, x).P$ waits for a message N on channel M and then behaves as $P\{N/x\}$; $*in(M, x).P$ acts as an unbounded replication of $in(M, x).P$; $out(M, N).P$ outputs N on M, synchronously, and then behaves as P; $P|Q$ is the parallel composition of P and Q; $if\ (M = N)\ then\ P\ else\ Q$ reduces to P if M is syntactically equal to N, to Q otherwise; $let\ (x, y) = M\ in\ P\ else\ Q$ behaves as $P\{M_1/x\}\{M_2/y\}$ when M is (M_1, M_2), as Q otherwise; finally, $let\ x = g(\widetilde{M})\ in\ P\ else\ Q$ acts as $P\{N/x\}$ if $g(\widetilde{M}) \Downarrow N$, as Q otherwise. Assumptions $(\gg F)$ and assertions $(\ll F)$ are inert process forms, built around the formulas of our affine logic, that express policy annotations.

Definition 1 (Safety). *A closed process P is* safe *iff whenever $P \rightarrow^* new\ \widetilde{a} : \widetilde{T}.(\ll G_1 \mid \ldots \mid \ll G_n \mid Q)$ one has $Q \equiv \gg F_1 \mid \ldots \mid \gg F_m \mid A$, with A containing no top-level assertions, and $F_1, \ldots, F_m \vdash G_1 \otimes \ldots \otimes G_n$.*

Unlike most of the existing definitions of safety [15,4,6,7], we take the tensor product of *all* the active assertions. That is required to remain faithful to the chosen logical framework, and enforce an affine use of each active assumption in our safety proofs. The definition extends readily to account for the presence of opponents. We define an opponent as a closed, *Un*-typed (cf. Section 4) process that does not contain any assertion.

Definition 2 (Robust Safety). *A closed process P is* robustly safe *iff $P \mid O$ is safe for every opponent O.*

The restriction to *Un*-typed processes is standard and does not involve any loss of generality; we ban assertions from opponent code, because otherwise an opponent could trivially break the safety property we target.

Refinement typing systems for distributed authorization. We review the main ideas and intuitions with a simple example, inspired by [15], of an on-line bookstore system governed by the following authorization policy:

$$\mathcal{A} \triangleq\ !\forall u, b.\, (Order(u, b) \multimap Clear(u, b))$$

The policy is stated as a persistent (reusable) formula: it establishes that an e-book order can be cleared for a user if that user has indeed ordered the e-book (note, in particular, the use of multiplicative implication to express the desired *injective* correspondence between order and clearance). The components are described by the annotated code below[1]

$$\begin{array}{ll} user & :: \gg Order(user, book) \mid out(net, (user, sign((user, book), k_{user}))) \\ bookstore & :: *in(net, (x_u, y)). \\ & \quad let\ (x_{vk}, x_{ek}) = keys(x_u)\ in\ let\ (x_u, x_b) = ver(y, x_{vk})\ in \\ & \quad \ll Clear(x_u, x_b) \mid out(net, enc(url(x_b), x_{ek})) \end{array}$$

[1] We assume that the bookstore keeps track of the public (encryption and verification) keys of each registered user, so that $keys(user) = (vk(k_{user}), ek(k_{user}))$. Also, for readability, we abuse the notation and use pattern-matching on input.

Consider now the system $\gg \mathcal{A} \mid user \mid bookstore$. In a system run, *user* authenticates her order of *book* to the *bookstore* by signing the request, and correspondingly *assumes* the formula *Order*(*user,book*) to declare her intention. The bookstore, in turn, receives the data from its input channel, verifies the signature and *asserts* the formula *Clear*(*user,book*) as a guard to clear the order. The system is safe, as the guard is entailed by the policy $\mathcal{A}$ and the assumption *Order*(*user,book*), which is available when the assertion *Clear*(*user,book*) is unleashed at top-level.

In a refinement type system, a safety proof can be derived by relying on the types of cryptographic keys. Key types have the general form $Key(\{x : T \mid C(x)\})$ representing keys with payload of (structural) type T for which the formula (credential) C may be assumed to hold. Given $k : Key(\{x : T \mid C(x)\})$, packaging a value m with k, as in $enc(m, k)$, typechecks provided that the formula $C(m)$ can be proved at the source site, using the assumptions available in the typing context associated with that site. Dually, extracting the payload from an encrypted packet, as in $dec(y, k)$, justifies the assumption of the formula $C(y)$ conveyed by the key type, hence the use of $C(y)$ in a proof of the credentials acting as access guards. In our example, we may use the type $Key(x_u : T_u, \{x_b : T_b \mid Order(x_u, x_b)\})$ for k_{user} to convey the credential $Order(x_u, x_b)$ from the user process to the bookstore site, and derive a static safety proof based on that.

3 Authorization, Authentication and Affine Refinements

Continuing with our example, consider extending the system with the new component:

$$dup :: *\, in(net, x).(out(net, x) \mid out(net, x))$$

to form the composition $\gg \mathcal{A} \mid user \mid bookstore \mid dup$. Unlike the original system, the extended one is unsafe (hence the original is not robustly safe), as the presence of the *dup* process causes *bookstore* to clear each order twice. Clearly, the problem arises from the absence of an authentication mechanism providing adequate guarantees of timeliness for the orders. The effect is captured by our definition of safety. To see that, first observe that a reduction sequence exists that unleashes two assertions *Clear*(*user,book*) for a single assumption *Order*(*user,book*). Then, note that $Order(user, book), \mathcal{A} \not\vdash Clear(user,book) \otimes Clear(user,book)$, as the assumption *Order*(*user,book*) is consumed in the proof of either of the two clearing assertions at the bookstore, thereby causing the derivation of the second assertion to fail.

Strong authentication is a long studied problem in static protocol analysis and verification. First formalized as *injective agreement* in [23], it has subsequently been approached with a variety of typing techniques, targeted at the analysis of various low-level mechanisms (timestamps, nonce handshakes, and session keys) [18,19,9,10,20,11]. Though such mechanisms are fundamental building blocks for distributed authorization frameworks, little (if any) of the work on strong authentication has resurfaced in existing typing systems for authorization [14,15,4,6,7]. Our proposal in the present paper aims at reconciling these

two streams of research, by building a unifying foundation for authentication and authorization, based on an affine refinement type system.

Just like traditional refinement typing systems, we employ key types to capture the transfer of authorization credentials within a protocol. However, since our refinements are affine formulas, we must control the use of keys so as to protect against any unintended duplication of the refinements, upon decryption. Specifically, when transferring a message $m : T$ packaged with, say, $k : Key(\{x : T \mid C(x)\})$ we must ensure that each extraction of $C(x)$ by the receiver correspond to a derivation of $C(m)$ at the source site. To accomplish that, our type system protects the refinement $C(x)$ with a *guard*, as in:

$$k : Key(w : U, \{x : T \mid G(w) \multimap C(x)\})$$

where $G(w)$ is a receiver-controlled formula that must be proved to derive the credential $C(x)$. In a nonce-handshake protocol, w represents the challenger-generated nonce, call it n, and $G(n)$ is the corresponding guard assumed by the challenger. A responder willing to prove the possession of a credential $C(m)$ for the payload m will be able to do so, as follows. Upon receiving the nonce, the responder transmits the pair (n, m) under the key k: that's possible when the responder has (or may derive) $C(m)$, because $C(m) \vdash G(n) \multimap C(m)$ in affine logic. At the challenger end, extracting the payload (w, x) and checking that $w = n$ makes it possible to derive $C(x)$, as $G(n), w = n, G(w) \multimap C(x) \vdash C(x)$. If we can ensure that $G(n)$ can be proved at most once, we also ensure that $C(x)$ is derived at most once, as desired.

Though the details vary for the different low-level mechanisms, the core intuitions we just outlined apply uniformly: data exchanged over the network is inherently exposed to replays, hence their credentials must be protected so that copying the data does not duplicate the credentials. In the type system, that is accomplished by embedding the credentials into multiplicative implications guarded by system-controlled formulas, which are built around *reserved* predicate symbols, and are guaranteed to be assumed in at most one position in the protocol code. As a result, key refinements become safely *copyable*, as the system-controlled guards guarantee that the credentials they embed are unleashed at most once, irrespective of any duplication the refinement may undergo.

In the next section, we provide full details of these mechanisms.

4 The Type System

The syntax of types T, U, V is defined by the following grammar.

$$\begin{aligned} T, U, V ::= {} & Un \mid Private \mid Ch(T) \mid \{x : T \mid F\} \mid (x : T, U) \mid T + U \\ \mid {} & \eta Key_{(x)}(T) \quad \eta \in \{Enc, Dec, Sig, Ver, Sym\} \\ \mid {} & \eta Pkt_{(x)}(T) \quad \eta \in \{Enc, Sig, Sym\}. \end{aligned}$$

The variable x is bound in $\{x : T \mid F\}$ with scope F, in $(x : T, U)$ with scope U, and in $\eta Key_{(x)}(T)$ and $\eta Pkt_{(x)}(T)$ with scope T. Un is the type of data

Table 1. Well-formed types and environments

$$\frac{\mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma) \quad \Gamma \vdash \diamond \quad T \neq \eta \mathit{Key}_{(x)}(U)}{\Gamma \vdash T} \text{ (Type-Base)}$$

$$\frac{\mathit{fnfv}(T) \setminus \{x\} \subseteq \mathit{dom}(\Gamma) \quad \Gamma \vdash \diamond \quad T \text{ copyable}}{\Gamma \vdash \eta \mathit{Key}_{(x)}(T)} \text{ (Type-Key)}$$

$$\frac{}{\varepsilon \vdash \diamond} \text{ (Env-Empty)}$$

$$\frac{\Gamma \vdash \diamond \quad \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)}{\Gamma, F \vdash \diamond} \text{ (Env-Form)}$$

$$\frac{u \notin \mathit{dom}(\Gamma) \quad \Gamma \vdash T \quad T \text{ a copyable, non-refinement type}}{\Gamma, u : T \vdash \diamond} \text{ (Env-Bind)}$$

coming from / flowing to the opponent (standard since [1]). *Private* is the type of untainted, secret data; $\mathit{Ch}(T)$ the type of channels with T payload; $\{x : T \mid F\}$ the type of $M : T$ such that $F\{M/x\}$ holds. A pair (M, N) has type $(x : T, U)$ if M has type T and N has type $U\{M/x\}$. A term of type $T + U$ is either $\mathit{inl}(M)$ where M has type T, or $\mathit{inr}(M)$ where M has type U. Finally, we devise two new types for cryptographic material – $\eta \mathit{Key}_{(x)}(T)$ and $\eta \mathit{Pkt}_{(x)}(T)$ – for keys with T payload and ciphertexts with T payload, respectively[2]. In both cases the binder x acts as a placeholder for the encryption key or the verification key: this technical device, first proposed in [13], is very effective and convenient to achieve a uniform treatment for nonce handshakes and session keys in our type system.

Typing environments and well-formed types. Typing environments, noted Γ, collect bindings for names and variables, as usual, and formulas occurring in assumptions. The domain of Γ, noted $\mathit{dom}(\Gamma)$, is the set of the values bound to a type in Γ. $\mathit{forms}(\Gamma)$ denotes the multiset of the formulas occurring in Γ. $\mathit{bindings}(\Gamma)$ is the environment obtained by erasing all the formulas from Γ. ε is the empty environment. Well-formed types and environments are defined in terms of the notions of *copyable* formulas and types, given below.

Definition 3 (Copyable Formulas and Types). *A formula F is copyable if it has either of the two forms $p(M_1, \ldots, M_n) \multimap F'$ with p reserved, or $!F'$. Copyable types, then, are defined inductively as follows:*

- *$\mathit{Un}, \mathit{Private}, \mathit{Ch}(T), \eta \mathit{Key}_{(x)}(T)$ and $\eta \mathit{Pkt}_{(x)}(T)$ are copyable;*
- *$\{x : T \mid F\}$, $(x : T, U)$ and $T + U$ are copyable if so are T, U and F.*

The rules for types and environments are in Table 1. Notice that, by (Type-Key), well-formed key types can only convey *copyable* payloads: hence, formulas may occur as refinements of key types only if they are guarded by system-reserved predicates, or they are prefixed by a bang modality: in the former case, the guard protects them against uncontrolled replication, in the latter, replication

[2] When x does not occur in T we often omit the binder and write simply $\eta \mathit{Key}(T)$, and $\eta \mathit{Pkt}(T)$, to ease the notation.

is harmless as the formula may be duplicated in the logic as well. A similar mechanism is enforced in the type system for injective agreement in [21].

Type environments only include bindings for non-refinement typed names and variables: that does not involve any loss of expressive power, as we may simply define the environment $\Gamma, u : \{x : T \mid F\}$ as $\Gamma, u : T, F\{u/x\}$. Also, type bindings must introduce copyable types, to protect against the unintended duplication of affine refinements (occurring in dependent pair or disjoint union types) upon the *environment splitting* distinctive of sub-structural type systems. We say that Γ splits as Γ_1 and Γ_2 ($\Gamma = \Gamma_1 \bullet \Gamma_2$) when $\mathit{bindings}(\Gamma) = \mathit{bindings}(\Gamma_1) = \mathit{bindings}(\Gamma_2)$ and $\mathit{forms}(\Gamma) = \mathit{forms}(\Gamma_1), \mathit{forms}(\Gamma_2)$. More in general, we write $\Gamma = \Gamma_1 \bullet \ldots \bullet \Gamma_n$ when $\Gamma = \Gamma' \bullet \Gamma_n$ and $\Gamma' = \Gamma_1 \bullet \ldots \bullet \Gamma_{n-1}$. Finally, we write $\Gamma \vdash F$ whenever $\mathit{forms}(\Gamma) \vdash F$, provided that $\Gamma \vdash \diamond$ and $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$.

Table 2. Typing rules for terms

$$\frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \text{ (Term-Env)}$$

$$\frac{\Gamma \vdash M : \mathit{DecKey}_{(x)}(T)}{\Gamma \vdash \mathit{ek}(M) : \mathit{EncKey}_{(x)}(T)} \text{ (Term-EncKey)}$$

$$\frac{\Gamma \vdash M : \mathit{SigKey}_{(x)}(T)}{\Gamma \vdash \mathit{vk}(M) : \mathit{VerKey}_{(x)}(T)} \text{ (Term-VerKey)}$$

$$\frac{\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash N : U\{M/x\}}{\Gamma_1 \bullet \Gamma_2 \vdash (M, N) : (x : T, U)} \text{ (Term-Pair)}$$

$$\frac{\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash F\{M/x\}}{\Gamma_1 \bullet \Gamma_2 \vdash M : \{x : T \mid F\}} \text{ (Term-Refine)}$$

$$\frac{\Gamma_1 \vdash M : T\{N/x\} \qquad \Gamma_2 \vdash N : \mathit{EncKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{enc}(M, N) : \mathit{EncPkt}_{(x)}(T)} \text{ (Term-AEnc)}$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash U}{\Gamma \vdash \mathit{inl}(M) : T + U} \text{ (Term-Left)}$$

$$\frac{\Gamma_1 \vdash M : T\{\mathit{vk}(N)/x\} \qquad \Gamma_2 \vdash N : \mathit{SigKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{sign}(M, N) : \mathit{SigPkt}_{(x)}(T)} \text{ (Term-Sign)}$$

$$\frac{\Gamma \vdash M : U \qquad \Gamma \vdash T}{\Gamma \vdash \mathit{inr}(M) : T + U} \text{ (Term-Right)}$$

$$\frac{\Gamma_1 \vdash M : T\{N/x\} \qquad \Gamma_2 \vdash N : \mathit{SymKey}_{(x)}(T)}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{senc}(M, N) : \mathit{SymPkt}_{(x)}(T)} \text{ (Term-SEnc)}$$

Typing rules for terms. Table 2 details the typing rules for terms. We omit the rules that define the subtype relation as well as the kinding rules for tainted and public types: all details can be found in [12]. The novel rules for cryptographic packets (Term-AEnc), (Term-Sign) and (Term-SEnc) exploit a form of dependent typing to track the shared information between encryption and decryption keys (respectively, signing and verification keys) [13].

Typing rules for processes. Table 3 presents the typing rules for actions and processes: we only discuss the most interesting points. The side-condition to

Table 3. Typing rules for actions and processes

$$\text{(Proc-Out)}\quad \frac{\Gamma_1 \vdash M : Ch(T) \qquad \Gamma_2 \vdash N : T \qquad \Gamma_3 \vdash P}{\Gamma_1 \bullet \Gamma_2 \bullet \Gamma_3 \vdash out(M,N).P}$$

$$\text{(Proc-In)}\quad \frac{\Gamma_1 \vdash M : Ch(T) \qquad \Gamma_2, x : T \vdash P}{\Gamma_1 \bullet \Gamma_2 \vdash in(M,x).P}$$

$$\text{(Proc-Repl)}\quad \frac{\Gamma_1 \vdash M : Ch(T) \qquad \Gamma_2, x : T \vdash P \qquad \Gamma_2 \text{ copyable}}{\Gamma_1 \bullet \Gamma_2 \vdash *in(M,x).P}$$

$$\text{(Proc-Stop)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

$$\text{(Proc-Cond)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash M : T \qquad \Gamma_2 \vdash N : T \\ \Gamma_3, !(M = N) \vdash P \qquad \Gamma_3 \vdash Q\end{array}}{\Gamma_1 \bullet \Gamma_2 \bullet \Gamma_3 \vdash \mathit{if}\ (M = N)\ \mathit{then}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-Split)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash M : (x : T, U) \\ \Gamma_2, x : T, y : U \vdash P \qquad \Gamma_2 \vdash Q\end{array}}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{let}\ (x,y) = M\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-Case-Left)}\quad \frac{\Gamma_1 \vdash M : T + U \qquad \Gamma_2, x : T \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{let}\ x = \mathit{casel}(M)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-Case-Right)}\quad \frac{\Gamma_1 \vdash M : T + U \qquad \Gamma_2, x : U \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \bullet \Gamma_2 \vdash \mathit{let}\ x = \mathit{caser}(M)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-ADec)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash M : EncPkt_{(y)}(T) \\ \Gamma_2 \vdash N : DecKey_{(y)}(T) \qquad \Gamma_3, x : T\{ek(N)/y\} \vdash P \qquad \Gamma_3 \vdash Q\end{array}}{\Gamma_1 \bullet \Gamma_2 \bullet \Gamma_3 \vdash \mathit{let}\ x = dec(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-Ver)}\quad \frac{\Gamma_1 \vdash M : SigPkt_{(y)}(T) \qquad \Gamma_2 \vdash N : VerKey_{(y)}(T) \qquad \Gamma_3, x : T\{N/y\} \vdash P \qquad \Gamma_3 \vdash Q}{\Gamma_1 \bullet \Gamma_2 \bullet \Gamma_3 \vdash \mathit{let}\ x = ver(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-SDec)}\quad \frac{\begin{array}{c}\Gamma_1 \vdash M : SymPkt_{(y)}(T) \\ \Gamma_2 \vdash N : SymKey_{(y)}(T) \qquad \Gamma_3, x : T\{N/y\} \vdash P \qquad \Gamma_3 \vdash Q\end{array}}{\Gamma_1 \bullet \Gamma_2 \bullet \Gamma_3 \vdash \mathit{let}\ x = sdec(M,N)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

$$\text{(Proc-Assert)}\quad \frac{\Gamma \vdash F}{\Gamma \vdash \ll F}$$

$$\text{(Proc-Par)}\quad \frac{\Gamma_1 \vdash A_1 \qquad \Gamma_2 \vdash A_2}{\Gamma_1 \bullet \Gamma_2 \vdash A_1 \mid A_2}$$

$$\text{(Proc-Extr)}\quad \frac{P \rightsquigarrow [\Gamma' \parallel A] \qquad \Gamma, \Gamma' \vdash A \qquad \mathit{fnfv}(P) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash P}$$

$$\text{(Weak)}\quad \frac{\Gamma_1, \Gamma_2 \vdash P \qquad \Gamma_1, F, \Gamma_2 \vdash \diamond}{\Gamma_1, F, \Gamma_2 \vdash P}$$

$$\text{(Contr)}\quad \frac{\Gamma_1, !F, !F, \Gamma_2 \vdash P}{\Gamma_1, !F, \Gamma_2 \vdash P}$$

$$(\otimes\text{-Left})\quad \frac{\Gamma_1, F, G, \Gamma_2 \vdash P}{\Gamma_1, F \otimes G, \Gamma_2 \vdash P}$$

$$(\multimap\text{-Left})\quad \frac{\Gamma_1' \vdash F \qquad \Gamma_2', G \vdash P \qquad \Gamma_1, \Gamma_2 = \Gamma_1' \bullet \Gamma_2'}{\Gamma_1, F \multimap G, \Gamma_2 \vdash P}$$

$$(\forall\text{-Left})\quad \frac{\Gamma_1, F(M), \Gamma_2 \vdash P}{\Gamma_1, \forall x.F(x), \Gamma_2 \vdash P}$$

$$(!\text{-Left})\quad \frac{\Gamma_1, F, \Gamma_2 \vdash P}{\Gamma_1, !F, \Gamma_2 \vdash P}$$

Table 4. The extraction relation $P \rightsquigarrow [\Gamma \parallel A]$

$$\text{(Extr-New)}\quad \frac{P \rightsquigarrow [\Gamma \parallel A] \qquad T \in \{Un, Private, Ch(U), SigKey_{(x)}(U), DecKey_{(x)}(U), SymKey_{(x)}(U)\}}{new\ a : T.P \rightsquigarrow [a : T, \Gamma \parallel A]}$$

$$\text{(Extr-Assume)}\quad \frac{}{\gg F \rightsquigarrow [F \parallel \mathbf{0}]} \qquad \text{(Extr-Empty)}\quad \frac{}{A \rightsquigarrow [\varepsilon \parallel A]} \qquad \text{(Extr-Par)}\quad \frac{P \rightsquigarrow [\Gamma_P \parallel A_P] \qquad Q \rightsquigarrow [\Gamma_Q \parallel A_Q]}{P \mid Q \rightsquigarrow [\Gamma_P, \Gamma_Q \parallel A_P \mid A_Q]}$$

(Proc-Repl-In), requiring that the continuation typechecks in a copyable environment, is needed for subject reduction as replicated processes may spawn an unbounded number of copies of their continuation [22]. (Proc-Cond) keeps track of the equality between two terms in the successful branch of a conditional check: since this information can be used an arbitrary number of times, it is made exponential. The rules for cryptography (Proc-ADec), (Proc-Ver) and (Proc-SDec) mirror the idea of the corresponding rules for cryptographic packets in Table 2. (Proc-Extr) is the only rule for proper processes, which are typechecked by first extracting the top-level restrictions and assumptions into a typing environment, and then using that environment to typecheck the residual action process. Extracting the assumptions is needed to protect against using them more than once in the same type derivation; extracting the restrictions keeps all names in scope (cf. Table 4).

The type system is completed by a set of structural rules to enable weakening and contraction, as well as the manipulation of the logical connectives in typing derivations. Since proofs in sub-structural logics require careful management of the environment, like in [24] these rules are needed to improve the expressiveness of our framework.

Theorem 1 (Robust Safety). *Let P be a closed process such that $fn(P) = \{a_1, \ldots, a_n\}$ and let $a_1 : Un, \ldots, a_n : Un \vdash P$, then P is robustly safe.*

5 Case Study: Cryptographic Sessions

We show the type system at work on two small, but realistic case studies that demonstrate the flexibility and effectiveness of our framework.

We start introducing additional notation for the system-controlled guards. First, we presuppose two system predicates $\text{KEY}(\cdot)$ and $\text{NONCE}(\cdot)$, to serve as guards in the refinements associated with session keys, and (long-term) keys in nonce-based protocols, respectively. As discussed earlier, $\text{NONCE}(\cdot)$ guards are used with keys such as $k : \eta Key(w : U, \{x : T \mid \text{NONCE}(w) \multimap C(x)\})$ to exchange a nonce $n : U$, packaged with a payload $M : T$ such that $C(M)$. The underlying verification pattern presupposes that (at most one occurrence of) the formula

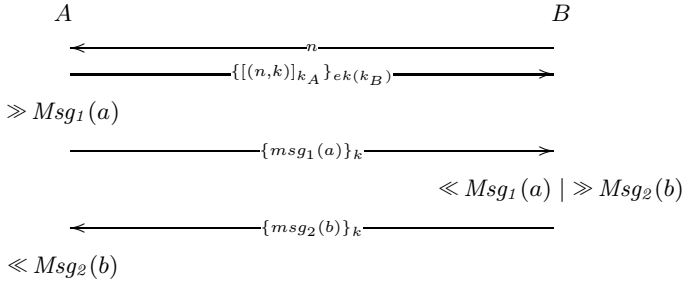
$\textsc{nonce}(n)$ be available at the receiver to obtain a proof that the sender possesses the credential $C(M)$. The pattern for session keys has the same rationale. In that case, it is built around keys such as $k : \mathit{SymKey}_{(y)}(\{x : T \mid \textsc{key}(y) \multimap C(x)\})$ intended for the exchange of payloads $M : T$ such that $C(M)$, with $\textsc{key}(k)$ acting as the controlling guard, predicating on the key k itself through the binder y.

Both patterns can be generalized to enable multiple checks of the same nonce and multiple uses of the same key within a session. That is achieved with key types of the form $\eta\mathit{Key}_{(y)}(\sum_{i=1}^{\ell}(w : U, \{x : T_i \mid \textsc{nonce}(w, M_i) \multimap C_i(x)\})$ and similarly $\mathit{SymKey}_{(y)}(\sum_{i=1}^{\ell}\{x : T_i \mid \textsc{key}(y, M_i) \multimap C_i(x)\})$, used in conjunction with assumptions of the form $\textsc{nonce}(n, M_i)$ and $\textsc{key}(k, M_i)$, respectively. The M_i's are closed, pairwise distinct terms that serve as tags to mark the ℓ different program points where the same nonce may be checked (the same key used)[3]. The following notation helps structure our specification patterns:

$$\begin{aligned}&\textsc{def}\ k = \textsc{SessionKey}[\textstyle\sum_{i=1}^{\ell}(M_i, T_i, C_i(x))]\ \textsc{in}\ P \triangleq\\ &\qquad\qquad \mathit{new}(k : \mathit{SymKey}_{(y)}(\textstyle\sum_{i=1}^{\ell}\{x : T_i \mid \textsc{key}(y, M_i) \multimap C_i(x)\})).\\ &\qquad\qquad (\gg \textsc{key}(k, M_1) \mid \cdots \mid \gg \textsc{key}(k, M_\ell) \mid P)\end{aligned}$$

$$\begin{aligned}&\textsc{def}\ n = \textsc{nonce}[U, \textstyle\sum_{i=1}^{\ell} M_i]\ \textsc{in}\ P \triangleq\ \mathit{new}(n : U).\\ &\qquad\qquad (\gg \textsc{nonce}(n, M_1) \mid \ldots \mid \gg \textsc{nonce}(n, M_\ell) \mid P)\end{aligned}$$

Bounded sessions. Our first example is a protocol that implements a bounded session, built around a finite, and fixed, flow of messages. It involves two agents that perform a nonce-handshake to exchange a symmetric key k and then use the key in a session that exchanges two messages, as shown in the diagram below:



There is no global policy defined here, and the assumptions and assertions of the specification are only meant to track the session steps, ensuring the timeliness of the messages exchanged. The interesting part of the encoding in our applied pi-calculus is in the choice of key types. We start with the type T_k of the shared session key k. Assuming a and b may be given type Un, we define:

$$T_k = \mathit{SymKey}_{(y)}(\textstyle\sum_{i=1}^{2}\{x : \mathit{Un} \mid \textsc{key}(y, i) \multimap \mathit{Msg}_i(x)\})$$

T_k presupposes two uses of k, to extract the two different types of messages conveying the affine formulas $\mathit{Msg}_1(a)$ and $\mathit{Msg}_2(b)$. In the applied pi-calculus

[3] When $\ell \leq 1$, and we need no tag, we simply omit them from the notation.

specification below, this type is introduced together with the two assumptions for the guard predicates $\text{KEY}(k, 1)$ and $\text{KEY}(k, 2)$ (we use natural numbers as the closed terms serving as tags). Based on T_k we may then construct the type T_{k_A} of A's signing key: $T_{k_A} = \mathit{SigKey}(y : \mathit{Un}, \{x : T_k \mid \text{NONCE}(y) \multimap \text{KEY}(x, 1)\})$. Notice that the credential protected by the nonce is the system guard $\text{KEY}(k, 1)$ that will allow B to use the shared k and extract the credential $\mathit{Msg}_1(a)$ marking the completion of the first exchange. The other guard, $\text{KEY}(k, 2)$ remains with A itself, to enable A's own use of the key at the completion of the protocol.

We are ready to define the protocol code: to ease the notation, we coalesce subsequent destructor applications in a single let statement:

$$
\begin{array}{l}
A \triangleq \mathit{in}(\mathit{net}, x_n).\\
\quad \text{DEF } k = \text{SESSIONKEY}[(1, \mathit{Un}, \mathit{Msg}_1(x)) + (2, \mathit{Un}, \mathit{Msg}_2(x))] \text{ IN}\\
\quad \mathit{out}(\mathit{net}, \mathit{enc}(\mathit{sign}((x_n, k), k_A), \mathit{ek}(k_B))).\\
\quad \gg \mathit{Msg}_1(a)\\
\quad \mid \mathit{out}(\mathit{net}, \mathit{senc}(\mathit{inl}(a), k)).\\
\quad\ \ \mathit{in}(\mathit{net}, x).\mathit{let}\ y = \mathit{caser}(\mathit{sdec}(x, k))\ \mathit{in} \ll \mathit{Msg}_2(y)\\
\\
B \triangleq \text{DEF } n = \text{NONCE}[\mathit{Un}] \text{ IN } \mathit{out}(\mathit{net}, n).\\
\quad \mathit{in}(\mathit{net}, x).\\
\quad \mathit{let}\ (y_n, y_k) = \mathit{ver}(\mathit{dec}(x, k_B), \mathit{vk}(k_A))\ \mathit{in}\\
\quad \mathit{if}\ (y_n = n)\ \mathit{then}\\
\quad \mathit{in}(\mathit{net}, z).\\
\quad \mathit{let}\ w = \mathit{casel}(\mathit{sdec}(z, y_k))\ \mathit{in} \ll \mathit{Msg}_1(w)\\
\quad \mid \gg \mathit{Msg}_2(b) \mid \mathit{out}(\mathit{net}, \mathit{senc}(\mathit{inr}(b), y_k))
\end{array}
$$

Typechecking the code goes as follows: we only comment on the most important steps, looking at the code of A and B separately.

At A's side, introducing the shared key k extends the typing environment with the assumptions $\text{KEY}(k, 1)$ and $\text{KEY}(k, 2)$. Then, to sign k with k_A, one derives $(x_n, k) : (y{:}\mathit{Un}, \{x{:}T_k \mid \text{NONCE}(y) \multimap \text{KEY}(x, 1)\})$ by (TERM-PAIR), (TERM-REFINE), and a proof of $\text{KEY}(k, 1), \text{KEY}(k, 2) \vdash \text{NONCE}(x_n) \multimap \text{KEY}(k, 1)$. Similarly, to send the first message encrypted under k, an application of (TERM-SENC) requires one to show $a : \{x : \mathit{Un} \mid \text{KEY}(k, 1) \multimap \mathit{Msg}_1(x)\}$, which in turn derives by (TERM-REFINE) based on a proof of $\mathit{Msg}_1(a) \vdash \text{KEY}(k, 1) \multimap \mathit{Msg}_1(a)$.

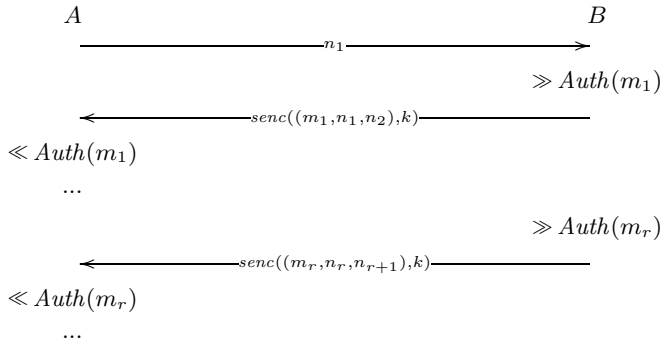
At B's side, creating the fresh nonce n extends the environment with the guard $\text{NONCE}(n)$. When B gets a response, she decrypts it and verifies the signature to extract the pair (y_n, y_k) and the formula $\text{NONCE}(y_n) \multimap \text{KEY}(y_k, 1)$ by (PROC-VER). Then, checking y_n against n extends the typing environment with the equality $!(y_2 = n)$, and the analysis of B proceeds breaking the implication as follows:

$$
\frac{\cdots, \text{NONCE}(n), !(y_n = n) \vdash \text{NONCE}(y_n) \qquad \cdots, \text{KEY}(y_k, 1) \vdash \mathit{in}(\mathit{net}, z).\cdots}{\cdots, \text{NONCE}(n), !(y_n = n), \text{NONCE}(y_n) \multimap \text{KEY}(y_k, 1) \vdash \mathit{in}(\mathit{net}, z).\cdots}\ (\multimap\text{-LEFT})
$$

The left premise is derived directly in the affine logical system. The right premise, in turn, leaves the guard $\text{KEY}(y_k, 1)$ for B to use y_k (the session key) in the continuation. Indeed, when the packet z reaches B, it is decrypted as w using

key y_k (and the sum left destructor). By applying (Proc-SDec) (and subsequently (Proc-Case-Left)), the environment is extended with the information $w : Un, \text{KEY}(y_k, 1) \multimap Msg_1(w)$. Note that the actual parameter y_k in the code replaces the formal parameter y in type T_k upon decryption, as dictated by (Proc-SDec). Now, consuming the guard $\text{KEY}(y_k, 1)$, we may derive the assertion $Msg_1(w)$ as desired.

Unbounded sessions. Though effective, the use of session keys illustrated in the previous example only applies to sessions in which the key is used a predefined (and finite) number of times. The next protocol, proposed in [18], shows how to account for unbounded sessions, exchanging an arbitrarily long stream of timely messages.



Unlike the previous protocol, here the message flow is always in the same direction, from B to A, and the message exchanged at step i conveys a payload m_i, which is authenticated by consuming nonce n_i, and a fresh nonce n_{i+1}, which is used to authenticate the next exchange. Again, the assumptions and assertions of the specification only serve for verifying the timeliness of each exchange. In this case, the timeliness proof relies on the following type for the shared key:

$$k : SymKey_{(y)}(x_1 : Un, (x_2 : Un, \{x_3 : Un \mid \text{NONCE}(x_2) \multimap (Auth(x_1) \otimes \text{NONCE}(x_3))\})).$$

At each use of k for decryption, A consumes the guard on the current nonce to obtain a proof of the expected authorization credential, and the nonce to repeat the process at the subsequent iteration.

The applied pi-calculus code for the protocol is as follows.

$$\begin{array}{ll}
A = & new\ a : Ch(\{x : Un \mid \text{NONCE}(x)\}). \\
& \text{DEF } n = \text{NONCE}[Un] \text{ IN} \\
& out(net, n).(out(a, n) \mid A^*) \\
A^* = & *in(a, x).in(net, y). \\
& \quad let\ (z_1, z_2, z_3) = sdec(y, k)\ in \\
& \quad if\ (z_2 = x)\ then \\
& \quad \ll Auth(z_1) \mid out(a, z_3)
\end{array}
\qquad
\begin{array}{ll}
B = & new\ b : Ch(Un). \\
& in(net, x).(out(b, x) \mid B^*) \\
B^* = & *in(b, x).new\ m : Un. \\
& \quad \gg Auth(m) \\
& \quad \mid \text{DEF } n = \text{NONCE}[Un] \text{ IN} \\
& \quad out(net, senc((m, x, n), k)). \\
& \quad out(b, n)
\end{array}$$

Both agents include replicated sub-processes whose iterations are controlled via synchronization over private channels. While this is a standard practice in the

pi-calculus, a remark is in order on the type chosen for channel a: this type is needed to provide nonce capabilities to the replicated process A^*, since rule (Proc-Repl-In) requires to typecheck this process in a copyable environment. The guard formula $\text{NONCE}(n)$ is used to typecheck the output of n on a, so that the associated capability can be recovered upon an input from the channel.

6 Conclusion

Authentication and authorization have been studied extensively in the literature on protocol verification, yet mostly as independent problems. We have proposed a unifying technique based on a novel affine refinement type system. The approach appears promising, as it supports a modular design of the framework, and is effective in the analysis of interesting classes of authentication protocols and authorization systems.

We are currently investigating the applications of our technique to further authentication mechanisms commonly employed in practice, e.g., timestamps and session identifiers, and to further protocols where the same nonce is checked by different principals. From our initial results, our approach appears to generalize smoothly to all such cases. We are also porting our framework from the applied pi-calculus to RCF [6], a concurrent functional programming language strongly related to F#. By recasting our technique to this new setting, we will be able to conduct our type-based analysis directly on application code.

Acknowledgments. Work partially supported by MIUR Project IPODS "*Interacting Processes in Open-ended Distributed Systems*".

References

1. Abadi, M.: Secrecy by typing in security protocols. Journal of the ACM 46(5), 749–786 (1999)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. 28th Symposium on Principles of Programming Languages (POPL), pp. 104–115. ACM Press (2001)
3. Abadi, M.: Logic in access control. In: Proc. 18th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 228–233. IEEE Computer Society Press (2003)
4. Backes, M., Hriţcu, C., Maffei, M.: Type-checking zero-knowledge. In: 15th ACM Conference on Computer and Communications Security (CCS 2008), pp. 357–370. ACM Press (2008)
5. Bauer, L., Jia, L., Sharma, D.: Constraining credential usage in logic-based access control. In: Proc. 23rd IEEE Symposium on Computer Security Foundations (CSF), pp. 154–168. IEEE Computer Society Press (2010)
6. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: Proc. 21st IEEE Symposium on Computer Security Foundations (CSF), pp. 17–32. IEEE Computer Society Press (2008)

7. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proc. 37th Symposium on Principles of Programming Languages (POPL), pp. 445–456. ACM (2010)
8. Blanchet, B.: From Secrecy to Authenticity in Security Protocols. In: Hermenegildo, M., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 342–359. Springer, Heidelberg (2002)
9. Bugliesi, M., Focardi, R., Maffei, M.: Compositional analysis of authentication protocols. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 140–154. Springer, Heidelberg (2004)
10. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of Typed Analyses of Authentication Protocols. In: Proc. 18th IEEE Computer Security Foundations Workshop (CSFW), pp. 112–125. IEEE Computer Society Press (2005)
11. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. Journal of Computer Security 15(6), 563–617 (2007)
12. Bugliesi, M., Calzavara, S., Eigner, F., Maffei, M.: Resource-aware authorization policies for statically typed cryptographic protocols. In: Proc. 24th IEEE Computer Security Foundations Symposium (CSF), pp. 83–98 (2011)
13. Focardi, R., Maffei, M.: Types for security protocols. In: Formal Models and Techniques for Analyzing Security Protocols. Cryptology and Information Security Series, vol. 5, ch. 7, pp. 143–181. IOS Press (2011)
14. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 141–156. Springer, Heidelberg (2005)
15. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization in distributed systems. In: Proc. 20th IEEE Symposium on Computer Security Foundations (CSF), pp. 31–45. IEEE Computer Society Press (2007)
16. Freeman, T., Pfenning, F.: Refinement types for ML. In: Wise, D.S. (ed.) PLDI, pp. 268–277. ACM (1991)
17. Girard, J.Y.: Linear logic: its syntax and semantics. In: Advances in Linear Logic. London Mathematical Society Lecture Note Series, vol. 22, pp. 3–42 (1995)
18. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. Journal of Computer Security 11(4), 451–519 (2003)
19. Gordon, A.D., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security 12(3), 435–484 (2004)
20. Haack, C., Jeffrey, A.: Timed spi-calculus with types for secrecy and authenticity. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 202–216. Springer, Heidelberg (2005)
21. Haack, C., Jeffrey, A.: Pattern-matching spi-calculus. Information and Computation 204(8), 1195–1263 (2006)
22. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems 21(5), 914–947 (1999)
23. Lowe, G.: A Hierarchy of Authentication Specifications. In: Proc. 10th IEEE Computer Security Foundations Workshop (CSFW), pp. 31–44. IEEE Computer Society Press (1997)
24. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 213–225. ACM Press (2003)
25. Troelstra, A.S.: Lectures on linear logic. CSLI Stanford. Lecture Notes Series, vol. 29 (1992)

Seamless Distributed Computing from the Geometry of Interaction

Olle Fredriksson and Dan R. Ghica

The University of Birmingham, U.K.

Abstract. In this paper we present a seamless approach to writing and compiling distributed code. By "seamless" we mean that the syntax and semantics of the distributed program remain the same as if it was executed on one node only, except for label annotations indicating on what node sub-terms of the program are to be executed. There are no restrictions on how node labels are to be assigned to sub-terms. We show how the paradigmatic (higher-order functional recursive) programming language PCF, extended with node annotations, can be used for this purpose. The compilation technique is directly inspired by game semantics and the Geometry of Interaction.

1 Introduction

The conventional view of a program running in a distributed system, commonly called a *distributed program*, is of processes running on the nodes of a network, exchanging information by passing messages. This is the view supported by the ubiquitous *Message Passing Interface* system [15]. Although an effective method for writing distributed programs, MPI-style programming is far more laborious than programming a stand-alone computer. In order to create a distributed version of a program developed to run on a single node the code needs to be partitioned into modules that will run on each node in the distributed system, and the communication between the modules needs to be reduced to data communication and handled explicitly by the programmer.

A more abstract, and therefore more convenient, way of programming the interaction between process running on different nodes is through *Remote Procedure Call* [16]. In RPC, *communication* is subsumed by *function call*, which entails a certain protocol between the caller and the callee, i.e. first the arguments are sent by the caller along with the function name, then the result is returned by the callee. The success of RPC is reflected in the large number of variations it spawned, such as Java's Remote Method Invocation, SOAP, CORBA's object request broker, etc. With the advent of cloud computing more comprehensive frameworks were developed based on this idea, such as Microsoft's Windows Communication Framework, Google's Web Toolkit or Facebook's Thrift framework, and aimed specifically at web-delivered computational services. Although significantly more convenient that MPI-style programming, RPC-style programming still requires a certain amount of boilerplate code and, more importantly, it is not compatible with higher-order functions and functional programming.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 34–48, 2013.
© Springer-Verlag Berlin Heidelberg 2013

A higher-order version of RPC seems to require the sending of functions (code) from the caller to the callee and vice versa, because functions can now be both arguments and results. One solution is that all nodes have access to a local instance of each function in the program and can send references to such functions (possibly paired with some free variables, forming a closure). Erlang [1] and Cloud Haskell [4] take this approach. Erlang, which runs in a virtual machine, even allows the sending of syntax trees for terms that do not exist on the remote node. However, both these approaches have the disadvantage that a program running on a single node needs to be "ported" to the distributed setting by including significant amounts of non-trivial boilerplate code.

What we will describe in this paper is a *seamless* approach to distributed programming: the distributed program is syntactically and semantically identical to the same program running on the single node, except for annotations (labels) indicating the names of the nodes where particular terms are to be executed. There is no language-induced restriction regarding the way locations are assigned: any syntactic sub-term of the program can be given an arbitrary node label, which will mean that it will be executed on the node of that label. There is no explicit communication between nodes, all the interaction being automatically handled "under the hood" by the generated code.

Example. To illustrate this point consider the same program written in Erlang versus PCF annotated with location information. Consider the PCF program $\mathbf{let}\ f\ =\ \lambda x.\ x * x\ \mathbf{in}\ f\ 3 + f\ 4$, and suppose that we want to delegate the execution of the multiplication operation on a node C while the rest of the program executes on the (main) node A. The annotated PCF code is simply: $(\mathbf{let}\ f\ =\ (\lambda x.\ x * x) \mathbin{@} C\ \mathbf{in}\ f\ 3 + f\ 4) \mathbin{@} A$

In Erlang, things are much more complicated. The f function can be set up as a server which receives request messages:

```
c(A_pid) -> receive X -> A_pid ! X * X end, c(A_pid).
main() ->
  C_pid = spawn(f, c, [self()]), C_pid ! 3,
  receive X -> C_pid ! 4, receive Y -> X + Y end
  end.
```

Arguably, the logical structure of the program is lost in the detail. Moreover, if we want to further delegate the application of f itself to a different server B, the annotated PCF is $(\mathbf{let}\ f\ =\ (\lambda x.\ x * x) \mathbin{@} C\ \mathbf{in}\ (f\ 3) \mathbin{@} B + (f\ 4) \mathbin{@} B) \mathbin{@} A$ whereas the Erlang version of the three-server distribution is even more complicated than the two-server version, which further obscures its logical structure:

```
c() -> receive {Pid, X} -> Pid ! X * X end, c().
b(A_pid, C_pid) ->
  receive
    request0 -> C_pid ! {self(), 3}, receive X -> A_pid ! X end;
    request1 -> C_pid ! {self(), 4}, receive X -> A_pid ! X end
  end,
```

```
  b(A_pid, C_pid).
main() ->
  C_pid = spawn(f2, c, []),
  B_pid = spawn(f2, b, [self(), C_pid]),
  B_pid ! request0,
  receive X -> B_pid ! request1, receive Y -> X + Y end
  end.
```

Contribution. The main technical challenge we address in this paper is handling higher-order and recursive computation. To be able to give a focussed and technically thorough treatment we will handle the paradigmatic functional programming language PCF [13], a language which is well understood semantically and which lies at the foundation of practical programming languages such as Haskell.

Conceptually, what makes the seamless distribution of PCF programs possible is an interpretation inspired by game semantics [6] and the Geometry of Interaction (GoI) [10]. These models are built on the principle of reducing function call to communication and computation to interaction.

Note that the idea of reducing computation to interaction also appeared in the context of process calculi [11]. This was a development independent of game semantics and GoI which happened around the same time. Compilation of distributed languages via process calculi is also possible, in languages such as Pict [12], but the methodology is different. Whereas we aim to hide all the communication by making it implicit in function call, Pict and related languages embedded process calculus syntax to develop communication protocols. A further significant development in compilation based on process calculi was the idea of *mobile code*, where software agents are explicitly sent across the network between running processes [3,14]. By contrast, in our methodology no code needs to be transmitted. Also note that GoI itself has been used before to compile PCF-like programs to unconventional architectures, namely reconfigurable digital circuits [7].[1]

The GoI model reduces a program to a static network of elementary nodes. Although this is eminently suitable for hardware synthesis, where the elementary nodes become elementary circuits and the network becomes the circuit interconnect, it is too fine grained for distributed compilation. We address this technical challenge by introducing a new style of *abstract machine* which has elementary instructions for both control (jumps) and communication. These machines can be (almost) arbitrarily *combined* by replacing communication with jumps, which gives a high degree of control over the granularity of the network.

Our compiler works in several stages. First it creates the fine-grained network of elementary communicating abstract machines. Then, using node annotations (labels), it combines all machines which arise out of the compilation of terms using the same label. The final step is to compile the abstract machines down to executable code using C for local execution and MPI for inter-machine communication.

[1] Tool available at `http://veritygos.org`.

2 PCF and Its GoI Model

We use conventional PCF with natural numbers as the only base type, extended with an annotation $t \mathbin{@} A$, for specifying the locus of a term's computation. This annotation is to be thought of as a compiler directive: the operational semantics and typing rules simply ignore the locus specifier and are otherwise standard [13].

Girard's Geometry of Interaction [8] is a model for linear logic used for the study of the dynamics of computation, seeing a proof in the logic as a net, executed through the passing of a token along its edges. It is well known that GoI can be extended to also interpret programming languages and that it is useful in compiling programs to low-level machine code [10]. In this paper, we will use it as an interpretation for terms in our language, but we will use the notion of a proof net quite literally in that our programs will be compiled into networks of communicating nodes that operate on and send a token to each other.

We give an interpretation of terms in our language similar to the GoI interpretations by Hoshino [9] and Mackie [10], where terms are coded into linear logic proof nets. The only difference is that the interfaces of our nets are determined by type instead of being homogeneous. In this regard, the interpretation is more similar to the hardware circuits presented by the second author [5]. The reason why this works is because our language is not polymorphic, which otherwise necessitates that the interfaces are more homogeneous, since polymorphic terms may be instantiated at different types.

Term interpretations are built by connecting the ports of graphical components, that we think of as the nodes in our network. Two connected components can communicate bidirectionally through data tokens, defined by the grammar:

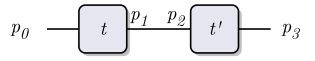
$$e ::= \bullet \mid 0 \mid \mathbf{S}\, e \mid \mathbf{inl}\, e \mid \mathbf{inr}\, e \mid \langle e, e \rangle.$$

We first give a reading of these components as partial maps between data tokens. In the next section we will give a low-level description of their inner workings as communicating abstract machines.

The standard GoI components are given in Figure 1: dereliction (d), promotion (δ) and contraction (c). The components are bidirectional and their behaviour is given by a function mapping the values of a port at a given moment to their values at the next moment. We denote the value on a port which sends/receives no data as $\bot$. Two well-formedness conditions of GoI nets are that at most one port is not $\bot$ (i.e. a single-token is received at any moment) and $\overline{\bot} = (\bot, \ldots, \bot)$ is a fixed-point for any net (i.e. no spontaneous output is created).

Let π_2, π_2 be the first and second projections. Components are connected by functional composition (in both directions) on the shared port, represented graphically as:

$$(t; t')(e, \bot) = t'(\pi_2 \circ t(e, \bot), \bot)$$
$$(t; t')(\bot, e) = t(\bot, \pi_1 \circ t'(\bot, e)).$$



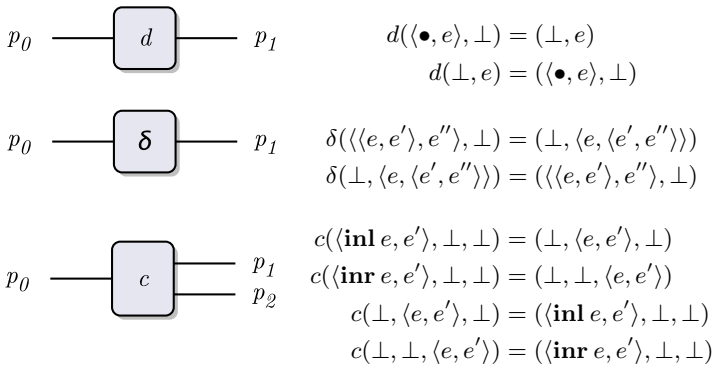
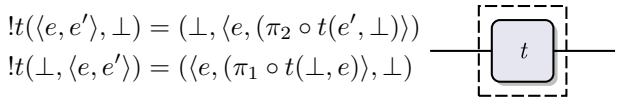
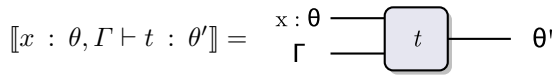


Fig. 1. Dereliction, promotion, and contraction

Exponentials. Tokens need to carry both data and 'routing' information, but we want the basic components to have no access to the routing information but to act on data only. The role of the exponential functor (!) is to remove this routing information from the enclosed component, pass the data to the component, then restore the routing information. Diagrammatically this is represented as a dotted box around a network, defined formally below:

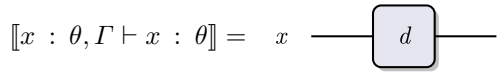


Types as interfaces. The interface of a net is determined by the typing judgement of the term it interprets. The **nat** type corresponds to one port; the function type, $\theta \to \theta'$, induces an interface which is the disjoint union of those for θ, θ'. A typing environment $\Gamma = x_1 : \theta_1, \ldots x_n : \theta_n$ induces an interface which is the disjoint union of the interfaces for each θ_i. A term with typing judgement $\Gamma, x : \theta \vdash t : \theta'$ interface given by the environment on the left and its type on the right. Diagrammatically this is:



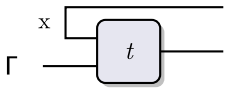
Note that type $\theta \times \theta'$ can be interpreted, as convenient, either as a pair of ports or as a single port sending or receiving pair-tokens. Because of the well-formedness conditions the pairs are always of shape $(e, \bot)$ or $(\bot, e)$.

Terms as networks. As the variables in the context correspond to the linear ! type, the GoI interpretation requires the use of dereliction:



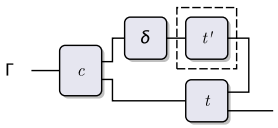
Abstraction and application. When interpreting an abstraction, the variable is added to the context of the inner term, just like in the typing rule, and exposed in the interface of the final component. We only show the diagrammatic definition, the formalisation is obvious:

$$[\![\Gamma \vdash \lambda x.\, t \;:\; \theta \to \theta']\!] =$$



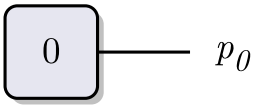
In the interpretation of application note the use of dereliction and exponentiation in the way the argument t' is connected to function t; this corresponds to the standard linear decomposition of call-by-name evaluation. Also note the use of contraction to explicitly 'share' the use of the free identifiers in Γ.

$$[\![\Gamma \vdash t\; t' \;:\; \theta']\!] =$$



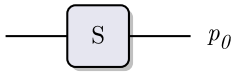
Constants. The interpretation of a constant is a simple component that answers with 0 when requested, i.e. $\mathbf{0}(\bullet) = 0$. Diagrammatically,

$$[\![\Gamma \vdash 0 \;:\; \mathbf{nat}]\!] =$$



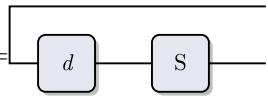
Successor. A new component is needed for the interpretation of the successor function. This handles the **S** operation directly on a natural number.

$$\mathbf{S}\,(\bot, \bullet) = (\bullet, \bot) \qquad \mathbf{S}\,(n, \bot) = (\bot, \mathbf{S}\,n).$$



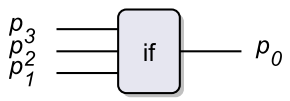
To make it possible to use this component in our interpretation it needs to be wrapped up in an abstraction and a dereliction to bring the argument to the right linear type.

$$[\![\Gamma \vdash \mathbf{S} \;:\; \mathbf{nat} \to \mathbf{nat}]\!] =$$



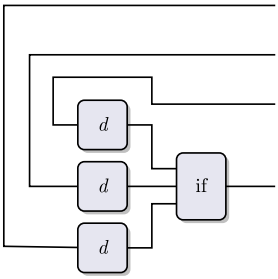
Conditionals. Similar to how addition was handled, conditionals are done by constructing a new component and then wrapping it up as a function.

$$\begin{aligned}
\mathrm{if}(\bullet, \bot, \bot, \bot) &= (\bot, \bullet, \bot, \bot) \\
\mathrm{if}(\bot, 0, \bot, \bot) &= (\bot, \bot, \bullet, \bot) \\
\mathrm{if}(\bot, \mathbf{S}\,n, \bot, \bot) &= (\bot, \bot, \bot, \bullet) \\
\mathrm{if}(\bot, \bot, n, \bot) &= (n, \bot, \bot, \bot) \\
\mathrm{if}(\bot, \bot, \bot, n) &= (n, \bot, \bot, \bot).
\end{aligned}$$



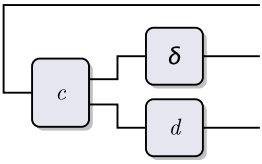
For the final interpretation of conditionals in the language, derelictions are added as the arguments are of exponential type:

$$[\![\Gamma \vdash \textbf{if} \cdot \textbf{then} \cdot \textbf{else} \cdot : \textbf{nat} \to \textbf{nat} \to \textbf{nat} \to \textbf{nat}]\!] =$$



Recursion. Recursion is interpreted as a component that is connected to itself as done by Mackie [10].

$$[\![\Gamma \vdash \mathbf{Y} \ : (\theta \to \theta) \to \theta]\!]$$



The abstract token machine interpretation given in this section is known to be sound [9,10].

Theorem 1 (GoI Soundness). *Let* $\vdash t : \textbf{nat}$ *be a closed PCF program (closed ground-type term) and* $[\![t]\!]$ *its GoI abstract-token machine representation. If* t *evaluates to* n *(*$t \Downarrow n$*) then* $[\![t]\!](\bullet) = n$.

3 The SIC Machine

To be able to describe the inner workings of the components and see how they can be compiled to executables we construct an abstract machine, the Stack-Interaction-Control (SIC) machine, which has a small instruction set tailor made for that purpose. The SIC machine works similar to Mackie's [10] but with the important distinction that it also allows sending and receiving messages to and from other machines, to model networked distribution. The machine descriptions and configurations are specified in Figure 2. An initial configuration for a machine description $\langle P, L\rangle$ is given by the function $\mathit{initial}(\langle P, L\rangle) = \langle \textbf{passive}, \bullet, P, L\rangle$.

3.1 SIC Semantics

The semantics of the machines are given as a transition relation in Figure 3. The machine instructions make it possible to manipulate the data token of an active machine, using the stack for state and intermediate results. The last two rules handle sending and receiving messages.

Label	l		
Port	p		
Instruction	I	$::= \texttt{inl} \mid \texttt{inr}$	Tags
		$\mid \texttt{fst} \mid \texttt{snd}$	Projections
		$\mid \texttt{unfst} \mid \texttt{unsnd}$	Reverse projections
		$\mid \texttt{flip} \mid \texttt{push} \mid \texttt{pop}$	Stack operations
		$\mid \texttt{zero} \mid \texttt{suc}$	Operations on natural numbers
Code	C	$::= I;\, C$	Instruction cons
		$\mid \texttt{jump}\, l$	Jump to label l
		$\mid \texttt{match}\, l_1\, l_2$	Conditional jump
		$\mid \texttt{if}\, l_1\, l_2$	Conditional jump (nat)
		$\mid \texttt{send}\, p$	Send on port p
Stack	S	$::= \bullet$	Empty
		$\mid d :: S$	Cons
Port map	$P : p \to l$		
Label map	$L : l \to C$		
Machine description		$::= \langle P, L \rangle$	
Machine state	M_{state}	$::= \textbf{active}\, C\, d$	
		$\mid \textbf{passive}$	
Machine configuration	M	$::= \langle M_{\text{state}},\, S,\, P,\, L \rangle$	

Fig. 2. SIC specification

Now it is possible to define a machine network where many of these machines can operate together. This machine network is described in the style of the Chemical Abstract Machine [2]. A machine network is simply a multiset ($\mathcal{M}$) of messages (port-data pairs (p, d)) that are "in the air" and a list of machine configurations (M). The transition relation for the network is:

$$\frac{M \to M'}{\langle \mathcal{M} \mid N_1, M, N_2 \rangle \to \langle \mathcal{M} \mid N_1, M', N_2 \rangle}\ \textsc{Silent}$$

$$\frac{M \xrightarrow{\texttt{send}\ \langle p,d \rangle} M'}{\langle \mathcal{M} \mid N_1, M, N_2 \rangle \to \langle \mathcal{M} \uplus \{\langle p, d \rangle\} \mid N_1, M', N_2 \rangle}\ \textsc{Send}$$

$$\frac{M \xrightarrow{\overline{\texttt{send}\ \langle p,d \rangle}} M'}{\langle \mathcal{M} \uplus \{\langle p, d \rangle\} \mid N_1, M, N_2 \rangle \to \langle \mathcal{M} \mid N_1, M', N_2 \rangle}\ \textsc{Receive}$$

Let $\mathit{active}(N)$ be a function that returns all the machines in N that are in **active** state.

Proposition 1. *For any sets of machine configurations N and N' and multisets of messages $\mathcal{M}$ and $\mathcal{M}'$, if $\langle \mathcal{M} \mid N \rangle \to \langle \mathcal{M}' \mid N' \rangle$, then $|\mathcal{M}'| + |\text{active}(N')| = |\mathcal{M}| + |\text{active}(N)|$.*

Proof. By cases on the machine network step relation and construction of the SIC machine's step relation.

$$
\begin{array}{lcl}
\langle \textbf{active}\ (\texttt{inl};\ C)\ d,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ (\textbf{inl}\ d),\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{inr};\ C)\ d,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ (\textbf{inr}\ d),\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{fst};\ C)\ \langle d_1, d_2\rangle,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ d_1,\ d_2 :: S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{snd};\ C)\ \langle d_1, d_2\rangle,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ d_2,\ d_1 :: S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{unfst};\ C)\ d_1,\ d_2 :: S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ \langle d_1, d_2\rangle,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{unsnd};\ C)\ d_2,\ d_1 :: S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ \langle d_1, d_2\rangle,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{flip};\ C)\ d,\ d_1 :: d_2 :: S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ d,\ d_2 :: d_1 :: S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{push};\ C)\ d,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ \bullet,\ d :: S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{pop};\ C)\ d_1,\ d_2 :: S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ d_1,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{zero};\ C)\ d,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ 0,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{suc};\ C)\ n,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ C\ \mathbf{S}\ n,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{jump}\ l)\ d,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ L(l)\ d,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{match}\ l_1\ l_2)\ (\textbf{inl}\ d),\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ L(l_1)\ d,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{match}\ l_1\ l_2)\ (\textbf{inr}\ d),\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ L(l_2)\ d,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{if}\ l_1\ l_2)\ 0,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ L(l_2)\ \bullet,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{if}\ l_1\ l_2)\ \mathbf{S}\ n,\ S,\ P,\ L\rangle & \to & \langle \textbf{active}\ L(l_1)\ \bullet,\ S,\ P,\ L\rangle \\
\langle \textbf{active}\ (\texttt{send}\ p)\ d,\ S,\ P,\ L\rangle & \xrightarrow{\texttt{send}\ \langle p,d\rangle} & \langle \textbf{passive},\ S,\ P,\ L\rangle \\
\langle \textbf{passive},\ S,\ P,\ L\rangle & \xrightarrow{\overline{\texttt{send}\ \langle p,d\rangle},\ p \in dom(P)} & \langle \textbf{active}\ L(P(p))\ d,\ S,\ P,\ L\rangle
\end{array}
$$

Fig. 3. SIC step relation

Case Silent : In this rule, $\mathcal{M}' = \mathcal{M}$, so $|\mathcal{M}'| = |\mathcal{M}|$. The SIC machine M that takes a step goes from **active** to **active** since all SIC machine step rules that do not send or receive messages are on that form, which also implies that $|active(N)| = |active(N')|$.

Case Send : Here $\mathcal{M}' = \mathcal{M} \uplus \{\langle p, d\rangle\}$ so $|\mathcal{M}'| = |\mathcal{M}| + 1$. The only SIC machine rule that applies here is the send rule, which takes the machine M from state **active** to **passive** . So $|active(N')| = |active(N)| - 1$, and thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| + 1 + |active(N)| - 1 = |\mathcal{M}| + |active(N)|$

Case Receive : In this case, $\mathcal{M} = \mathcal{M}' \uplus \{\langle p, d\rangle\}$ so $|\mathcal{M}'| = |\mathcal{M}| - 1$. The only SIC machine rule that applies is the receive rule, which takes the machine M from state **passive** to **active** , meaning that $|active(N')| = |active(N)| + 1$. Thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| - 1 + |active(N)| + 1 = |\mathcal{M}| + |active(N)|$.

In particular, this proof means that if we start out with one message and no active machines, there can be at most one active machine at any point in the network's execution – the execution is *single-token*.

3.2 Components as SIC Code

In each of the components we take all ports p_x and labels l_x to be distinct. Components are connected by giving an output port the same name as the input port of the component that it is connected to. To emphasise the input/output role of a port we sometimes write them as p_x^i when serving as input and p_x^o when serving as output. The machine descriptions for the different components are described by giving their port mappings P and label mappings L as a tuple.

The following three machines are stateless. They use the stack internally for intermediate results, but ultimately return the stack to the initial empty state.

Dereliction, d, removes the first component of the token tuple when going from left to right, and adds it when going in the other direction:

$$\text{dereliction} = \left\langle \begin{array}{ll} p_0^i \mapsto l_0 & \\ p_1^i \mapsto l_1 & , \end{array} \begin{array}{l} l_0 \mapsto \texttt{snd}; \texttt{pop}; \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{push}; \texttt{unfst}; \texttt{send}\, p_0^o \end{array} \right\rangle$$

Promotion, δ, reassociates the data token to go from $\langle\langle e, e'\rangle, e''\rangle$ to $\langle e, \langle e', e''\rangle\rangle$ and back:

$$\text{promotion} = \left\langle \begin{array}{ll} p_0^i \mapsto l_0 & \\ p_1^i \mapsto l_1 & , \end{array} \begin{array}{l} l_0 \mapsto \texttt{fst}; \texttt{snd}; \texttt{flip}; \texttt{unfst}; \texttt{unsnd}; \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{snd}; \texttt{fst}; \texttt{flip}; \texttt{unsnd}; \texttt{unfst}; \texttt{send}\, p_0^o \end{array} \right\rangle$$

Contraction, c, uses matching on the first component of the token to choose the right port to send on when going from left to right.

$$\text{contraction} = \left\langle \begin{array}{ll} p_0^i \mapsto l_0 & \\ p_1^i \mapsto l_1 & , \\ p_2^i \mapsto l_2 & \end{array} \begin{array}{l} l_0 \mapsto \texttt{fst}; \texttt{match}\, l_3\, l_4 \\ l_1 \mapsto \texttt{fst}; \texttt{inl}; \texttt{unfst}; \texttt{send}\, p_0^o \\ l_2 \mapsto \texttt{fst}; \texttt{inr}; \texttt{unfst}; \texttt{send}\, p_0^o \\ l_3 \mapsto \texttt{unfst}; \texttt{send}\, p_1^o \\ l_4 \mapsto \texttt{unfst}; \texttt{send}\, p_2^o \end{array} \right\rangle$$

Dotted boxes with k inputs are defined as follows: Let $P_{\text{in}}^i = \{p_0^i, \ldots, p_{k-1}^i\}$, $P_{\text{out}}^i = \{p_k^i, \ldots, p_{2k-1}^i\}$, Let $P_{\text{in}}^o = \{p_0^o, \ldots, p_{k-1}^o\}$, $P_{\text{out}}^o = \{p_k^o, \ldots, p_{2k-1}^o\}$, and $L_{\text{out}} = \{l_k, \ldots, l_{2k-1}\}$. The box for these sets of ports and labels is then the following (note that boxes use the stack for storing their state):

$$\text{box} = \left\langle p_i^i \mapsto l_i \mid p_i^i \in P_{\text{in}}^i \cup P_{\text{out}}^i \quad , \quad \begin{array}{l} l_i \mapsto \texttt{snd}; \texttt{send}\, p_{i+k}^o \mid l_i \in L_{\text{in}} \\ l_i \mapsto \texttt{unsnd}; \texttt{send}\, p_{i-k}^o \mid l_i \in L_{\text{out}} \end{array} \right\rangle$$

For a constant, the component's abstract machine is defined as follows:

$$\text{constant} = \left\langle p_0^i \mapsto l_0 \quad , \quad l_0 \mapsto \texttt{zero}; \texttt{send}\, p_0^o \right\rangle$$

The successor machine first asks for its argument, then runs the successor instruction on that.

$$\text{suc} = \left\langle \begin{array}{ll} p_0^i \mapsto l_0 & \\ p_1^i \mapsto l_1 & , \end{array} \begin{array}{l} l_0 \mapsto \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{suc}; \texttt{send}\, p_0^o \end{array} \right\rangle$$

The conditional uses the matching instruction `if` to choose the right branch depending on the natural number of the token.

$$\text{if} = \left\langle \begin{array}{ll} p_0^i \mapsto l_0 & \\ p_1^i \mapsto l_1 & , \\ p_2^i \mapsto l_2 & \\ p_3^i \mapsto l_2 & \end{array} \begin{array}{l} l_0 \mapsto \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{if}\, l_{\text{true}}\, l_{\text{false}} \\ l_{\text{true}} \mapsto \texttt{send}\, p_2^o \\ l_{\text{false}} \mapsto \texttt{send}\, p_3^o \\ l_2 \mapsto \texttt{send}\, p_0^o \end{array} \right\rangle$$

To connect the port p_0 of machine M to p_1 of M' we rename them in the machine definitions to the same port name p. A port must be connected to at most one other port; in this case the net is said to be *deterministic*, as each message will be received by at most one other machine. A port of a machine which is not connected to a port of another machine is said to be a port *of the network*. By $\mathit{inputs}(M)$ ($\mathit{outputs}(M)$) we mean the inputs (outputs) of a machine, whereas by $\mathit{inputs}(N)$ ($\mathit{outputs}(N)$) we mean the inputs (outputs) of a net. Similarly, by $\pi(M)$ ($\pi(N)$) we mean the ports of a machine (network).

Theorem 2 (Soundness). *Let* $\vdash t : \mathbf{nat}$ *be a closed PCF program (closed ground-type term),* $[\![t]\!]$ *its GoI abstract-token machine representation and* N *its SIC-net implementation. If* t *evaluates to* n *(*$t \Downarrow n$*) then* $[\![t]\!](\bullet) = n$ *and* $\langle \{\langle p^i, \bullet\rangle\} \mid N\rangle \to^* \langle \{\langle p^o, n\rangle\} \mid N\rangle$.

4 Combining Machines

When writing distributed applications, the location at which a computation is performed is vital. Traditional approaches are sometimes explicit about that, for instance by using message passing. Using our current term interpretation, and thinking of each abstract machine as running on a different node in a network, we get the communication in the network handled automatically, but will have one abstract machine for each (very small) component. The interpretation produces extremely fine grained networks where each node does very little work before passing the token along to another node. It is expected that the communication is one of the most performance critical parts in a distributed network, which is why it would be better if bigger chunks of computations happened on the same node before the token was passed along.

To make this possible, we devise a way to combine the descriptions of two abstract machines in a deterministic network to get a larger abstract machine with the same behaviour as the two original machines. Informally, the way to combine two machines is to remove ports that are used internally between the two machines (if any) and replace sends on those ports with jumps. The algorithm for combining components $M_1 = \langle P_1, L_1\rangle$ and $M_2 = \langle P_2, L_2\rangle$ is described formally below.

We use Δ for the symmetric difference of two sets. If $f : A \to B$ is a function we write as $f \restriction A'$ the restriction of f to the domain $A' \subseteq A$ and we extend it in the obvious way to relations. We use the standard notation $C[s/s']$ to denote the replacing of all occurrences of a string s by s' in C. We write $C[s(x)/s'(x) \mid x \in A]$ to denote the substitution of all strings of shape $s(x)$ by strings of shape $s'(x)$ with x in a list A, defined inductively as

$$C[s(x)/s'(x) \mid x \in \emptyset] = C$$
$$C[s(x)/s'(x) \mid x \in a :: A] = \big(C[s(a)/s'(a)]\big)[s(x)/s'(x) \mid x \in A]$$

The combination of two machines is defined by keeping the ports which are not shared and by replacing in the code the send operations to shared ports by jumps to labels given by the port mappings.

$$\mathit{combine}(M_1, M_2) = \langle (P_1 \cup P_2) \upharpoonright (\pi(M_1) \,\Delta\, \pi(M_2)),$$
$$(L_1 \cup L_2)[\texttt{send}\, p/\texttt{jump}\, P(p) \mid p \in \pi(M_1) \cap \pi(M_2)]\rangle.$$

There are two abuses of notation above. First, the union $P_1 \cup P_2$ above is on functions taken as sets of pairs and it may not result in a proper function. However, the restriction to $\pi(M_1) \,\Delta\, \pi(M_2)$ always produces a proper function. Second, $\pi(M_1) \cap \pi(M_2)$ is a set and not a list. However, the result of this substitution is independent of the order in which the elements of this set are taken from any of its possible list representations.

A network is said to be combinable if combining any of its components does not change the overall network behaviour

Definition 1. *A deterministic network of machines $N = M_1, \ldots, M_k$ is* combinable *if whenever*

$$\langle \{\langle p, d\rangle\} \mid N\rangle \to^* \langle \{\langle p', d'\rangle\} \mid N'\rangle$$

for some N', p in inputs(N)*, p' in* outputs(N)*, then for any $N_{combined}$ obtained from N by replacing M_i, M_j with* combine(M_i, M_j) *for some $i \neq j$ we have that*

$$\langle \{\langle p, d\rangle\} \mid N_{combined}\rangle \to^* \langle \{\langle p', d'\rangle\} \mid N'_{combined}\rangle$$

for some $N'_{combined}$.

Note that the combined net is not equivalent to the original net (for a suitable notion of equivalence such as bisimilarity) because it will have fewer observable messages being exchanged.

Lemma 1. *If a net N is combinable then $N_{combined}$ is also combinable.*

The set of combinable machines is hard to define exactly, so we would just like to find a sound characterisation of such machines which covers all the basic components we used and their combinations.

Definition 2. *A machine description $M = \langle P, L\rangle$ is* stack-neutral *if for all stacks S and S', p in* inputs(M)*, p' in* outputs(M)*, if*

$$\langle \{\langle p, d\rangle\} \mid \langle \mathbf{passive},\, S,\, P,\, L\rangle\rangle \to^* \langle \{\langle p', d'\rangle\} \mid \langle \mathbf{passive},\, S',\, P,\, L\rangle\rangle$$

then $S = S'$.

Definition 3. *A machine network N of k machines described by port mappings P_i and label mappings L_i is* stack-neutral*, if for all stacks S_i and S'_i, p in* inputs(N)*, p' in* outputs(N)*, if*

$$\langle \{\langle p, d\rangle\} \mid [\langle \mathbf{passive},\, S_i,\, P_i,\, L_i\rangle \mid i \in \{1, \ldots, k\}]\rangle \to^*$$
$$\langle \{\langle p', d'\rangle\} \mid [\langle \mathbf{passive},\, S'_i,\, P_i,\, L_i\rangle \mid i \in \{1, \ldots, k\}]\rangle$$

then all $S_i = S'_i$.

Note that this definition is more general than having a list of stack-neutral machines, as a stack-neutral network's machines may use the stack for state after it has been exited as long as it's cleared before an output on a network port.

Proposition 2. *If two machine networks N_1 and N_2 (of initially passive machines) are stack-neutral, combinable and N_1, N_2 is deterministic, then N_1, N_2 is stack-neutral and combinable.*

For any SIC-net N let $box(N)$ be N with an additional box machine M with input ports $outputs(N)$ and output ports $inputs(N)$, defined as in Sec. 3.2.

Proposition 3. *If a machine network N is stack-neutral and combinable then* box(N) *is stack-neutral and combinable,*

From the following two results it follows by induction on the structure of the generated nets that

Theorem 3. *If $\Gamma \vdash t : \theta$ is a PCF term, $[\![t]\!]$ its GoI abstract-token machine representation and N the implementation of $[\![t]\!]$ as a SIC-net then N is combinable.*

With the ability to combine components, we can now exploit the $t \mathbin{@} A$ annotations in the language. They make it possible to specify where a piece of code should be located (A is a node identifier). When this construct is encountered in compilation, the components generated in compiling t are tagged with A (possibly overwriting older tags).

Next, the components with the same tag are combined using the algorithm above and their combined machine placed on the node identified by the tag. This allows the programmer to arbitrarily choose where the compiled representation of a part of a term is placed. Soundness (Thm. 2) along with the freedom to combine nets (Thm. 3) ensures that the resulting network is a correct implementation of any (terminating) PCF program.

5 Compiling PCF

We developed an experimental compiler that compiles to C, using MPI for communication, using SIC abstract machines as an intermediate formalism. [2] Each machine description in a network is mapped to a C source file in a fairly straightforward manner, using a function for each machine instruction and global variables for the data token and the stack. An example of a predefined instruction is that for the `flip` instruction:

```
inline void flip() {
  Data d1 = pop_stack();
  Data d2 = pop_stack();
  push_stack(d1);
  push_stack(d2);}
```

[2] Download from `http://veritygos.org/dpcf`

An abstract machine's label l corresponds to a C function `void l()` whose definition is a list of calls to the predefined machine instruction functions. In this representation, jumps are function calls. All functions are small and not used recursively so can be efficiently inlined.

Each process in MPI has a unique identifier called its *rank*, and messages can also be assigned a *tag*. A port in a SIC machine is uniquely determined by its tag, but also has to be assigned a rank so that the message can be sent to the correct node. This is resolved at compile-time. The main loop for a machine listening on ports corresponding to tags 0 and 1 looks like this:

```
while(1) {
  int port = receive();
  switch (port) {
    case 0: l0(); break;
    case 1: l1(); break;
    default: break;}}
```

Here `l0` and `l1` are functions corresponding to the labels associated with the ports. The predefined function `receive` calls `MPI_Recv`, which is an MPI function that blocks until a message is received. A process in this state thus corresponds to a machine in **passive** state. Upon receiving a message, the `receive` function de-serialises the message and assigns it to the global data token variable before returning the message's tag. The predefined function for the `send` instruction now has to take two parameters: the destination node's rank and the port's tag:

```
inline void send(int node, int port);
```

The function takes care of serialising the data token sending it to the correct node using `MPI_Send`.

When all machines have been compiled to C, these can in turn be compiled to executables and run on different machines in a network where they use message passing for communication.

6 Conclusion and Future Work

We have shown a programming language and compilation model for seamless distributed computing, that provides freedom in choosing the location at which a computation takes place with implicitly handled communication. This was achieved by basing the model on the Geometry of Interaction and constructing a way to produce nodes that are more coarse grained than the standard elementary nodes, and showing that this is still correct.

So far semantics of a term are sequential – there is nothing taking place in parallel. The next step will be to investigate how to extend the system with a safe and flexible parallelisation mechanism. A start is to identify internally sequential partitions of the network that can safely be run in parallel. Then the evaluation of a function's arguments can be performed in parallel with the function as long

as they are in different partitions of the network. Another idea is to add a more specific construct for parallelisation, e.g. one for map-reduce.

An important point that has not been discussed in this paper is fault-tolerance: A distributed system needs to be able withstand nodes crashing, becoming unavailable or being added by dynamically reassigning the locus of execution. This is also something that we would like to look into in the future.

References

1. Armstrong, J.: A history of Erlang. In: HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pp. 6–1–6–26. ACM, New York (2007)
2. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. 96(1), 217–248 (1992)
3. Carzaniga, A., Picco, G.P., Vigna, G.: Designing distributed applications with mobile code paradigms. In: Adrion, W.R., Fuggetta, A., Taylor, R.N., Wasserman, A.I. (eds.) ICSE, pp. 22–32. ACM (1997)
4. Epstein, J., Black, A.P., Jones, S.L.P.: Towards Haskell in the cloud. In: Claessen, K. (ed.) Haskell, pp. 118–129. ACM (2011)
5. Ghica, D.R.: Geometry of Synthesis: a structured approach to VLSI design. In: Hofmann, M., Felleisen, M. (eds.) POPL, pp. 363–375. ACM (2007)
6. Ghica, D.R.: Applications of game semantics: From program analysis to hardware synthesis. In: LICS, pp. 17–26. IEEE Computer Society (2009)
7. Ghica, D.R.: Function interface models for hardware compilation. In: 2011 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 131–142. IEEE (2011)
8. Girard, J.: Geometry of interaction 1: Interpretation of system F. Studies in Logic and the Foundations of Mathematics 127, 221–260 (1989)
9. Hoshino, N.: A modified GoI interpretation for a linear functional programming language and its adequacy. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 320–334. Springer, Heidelberg (2011)
10. Mackie, I.: The geometry of interaction machine. In: POPL, pp. 198–208 (1995)
11. Milner, R.: Functions as processes. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 167–180. Springer, Heidelberg (1990)
12. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, pp. 455–494. MIT Press (2000)
13. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. 5(3), 223–255 (1977)
14. Sewell, P., Wojciechowski, P.T., Unyapoth, A.: Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. ACM Trans. Program. Lang. Syst. 32(4) (2010)
15. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI-The Complete Reference, 2nd (revised) edn. The MPI Core, vol. 1. MIT Press, Cambridge (1998)
16. White, A.M.: A high-level framework for network-based resource sharing. Augmentation Research Centre, RFC 707 (1975)

A Beginner's Guide to the Dead*L*ock *A*nalysis *M*odel

Elena Giachino and Cosimo Laneve

Dipartimento di Informatica, Università di Bologna – INRIA Focus Team
{giachino,laneve}@cs.unibo.it

Abstract. This paper is an introduction to the framework for the deadlock analysis of object-oriented languages we have defined in [6,5]. We present a basic JAVA-like language and the deadlock analysis model in an accessible way. We also overview the algorithm for deciding deadlock-freeness by discussing a number of paradigmatic examples. We finally explore the techniques for coping with extensions of the object-oriented language.

Keywords: Static deadlock analyzers, object-oriented languages, circular dependencies, lam, livelocks.

1 Introduction

Modern systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In these systems, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. An alternative description is that the correct termination of each of the two process activities *depends* on the termination of the other. Since there is a *circular dependency*, termination is not possible.

Deadlocks may be particularly insidious to detect in systems where the basic communication operation is asynchronous and the synchronization explicitly occurs when the value is strictly needed. Speaking a JAVA idiom, methods are *synchronized*, that is each method of the same object is executed in mutual exclusion, and method invocations are *asynchronous*, that is the caller thread continues its execution without waiting for the result of the called method. Additionally, an operation of *join* explicitly synchronizes callee and called methods (possibly returning the value of the called method). In this context, when a thread running on an object x performs a join operation on a thread on y then it blocks every other thread that is competing for the lock on x. This blocking situation corresponds to a dependency pair (x, y), meaning that the progress on x is possible provided the progress of threads on y. A deadlock then corresponds to a circular dependency in some configuration.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 49–63, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Further difficulties arise in the presence of infinite (mutual) recursion: consider, for instance, systems that create an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and really hard to predict. In addition, deadlocks may not be detected during testing, and even if they are it can be difficult to reproduce them and find their causes.

Deadlock detection has been largely investigated both with static and run-time techniques [2,10,1] (just to cite few references). Static analysis guarantees that all executions of a program are deadlock-free, at the cost of being not precise because it may discard safe programs (false positives). Run-time checking cannot be exhaustive. However, whenever applicable, it produces fewer false positives than static analysis.

We adopt a static approach, while retaining the precision of a run-time checker in a large number of cases. Our deadlock detection framework consists of an inference algorithm that extracts abstract behavioral descriptions out of the concrete program. These abstract descriptions, called *lam programs*, an acronym for *deadLock Analysis Model*, retain necessary informations for the deadlock analysis (typically all the synchronization informations are extracted, while data values are ignored). Then a decision algorithm evaluates the abstract program for verifying its circularity-freeness (every state has no circular dependency). In turn, this property implies the deadlock-freeness of the original program. Our approach is then both *flexible* (since only the inference algorithm has to be adapted to the language, while the analyzer is language-independent) and *precise* (since the analyzer is a decision algorithm on a large class of lam programs – see below). The major benefits of our technique are that (i) it does not use any pre-defined partial order of resources and that (ii) it accounts for dynamic resource creation.

To overview our analyzer, we observe that, in presence of recursion in the code, the evaluation of the abstract description may end up into an infinite sequence of states, without giving back any answer. Instead, our analysis always terminates. The theoretical framework we have designed allow us to determine when to stop the evaluation. Informally, when the abstract program is *linear* – it has (mutual) recursions of the kind of the factorial function –, then states reached after some point are going to be equivalent to past states. That point, called *saturated state*, may be determined in a similar way as the orbit of a permutation [3] (actually, our theory builds on a generalization of the theory of permutations [6]). A saturated state represents the end of a pattern that is going to repeat indefinitely. Therefore it is useless to analyze it again and, if a deadlock has not been encountered up to that point, then it cannot be produced afterwords. Analogously, if a deadlock has been encountered, then a similar deadlock must be present each time the same pattern recurs. When the abstract program is nonlinear – it has (mutual) recursions of the kind of the fibonacci function – our technique is not precise because it introduce fake dependency pairs. However, it is sound.

The aim of this paper is to present our deadlock technique informally, by means of examples and without going into the (many) theoretical details.

The interested reader may find the theoretical developments in [6] and the application of our abstract descriptions to a programming language in [5,4].

The structure of the paper is as follows. In Section 2, we introduce the programming language and the analysis model. In Section 3, we overview the algorithm for detecting circularities in the analysis model. In Section 4, we discuss a number of programs and their associated abstract models. In Section 5, we explore two relevant extensions of the programming language: field assignment and an operation for releasing the lock. We conclude in Section 6.

2 Languages and Models

2.1 The Language FJf

The programs that are analyzed in this paper will be written in a JAVA-like language. Instead of using JAVA, which is quite verbose, we stick to a dialect of the ABS language [8], called FJf. To enhance readability, the semantics of FJf will be given in an indirect way by discussing the compilation patterns of the main constructs of FJf into JAVA (the reader is referred to [4] for a direct operational semantics).

FJf syntax uses four disjoint infinite sets of names: *class names*, ranged over by A, B, C, $\cdots$, *field names*, ranged over by f, g, $\cdots$, *method names*, ranged over by m, n, $\cdots$, and *variables*, ranged over by x, y, $\cdots$. The special name this is assumed to belong to the set of variables. The notation $\widetilde{\mathtt{C}}$ is a shorthand for $\mathtt{C}_1\,;\cdots;\mathtt{C}_n$ and similarly for the other names. Sequences of pairs are abbreviated as $\mathtt{C}_1\ \mathtt{f}_1\,;\cdots;\mathtt{C}_n\ \mathtt{f}_n$ with $\widetilde{\mathtt{C}}\ \widetilde{\mathtt{f}}$. The syntactic categories of *class declarations* CL, *method declarations* M, *expressions* e, and *types* T are defined as follows

$$
\begin{array}{lcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\widetilde{\mathtt{T}}\ \widetilde{\mathtt{f}}\ ;\ \widetilde{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{T\ m}\ (\widetilde{\mathtt{T}}\ \widetilde{\mathtt{x}})\{\ \mathtt{return\ e}\ ;\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e!m}(\widetilde{\mathtt{e}}) \mid \mathtt{new\ C}(\widetilde{\mathtt{e}}) \mid \mathtt{e;e} \mid \mathtt{e.get} \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{Fut}(\mathtt{T})
\end{array}
$$

where sequences of field declarations $\widetilde{\mathtt{T}}\ \widetilde{\mathtt{f}}$, method declarations $\widetilde{\mathtt{M}}$, and parameter declarations $\widetilde{\mathtt{T}}\ \widetilde{\mathtt{x}}$ are assumed to contain no duplicate names. A program is a pair (CT, e), where the *class table* CT is a finite mapping from class names to class declarations CL and e is an expression, called main expression.

According to the syntax, every class has a superclass declared with extends. To avoid circularities, we assume a distinguished class name Object with no field and method declarations and whose definition does not appear in the class table. We always omit the declaration "extends Object".

The main features of FJf are:

futures – Fut(T) these terms are called *futures* of type T and represent pointers to values of type T that may be not available yet. Fut(T) are the types of method invocations that have T as return type.

asynchronous method invocation – `x!m(y)`**:** the caller continues its execution when a method `m` of `x` is invoked without waiting for the result. The called method is evaluated in parallel (on a new spawned thread). The invocation `x!m(y)` has the same behavior as the following code written in JAVA

```
new Thread ( new Runnable() { public void run() { x.m(y); }
           }).start();
```

where, for the sake of conciseness, the code for exception handling is omitted. In particular, in JAVA, every `join()` statement and every synchronized method invocation should handle an `InterruptedException`.

mutual exclusion**:** in `FJf` every method invocations spawns a new threads. However at most one thread per object is executed at the same time. This is translated in JAVA by declaring every method to be `synchronized`.

thread synchronization for value retrieval**:** in `FJf` the caller retrieves the result of an invocation by the operation `t.get`, where `t` is a reference of the invocation. The `get` operation is blocking until the returned value is produced. The `FJf` code:

```
x!m(y).get;
```

corresponds to the synchronization behavior of the following code in JAVA (without exception handling management)

```
Thread t = new Thread ( new Runnable() {
                          public void run() { x.m(y); }
                        });
t.start();
t.join();
```

The operations `get` in `FJf` and `join` in JAVA have not the exact same meaning: while `get` synchronizes the two threads and retrieves the value, `join` is just a synchronization between the callee's and the caller's threads. However, from the point of view of deadlock analysis, the behaviors of the two operations are equivalent.

We omit examples at this stage: several `FJf` codes will be discussed in Section 4.

2.2 Analysis Models for Detecting Circularities

In [6], we have developed a theoretical framework for defining relations on names (every pair of a relation defines a dependency between two tasks, the first one is waiting for the result of the second) and for determining whether a definition may produce a circular dependency – a *deadlock* – or not. The language for defining relation is called *lam* – an acronym for deadLock Analysis Model. Lams are defined by terms that use a set of *names*, ranged over by x, y, z, $\cdots$, and a disjoint set of *function names*, ranged over `m`, `m′`, `n`, `n′`, $\cdots$. A *lam program* is a

tuple $\left(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\right)$ where $\mathtt{m}_i(\widetilde{x_i}) = \mathtt{L}_i$ are *function definitions* and $\mathtt{L}$ is the *main lam*. The syntax of $\mathtt{L}_i$ and $\mathtt{L}$ is

$$\mathtt{L} \quad ::= \quad \mathtt{0} \quad | \quad (x, y) \quad | \quad \mathtt{m}(\widetilde{x}) \quad | \quad \mathtt{L} \| \mathtt{L} \quad | \quad \mathtt{L}; \mathtt{L}$$

such that (i) all function names occurring in $\mathtt{L}_i$ and $\mathtt{L}$ are defined, and (ii) the arity of function invocations matches that of the corresponding function definition.

It is possible to associate a lam function to each method of a `FJf` program. The purpose of the association is to abstract the object dependencies that a method will generate out of its definition. For instance, the `FJf` method declaration

```
C m1( C y, C z ) { return  ( y!m2( z ) ).get ; }
```

has the associated function declaration in a lam program

$$\mathtt{m1}(x, y, z) = (x, y) \| \mathtt{m2}(y, z)$$

where the first argument is the name of the object `this`, which is x for `m1` and y in the invocation of `m2`. The association method-definition/lam-function is defined by an inference system in [5]. In this paper, in order to be as simple as possible, we keep this association informal.

The semantics of a lam program requires a couple of preliminary notions. Let $\mathbb{V}, \mathbb{V}', \mathbb{I}, \cdots$ be partial orders on names (relations that are reflexive, antisymmetric, and transitive) and let $\mathbb{V} \oplus \widetilde{x} < \widetilde{z}$, with $\widetilde{x} \in \mathbb{V}$ and $\widetilde{z} \notin \mathbb{V}$, be the least partial order $\mathbb{V}'$ that satisfies the following rules

$$\mathbb{V} \subseteq \mathbb{V}' \qquad \frac{x \in \widetilde{x} \quad (x, y) \in \mathbb{V} \quad z \in \widetilde{z}}{(y, z) \in \mathbb{V}'}$$

That is, $\widetilde{z}$ become *maximal names* of $\mathbb{V} \oplus \widetilde{x} < \widetilde{z}$. Let *lam contexts*, noted $\mathfrak{L}[\ ]$, be terms derived by the following syntax:

$$\mathfrak{L}[\ ] \quad ::= \quad [\ ] \quad | \quad \mathtt{L} \| \mathfrak{L}[\ ] \quad | \quad \mathtt{L}; \mathfrak{L}[\ ]$$

As usual $\mathfrak{L}[\mathtt{L}]$ is the lam where the hole of $\mathfrak{L}[\ ]$ is replaced by $\mathtt{L}$. Finally, let $var(\mathtt{L})$ be the set of names occurring in $\mathtt{L}$.

The operational semantics of a program $\left(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}_{\ell+1}\right)$ is defined by a *transition relation* between *states* that are pairs $\langle \mathbb{V},\ \mathtt{L} \rangle$ and satisfying the rule:

$$\frac{\begin{array}{l}\text{(RED)}\\ \mathtt{m}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L}) \setminus \widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ are fresh}\\ \qquad\qquad \mathtt{L}[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x}] = \mathtt{L}'\end{array}}{\langle \mathbb{V},\ \mathfrak{L}[\mathtt{m}(\widetilde{u})] \rangle \longrightarrow \langle \mathbb{V} \oplus \widetilde{u} < \widetilde{w},\ \mathfrak{L}[\mathtt{L}'] \rangle}$$

By (RED), a lam is evaluated by successively replacing function invocations with the corresponding lam instances. At every evaluation step, free names in

a lam definition $\mathtt{m}(\widetilde{u}) = \mathtt{L}$, namely $var(\mathtt{L}) \setminus \widetilde{x}$, are replaced by *fresh names*. This replacement models name creation and correspond to the `new` operation in `FJf`. For example, if $\mathtt{m}(x) = (x,y)\|\mathtt{m}(y)$ and $\mathtt{m}(u)$ occurs in the main lam, then $\mathtt{m}(u)$ is replaced by $(u,v)\|\mathtt{m}(v)$, where v is a *fresh maximal name* in some partial order. The initial state of a program with main lam $\mathtt{L}$ is $\langle \mathbb{I}_{\widetilde{x}}, \mathtt{L}\rangle$, where $\widetilde{x} = x_1, \cdots, x_n = var(\mathtt{L})$ and $\mathbb{I}_{\widetilde{x}} = \{(x,x) \mid x \in \widetilde{x}\}$ (we are abusing of the set-notation).

Lams record sets of relations on names. To make explicit these relations, let $\flat(\cdot)$, called *flattening*, be the function inductively defined as follows

$$\flat(\mathtt{0}) = \mathtt{0}, \qquad \flat((x,y)) = (x,y), \qquad \flat(\mathtt{m}(\widetilde{x})) = \mathtt{0},$$
$$\flat(\mathtt{L}\|\mathtt{L}') = \flat(\mathtt{L})\|\flat(\mathtt{L}'), \qquad \flat(\mathtt{L};\mathtt{L}') = \flat(\mathtt{L});\flat(\mathtt{L}').$$

For example

$$\flat(\mathtt{m}(x,y,z);(x,y)\|\mathtt{n}(y,z)\|\mathtt{m}(u,y,z);\mathtt{n}(u,v)\|(u,v);(v,u)) = (x,y);(u,v);(v,u)$$

that is, the argument of $\flat(\cdot)$ defines three relations: $\{(x,y)\}$ and $\{(u,v)\}$ and $\{(v,u)\}$. It is easy to verify that, up-to the axioms

$$(x,y)\|(x,y) = (x,y) \qquad (\mathtt{L};\mathtt{L}')\|(x,y) = \mathtt{L}\|(x,y);\mathtt{L}'\|(x,y)$$

$$\frac{\mathtt{L} = (x_1,y_1)\|\cdots\|(x_n,y_n)}{\mathtt{L};\mathtt{L} = \mathtt{L}}$$

$\flat(\mathtt{L})$ always returns *sequences of pairwise different parallels of pairs* (i.e. sets of pairwise different relations).

Definition 1. *A lam* $\mathtt{L}$ has a circularity *if*

$$\flat(\mathtt{L}) = (x_1,x_2)\|(x_2,x_3)\|\cdots\|(x_m,x_1)\|\mathtt{L}';\mathtt{L}''$$

for some $x_1,\cdots,x_m$. *A state* $\langle \mathbb{V}, \mathtt{L}\rangle$ *has a circularity if* $\mathtt{L}$ *has a circularity. A program* $\left(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\right)$ *is* circularity-free *if no state yielded by evaluating* $\langle \mathbb{I}_{var(\mathtt{L})}, \mathtt{L}\rangle$ *has a circularity.*

3 The Algorithm for Deciding Circularity-Freeness

In case of non-recursive lam programs, since the evaluations always terminate, the circularity-freeness problem is (easily) decidable. Otherwise – when functions are (mutually) recursive – the evaluation would not terminate and would produce infinite relations due to the creation of names. Nevertheless, the problem of circularity-freeness is decidable for a large set of mutual recursive lam programs – the linear ones.

Definition 2. *A lam program* $\left(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\right)$ *is* linear *if, for every function* $\mathtt{m}_{i_0}$, *there is at most one sequence* $\mathtt{m}_{i_0}\cdots\mathtt{m}_{i_m}$ *such that, for every* $0 \le j \le m$, $\mathtt{L}_{i_j}$ *contains exactly one invocation of* $\mathtt{m}_{i_{j+1\%m}}$ *(the operation* % *is the remainder of the natural division).*

For example, a factorial-like programs, such as $\big(\texttt{fact}(x) = (x,y)\|\texttt{fact}(y),\ \texttt{fact}(x)\big)$, are linear, while fibonacci-like ones, such as $\big(\texttt{fib}(x) = (x,y)\|(x,z)\|\ \texttt{fib}(y)\|\texttt{fib}(z),\ \texttt{fib}(x)\big)$, are not.

The idea of our technique is to recognize the pattern of the recursion in order to be able to determine the states when the evaluation flow is going to repeat the same pattern (with different names) and only produce pattern of dependencies that were already discovered in the past of the evaluation. Once our algorithm recognizes these states, called *saturated states*, it just interrupts avoiding non-terminating evaluations.

Theorem 1 ([6]). *The problem of circularity-freeness in linear lam programs is decidable.*

The theoretical details of this theorem are out of the scope of this paper. Here we illustrate the algorithm on two sample programs. The first one is $\big(\texttt{fact}(x) = (x,y)\|\texttt{fact}(y),\ \texttt{fact}(x)\big)$. In this case, our theory affirms that the saturated state is reached in two steps of evaluation. That is

$$\begin{aligned}\big\langle \mathbb{I}_x,\ \texttt{fact}(x)\big\rangle &\longrightarrow \big\langle \mathbb{I}_x \oplus (x{<}y),\ (x,y)\|\texttt{fact}(y)\big\rangle \\ &\longrightarrow \big\langle \mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z),\ (x,y)\|(y,z)\|\texttt{fact}(z)\big\rangle\ .\end{aligned}$$

We observe that, if we perform a further step of evaluation, we get

$$\big\langle \mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z) \oplus (z{<}u),\ (x,y)\|(y,z)\|(z,u)\|\texttt{fact}(u)\big\rangle$$

and there is an injective partial map ρ on $\mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z) \oplus (z{<}u)$, namely $\rho = [z \mapsto x, u \mapsto y]$, such that

1. $\texttt{fact}(u)$ is mapped to an invocation that is already evaluated – that is $\texttt{fact}(x)$;
2. dependencies produced by $\texttt{fact}(z)$, namely (z,u) are mapped to dependencies that have been already produced, namely (x,y).

These properties allow us to decide that the evaluation is repeating the same pattern (on new names) and conclude that no circularity will be ever manifested since the saturated state is circularity-free.

When the lam program is nonlinear, our technique is imprecise but sound: the nonlinear lam program is transformed into a linear one *by contracting* multiple method invocations to one. This contraction *introduces fake dependencies* (i.e. false positives in terms of circularities). Once the linear transformed program is obtained, then the analysis is run on it. It turns out that these additional dependencies cannot be eliminated because of a cardinality argument. More precisely, the evaluation of a method invocation $\texttt{m}(\widetilde{u})$ in a linear program may produce at most one invocation of $\texttt{m}$, while an invocation of $\texttt{m}(\widetilde{u})$ in a nonlinear program may produce two or more invocations of $\texttt{m}$. When the invocations of $\texttt{m}$ create names, *contracting* different invocations into one means reducing an exponential number of new names to a linear number. This, in terms of the analysis means losing precision. Nevertheless, we prove the soundness of our technique:

if the transformed linear program is circularity-free then the original nonlinear one is also circularity-free. For example, consider the fibonacci-like program

$$\big(\ \mathtt{fib}(x) = (x,y) \| (x,z) \| \mathtt{fib}(y) \| \mathtt{fib}(z),\ \mathtt{fib}(x)\ \big)$$

The transformed program is

$$\big(\ \mathtt{fib}^{aux}(x,x') = (x,y) \| (x,z) \| (x',y) \| (x',z) \| \mathtt{fib}^{aux}(y,z)\,, \\ \mathtt{fib}^{aux}(x,x)\big)$$

which is linear but adds dependencies. In this transformation, the two invocations of `fib` have been contracted into one – the $\mathtt{fib}^{aux}$ invocation – that carries two arguments, one for every invocation – notice that the invocation $\mathtt{fib}^{aux}(\mathtt{y},\mathtt{z})$ corresponds to the two invocations $\mathtt{fib}(\mathtt{y})$ and $\mathtt{fib}(\mathtt{z})$. In the body of $\mathtt{fib}^{aux}$, the creation of the two names in `fib` is simulated by creating two names as well. However, these two names *are the same* both for the first argument and for the second one (that correspond to the two recursive invocations of `fib`). This results into fake dependencies between names originally belonging to two different and independent invocations. In particular, after two steps of evaluation, $\mathtt{fib}^{aux}(x,x)$ gives,

$$(x,y) \| (x,z) \| (y,y') \| (y,z') \| (z,y') \| (z,z') \| \mathtt{fib}^{aux}(y',z')$$

whilst the corresponding lam of the original nonlinear program is

$$(x,y) \| (x,z) \| (y,y') \| (y,y'') \| (z,z') \| (z,z'') \| \mathtt{fib}(y') \| \mathtt{fib}(y'') \| \mathtt{fib}(z') \| \mathtt{fib}(z'')\ .$$

We demonstrate that, if no circularity is manifested by the saturated state of $\mathtt{fib}^{aux}$ then the original fibonacci program is circularity-free. Since the circularity-freeness of $\mathtt{fib}^{aux}$ is decidable, by Theorem 1, then, in case of circularity-freeness, we are able to state the same property for `fib`.

4 The Technique by Examples

We illustrate our analysis technique by discussing a number of programs written in `FJf`. Every example is discussed as follows:

1. we first present the `FJf` description;
2. then we give the associated lam, by keeping informal the association technique (see [5] for on a inference system defining the formal association);
3. we finally inspect and evaluate lams looking for circularities.

Getting started: a simple deadlocked program. Consider the following class `C` with three synchronized methods `m1`, `m2`, and `m3`:

```
class C {
     C m1( C y, C z ) { return y!m2(z).get ; }
     C m2( C z ){ return z!m3( ).get ; }
     C m3( ) { return new C( ) ; }
}
```

An invocation `x!m1(y,z)` spawns a thread (in the thread pool of `x`) that will invoke `y!m2(z)` and immediately blocks because of the `get` (hence it will not release the lock on `x`), waiting for the value that is returned by `m2`. The invocation `y!m2(z)`, in turn, causes a new thread to be spawned (in the thread pool of `y`) that invokes `z!m3()` and blocks waiting for its result. The lam functions associated to the above methods are

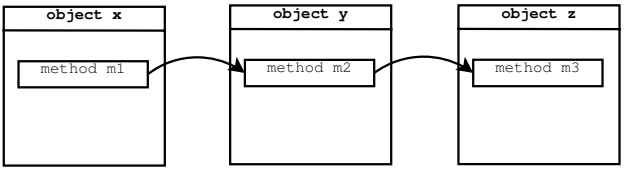
$$\begin{aligned}\texttt{m1}(x,y,z) &= (x,y)\|\texttt{m2}(y,z)\\ \texttt{m2}(x,y) &= (x,y)\|\texttt{m3}(y)\\ \texttt{m3}(x) &= \texttt{0}\end{aligned}$$

That is, to every method declaration we associate a lam declaration with an additional first argument representing the object `this` (which is always x). The body of the lam declaration ignores every local computation and translates `FJf` method invocations into lam function invocations and `get` operations into dependency pairs.

If the code of the main is

```
new C()!m1( new C(), new C() )
```

then the evaluation of the program will lock objects as depicted below



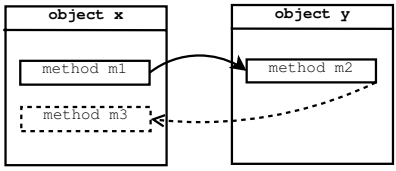
and no deadlock will appear. Since in this case the lam program is not recursive, our analysis technique is straightforward: just evaluate completely the lam program and verify whether the final lam is circularity-free. In fact, we get:

$$\begin{aligned}\langle \mathbb{I}_{x,y,z},\, \texttt{m1}(x,y,z)\rangle &\longrightarrow \langle \mathbb{I}_{x,y,z},\, (x,y)\|\texttt{m2}(y,z)\rangle \longrightarrow \langle \mathbb{I}_{x,y,z},\, (x,y)\|(y,z)\|\texttt{m3}(z)\rangle\\ &\longrightarrow \langle \mathbb{I}_{x,y,z},\, (x,y)\|(y,z)\|\texttt{0}\rangle\end{aligned}$$

which has no circularity. On the contrary, if the code of `main` is

```
C x = new C(); x!m1( new C(), x )
```

then a deadlock will occur. In fact, in this case $x = z$ and the thread executing `y!m2(x)` will block waiting for the result of `x!m3()`. In turn, this method will never be executed since the object `x` is locked by the initial invocation, as depicted below



This circularity is exactly what our analysis is able to catch. In this case, the main lam is $\mathtt{m1}(x,y,x)$ and the computation is

$$\begin{array}{l}\langle \mathbb{I}_{x,y},\, \mathtt{m1}(x,y,x)\rangle \longrightarrow \langle \mathbb{I}_{x,y},\, (x,y)\|\mathtt{m2}(y,x)\rangle \longrightarrow \langle \mathbb{I}_{x,y},\, (x,y)\|(y,x)\|\mathtt{m3}(x)\rangle \\ \qquad\qquad\qquad\quad \longrightarrow \langle \mathbb{I}_{x,y},\, (x,y)\|(y,x)\|\mathtt{0}\rangle\end{array}$$

which has a final state with a circularity.

Scheduler's choices and deadlocks. Deadlocks that are usually difficult to detect are those caused by scheduler's choices. Consider the following class `D`

```
class D {
   D m1( D y, D z ) { return new D()!m4(y,z).get ; z!m2(y) ; new D() ; }
   D m2( D y ){ return y!m3( ).get ; }
   D m3( ) { return new D( ) ; }
   D m4( D y, D z ) { return y!m2(z) ; new D() ; }
}
```

and evaluate `(new D)!m1(new D, new D)`. This evaluation is nondeterministic: it may yield a deadlock or not according to the scheduling of the threads. In particular, the execution terminates successfully if the lock of the object `z` is grabbed by `z!m3()` (inside `m2`) before the invocation of `z!m2(y)` (inside `m1`) obtains the lock. If, on the contrary, the lock on `z` is taken by the invocation of `z!m2(y)` inside `m1`, then a deadlock occurs. Let us analyze the program with our technique. The lam functions associated to the above methods are:

$$\begin{array}{l}\mathtt{m1}(x,y,z) = (x,u)\|\mathtt{m4}(u,y,z)\ ;\ \mathtt{m2}(z,y)\|\mathtt{m2}(y,z)\ ,\\ \mathtt{m2}(x,y) = (x,y)\|\mathtt{m3}(y)\ ,\\ \mathtt{m3}(x) = \mathtt{0}\ ,\\ \mathtt{m4}(x,y,z) = \mathtt{m2}(y,z)\ ,\end{array}$$

In this case, the lam of `m1` has shape $\mathtt{T}_1\,;\,\mathtt{T}_2$ because the corresponding code spawns two threads in sequence: the invocation of `m2` after the termination of the invocation of `m4`. In turn, since the invocation of `m4` spawns an asynchronous behavior (the asynchronous invocation of `m2`), then $\mathtt{T}_2$ also contains an invocation of `m2` caused by `m4`. That is, the leftmost invocation of `m2` in $\mathtt{T}_2$ is due to the code of `m1`, the rightmost one is due to the invocation of `m4`.

As before, this lam program is not recursive and, in order to analyze it, we have to compute the final state, assuming $\mathtt{m1}(x,y,z)$ as the main function:

$$\begin{array}{l}\langle \mathbb{I}_{x,y,z},\, \mathtt{m1}(x,y,z)\rangle \longrightarrow \langle \mathbb{I}_{x,y,z} \oplus (x,y,z<u),\, (x,u)\|\mathtt{m4}(u,y,z)\ ;\ \mathtt{m2}(z,y)\|\mathtt{m2}(y,z)\rangle \\ \qquad\qquad\qquad\quad \longrightarrow \langle \mathbb{I}_{x,y,z} \oplus (x,y,z<u),\, (x,u)\|\mathtt{m2}(y,z)\ ;\ \mathtt{m2}(z,y)\|\mathtt{m2}(y,z)\rangle \\ \qquad\qquad\quad \longrightarrow \ldots \longrightarrow \langle \mathbb{I}_{x,y,z} \oplus (x,y,z<u),\, (x,u)\|(y,z)\ ;\ (z,y)\|(y,z)\rangle\end{array}$$

thus manifesting a circularity.

The cooperative factorial function. Let us extend the language of Section 2.1 with the primitive type `int`, the arithmetic operations and the conditional expression. The meanings of these features are standard. Then consider the following class `Maths` where the method `fact(n)` computes the factorial of the of the argument

(when positive) by performing a recursive invocation on a newly created object. The evaluation produces a finite chain of newly created threads waiting for the termination of the next thread on a new object.

```
class Maths {
   int fact(int n) { return if (n==0) then 1 ;
                             else n*((new Maths)!fact(n-1).get) ; }
}
```

The evaluations of `fact(n)` never yield a deadlock, since no circular dependency can be constructed. This may be verified by translating the program into the lam formalism

$$\big(\ \mathtt{fact}(x) = (x,y) \| \mathtt{fact}(y),\ \mathtt{fact}(x)\big)$$

(the integer parameter is abstracted away, and the method shows only the parameter x representing the object `this`) and evaluating $\mathtt{fact}(x)$ till the saturated state, as discussed in Section 3. We notice that the conditional expression is translated into a lam by gathering the dependencies of the two branches, a standard technique in static analysis (in this case, the dependencies of the `then`-branch are empty).

A complex recursive pattern. As an example of recursive program, consider the following `FJf` class:

```
class R { R m1(R y, R z){ return y!m1(z,new R()).get ; z } }
```

The method `m1` is a recursive method (of the same kind of the foregoing factorial function) that invokes `m1` on its first argument with the second argument that is a new object. It does not generate any circularity when invoked with different arguments. However, several concurrent instances of its may produce a circular dependency. For instance, consider the main expression containg *two* invocations of `m1` with the swapped arguments:

```
R x = new R() ; R y = new R() ; R z = new R() ; x!m1(y,z); x!m1(z,y)
```

(we are using *local variables*; these may be easily encoded in our calculus as arguments of additional auxiliary methods). The associated lam program is

$$\big(\ \mathtt{m1}(x,y,z) = (x,y) \| \mathtt{m1}(y,z,w)\ ,\ \mathtt{m1}(x,y,z) \| \mathtt{m1}(x,z,y)\ \big)$$

which is linear. Our theory guarantees that, unfolding six times the two invocations of `m1`, it is possible to establish the circularity-freeness of the program. This means that, in order to get the saturated state we have a computation of length twelve. In particular, after four steps of the computation, we get

$$\begin{array}{rl} \big\langle \mathbb{I}_{x,y,z},\ \mathtt{m1}(x,y,z) \| \mathtt{m1}(x,z,y) \big\rangle & \longrightarrow^2\ \big\langle \mathbb{V}_1,\ (x,y) \| (x,z) \| \mathtt{m1}(y,z,u) \| \mathtt{m1}(z,y,v) \big\rangle \\ & \longrightarrow^2\ \big\langle \mathbb{V}_2,\ (x,y) \| (x,z) \| (y,z) \| (z,y) \| \mathtt{m1}(z,u,u') \| \mathtt{m1}(y,v,v') \big\rangle \end{array}$$

where $\mathbb{V}_1 = \mathbb{I}_{x,y,z} \oplus (x,y,z < u) \oplus (x,y,z < v)$ and $\mathbb{V}_2 = \mathbb{V}_1 \oplus (y,z,u < u') \oplus (y,z,v < v')$. Since the last state has a circular dependency, we can stop the analysis here and deduce that the corresponding program deadlocks.

Another complex recursive pattern. The following `FJf` class manifest another issue about saturation. Let

```
class Rec {
    Rec m1(Rec y, Rec z, Rec w){
        return z!m2(y) ; this!m2(z) ; w!m2(z) ; y!m1(this,w,new Rec()) ; z ;
    }
    Rec m2(Rec y){ return y!m3().get; }
    Rec m3() { new Rec(); }
}
```

and let `new Rec()!m1(new Rec(), new Rec(), new Rec())` be the main expression. The lam function associated to the above methods are

$$\begin{array}{l}\mathtt{m1}(x,y,z,w) = \mathtt{m2}(z,y) \| \mathtt{m2}(x,z) \| \mathtt{m2}(w,z) \| \mathtt{m1}(y,x,w,w') \ , \\ \mathtt{m2}(x,y) = (x,y) \| \mathtt{m3}(y) \ , \\ \mathtt{m3}(x) = \mathtt{0} \ , \end{array}$$

and our theory guarantees that the saturated state is obtained after four unfoldings of `m1` (and completely unfolding the auxiliary method invocations `m2` and `m3`). However, we notice that it is possible to obtain a state with a circularity after two unfoldings of $\mathtt{m1}(x,y,z)$ (which we assume as the main function):

$$\begin{array}{rl} \langle \mathbb{I}_{x,y,z},\, \mathtt{m1}(x,y,z) \rangle & \longrightarrow^* \langle \mathbb{V},\ (z,y) \| (x,z) \| (w,z) \| \mathtt{m1}(y,x,w) \rangle \\ & \longrightarrow^* \langle \mathbb{V}',\ (z,y) \| (x,z) \| (w,z) \| (w,x) \| (y,w) \| (w',w) \| \mathtt{m1}(x,y,w') \rangle \end{array}$$

where $\mathbb{V} = \mathbb{I}_{x,y,z} \oplus (x,y,z < w)$ and $\mathbb{V}' = \mathbb{V} \oplus (y,x,w < w')$. In particular, the last state has circularity $(z,y) \| (y,w) \| (w,z)$. However, if we perform a further unfolding of $\mathtt{m1}(x,y,w')$ we get the relation

$$(z,y) \| (x,z) \| (w,z) \| (w,x) \| (y,w) \| (w',w) \| (w',y) \| (x,w') \| (w'',w')$$

that manifests the additional circularity (w,x) (x,w') (w',w), which has no counterpart in the previous states (it cannot be mapped to a past circularity). That is, in order to have a complete account of circular dependencies, it is necessary to compute the lam till the saturated state.

The cooperative fibonacci function. A paradigmatic program that yields a non-linear lam is the one computing fibonacci numbers. Consider to augment the above class `Maths` with the following method `fib`

```
int fib(int n) {
  return if n==0 then 1 ;
     else if n==1 then 1 ;
     else new Maths()!fib(n-1).get + new Maths()!fib(n-2).get ; }
}
```

that implements the standard recursive algorithm of fibonacci. As in the factorial example, the above code is a cooperative solution: the recursive invocations are performed on new objects every time, so that the result is computed in

parallel threads. The values of the spawned threads are retrieved by `get` operations, which corresponds to synchronization points. To analyze this program, we consider the associated lam:

$$\big(\ \texttt{fib}(x) = (x, y) \| (x, z) \| \texttt{fib}(y) \| \texttt{fib}(z),\ \texttt{fib}(x)\ \big)$$

(as before, the integer parameter is abstracted away and the method only carries the parameter x representing the object `this`). This lam program is not linear and, as discussed in Section 3, it is circularity-free.

5 Additional Issues

Two relevant extensions of the language `FJf` are field assignment and the operation `await`. In this section we indicate the techniques we are developing for coping with them.

Fields and assignments. The update operation increases the difficulties of the deadlock analysis. In particular, a field of an object that is modified by concurrent threads has a nondeterminate final value. This, in turn, may cause unpredictable behaviors due to the scheduling of the threads (see also Section 4). In [7] there is a preliminary study of this issue. For instance, consider the two classes `E` and `F` below

```
class E {
      E m1( F y ) { return  y!n1 ; new E() ; }
      E m2( ) { return new E( ); }
 }
class F {
      E f;
      E n1( ) { return  f = this ; }
      E n2( ) { return f!m2().get ; }
 }
```

and the main expression

```
F x = new F( new E() ) ; new E()!m1(x) ; x!n2() ;
```

(as before, we are using *local variables*). The evaluation of this expression yields a thread (spawned inside `m1`) that modifies the field `f` of `x` (because of the invocation `x!n1`). This thread is concurrent with the invocation `x!n2()`. If the execution flow is such that the invocation `f.m2()` (inside `x!n2()`) is evaluated before the assignment, then the execution will terminate (with a new object stored in `f`), otherwise `f` will contain a reference to `this` and the computation deadlocks.

In order to take into account the updates, we extend lams with the possibility of specifying *sets* of objects, which model the possible values that may be stored in fields. In particular, the lam program associated to the above code is

$$\big(\ \texttt{m1}(x, y, \mathcal{Y}) = \texttt{n1}(y, \mathcal{Y}),\ \texttt{m2}(x) = 0,\ \texttt{n1}(z, \mathcal{Z}) = 0,\ \texttt{n2}(z, \mathcal{Z}) = \texttt{m2}(\mathcal{Z}) \| (z, \mathcal{Z}),\ \texttt{m1}(x, y, \{x, y\}) \| \texttt{n2}(y, \{y, z\})\ \big)$$

where $\mathcal{Y}$ is the set of possible values the fields of the object y may be assigned. Analogously for $\mathcal{Z}$ with respect to z. Notice that the first arguments of `m1` and `m2` are of class `E`, therefore they do not have any fields. This is why they do not have any associated set. The body of `n2` has the pair $(z, \mathcal{Z})$, which represent the set of pairs $\{(z, z') \mid z' \in \mathcal{Z}\}$. The execution of the lam program will be the following:

$$\begin{aligned}\langle \mathbb{I}_{x,y,z},\, \mathtt{m1}(x, y, \{x, y\}) \| \mathtt{n2}(y, \{y, z\}) \rangle &\longrightarrow \langle \mathbb{I}_{x,y,z},\, \mathtt{n1}(y, \{x, y\}) \| \mathtt{n2}(y, \{y, z\}) \rangle \\ &\longrightarrow \langle \mathbb{I}_{x,y,z},\, \mathtt{n2}(y, \{y, z\}) \rangle \\ &\longrightarrow \langle \mathbb{I}_{x,y,z},\, (y, \{y, z\}) \| \mathtt{m2}(\{y, z\}) \rangle \\ &\longrightarrow \langle \mathbb{I}_{x,y,z},\, (y, \{y, z\}) \rangle \longrightarrow \langle \mathbb{I}_{x,y,z},\, (y, y) \| (y, z) \rangle\end{aligned}$$

The final state has a circular dependency, i.e. (y, y).

Await and livelocks. A further synchronization operator of ABS is `await` (see [8,5] for its formal semantics and examples). The operation `await` suspends the current thread, by releasing the lock on its own object, while waiting for a thread termination. Later on, the thread competes again for grabbing the lock and, when acquired, it tries again for the result. If it is available, the computation proceeds, otherwise the lock is released again and so on. With this semantics, a circular dependency does not correspond to a blocking situation, but to a situation where one or more threads are caught in an infinite loop of getting and releasing the lock. This phenomenon is usually called a *livelock*.

In the presence of livelocks, a circular dependency may not necessarily be a bad situation. Let us discuss this issue. A dependency pair (x, y) in a livelock analysis means that *some thread on* x is waiting – not busy-waiting – for the result of *some thread on* y. Under this meaning, the term $(x, y) \| (y, x)$, which is signaled as an incorrect state by our technique, may be safe because the involved threads can be different. In fact, in this case, since the threads do not busy-wait on the objects x and y, the computation terminates successfully.

In order to distinguish between the two types of circularities, we extend the names used in lams with *thread names*. In this extension – lams with two sorts of names, thread names have a "type", which is the (object) name –, the livelocks are those manifested by terms such as $(\mathtt{t}, \mathtt{t}') \| (\mathtt{t}', \mathtt{t})$; while terms as $(\mathtt{t}, \mathtt{t}') \| (\mathtt{t}', \mathtt{t}'')$, even if `t` and $\mathtt{t}''$ have the same object type, are painless.

6 Conclusion

This paper surveys a technique for the static analysis of deadlocks. This technique associates abstract descriptions, called lams, to programs and then evaluates such descriptions to catch circular dependencies. The technique does not use any pre-defined partial order of resources and does account for dynamic resource creation. We stuck to a popularizing exposition; therefore the technical details have been omitted and several examples should help in clarifying the difficult points.

We are currently experiencing our technique in the HATS European project (`www.hats-project.eu`) on large programs written in an object-oriented language with futures [4]. An inference systems for deriving lams [5], for a subset of

this language, has been defined and additional annotations allow us to instruct the inference system when structured data types and iterations occur.

Lams have been used in the first place for detecting the so-called *resource allocation deadlocks*, as encountered in e.g. operating systems. However, our technique seems also adequate for deadlocks due to process synchronizations, as those in process calculi [11,10,9]. A thorough comparison with these works is scheduled for the next future.

An interesting direction of research is the application of our algorithm for static analysis to verify properties different than deadlocks. This might boil down to devise languages different than lams and to different definitions of saturated states.

References

1. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L.: Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development 54(5), 3 (2010)
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program: preventing data races and deadlocks. In: Proc. OOPSLA 2002, pp. 211–230. ACM (2002)
3. Comtet, L.: Advanced Combinatorics: The Art of Finite and Infinite Expansions, Dordrecht, Netherlands (1974)
4. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis in practice (submitted, 2013), `http://www.cs.unibo.it/~laneve`
5. Giachino, E., Laneve, C.: Deadlock and livelock analysis in concurrent objects with futures, Technical Report (2013), `http://www.cs.unibo.it/~laneve`
6. Giachino, E., Laneve, C.: Mutations, flashbacks and deadlocks (submitted, 2013), `http://www.cs.unibo.it/~laneve`
7. Giachino, E., Lascu, T.A.: Lock analysis for an asynchronous object calculus. In: ICTCS 2012 (2012)
8. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
9. Kobayashi, N.: Type systems for concurrent programs. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003)
10. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. and Comput. 100, 41–77 (1992)

Formal Modeling and Reasoning about the Android Security Framework

Alessandro Armando[1,2], Gabriele Costa[2], and Alessio Merlo[3]

[1] Fondazione Bruno Kessler
armando@fbk.eu
[2] Università degli Studi di Genova
gabriele.costa@unige.it
[3] Università E-Campus
alessio.merlo@uniecampus.it

Abstract. Android OS is currently the most widespread mobile operating system and is very likely to remain so in the near future. The number of available Android applications will soon reach the staggering figure of 500,000, with an average of 20,000 applications being introduced in the Android Market over the last 6 months. Since many applications (e.g., home banking applications) deal with sensitive data, the security of Android is receiving a growing attention by the research community. However, most of the work assumes that Android meets some given high-level security goals (e.g. sandboxing of applications). Checking whether these security goals are met is therefore of paramount importance. Unfortunately this is also a very difficult task due to the lack of a detailed security model encompassing not only the interaction among applications but also the interplay between the applications and the functionalities offered by Android. To remedy this situation in this paper we propose a formal model of Android OS that allows one to formally state the high-level security goals as well as to check whether these goals are met or to identify potential security weaknesses.

1 Introduction

Modern smartphones not only act as cell phones, but also as handheld personal computers, where users manage their personal data, interact with online payment systems, and so on. As stated in [12], *"a central design point of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc."*. Android strives to achieve this security goal through a cross-layer security architecture, the Android Security Framework (ASF), leveraging the access control mechanisms offered by the underlying Linux kernel.

Recent work (e.g., [2,23,19,16]) unveiled a plethora of vulnerabilities occurring at different layers of the Android stack and a number of extensions to the Android

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 64–81, 2013.
© Springer-Verlag Berlin Heidelberg 2013

native security policies (e.g., [17]) and to the framework itself (e.g., [13,8]) have been put forward. However, a systematic assessment of the ASF and of the proposed solutions is very difficult to achieve. Mainly, this is due to the lack of a detailed security model encompassing not only the interaction among applications but also the interplay between the applications and the functionalities offered by Android.

In this work we focus on modeling the Android OS in order to overcome the aforementioned aspects. The contribution of this paper is twofold. Firstly, we propose a formal model of Android that allows us to formally describe the security-relevant aspects of the ASF. Secondly, we present a type and effect system that we use for both producing the model of a platform and verifying whether it meets some expected security goals. For modeling, we adopt a process algebra-like formalism, namely *history expressions* [6], that can be exploited for different purposes, as we detail in the following.

Structure of the paper. In Section 2 we briefly introduce the architecture of Android and the principal interactions. In Section 3 we describe the ASF and its enforcement mechanisms. In Section 4 we present our formal model for the Android Security Framework. In Section 5 we present our type and effect system, we prove its key properties and we describe its possible exploitations. Finally, in Section 6 we draw some concluding remarks.

2 Android Architecture

The Android stack can be represented with 5 functional levels: Application, Application Framework, Application Runtime, Libraries and the underlying Linux kernel.

1. *Application Layer.* It includes both system (home,browser,email,..) and user-installed Java applications. Applications are made of *components* corresponding to independent execution modules, that interact with each others. There exist four kinds of components: 1) *Activity*, representing a single application screen with a user interface, 2) *Service*, which is kept running in background without interaction with the user, 3) *Content Provider*, that manages application data shared among components of (potentially) distinct applications, and 4) *Broadcast Receiver* which is able to respond to system-wide broadcast announcements coming both from other components and the system. Components are defined in *namespaces* that map components to a specific name which allow to identify components in the system.
2. *Application Framework.* It provides the main OS services by means of a set of APIs. This layer also includes services for managing the device and interacting with the underlying Linux drivers (e.g. *Telephony Manager* and *Location Manager*).
3. *Android Runtime.* This layer comprises the Dalvik virtual machine, the Android's runtime's core component which executes applications.
4. *Libraries.* It contains a set of C/C++ libraries providing useful tools to the upper layers and for accessing data stored on the device. Libraries are widely used by the Application Framework services.

5. *Linux kernel.* Android relies on a Linux kernel for core system services. Such services include process management and drivers for accessing physical resources and Inter-Component Communication (ICC).

2.1 Interactions in Android

In Android, interactions can be *horizontal* (i.e. application to application) or *vertical* (i.e. application to underlying levels). Horizontal interactions are used to exploit functionalities provided by other applications, while vertical ones are used to access system services and resources. Component services are invoked by means of a message passing paradigm, while resources are referred by a special formatted URI. Android URIs can also be used to address a content provider database.

Horizontal interactions are based on a message abstraction called *intent*. Intent messaging is a facility for dynamic binding between components in the same or different applications. An intent is a passive data structure holding an abstract description of an operation to be performed (called *action*) and optional data in URI format. Intents can be *explicit* or *implicit*. In the former case, the destination of the action is explicitly expressed in the intents (through the *name* of the receiving application/component), while in the latter case the system has to determine which is the target component accordingly to the action to be performed, the optional data value and the applications currently installed in the system.

Intent-based communications are granted by a kernel driver called Binder which offers a lightweight capability-based remote procedure call mechanism. Although intent messaging passes through the Binder driver, it is convenient to maintain intent's level of abstraction for modeling purpose. In fact, every Android application defines its entry points using *intent filters* which are lists of intent's actions that can be dispatched by the application itself. Furthermore, an intent can be used to start activities, communicate with a service or send broadcast messages.

Vertical interactions are used by applications to access system resources and functionalities which are exposed through a set of APIs. Although system calls can cause a cascade of invocations in the lower layers, possibly reaching the kernel, all of them are mediated by the application framework APIs. Hence, APIs mask internal platform details to the invoking applications. Internally, API calls are handled according to the following steps. When an application invokes a public API in the library, the invocation is redirected to a private interface, also in the library. The private interface is an RPC stub. Then, the RPC stub initiates an RPC request with the system process that asks a system service to perform the requested operation.

3 Android Security Framework

The Android Security Framework (ASF) consists of a set of decentralized security mechanisms spanning on all layers of the Android stack. The ASF enforces an informal and cross-layer security policy focused on the concept of *permission*.

3.1 Android Permissions

In Android, a permission is a string expressing the ability to perform a specific operation. Permissions can be system-defined or user-defined. Each application statically declares the set of permissions it requires to work properly. Such a set is generally a superset of the permissions effectively used at runtime. During installation of an application, the user must grant the whole set of required permissions, otherwise the installation is aborted. Once installed, an application cannot modify such permissions.

Each application package contains an XML file, called *Android Manifest*, containing two types of permissions:

- **declared permissions** are defined by the application itself and represent access rights that other applications must have for using its resources.
- **requested permissions** representing the permissions held by the application.

Since permissions specified in the manifest are static (i.e., they cannot possibly change at runtime), they are not suited to regulate access to resources that are dynamically defined by the application (e.g., shared data from a content provider). For this reason Android APIs include special methods for dynamically (i) *granting*, (ii) *revoking* and (iii) *checking* access permissions. These methods handle per-URI permissions thereby letting the application give temporary access privileges for some owned URI to other applications.

3.2 Android Security Policy

The Android security policy defines restrictions on the interactions among applications and between each application and the system. The Android security policy is globally enforced by the ASF. Both the policy and the ASF strongly rely on permissions associated with the components. We detail here the security policy related to the architecture and the interactions explained in Sec. 2.

Horizontal interactions. Horizontal interactions between components are carried out through permissions associated with intents. Each application can declare a list of permissions for their incoming intents. When application A sends an intent I to application B, the platform delivers I only if A has the privileges (granted at installation time) requested by B. Otherwise, the intent does not reach B (still, it could be delivered to other recipients).

Vertical interactions. By default, an Android application can only access a limited range of system resources. These restrictions are implemented in different forms. Some capabilities are restricted by an intentional absence of APIs mediating sensitive accesses. For instance, there is no API that allows for the direct manipulation of the SIM card. In other cases, the sensitive APIs are reserved for trusted applications and are protected through permissions.

Each API in the library layer is associated with a certain permission. Once a component invokes an API, the privileges of the component are checked against

the permission required for the API. If an application attempts to invoke an API without having the proper privileges, a security exception is thrown and the invocation fails.

Linux layer and IPC. The Android platform takes advantage of the Linux user-based access control model as a means to identify and isolate application resources. The Android system assigns a unique user ID (UID) to each Android application and runs it as that user in a separate Linux process. This sets up a kernel-level *Application Sandbox*. This approach uses the native Linux isolation for users to implement a security policy that avoids direct communications among Android applications by forcing all their interactions to rely upon the IPC system. However, such a policy does not prevent a Linux process (running an application) from communicating through one of the native UNIX mechanisms such as sockets or files. Notice that the Linux permissions apply on such channels.

The previous analysis shows the cross-layer nature of Android Security policy. The ASF is distributed and involves distinct security-relevant aspects (from UID and GID at Linux layer to human-readable and high level Android permissions). Assessing the effectiveness of all the security mechanisms and their interplay is difficult due to such heterogeneity and the lack of a detailed and comprehensive model of the security-relevant aspects of Android.

4 Android Model

In this section we describe how we model Android applications and components. In particular, we introduce a framework for defining an application in terms of its (*i*) components, (*ii*) manifest and (*iii*) name space. Moreover, we present a formal semantics for describing computations in our model. We believe that our framework can be used to accurately describe most of the security-relevant aspects of the Android OS. Indeed, even though Java-like languages have been proposed for the application of formal methods, e.g., see [7,15], here we aim at focussing on application-to-application and application-to-system interactions which do not depend on object orientation. To this purpose, we show, through examples, that our model covers a number of security flaws that have been recently reported.

4.1 Applications and Components

In Table 1 we report the syntax of the elements of our framework.

Intuitively, an application A is a triple consisting of a manifest M, a naming function Δ and a finite list of components $\bar{C} = C_1 \dots C_n$. A manifest M contains three parts: requested permissions Π, declared permissions P and an intents resolution function Λ. The permission request part Π is a finite sequence of (*i*) intent permission requests $\rho\alpha$ and (*ii*) system permission requests $\rho\sigma$. Instead, the permission declaration P is a list of pairs $(\alpha, \bar{u})$ binding intent names α, α'

Table 1. Syntax of applications and components

$$
\begin{array}{rll}
A & ::= \langle M, \Delta, \bar{C} \rangle & \text{Application} \\
M & ::= \Pi; P; \Lambda & \text{Manifest} \\
\Pi & ::= \varepsilon \mid \rho\alpha.\Pi \mid \rho\sigma.\Pi & \text{Permission requests} \\
P & ::= \varepsilon \mid (\alpha, \bar{u}).P & \text{Exported permissions} \\
\Lambda & ::= \varepsilon \mid (\alpha \mapsto \eta).\Lambda & \text{Intent binding} \\
\Delta & ::= \emptyset \mid \Delta\{C/\eta\} & \text{Name space} \\
C & ::= \mathtt{skip} \mid \mathtt{icast}\, E \mid \mathtt{ecast}\, \eta\, E \mid \mathtt{grant}_\sigma\, \eta\, E \mid & \text{Statements} \\
 & \quad \mathtt{revoke}_\sigma\, \eta\, E \mid \mathtt{check}_\sigma\, \eta\, E \mid \mathtt{new}\, x\, \mathtt{in}\, C \mid \mathtt{receive}_\alpha\, x \mapsto C \mid \\
 & \quad \mathtt{apply}\, E\, \mathtt{to}\, E' \mid \mathtt{system}_\sigma\, E \mid \mathtt{if}\, (E = E')\{C\}\, \mathtt{else}\, \{C'\} \mid C; C' \\
E & ::= \mathtt{null} \mid u \mid x \mid I_\alpha(E, E') \mid E.\mathtt{d} \mid & \text{Expressions} \\
 & \quad E.\mathtt{e} \mid \mathtt{proc}\, f(x)\{C\}
\end{array}
$$

to lists of resources $\bar{u} = u_1, \ldots, u_k$. The function Λ maps each intent name α to a set of (identifiers of) components $\{\eta_1, \ldots, \eta_n\}$ that can serve it, namely the available *receivers*. Finally, Δ resolves components identifier η, η' into actual components C, C'.

Software components are obtained from the composition of *statements* and *expressions*. Expressions, ranged over by E, E', can be $\mathtt{null}$, resources u, u', variables x, y, intents constructors $I_\alpha(E, E')$, data and extra field getters ($E.\mathtt{d}$ and $E.\mathtt{e}$, respectively) or procedure declarations $\mathtt{proc}\, f(x)\{C\}$ (where f is bound in C).

Similarly, statements, denoted by C, C', can be a $\mathtt{skip}$ command, an implicit intent cast $\mathtt{icast}\, E$, an explicit intent cast $\mathtt{ecast}\, \eta\, E$, an access permission grant $\mathtt{grant}_\sigma\, \eta\, E$, a permission revocation $\mathtt{revoke}_\sigma \eta\, E$, a permission checking $\mathtt{check}_\sigma\, \eta\, E$, a fresh resource creation $\mathtt{new}\, x\, \mathtt{in}\, C$, an intent receiver $\mathtt{receive}_\alpha\, x \mapsto C$, an application of a procedure to a parameter $\mathtt{apply}\, E\, \mathtt{to}\, E'$, a system call $\mathtt{system}_\sigma\, E$, a conditional branching $\mathtt{if}\, (E = E')\{C\}\, \mathtt{else}\, \{C'\}$ or a sequence $C; C'$.

4.2 Operational Semantics

The behaviour of programs follows the small step semantics rules given in Table 2. Computations are sequences of reductions steps from a source configuration to a target one. For expressions, a configuration only contains the element under evaluation E. The operational semantics reduces expressions E, E' to *values* v, v'. A value can be either the void element $\perp$, a resource u, a intent $I_\alpha(u, v)$, or a procedure $\mathtt{proc}\, f(x)\{C\}$. If no reductions apply to a configuration E (where E is not a value), we write $E \not\to$ and we say it to be *stuck*.

The semantic rules for commands are more tricky. Basically, we evaluate statements under a configuration U, Φ, C where C is the program under computation, U is a resources ownership function (i.e., $U(\eta) = \mathcal{U}$ means that the component

Table 2. Semantics of expressions and statements (fragment)

$$(\texttt{E-NULL})\ \texttt{null} \to \bot \quad (\texttt{E-FLD})\ \frac{E \to E'}{E.f \to E'.f} \quad (\texttt{E-DATA})\ I_\alpha(u,v).\texttt{d} \to u$$

$$(\texttt{E-EXT})\ I_\alpha(u,v).\texttt{e} \to v \quad (\texttt{E-INT}_\texttt{L})\ \frac{E \to E''}{I_\alpha(E,E') \to I_\alpha(E'',E')} \quad (\texttt{E-INT}_\texttt{R})\ \frac{E \to E'}{I_\alpha(v,E) \to I_\alpha(v,E')}$$

$$(\texttt{S-SKIP})\ U,\Phi,\texttt{skip} \rightsquigarrow U,\Phi,\cdot \qquad (\texttt{S-GRNT})\ \frac{\texttt{self} = \eta' \quad u \in U(\eta') \quad \Phi' = \Phi \cup \{(\eta,\sigma,u)\}}{U,\Phi,\texttt{grant}_\sigma\,\eta\,u \rightsquigarrow U,\Phi',\cdot}$$

$$(\texttt{S-REVK})\ \frac{\texttt{self} = \eta' \quad u \in U(\eta') \quad \Phi' = \Phi \setminus \{(\eta,\sigma,u)\}}{U,\Phi,\texttt{revoke}_\sigma\,\eta\,u \rightsquigarrow U,\Phi',\cdot}$$

$$(\texttt{S-ICST})\ \frac{\texttt{self} = \eta \quad \eta' \in \Lambda(\alpha) \quad \eta,\alpha,u \models \Phi}{U,\Phi,\texttt{icast}\ I_\alpha(u,v) \overset{\alpha_\eta^{\eta'}(u,v)}{\rightsquigarrow_\Lambda} U,\Phi,\cdot} \qquad (\texttt{S-ECST})\ \frac{\texttt{self} = \eta \quad \eta' \in \Lambda(\alpha) \quad \eta,\alpha,u \models \Phi}{U,\Phi,\texttt{ecast}\ \eta'\ I_\alpha(u,v) \overset{\alpha_\eta^{\eta'}(u,v)}{\rightsquigarrow_\Lambda} U,\Phi,\cdot}$$

$$(\texttt{S-CHK})\ \frac{\eta,\sigma,u \models \Phi}{U,\Phi,\texttt{check}_\sigma\,\eta\,u \rightsquigarrow U,\Phi,\cdot} \qquad (\texttt{S-SYS})\ \frac{\texttt{self} = \eta \quad \eta,\sigma,u \models \Phi}{U,\Phi,\texttt{system}_\sigma\,u \overset{\sigma_\eta(u)}{\rightsquigarrow} U,\Phi,\cdot}$$

$$(\texttt{S-NEW})\ \frac{\texttt{self} = \eta \quad \text{fresh}\,u}{U,\Phi,\texttt{new}\,x\,\texttt{in}\,C \rightsquigarrow U \cup \{\eta/u\},\Phi,C[u/x]}$$

$$(\texttt{S-APP})\ U,\Phi,\texttt{apply proc}\,h(y)\{C\}\,\texttt{to}\,v \rightsquigarrow U,\Phi,C[v/y,\texttt{proc}\,h(y)\{C\}/h]$$

$$(\texttt{S-CND})\ U,\Phi,\texttt{if}\,(v = v')\{C_{tt}\}\,\texttt{else}\,\{C_{ff}\} \rightsquigarrow U,\Phi,C_{\mathcal{B}(v=v')}$$

$$(\texttt{S-SEQ})\ \frac{U,\Phi,C \overset{b}{\rightsquigarrow} U',\Phi',C''}{U,\Phi,C;C' \overset{b}{\rightsquigarrow} U',\Phi',C'';C'} \qquad (\texttt{S-SEQ}^-)\ U,\Phi,\cdot;C \rightsquigarrow U,\Phi,C$$

(identified by) η owns the resources in $\mathcal{U}$) and Φ is the system policy (we write $\eta,\beta,u \models \Phi$ for $(\eta,\beta,u) \in \Phi$ with $\beta \in \{\alpha,\sigma\}$). Slightly abusing the notation, we also use $\cdot$ in the configuration to represent computation termination.

Computational steps consist of transitions from a source configuration to a target one. Transitions have the form $U,\Phi,C \overset{a}{\rightsquigarrow_\Lambda} U',\Phi',C'$ where a is an observable action, i.e., an intent or a system call, that the computation can perform and Λ is an intents destination table, i.e, $\Lambda(\alpha) = \{\eta_1,\ldots,\eta_k\}$ means that $\eta_1,\ldots,\eta_k$ are the candidates for handling an intent α. When not necessary, we feel free to omit a and Λ from the transitions.

According to the rules of table 2,[1] the commands behave as follows. The statement `skip` does not change the system state and terminates (S−SKIP). Both implicit (rule (S−ICST)) and explicit (rule (S−ECST)) casts produce an observable action $\alpha_\eta^{\eta'}(u,v)$ and reduce to $\cdot$. The only difference is that the receiver η' for an implicit cast can be any of the elements of the destination table Λ, while

[1] For brevity, table 2 only reports the rules which are more interesting our presentation. The full semantics can be found at `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`

an explicit cast declares the destination (which still must be a legal one, i.e., $\eta' \in \Lambda$). Note that, these reduction steps take place only if they are allowed by the current policy Φ. Permission granting and revocation (rules (S−GRNT) and (S−REVK)) are symmetrical. Indeed, granting a permission causes the current policy to be extended with a possibly new, allowed action, while revocation removes some existing privileges. Both the operations require u to be owned by the executing component, i.e., $u \in U(\eta')$ where $\texttt{self} = \eta'$. Then, a security check $\texttt{check}_\sigma\, \eta\, u$ interrupts the computation if η has no rights to access to u through σ (rule (S−CHK)). A system call $\texttt{system}_\sigma u$ is performed (rule (S−SYS)) if the current component is allowed to invoke it and generates a corresponding access action $\sigma_\eta(u)$ (where η is the source of the access σ). Resource creation (S−NEW) causes a statement C to be evaluated under a state in which a fresh resource u is associated to the variable x. As expected, the owner of the resource is the current component. Procedure application (rule (S−APP)) reduces to the computation of the procedure body C where the formal parameter y and the variable h are replaced with the actual parameter v and the procedure definition, respectively. A conditional statement (rule (S−CND)) reduces to one of its branches depending on the value of its guard (we write $\mathcal{B}(v = v')$ as an abbreviation of the two conditions $v = v'$ and $v \neq v'$ which evaluate to either *tt* or *ff*). Finally, a sequence of statements $C; C'$ behaves like C until it terminates and then reduces to the execution of C' (rules (S−SEQ) and (S−SEQ$^-$)).

In addition to the standard syntax, we define the following abbreviations which we adopt for the sake of presentation.

$$\texttt{if}\,(E \neq E')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E = E')\{C'\}\,\texttt{else}\,\{C\}$$

$$\texttt{if}\,(E_1 = E_1' \wedge E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E_1 = E_1')\{\,\texttt{if}\,(E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\}\}\,\texttt{else}\,\{C'\}$$

$$\texttt{if}\,(E_1 = E_1' \vee E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E_1 \neq E_1')\{\,\texttt{if}\,(E_2 \neq E_2')\{C'\}\,\texttt{else}\,\{C\}\}\,\texttt{else}\,\{C\}$$

$$\texttt{if}\,(E \in \mathcal{U})\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(\bigvee_{u \in \mathcal{U}} E = u)\{C\}\,\texttt{else}\,\{C'\}$$

$$\texttt{while}\,(E \neq v)\,\texttt{do}\,\{C\} \triangleq \texttt{apply proc}\, w(x)\{\,\texttt{if}\,(E \neq x)\{C;\,\texttt{apply}\, w\,\texttt{to}\, E\}\,\texttt{else}\,\{\,\texttt{skip}\,\}\,\texttt{to}\, v$$

Finally, we say that a configuration is *stuck* (we write $U, \Phi, S \not\rightsquigarrow_\Lambda$) if $S \neq \cdot$ and the configuration admits no transitions. In real Android systems, this situation corresponds to program termination or exception raising, but this aspect does not impact on our framework. If a configuration reduces to a stuck one, we say it to *go wrong*.

Example 1. Consider the following statement.

$$C = \texttt{apply proc}\, f(x)\{\texttt{system}_\sigma\, x;\, \texttt{icast}\, I_\alpha(x, \texttt{null})\}\,\texttt{to}\, u$$

We simulate a computation under a configuration U, Φ, C where $\Phi = \{(\eta, \sigma, u)\}$ and $\texttt{self} = \eta$. The resulting computation follows.

$U, \Phi,\ \texttt{apply proc}\, f(x)\{\texttt{system}_\sigma\, x;\, \texttt{icast}\, I_\alpha(x, \texttt{null})\}\,\texttt{to}\, u \rightsquigarrow_\Lambda U, \Phi, \texttt{system}_\sigma\, u;\, \texttt{icast}\, I_\alpha(u, \texttt{null})$
$\overset{\sigma_\eta(u)}{\rightsquigarrow_\Lambda} U, \Phi, \cdot;\, \texttt{icast}\, I_\alpha(u, \texttt{null}) \rightsquigarrow_\Lambda U, \Phi,\ \texttt{icast}\, I_\alpha(u, \texttt{null})$

The first step consists of a procedure application to an argument u. This reduces the statement to the procedure body where the variable x is replaced by u. The next step is a system call σ. Since η is allowed to perform access, i.e., $\eta, \sigma, u \models \Phi$, the statement fires the corresponding action and reduces to an implicit cast statement. Then, as $\eta, \alpha, u \not\models \Phi$, the computation cannot proceed further and the configuration is stuck.

4.3 Execution Context

As described in Section 2, application manifests declare (i) *activities*, (ii) *receivers* and (iii) *content providers*. The information contained in the manifest contribute to defining how the components interact with each other and with the platform. We describe this mechanism by means of an *execution context* (and its semantics) which we define below.

Definition 1. *An execution context (context for short) is* $\mathbf{P} = U, \Phi, [C_1]_{\eta_1} \cdots [C_n]_{\eta_n}$. *The operational semantics of a context is defined by the rules*

$$(\texttt{CTX}-\texttt{S}) \quad \frac{U, \Phi, C_j \overset{b}{\rightsquigarrow}_\Lambda U', \Phi', C'}{U, \Phi, [C_1]_{\eta_1} \cdots [C_j]_{\eta_j} \cdots [C_n]_{\eta_n} \overset{b}{\Rightarrow}_\Lambda U', \Phi', [C_1]_{\eta_1} \cdots [C']_{\eta_j} \cdots [C_n]_{\eta_n}}$$

$$(\texttt{CTX}-\texttt{I}) \quad \frac{U, \Phi, C_i \overset{\alpha^{\eta_j}_{\eta_i}(u,v)}{\rightsquigarrow}_\Lambda U', \Phi', C'}{U, \Phi, \cdots [C_i]_{\eta_i} \cdots [\texttt{receive}\, x \mapsto C]_{\eta_j} \cdots \Rightarrow_\Lambda U', \Phi', \cdots [C']_{\eta_i} \cdots [C\{I_\alpha(u,v)/x\}]_{\eta_j} \cdots}$$

Intuitively, the state of a platform is entirely defined by its execution context, i.e., the configuration of the components running on it. Each component C is wrapped by a local context $[\cdot]_\eta$ labelled with its name. The execution context changes according to the computational steps performed by the components running on it and can see any action b (rule (CTX−S)). Also, the context provides the support for the intent-based communications (rule (CTX−I)). In practice, the context observes an action $\alpha^{\eta_j}_{\eta_i}(u, v)$ fired by a component η_i and delivers it to the right destination η_j.

When a platform is initialised, e.g., at system boot, a default, starting context is created. We now present the procedure that, given a system $S = A_1, \ldots, A_n$, returns the corresponding initial context. To do that, we introduce some preliminary notions.

Definition 2. *Given an application* $A = \langle M, \Delta, \bar{C} \rangle$ *such that* $M = \Pi; P; \Lambda$ *we define:*

– *the* permissions set *of* A, *in symbols* $\mathsf{Perm}(A) = \{\!|\ P\ |\!\}$, *where*

$$\{\!|\ \varepsilon\ |\!\} = \emptyset \qquad \{\!|\ (\alpha, \bar{u}).P'\ |\!\} = \bigcup_{u_i \in \bar{u}} \{(\alpha, u_i)\} \cup \{\!|\ P'\ |\!\}$$

– *the* privileges set *of* A, *in symbols* $\mathsf{Priv}_{\mathcal{P}}(A) = \bigcup_{\eta \in dom(\Delta)} \langle\!\langle \Pi \rangle\!\rangle^{\eta}_{\mathcal{P}}$, *where* $\mathcal{P}$ *is a permissions set and*

$$\langle\!\langle \varepsilon \rangle\!\rangle^{\eta}_{\mathcal{P}} = \emptyset \qquad \langle\!\langle \rho\sigma.\Pi \rangle\!\rangle^{\eta}_{\mathcal{P}} = \langle\!\langle \Pi \rangle\!\rangle^{\eta}_{\mathcal{P}} \cup \bigcup_{u} \{\sigma(u)\} \qquad \langle\!\langle \rho\alpha.\Pi' \rangle\!\rangle^{\eta}_{\mathcal{P}} = \langle\!\langle \Pi' \rangle\!\rangle^{\eta}_{\mathcal{P}} \cup \bigcup_{\tau} \{\alpha_\eta(u,\tau) \mid (\alpha, u) \in \mathcal{P}\})$$

Briefly, $\mathsf{Perm}(A)$ is the set of new permissions which A exposes in its manifest while $\mathsf{Priv}_{\mathcal{P}}(A)$ is the set of privileges it requests. We also write $\mathsf{Perm}(S)$ and $\mathsf{Priv}_S(A)$, where $S = A_1, \ldots, A_n$, as a shorthand for $\bigcup_i \mathsf{Perm}(A_i)$ and $\mathsf{Priv}_{\mathsf{Perm}(S)}(A)$, respectively. Even though Android does not check intents' extras, we annotate privileges with types τ, τ' (see Section 5). Intuitively, the expression $\bigcup_\tau \{\alpha_\eta(u,\tau) \mid \ldots\}$ denotes the set of intents α coming from η and carrying data u, no matter what extra (of type) τ they contain. We can now explain how we create an initial context.

Definition 3. *Given a system* $S = A_1, \ldots, A_n$, *such that* $A_i = \langle M_i, \Delta_i, C^i_1 \ldots C^i_{k_i} \rangle$[2] *and* $M_i = \Pi_i; P_i; \Lambda_i$, *we define*

– $\mathbf{U}_S = \lambda\eta.\emptyset$;
– $\mathbf{\Phi}_S = \bigcup_{A_i \in S} \{(\eta, \sigma, u) \mid \sigma_\eta(u) \in \mathsf{Priv}_S(A_i)\} \cup \bigcup_{A_i \in S} \{(\eta, \alpha, u) \mid \alpha_\eta(u,\tau) \in \mathsf{Priv}_S(A_i)\}$;
– $\mathbf{\Lambda}_S = \lambda\alpha. \bigcup_i \Lambda_i(\alpha)$;

Then, the default context for S *is* $\mathbf{U}_S, \mathbf{\Phi}_S, [C^1_1]_{\eta^1_1} \cdots [C^n_{k_n}]_{\eta^n_{k_n}}$ *where* $C^i_j = \Delta_i(\eta^i_j)$. *The computation is then driven by* $\Rightarrow_{\mathbf{\Lambda}_S}$.

In words, when a platform is initialised, all the components are loaded in the execution context. Also, the applications contribute to create the ownership function $\mathbf{U}_S$, the policy $\mathbf{\Phi}_S$ and the destinations table $\mathbf{\Lambda}_S$. Initially, we assume no resources to be owned by the applications, i.e., $\mathbf{U}_S = \lambda\eta.\emptyset$. Note that still resources can exist and we call them *static* or *system* resources. Instead, the system policy $\mathbf{\Phi}_S$ is obtained from the union of all the privileges requested by the applications (according to the existing permissions). In particular, we combine the privileges for the system calls, i.e., (η, σ, u) and those for intents, i.e., (η, α, u). The destination table $\mathbf{\Lambda}_S$ is straightforward: for each intent α it returns the set of all the declared receivers. Finally, all the components are labelled with the unique name[3] that is declared in the name space function of their application.

Example 2. We propose the following implementation of the Denial of Service (DoS) attack reported in [2]. The *zygote* socket is a system resource of the Android platform. Briefly, upon receiving a request (intent `fork`) from an application, the system connects to the zygote (system call `zygote`) for creating and starting a new process. For balancing the computational load, the system service grants that only certain processes can be allocated (we assume a finite set $T = \{t_1, \ldots, t_k\}$). We model the corresponding component as

$$C_Z = \mathtt{receive}_{\mathtt{fork}} w \mapsto \mathtt{if}\ (w.\mathtt{d} \in T)\{\mathtt{system}_{\mathtt{zygote}} w.\mathtt{d}\}\ \mathtt{else}\ \{\ \mathtt{skip}\ \}$$

[2] We also assume that $\forall i, j. dom(\Delta_i) \cap dom(\Delta_j) = \emptyset$.
[3] Recall that we assumed $\forall i, j. i \neq j \Rightarrow dom(\Delta_i) \cap dom(\Delta_j) = \emptyset$.

and then the service application is $A_Z = \langle M_Z; \Delta_Z; C_Z \rangle$ with $M_Z = \rho\,\mathtt{zygote}.\varepsilon;(\mathtt{fork}, T).\varepsilon;(\mathtt{fork} \mapsto \eta_Z).\varepsilon$ and $\Delta_Z(\eta_Z) = C_Z$.

Due to a wrong implementation of the access permissions, any application having the network privileges can communicate with the zygote socket. Hence, the application $A = \langle M, \Delta, C \rangle$ where $M = \rho\,\mathtt{zygote}.\varepsilon;\varepsilon;(\mathtt{start} \mapsto \eta).\varepsilon$ and $\Delta(\eta) = C$ with $C = \mathtt{new}\,x\,\mathtt{in}\,\mathtt{system}_{\mathtt{zygote}}x$.

The elements of the initial context for $S = A, A_Z$ (see definition 3) are

- $\mathbf{U}_S = \lambda\eta.\emptyset$ and $\mathbf{\Phi}_S = \{(\eta_Z, \mathtt{zygote}, \_), (\eta, \mathtt{zygote}, \_)\}$ (where $\_$ means "any value");
- $\mathbf{\Lambda}_S$ such that $\mathbf{\Lambda}_S(\mathtt{fork}) = \{\eta_Z\}$.

Hence, the initial context is $\mathbf{U}_S, \mathbf{\Phi}_S, [C]_\eta[C_Z]_{\eta_Z}$. A possible reduction for it is (CTX−S)

$$\mathbf{U}_S, \mathbf{\Phi}_S, [\mathtt{new}\,x\,\mathtt{in}\,\mathtt{system}_{\mathtt{zygote}}x]_\eta[C_Z]_{\eta_Z} \Rightarrow_{\mathbf{\Lambda}_S} \mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\mathtt{system}_{\mathtt{zygote}}u]_\eta[C_Z]_{\eta_Z}$$

where u is a fresh resource. A further step is again (CTX−S)

$$\mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\mathtt{system}_{\mathtt{zygote}}u]_\eta[C_Z]_{\eta_Z} \Rightarrow_{\mathbf{\Lambda}_S} \mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\cdot]_\eta[C_Z]_{\eta_Z}$$

This last reduction is legal for the platform since $(\eta, \mathtt{zygote}, u) \in \mathbf{\Phi}_S$. However, as $u \notin T$, this operation corresponds to a violation of the requirement described above.

5 Type and Effect

In this section we present our type and effect system for the language introduced in Section 4. Also, we conclude this section with a brief dissertation about the advantages and the possible applications of history expressions for the analysis and verification of security properties which we plan to investigate in future work.

5.1 History Expressions

The type and effect system assigns types to expressions and *history expressions* to statements. Intuitively, a history expression represents the security-relevant, side effects produced by computations. History expressions are defined through the following syntax.

Definition 4. *(Syntax of history expressions)*

$$H, H' ::= \varepsilon \mid h \mid \alpha_\eta(u, \tau) \mid \bar{\alpha}_\eta h.H \mid \sigma_\eta(u) \mid \uparrow^\eta_{\sigma,u} \mid \downarrow^\eta_{\sigma,u} \mid ?^\eta_{\sigma,u} \mid \nu u.H \mid H \cdot H' \mid H + H' \mid H \parallel H' \mid \mu h.H \mid H\backslash_L$$

Briefly, they can be empty ε, variables h, h', parametric actions α_η, input prefixed expressions $\bar{\alpha}_\eta h.H$, system actions σ_η, permission granting $\uparrow^\eta_{\sigma,u}$, permission revocations $\downarrow^\eta_{\sigma,u}$, permission checks $?^\eta_{\sigma,u}$, resource creation $\nu u.H$, sequences

Table 3. History expressions semantics

$$\alpha_\eta(u,\tau) \xrightarrow{\alpha_\eta(u,\tau)} \varepsilon \qquad \sigma_\eta(u) \xrightarrow{\sigma_\eta(u)} \varepsilon \qquad \restriction^\eta_{\sigma,u} \xrightarrow{\restriction^\eta_{\sigma,u}} \varepsilon \qquad \downarrow^\eta_{\sigma,u} \xrightarrow{\downarrow^\eta_{\sigma,u}} \varepsilon \qquad ?^\eta_{\sigma,u} \xrightarrow{?^\eta_{\sigma,u}} \varepsilon \qquad H \dot{\to} H$$

$$\nu u.H \dot{\to} H \qquad \frac{H \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H'' \parallel H'} \qquad \frac{H' \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H' \parallel H''} \qquad \frac{H \xrightarrow{\alpha_{\eta'}(u,\tau)} H''}{H \parallel \bar{\alpha}_\eta h.H' \dot{\to} H'' \parallel H'\{\alpha_\eta(u,\tau)/h\}}$$

$$\frac{H \xrightarrow{a} H''}{H \cdot H' \xrightarrow{a} H'' \cdot H'} \qquad \frac{H \xrightarrow{a} H' \quad a \in L}{H\backslash_L \xrightarrow{a} H\backslash_L} \qquad \frac{H \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''} \qquad \frac{H' \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''} \qquad \frac{H\{H/h\} \xrightarrow{a} H'}{\mu h.H \xrightarrow{a} H'}$$

$$[\![H]\!] = \{a_1 \dots a_n \mid \exists H'.H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H'\}$$

$H \cdot H'$, non deterministic choices $H + H'$, concurrent compositions $H \parallel H'$, recursions $\mu h.H$ or action restrictions $H\backslash_L$.

We define the semantics of history expressions through a *labelled transition system* (LTS) according to the rules in Table 3.

As expected, most of the transitions of Table 3 are common to many process algebrae semantics. In particular, history expressions $\alpha_\eta(u,\tau)$, $\sigma_\eta(u)$, $\restriction^\eta_{\sigma,u}$, $\downarrow^\eta_{\sigma,u}$ and $?^\eta_{\sigma,u}$ simply fire the corresponding actions and reduce to ε. A sequence $H \cdot H'$ behaves like H until $H = \varepsilon$ (in which case we force $\varepsilon \cdot H' = H'$), while a resource creation $\nu u.H$ reduces to H producing no visible effects. Instead, a restriction $H\backslash_L$ makes the same transitions as H, provided they are allowed by L, i.e., $a \in L$. Two concurrent history expressions $H \parallel H'$ admit different reductions: either one of the two sub-expressions independently performs one step or both of them synchronise on a certain action. In order to perform a synchronisation, one of the two must be a *receiver* for an action emitted by the other, i.e., $\bar{\alpha}_{\eta'} h.H$. Note that received actions are relabelled with the identity of the receiver. Instead, Non conditional choice $H + H'$ can behave like H or H', respectively. Finally, a recursive history expression $\mu h.H$ can reduce to H where the instances of the variable h have been replaced by the recursive expression.

Denotational semantics function $[\![\cdot]\!]$ maps each history expression H into a set of finite execution traces which H can generate.

5.2 Type and Effect System

Before presenting our type and effect system, we need to introduce two preliminary definitions for *types* and *type environment*.

Definition 5. *(Types and type environment)*

$$\tau, \tau' ::= \mathbf{1} \mid \mathcal{U} \mid \mathcal{I}_\alpha(\mathcal{U}, \tau) \mid \tau \xrightarrow{H} \mathbf{1} \qquad\qquad \Gamma, \Gamma' ::= \emptyset \mid \Gamma\{\tau/x\}$$

Table 4. Typing rules

$$(\mathtt{T_E{-}NULL})\ \Gamma \vdash \mathtt{null} : \mathbf{1} \qquad (\mathtt{T_E{-}RES})\ \Gamma \vdash u : \{u\} \qquad (\mathtt{T_E{-}VAR})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad (\mathtt{T_E{-}PROC})\ \frac{\Gamma\{\tau/y, \tau \xrightarrow{H} \mathbf{1}/h\} \triangleright_O^\eta C : H}{\Gamma \vdash \mathtt{proc}\, h(y)\{C\} : \tau \xrightarrow{H} \mathbf{1}}$$

$$(\mathtt{T_E{-}INT})\ \frac{\Gamma \vdash E : \mathcal{U} \quad \Gamma \vdash E' : \tau}{\Gamma \vdash I_\alpha(E, E') : \mathcal{I}_\alpha(\mathcal{U}, \tau)} \qquad (\mathtt{T_E{-}DATA})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \vdash E.\mathtt{d} : \mathcal{U}} \qquad (\mathtt{T_E{-}EXT})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \vdash E.\mathtt{e} : \tau}$$

$$(\mathtt{T_S{-}SKIP})\ \Gamma \triangleright_O^\eta \mathtt{skip} : \varepsilon \qquad (\mathtt{T_S{-}ICST})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \triangleright_O^\eta \mathtt{icast}\, E : \sum_{u \in \mathcal{U}} \alpha_\eta(u, \tau)} \qquad (\mathtt{T_S{-}ECST})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \triangleright_O^\eta \mathtt{ecast}\, \eta'\, E : \sum_{u \in \mathcal{U}} \alpha_\eta(u, \tau)}$$

$$(\mathtt{T_S{-}SYS})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \triangleright_O^\eta \mathtt{system}_\sigma\, E : \sum_{u \in \mathcal{U}} \sigma_\eta(u)} \qquad (\mathtt{T_S{-}CHK})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \triangleright_O^{\eta'} \mathtt{check}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U}} ?_{\sigma,u}^\eta}$$

$$(\mathtt{T_S{-}GRNT})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \triangleright_O^{\eta'} \mathtt{grant}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U} \cap O} !_{\sigma,u}^\eta} \qquad (\mathtt{T_S{-}REVK})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \triangleright_O^{\eta'} \mathtt{revoke}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U} \cap O} !_{\sigma,u}^\eta}$$

$$(\mathtt{T_S{-}APP})\ \frac{\Gamma \vdash E : \tau \xrightarrow{H} \mathbf{1} \quad \Gamma \vdash E' : \tau}{\Gamma \triangleright_O^\eta \mathtt{apply}\, E\, \mathtt{to}\, E' : H} \qquad (\mathtt{T_S{-}SEQ})\ \frac{\Gamma \triangleright_O^\eta C : H \quad \Gamma \triangleright_O^\eta C' : H'}{\Gamma \triangleright_O^\eta C; C' : H \cdot H'}$$

$$(\mathtt{T_S{-}NEW})\ \frac{\Gamma\{\{u\}/x\} \triangleright_{O \cup \{u\}}^\eta C : H \quad \text{fresh } u}{\Gamma \triangleright_O^\eta \mathtt{new}\, x\, \mathtt{in}\, C : \nu u.H} \qquad (\mathtt{T_S{-}RECV})\ \frac{\Gamma\{\mathcal{I}_\alpha(\mathcal{U}, \tau)/x\} \triangleright_O^\eta C : H}{\Gamma \triangleright_O^\eta \mathtt{receive}_\alpha\, x \mapsto C : \bar{\alpha}_\eta h.H}$$

$$(\mathtt{T_S{-}CND})\ \frac{\Gamma \triangleright_O^\eta C : H \quad \Gamma \triangleright_O^\eta C' : H}{\Gamma \triangleright_O^\eta \mathtt{if}\, (E = E')\{C\}\, \mathtt{else}\, \{C'\} : H} \qquad (\mathtt{T_S{-}WKN})\ \frac{\Gamma \triangleright_O^\eta C : H' \quad H' \sqsubseteq H}{\Gamma \triangleright_O^\eta C : H}$$

A type can be a *unit* $\mathbf{1}$, a finite set of resources $\mathcal{U} = \{u_1, \ldots, u_n\}$, an intent $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ or an annotated arrow $\tau \xrightarrow{H} \mathbf{1}$. We use annotated types in the style of [4,22] (to which we refer the reader for more details) for denoting the latent effect that a procedure can generate when applied to a target input. A type environment Γ maps variable names into types and can be either empty $\emptyset$ or a new binding in an existing environment $\Gamma\{\tau/x\}$.

Type judgements assign types to expressions and history expressions to statements. For expressions, the syntax is $\Gamma \vdash E : \tau$ and shall be read "expression E has type τ under environment Γ". Similarly, for statements we have $\Gamma \triangleright_O^\eta C : H$ with the meaning that, under environment Γ, statement C (which is part of package η) generates effect H. Also, we use O to denote the set of resources owned by the package η. The rules of the type and effect system are reported in Table 4.

In words, the expression `null` has type $\mathbf{1}$ and a resource u has type $\{u\}$ (rules ($\mathtt{T_E{-}NULL}$) and ($\mathtt{T_E{-}RES}$)). Instead, the type of a variable x is provided by the environment Γ (rule ($\mathtt{T_E{-}VAR}$)). Procedures require more attention (rule ($\mathtt{T_E{-}PROC}$)). Indeed, we say that a procedure $\mathtt{proc}\, f(x)\{C\}$, has arrow type $\tau \xrightarrow{H} \mathbf{1}$ where τ is the type of its input and H is the latent effect obtained by

typing C (see rules for statements). Also, typing C requires to recursively keep trace of the type of x and of the procedure f. Typing intents (rule ($\mathtt{T_E{-}INT}$)) is quite intuitive: an intent $I_\alpha(E, E')$ has type $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ where $\mathcal{U}$ and τ are the types of the sub-expressions E and E'. Conversely, the type of the data and extra fields (rules ($\mathtt{T_E{-}DATA}$) and ($\mathtt{T_E{-}EXT}$)) of an intent of type $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ have type $\mathcal{U}$ and τ, respectively.

Typing rules for statements are also straightforward. A `skip` command (rule ($\mathtt{T_S{-}SKIP}$)) generates the void effect ε, while casting an intent (both implicitly or explicitly, rule ($\mathtt{T_S{-}ICST}$) and ($\mathtt{T_S{-}ECST}$)) inside a component η, can generate an action $\alpha_\eta(u, \tau)$ for each possible instance of u compatible with the intent type (we use $\sum H_i$ as a shorthand for the finite summation $H_1 + H_2 + \ldots$). Similarly, system calls, permission granting, revocation and checks produce corresponding, observable actions (rules ($\mathtt{T_S{-}SYS}$), ($\mathtt{T_S{-}GRNT}$), ($\mathtt{T_S{-}REVK}$) and ($\mathtt{T_S{-}CHK}$), respectively). In particular, a command $\mathtt{system}_\sigma(E)$ is typed to the sum of all the possible accesses σ to the resources denoted by E. Instead, permission granting (revocation) evaluates to the special action $\uparrow^\eta_{\sigma,u}$ ($\downarrow^\eta_{\sigma,u}$). Then, permission checks produce the special actions $?^\eta_{\sigma,u}$. Applying a procedure to a parameter (rule ($\mathtt{T_S{-}APP}$)) results in its latent effect to be carried out. The sequence of statements ($\mathtt{T_S{-}SEQ}$) is typed to the sequence of their effects, the resource creation command ($\mathtt{T_S{-}NEW}$) results in the history expression $\nu u.H$, a receiver (rule ($\mathtt{T_S{-}RECV}$)) has effect $\bar{\alpha}_\eta h.H$ and a conditional branching (rule ($\mathtt{T_S{-}CND}$)) has effect equal to those of its two branches. Finally, we include a rule, called *weakening* ($\mathtt{T_S{-}WKN}$), for extending the effect of statements (where $H' \sqsubseteq H$ iff $[\![H']\!] \subseteq [\![H]\!]$).

Example 3. Consider the following two statements:

$$C = \mathtt{apply}\,(\,\mathtt{proc}\,f(y)\{\,\mathtt{receive}\,x \mapsto \mathtt{system}_\sigma\,(x.\mathtt{d});\ \mathtt{apply}\,f\,\mathtt{to}\,y\})\,\mathtt{to}\,\mathtt{null}$$
$$C' = \mathtt{icast}\,I_\alpha(u, \mathtt{null})$$

We type them as follows:

$$\emptyset \triangleright^\eta_\emptyset\ \mathtt{apply}\,(\,\mathtt{proc}\,f(y)\{\,\mathtt{receive}\,x \mapsto \mathtt{system}_\sigma\,(x.\mathtt{d});\ \mathtt{apply}\,f\,\mathtt{to}\,y\})\,\mathtt{to}\,\mathtt{null} : \mu h.\bar{\alpha}_\eta h'.\sigma_\eta(u) \cdot h$$
$$\emptyset \triangleright^{\eta'}_\emptyset\ \mathtt{icast}\,I_\alpha(u, \mathtt{null}) : \alpha_{\eta'}(u, \mathbf{1})$$

The complete derivations are reported at `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`.

A fundamental property of our type system is that it generates history expressions which *correctly* represent the behaviour of the statements they are extracted from. This is granted by the following lemma.

Lemma 1. *For each C such that $\emptyset \triangleright^\eta_O C : H$ and for each Φ, Λ and U such that $U(\eta) = O$, for all arbitrary long sequences of actions performed by U, Φ, C there exists a trace in $[\![H]\!]$ denoting it.*

As far as the overall behaviour of a system depends on several components and their permissions and privileges, typing each single component is not sufficient to

create a model of an entire platform. Hence, we define a compositional operator, based on our typing rules, which, given a system generates a corresponding model.

Definition 6. *Given a system* $S = A_1, \ldots, A_n$ *such that* $A_i = \langle M_i, \Delta_i, C_1^i \ldots C_{k_i}^i \rangle$ *we define*

$$\text{HE}(S) = (H_1^1 \backslash_{L_{A_1}}) \parallel \ldots \parallel (H_{k_1}^1 \backslash_{L_{A_1}}) \parallel \ldots \parallel (H_1^n \backslash_{L_{A_n}}) \parallel \ldots \parallel (H_{k_n}^n \backslash_{L_{A_n}}) \qquad \text{where}$$

- $\emptyset \triangleright_{\emptyset}^{\eta} C_j^i : H_j^i$ *(with* $\Delta_i(\eta) = C_j^i$*);*
- $L_{A_i} = \{\alpha_\eta(u, \tau) \mid \alpha_\eta(u, \tau) \in \mathsf{Priv}_S(A_i)\}$.

The operator $\text{HE}(S)$ generates a history expression which correctly models S as stated by the following theorem.

Theorem 1. *For each* $S = A_1, \ldots, A_n$ *such that* $A_i = \langle M_i, \Delta_i, C_1^i \ldots C_{k_i}^i \rangle$ *for any arbitrary long computation performed by* $\mathbf{U}_S, \mathbf{\Phi}_S, [C_1^1]_{\eta_1^1} \cdots [C_{k_n}^n]_{\eta_{k_n}^n}$ *there exists a trace in* $[\![\text{HE}(S)]\!]$ *denoting it.*

Such property guarantees that any possible behaviour that a platform has at runtime is contained in its model which we can analyse statically.

5.3 Future Directions

We showed that type and effect systems can be used to compute an over-approximation of the behaviours of programs called the history expressions. History expressions can be exploited for different kinds of analysis, e.g., validation against security policies [5] or deployment of extra security checks [22]. We plan to investigate the existing techniques which rely on history expressions for verifying whether they apply, as we believe, to our model.

Another possibility is to exploit type systems as proof systems. Let H be a history expression and C be a statement. Typing $\emptyset \triangleright_{\eta}^{O} C : H$ corresponds to proving that the behaviour of C is bounded by H. This means that, if we specify security policies through history expressions, then we can verify a program by typing it to that particular history expression. Also, we can obtain similar results by typing a statement and checking whether the obtained history expression is a subtype (relation $\sqsubseteq$) of the policy ones. A convenient way to do that can be via simulation-based techniques, which can be applied here since $\sqsubseteq$ is indeed a simulation relation (see `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`).

6 Conclusion and Related Work

In this work we presented an approach which aims at modeling the Android Application Framework. Furthermore, such model is automatically inferred by means of a type and effect system. The type and effect system can either generate or verify history expressions from the Android applications (components

and manifests). The resulting model is safe in the sense that it correctly represents all the possible runtime computations of the applications. Moreover, the history expressions representing (the components of) each single application can be combined together in order to create a global model for a specific Android platform. History expressions, originally proposed by Bartoletti et al. [5], have been successfully applied to the security analysis of Java applications [3] and web services [4], and we plan to apply similar approaches to Android.

Related work. Only recently researchers focussed on the formal modeling and analysis of the Android platform and its security aspects. In [21] the authors formalise the permission scheme of Android. Briefly, their formalisation consists of a state-based model representing entities, relations and constraints over them. Also, they show how their formalism can be used to automatically verify that the permissions are respected. Unlike our proposal, their language only describes permissions and obligations and does not capture application interactions which we infer from actual implementations. In particular, their framework provides no notion of interaction with the platform, while we represent it through system calls.

Similarly to the present work, Chaudhuri [10] proposes a language-based approach to infer security properties from Android applications. Moreover, this work propose a type system that guarantees that well-typed programs respects user data access permissions. The type and effect system that we presented here extends the proposal of [10] as it also infers/verifies history expressions. History expressions can denote complex interactions and behaviours and which allow for the verification and enforcement of a rich class of security policies [1].

Most of the literature on Android security contains proposals for i) extending the native security policy, ii) enhancing the ASF with new tools for specific security-related checks, and iii) detecting vulnerabilities and security threats. Regarding the first category, in [20] Android security policy is analysed in terms of efficacy and some extensions are proposed. Besides, in [17] authors propose an extension to the basic Android permission systems and corresponding new policies. Moreover, in [24] new privacy-related security policies are proposed for addressing security problems related to users' personal data.

Related to ASF, many proposal have been made to extend native security mechanisms. For instance, [13] and [18] are focused on permissions: the first proposes a monitoring tool for assessing the actual privileges of Android applications while the latter describes SAINT, a modification to Android stack that allows to manage install-time permissions assignment. Other tools are mainly focused on malware detection (e.g. XManDroid [8] and Crowdroid [9]) and application certification (e.g. Scandroid [14] and Comdroid [11]).

Some works have been carried out to detect vulnerabilities which are often independent from the Android version. Many of them show that the Android platform may suffer from DoS attacks [2], covert channels [19], web attacks [16] and privilege escalation (see [8]).

All the analysed approaches are unrelated and may work independently on the same Android stack. However, since different approaches often share common

security features they should integrate one another. Such result is currently unachievable, due to the lack of common and comprehensive reference model for the security of the Android platform.

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium, pp. 107–121 (2003)
2. Armando, A., Merlo, A., Migliardi, M., Verderame, L.: Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 13–24. Springer, Heidelberg (2012)
3. Bartoletti, M., Costa, G., Degano, P., Martinelli, F., Zunino, R.: Securing Java with Local Policies. Journal of Object Technology 8(4), 5–32 (2009)
4. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security (JCS) 17(5), 799–837 (2009)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 32–47. Springer, Heidelberg (2007)
6. Bartoletti, M., Degano, P., Ferrari, G.-L., Zunino, R.: Local policies for resource usage analysis. ACM Transactions on Programming Languages and Systems 31(6), 1–43 (2009)
7. Bierman, G.M., Parkinson, M.J., Pitts, A.M.: MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge (2003)
8. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt (April 2011)
9. Burguera, I., Zurutuza, U., Nadjm-Therani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011 (2011)
10. Chaudhuri, A.: Language-based security on Android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, pp. 1–7. ACM, New York (2009)
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys 2011, pp. 239–252. ACM, New York (2011)
12. Android Developers. Security and permissions, `http://developer.android.com/guide/topics/security/security.html`
13. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638 (2011)
14. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications
15. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 132–146 (1999)

16. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on webview in the android system. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 343–352. ACM, New York (2011)
17. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, pp. 328–332. ACM, New York (2010)
18. Ongtang, M., Mclaughlin, S., Enck, W., Mcdaniel, P.: Semantically rich application-centric security in android. In: ACSAC 2009: Annual Computer Security Applications Conference (2009)
19. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proceedings of the 18th Annual Network & Distributed System Security Symposium (2011)
20. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google android: A comprehensive security assessment. IEEE Security Privacy 8(2), 35–44 (2010)
21. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework. In: Proceedings of the 2010 IEEE Second International Conference on Social Computing, SOCIALCOM 2010, pp. 944–951. IEEE Computer Society, Washington, DC (2010)
22. Skalka, C., Smith, S.: History effects and verification. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 107–128. Springer, Heidelberg (2004)
23. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, pp. 317–326. ACM, New York (2012)
24. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) TRUST 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)

A Type System for Flexible Role Assignment in Multiparty Communicating Systems

Pedro Baltazar[1], Luís Caires[3], Vasco T. Vasconcelos[2], and Hugo Torres Vieira[3]

[1] Instituto de Telecomunicações, IST, Universidade Técnica de Lisboa
[2] LaSIGE, Faculdade de Ciências, Universidade de Lisboa
[3] CITI-DI, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Abstract. Communication protocols in distributed systems often specify the roles of the parties involved in the communications, namely for enforcing security policies or task assignment purposes. Ensuring that implementations follow role-based protocol specifications is challenging, especially in scenarios found, e.g., in business processes and web applications, where multiple peers are involved, single peers impersonate several roles, or single roles are carried out by several peers. We present a type-based analysis for statically verifying role-based multi-party interactions, based on a simple π-calculus model and prior work on conversation types. Our main result ensures that well-typed systems follow the role-based protocols prescribed by the types, including systems where roles are flexibly assigned to processes.

1 Introduction

Communication is a central feature of nowadays software systems, as more and more often systems are built using computational resources that are concurrently available and distributed in the web. Examples range from operating systems where functionality is distributed between distinct threads in the system, to services available on the Internet, which rely on third-party (remote) service providers to carry out subsidiary tasks, following the emerging model of SaaS (software as a service) and cloud computing. Building software from the composition of communicating interacting pieces is very flexible, at least in principle, since resources can be dynamically discovered and chosen according to criteria such as declared functionality, availability and work load. In such a setting, all interacting parties must agree on communication protocols without relying on centralized control. Verification mechanisms that automatically check whether the code meets some common protocol specification become then of crucial importance.

A protocol specification describes a set of message exchanges, recording when these should occur as well as the parties involved in the interaction. A party involved in a protocol may have a spatial meaning, for instance denoting a distinguished site or process, or, more generally, a party may have a behavioral meaning, a *role* in the interaction that may be realized by one or more processes or sites. Conversely, a process may impersonate different roles throughout its

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 82–96, 2013.
© Springer-Verlag Berlin Heidelberg 2013

execution. Such flexibility is essential to address systems, e.g., where a leader role is impersonated by different sites at different stages of the protocol, and the role of each site changes accordingly.

A challenge that arises is then to devise techniques to verify whether a system complies to a protocol specification, given such dynamic and distributed implementation of roles, just by inspecting the source code. A particular situation where roles must be traced is when checking conformance against security policies like, for example, those involving separation of duties.

In this paper we present a type-based analysis for verifying whether systems defined in a model programming language follow the role-based protocol descriptions as prescribed by types. Our development is based on conversation type theory [4], extending it with the ability to specify and analyze the roles involved in the interactions. The underlying model of our analysis is an extremely parsimonious extension of the π-calculus [13,15], where communication actions specify a message label and the role performing the action, inspired by TyCO [16]. Conversations generalize sessions [10,12] with support to multiparty interaction, addressing dynamically established collaborations between an unanticipated number of partners. A distinguishing feature of the conversation types approach is that multiple parties interact using labeled messages in a single medium of communication, while other works support multiparty communication via message queues [11] and indexed communication channels [2]. We choose to adopt the simplest possible setting where session-like multiparty interaction may be studied, and extend it in a minimal way so as to support general reasoning about roles. So, apart from retaining the simplicity of conversation types, our theory addresses systems where a single role may be realized by several parties and where processes may dynamically change the role on behalf of which they are interacting, as needed to model communicating workflows as present in actual business processes. This contrasts with related approaches (see, e.g., [7,11]) where roles have a "spatial" meaning, as they are mapped into the structure of systems or sites in a static way.

In the remainder of this section we informally describe our type analysis by going through some examples. Consider the protocol specification given by type

$$\mathsf{Sender} \to \mathsf{Receiver}\ \mathit{hello}().\mathsf{Sender} \to \mathsf{Receiver}\ \mathit{bye}()$$

which captures a binary interaction where messages *hello* and *bye* are sequentially exchanged, and the communicating partners are identified by Sender and Receiver, which send and receive the messages, respectively (read $\to$ as "sends to"). A non surprising implementation of this interaction is given by process

$$\mathit{chat} \triangleleft_{\mathsf{Sender}} \mathit{hello}().\mathit{chat} \triangleleft_{\mathsf{Sender}} \mathit{bye}() \mid \mathit{chat} \triangleright_{\mathsf{Receiver}} \mathit{hello}().\mathit{chat} \triangleright_{\mathsf{Receiver}} \mathit{bye}()$$

where two concurrent processes interact on channel *chat* following the protocol above. The process on the left sends the two messages under role Sender ($\triangleleft_{\mathsf{Sender}}$), as described by type $!\mathsf{Sender}\,\mathit{hello}().!\mathsf{Sender}\,\mathit{bye}()$, while the process on the right receives the two messages under role Receiver ($\triangleright_{\mathsf{Receiver}}$), described by type $?\mathsf{Receiver}\,\mathit{hello}().?\mathsf{Receiver}\,\mathit{bye}()$.

In this first example there is a perfect match between processes and the roles under which the processes interact. However, this does not need to be the case. Consider a different implementation of the same protocol

$$chat \triangleleft_{\mathsf{Sender}} hello().chat \triangleright_{\mathsf{Receiver}} bye() \mid chat \triangleright_{\mathsf{Receiver}} hello().chat \triangleleft_{\mathsf{Sender}} bye()$$

where the process on the left sends message *hello* as Sender and then receives message *bye* as Receiver, described by type !Sender *hello*(). ?Receiver *bye*(), and the process on the right first acts as Receiver and then as Sender, described by type ?Receiver*hello*().!Sender*bye*(). Notice each role is carried out by two distinct processes and each process implements two distinct roles.

Our type analysis ensures that both implementations follow the prescribed protocol, since the protocol Sender → Receiver *hello*().Sender → Receiver *bye*() is decomposed in "complementary" types that describe the behavior of the individual processes (for instance, in type !Sender *hello*(). ?Receiver *bye*() and type ?Receiver *hello*(). !Sender *bye*()). Although very simple, this example already distinguishes our approach from previous works, since the ability to specify roles is absent in [4] while [7,11] do not support such role distribution. Conceivably channel delegation (channel-passing) supported by previous works may be used to represent a similar notion but, to model this example in particular, two channel delegations would be necessary, which implies it would not be possible to directly observe that the two interactions take place in a related medium (in our case the *chat* channel) and the ability to audit role participation locally would be lost (as the personification of a different role would be a consequence of channel-passing).

Now consider a more realistic scenario (adapted from [4]) described by type

$$\mathsf{Buyer} \to \mathsf{Seller}\, buy().\ \mathsf{Seller} \to \mathsf{Buyer}\, price().\ \mathsf{Seller} \to \mathsf{Shipper}\, product().\ \mathsf{Shipper} \to \mathsf{Buyer}\, details() \quad (1)$$

which captures the interactions in a purchase system involving three parties. Messages *buy*, *price*, *product* and *details* are exchanged between a Buyer, a Seller, and a Shipper. First, the buyer sends the seller a buy request, then the seller replies the price back to the buyer. After that, the seller informs the shipper of the chosen product and the shipper sends the buyer the delivery details.

Fig. 1 shows a possible implementation of the purchase interaction system. Using the **new** construct, process `Buyer` creates a fresh channel *chat* that will host the purchase interaction described by (1). This newly created name is passed to a shop, via message *buyService*. Code $shop \triangleleft_{\mathsf{Buyer}} buyService(chat)$ represents the output of message *buyService* on channel *shop*, passing name *chat* under role Buyer. The `Buyer` process then sends message *buy*, after which it is simultaneously active to receive *price* and to send name *chat* on *mailBox storeService*.

The `Shop` process starts by receiving a channel name (that instantiates variable x) in message *buyService*. Then, in this received channel the `Shop` impersonates the Seller role and receives message *buy*, after which it sends message *price*. At this point, process `Shop` simultaneously impersonates Seller and Shipper,

$$
\begin{array}{lll}
\texttt{Buyer} & \triangleq & (\textbf{new}\ \mathit{chat}) \\
 & & \mathit{shop} \lhd_{\mathsf{Buyer}} \mathit{buyService}(\mathit{chat}). \\
 & & \quad \mathit{chat} \lhd_{\mathsf{Buyer}} \mathit{buy}(). \\
 & & \quad\quad (\mathit{chat} \rhd_{\mathsf{Buyer}} \mathit{price}() \mid \mathit{mailBox} \lhd_{\mathsf{Buyer}} \mathit{storeService}(\mathit{chat})) \\
\texttt{Shop} & \triangleq & \mathit{shop} \rhd_{\mathsf{Shop}} \mathit{buyService}(x). \\
 & & \quad x \rhd_{\mathsf{Seller}} \mathit{buy}(). \\
 & & \quad\quad x \lhd_{\mathsf{Seller}} \mathit{price}(). \\
 & & \quad\quad\quad (x \lhd_{\mathsf{Seller}} \mathit{product}() \mid x \rhd_{\mathsf{Shipper}} \mathit{product}().x \lhd_{\mathsf{Shipper}} \mathit{details}()) \\
\texttt{Mail} & \triangleq & \mathit{mailBox} \rhd_{\mathsf{Mail}} \mathit{storeService}(x). \\
 & & \quad x \rhd_{\mathsf{Buyer}} \mathit{details}() \\
\texttt{System} & \triangleq & (*\texttt{Buyer} \mid *\texttt{Mail} \mid *\texttt{Shop})
\end{array}
$$

Fig. 1. Code for the Purchase System

which exchanges message *product*, after which message *details* is sent. Notice that this particular `Shop` carries out both the role of the Seller and the role of the Shipper, allowing to represent a shop equipped with its own shipping service.

The `Mail` process defines a message storage service that impersonates the buyer in receiving the shipping delivery details. Notice that the buyer passes name *chat* to the mailbox, allowing in this way a third party to dynamically join the ongoing interaction, while still interacting on the delegated channel (via message *price*). Hence, in this system the Buyer role is actually carried out by two distinct processes (`Buyer` and `Mail`), which can be simultaneously active.

The implementation shown in Fig. 1 involves three distinguished processes that carry out the three roles identified in the protocol, albeit not in a one-to-one-correspondence. The type given in (1) captures the interaction in channel *chat*, which is passed from the buyer to the shop and to the mailbox in messages *buyService* and *storeService*, respectively. In order to analyze the protocol distribution between the three parties, we must consider the "slices" of protocol that are delegated in messages. Namely, the overall protocol is *split* in the type that captures the behavior that is sent to the shop (via message *buyService*)

$$?\mathsf{Seller}\ \mathit{buy}().\ !\mathsf{Seller}\ \mathit{price}().\ \mathsf{Seller} \to \mathsf{Shipper}\ \mathit{product}().\ !\mathsf{Shipper}\ \mathit{details}()$$

and in the type that captures the behavior retained by the buyer

$$!\mathsf{Buyer}\ \mathit{buy}().?\mathsf{Buyer}\ \mathit{price}().\Diamond ?\mathsf{Buyer}\ \mathit{details}()$$

The $\Diamond$ type expresses the fact that the input of message *details* occurs "sometime in the future", i.e., it does not necessarily occur exactly after the input of message *price*. In fact the `Buyer` process illustrated in Fig. 1 does not guarantee that the input is active only after the reception of message *price*. However, the sequentiality of the message exchanges is ensured by the `Shop` process, since the output of message *details* only occurs after the output of message *price*. A type $\Diamond B$ denotes a behavior that must occur sometime, but not necessarily "now" — $\Diamond B$ types obey the basic laws of the eventually temporal logic operator.

When typing the buyer process there is a further type decomposition, at the level of messages *price* and *details*, resulting in types ?Buyer *price*() and ◇?Buyer *details*(), the former being retained by the buyer process and the latter delegated to the mailbox. When typing the shop process there is another type decomposition, at the level of message *product*, resulting in types !Seller*product*() and ?Shipper*product*().!Shipper *details*(), which explain the behaviors of the parallel processes in the shop code. All decompositions sketched above are captured by a *type split*, ∘, relation that explains how protocols may be split in two complementary slices, along with subtyping. $\Diamond B$ types are crucial to the definition of type split, as they provide algebraic support to the flexibility required to sequentially order message exchanges among multiple parties.

In the previous example, the fact that message *details* is exchanged after message *price* is not observable just by looking at the source code of the buyer and mail. However, such ordering is guaranteed by the shop. If we specify that the buyer, in general, exhibits such behaviors concurrently (for example, ?Buyer *price*() | ?Buyer *details*()) we would require (order preserving) decompositions of protocols into multiple threads of behavior. The flexibility introduced by $\Diamond B$ types solves this problem as they support the specification of orderings that are guaranteed via synchronization. For example, the type ?Buyer *price*().◇?Buyer *details*() says that the reception of message details takes place (sometime) after the reception of message price. On the other hand, the type !Seller*price*().Seller → Shipper*product*().!Shipper *details*() says that the output of details necessarily occurs immediately after the output of message price. The combination of the two typed guarantees the overall ordering: first message *price*, then *product* and finally *details*.

The purchase interaction of the system shown in Fig. 1 follows the protocol specification given in (1). Notice that the Buyer role is distributed between two processes (`Buyer` and `Mail`), and that roles Seller and Shipper are carried out by a single process (`Shop`). From the point of view of our type analysis the system follows the prescribed protocols, regardless of the spatial configuration of the processes that implement the roles.

2 Process Language

In this section we present the process model, first by introducing the syntax and second by defining the operational semantics. Our process language is the π-calculus [13,15] extended with labeled communication and role-based annotations. The syntax, inspired in TyCO [16], is illustrated in Fig. 2, where we consider given an infinite sets of labels $\mathcal{L}$, of channel names $\mathcal{N}$ and of roles $\mathcal{R}$. Labels, used to index communication, are identifiers that may neither be created nor communicated (e.g., XML tags). Names are used to identify mediums of communication. For typing purposes, we distinguish two distinct usages of channels: public (*shared*) communication mediums (e.g., gateways to service providers, like the *shop* and *mailBox* channels in the example) and private (*linear*) mediums, where a set of related interactions among several parties may take place

$$P ::= \mathbf{0} \;[]\; (\mathbf{new}\ x)P \;[]\; P_1 \mid P_2 \;[]\; *P \;[]\; x \triangleright_r \{l_i(x_i).P_i\}_{i\in I} \;[]\; x \triangleleft_r l(y).P$$
$$l \in \mathcal{L}(abels) \qquad x, y \in \mathcal{N}(ames) \qquad r, s \in \mathcal{R}(oles)$$

Fig. 2. Process Syntax

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$
$$(\mathbf{new}\ x)(\mathbf{new}\ y)P \equiv (\mathbf{new}\ y)(\mathbf{new}\ x)P$$
$$P_1 \mid (\mathbf{new}\ x)P_2 \equiv (\mathbf{new}\ x)(P_1 \mid P_2) \;\; (\text{if } x \notin \text{fn}(P_1))$$
$$(\mathbf{new}\ x)\mathbf{0} \equiv \mathbf{0} \qquad *P \equiv *P \mid P \qquad P_1 \equiv P_2 \;\; (\text{if } P_1 \equiv_\alpha P_2)$$

Fig. 3. Structural Congruence

(capturing, e.g., service instance interactions, like the *chat* channel in the example). Roles are used to identify the parties involved in communications.

A process is either an inactive process $\mathbf{0}$, a name restriction $(\mathbf{new}\ x)P$ where name x is known only to process P, a parallel composition $P_1 \mid P_2$ where P_1 and P_2 are simultaneously active, or a replication $*P$ where unlimited copies of P are simultaneously active. Process constructs described up to here (the static fragment) correspond exactly to the ones found in π-calculus. As for communication primitives, we extend the (monadic) π-calculus output and input primitives with labeled communication and role annotations. Process $x \triangleleft_r l(y).P$ is able to send a message on channel x, under role r, labeled by l. Upon synchronization the name y is sent and the continuation P activated. Notice that the r annotation identifies the role in which the emission is performed. The input summation process $x \triangleright_r \{l_i(x_i).P_i\}_{i\in I}$ is able to receive one message in name x, under role r, labeled by any of the l_i labels, where i ranges over index set I (we assume that all labels l_i in an input prefix are distinct). Upon synchronization with an l_j labeled message, the respective parameter x_j is instantiated and the respective continuation P_j activated. In $(\mathbf{new}\ x)P$ all occurrences of x are bound in P, and in $x \triangleright_r \{l_i(x_i).P_i\}_{i\in I}$ all occurrences of x_i are bound in P_i, for each $i \in I$.

We introduce some auxiliary notions: we use $fn(P)$ to denote the set of free names of process P, defined as expected, and $P[x \leftarrow y]$ to denote the process obtained by replacing all free occurrences of x by y in P. As usual, we omit inactive continuations (e.g., $x \triangleleft_r l(y)$ stands for $x \triangleleft_r l(y).\mathbf{0}$).

The operational semantics is given by a reduction relation and by a structural congruence. We consider the standard definition of structural congruence, denoted by $\equiv$, defined as the least congruence that satisfies the rules in Fig. 3. Structural congruence is used in the definition of the reduction relation to syntactically rearrange the process, in order to allow reduction to be defined, as usual, by capturing the basic case for synchronization and identifying the active contexts in which a synchronization may take place.

$$\frac{k \in I}{x \triangleright_r \{l_i(x_i).P_i\}_{i\in I} \mid x \triangleleft_s l_k(y).P \xrightarrow{x:s\to rl_k} P_k[x_k \leftarrow y] \mid P} \quad \text{(Red-Comm)}$$

$$\frac{P \xrightarrow{\lambda} P' \qquad \lambda \in \{\tau, x : s \to rl\}}{(\mathbf{new}\ x)P \xrightarrow{\tau} (\mathbf{new}\ x)P'} \qquad \frac{P \xrightarrow{x:s\to rl} P' \qquad y \neq x}{(\mathbf{new}\ y)P \xrightarrow{x:s\to rl} (\mathbf{new}\ y)P'} \quad \text{(Red-New1,Red-New2)}$$

$$\frac{P_1 \xrightarrow{\lambda} P_1'}{P_1 \mid P_2 \xrightarrow{\lambda} P_1' \mid P_2} \qquad \frac{P_1 \equiv P_1' \qquad P_1' \xrightarrow{\lambda} P_2' \qquad P_2' \equiv P_2}{P_1 \xrightarrow{\lambda} P_2} \quad \text{(Red-Par,Red-Struct)}$$

Fig. 4. Reduction Relation

For typing purposes, and since we intend to match process behaviors against type specifications, our reduction relation records (public) synchronization information in labels. Reduction labels (ranged over by λ) are of two forms: a τ label captures a private internal interaction, whereas an $x : s \to rl$ label captures an l-labeled message exchange on channel x, between roles s(ender) and r(eceiver).

We may now present the reduction relation, defined by the rules given in Fig. 4, where we use $P_1 \xrightarrow{\lambda} P_2$ to represent that process P_1 reduces to P_2 with label λ. Rule (Red-Comm) says that two parallel input and output processes may exchange message l_k on channel x, the interaction being captured by label $x : s \to rl_k$, where also the roles involved in the interaction are recorded. As the result of the synchronization, name y activates the continuation (respective to l_k) instantiating parameter x_k. The continuation of the output process is also activated as a consequence of the synchronization. Rule (Red-Par) closes reduction under parallel contexts, while rules (Red-New1) and (Red-New2) close reduction under name restriction. (Red-New1) captures synchronization in private names in the scope of the name restriction, either by "hiding" a public synchronization in the restricted name or by allowing private synchronizations. (Red-New2) captures public synchronizations in the scope of the name restriction, not involving the restricted name. (Red-Struct) closes reduction under structural congruence.

3 Type System

In this section we present our type system. The type language is given in Fig. 5, where we distinguish between behavioral types that describe linear interactions (B) from types that describe shared interactions (T) (cf. conversation [4] or session [12] initiation primitives). We also use message (argument) types (M) that specify either a linear protocol or a shared message type, and communication prefixes (ρ) that describe role-based communication actions.

A behavioral type B specifies the inactive behavior **end**, the parallel composition $B_1 \mid B_2$ of two independent behaviors B_1 and B_2, the sometime $\Diamond B$, which says that behavior B may occur at any point in time, or a menu of labeled actions $\rho\{l_i(M_i).B_i\}_{i\in I}$, each one specifying the type of the name communicated

$$B ::= \mathbf{end} \,[]\, B \mid B \,[]\, \Diamond B \,[]\, \rho\{l_i(M_i).B_i\}_{i\in I}$$
$$T ::= l(B) \qquad M ::= B \,[]\, T \qquad \rho ::= !s \,[]\, ?r \,[]\, s \to r$$

Fig. 5. Conversation Types Syntax

$$\vdash \mathbf{end} \qquad \frac{B_1 \# B_2 \quad \vdash B_1 \quad \vdash B_2}{\vdash B_1 \mid B_2} \qquad \frac{\forall i \in I \quad \vdash B_i \quad \rho\{l_i(M_i).\mathbf{end}\} \# B_i}{\vdash \rho\{l_i(M_i).B_i\}_{i\in I}}$$
$$\vdash \Diamond\mathbf{end} \qquad \frac{\vdash B_1 \mid B_2 \quad \vdash \Diamond B_1 \quad \vdash \Diamond B_2}{\vdash \Diamond(B_1 \mid B_2)} \qquad \frac{\forall i \in I \quad \vdash \rho\{l_i(M_i).B_i\} \quad \rho \in \{!s, ?r\}}{\vdash \Diamond\rho\{l_i(M_i).B_i\}_{i\in I}}$$

Fig. 6. Well-Formed Type Predicate

in the message M_i, and the respective continuation behavior B_i. Depending on the communication prefix ρ, an action menu represents either an input branching (when ρ is $?r$), an output choice (when ρ is $!s$)—cf. branch and choice session types [12]—or an internal choice $s \to r$, i.e., a matched communication between an output and an input. Notice that the communication roles are identified in the communication prefixes: the *s*ender role in $!s$, the *r*eceiver role in $?r$, and the two roles involved in the interaction in $s \to r$ (s sends to r). Notice also that input and output actions (interface types that capture interactions with the environment) are mixed with matched actions (capturing internal interactions) at the same level in the type language.

The conversation type language is extended with role-based annotations and sometime types ($\Diamond B$). Although a specification is not expected to use $\Diamond B$ types, these are crucial to allow the decomposition of protocols into slices, some of which related to interactions that occur later in the protocol.

A message argument type M either specifies a behavioral linear type B, in case a linear name is communicated in the message, or a shared type T, in case a shared name is communicated in the message. A shared type T abbreviates $l(B)$, identifying the label of the message exchanged and the (linear) type of the name sent in the message — to simplify the presentation we consider that only linear names can be communicated in shared messages (communicating shared names can be easily encoded).

We now introduce some auxiliary notions, namely the type apartness, well-formed types, and matched types, all defined as predicates. Type apartness is used to identify non-interfering concurrent behaviors that may be safely composed in a linear interaction. To define type-apartness we use $lab(B)$ to denote the set of labels occurring in type B, defined as expected. We say that two types B_1 and B_2 are apart, and we write $B_1 \# B_2$, if the set of labels of B_1 is disjoint from the set of labels of B_2 ($lab(B_1) \cap lab(B_2) = \emptyset$). Building on apartness, we introduce well-formed type predicate, noted $\vdash B$, given by the rules in Fig. 6. Informally, in a well-formed type labels do not appear repeatedly in parallel (to ensure race-free behavior) or in sequence (useful to simplify presentation).

$$\frac{B_1 <: B_1'}{B_1 \mid B_2 <: B_1' \mid B_2} \qquad \frac{\forall i \in I \quad B_i <: B_i'}{\rho\{l_i(M_i).B_i\}_{i\in I} <: \rho\{l_i(M_i).B_i'\}_{i\in I}} \qquad \frac{\vdash \Diamond B}{B <: \Diamond B}$$

$$(B_1 \mid B_2) \mid B_3 \equiv B_1 \mid (B_2 \mid B_3) \qquad B_1 \mid B_2 \equiv B_2 \mid B_1 \qquad B \mid \mathbf{end} \equiv B$$

$$\Diamond(B_1 \mid B_2) \equiv \Diamond B_1 \mid \Diamond B_2 \qquad \Diamond\mathbf{end} \equiv \mathbf{end}$$

Fig. 7. Subtyping Relation

$$\frac{\vdash B}{B = \mathbf{end} \circ B} \qquad \frac{B_1 = B_1' \circ B_1'' \quad B_2 = B_2' \circ B_2'' \quad \vdash B_1 \mid B_2}{B_1 \mid B_2 = B_1' \mid B_2' \circ B_1'' \mid B_2''} \qquad \text{(S-END,S-PAR)}$$

$$\frac{\forall i \in I \quad B_i = B_i' \circ B_i'' \quad \{\rho_1, \rho_2\} = \{!r_1, ?r_2\} \quad \vdash r_1 \to r_2\{l_i(M_i).B_i\}_{i\in I}}{r_1 \to r_2\{l_i(M_i).B_i\}_{i\in I} = \rho_1\{l_i(M_i).B_i'\}_{i\in I} \circ \Diamond\rho_2\{l_i(M_i).B_i''\}_{i\in I}} \qquad \text{(S-TAU)}$$

$$\frac{\forall i \in I \quad B_i = B_i' \circ \Diamond B \quad \vdash \rho\{l_i(M_i).B_i\}_{i\in I}}{\rho\{l_i(M_i).B_i\}_{i\in I} = \rho\{l_i(M_i).B_i'\}_{i\in I} \circ \Diamond B} \qquad \text{(S-BRK)}$$

$$\frac{\forall i \in I \quad B_i = B_i' \circ \Diamond B \quad \vdash \Diamond\rho\{l_i(M_i).B_i\}_{i\in I}}{\Diamond\rho\{l_i(M_i).B_i\}_{i\in I} = \Diamond\rho\{l_i(M_i).B_i'\}_{i\in I} \circ \Diamond B} \qquad \text{(S-BRKS)}$$

$$\frac{B = B_2 \circ B_1}{B = B_1 \circ B_2} \qquad \frac{B_1' = B_2' \circ B_3' \quad B_1 \equiv B_1' \quad B_2 \equiv B_2' \quad B_3 \equiv B_3'}{B_1 = B_2 \circ B_3} \qquad \text{(S-SYM,S-EQU)}$$

Fig. 8. Type Split Relation

Also well-formed $\Diamond$ types are not applied directly to message exchanges ($s \to r$), since we are interested in specifying message exchanges that happen exactly at some point in the protocol. Also used by our typing is the notion of matched types, which captures systems where all input actions have a matching output. We say that type B is matched, noted $matched(B)$, if all communication prefixes in B are of the form $s \to r$.

The subtyping relation between behavioral types, noted $B_1 <: B_2$, is the least reflexive and transitive relation satisfying the rules in Fig. 7, where we write $B_1 \equiv B_2$ when $B_1 <: B_2$ and $B_2 <: B_1$. We remark on the use of subtyping to introduce flexibility at the level of $\Diamond$ types: type B is a subtype of $\Diamond B$, which, intuitively, means that carrying out behavior B immediately is a safe implementation of eventually carrying out behavior B.

We may now introduce type split, a ternary relation that explains how a behavioral type may be safely decomposed in two slices of behavior, capturing, in a compositional way, the behavioral contribution of distinct processes to the overall interaction. The split relation is defined by the rules given in Fig. 8, where $B = B_1 \circ B_2$ denotes that type B may be decomposed in parts B_1 and B_2.

We briefly discuss the splitting rules. Rule (S-END) specifies that a behavioral type may be decomposed in itself and the inactive behavior, typing processes that contribute "all or nothing" to the interaction. Rule (S-PAR) explains the decomposition of two independent behaviors in two slices of behaviors each,

capturing the decomposition of a system in two processes that contribute both to independent interactions.

Rule (S-TAU) separates a matched communication, between roles r_1 and r_2, in the respective output by role r_1 and input by role r_2, given a splitting of the continuation behaviors. The rule captures the decomposition of a system in two processes that synchronize in a message, each with a given role in the interaction, where one of them carries out the behavior immediately, while the other may carry out the behavior at some point in time ($\Diamond$). In such way, since one of the behaviors occurs immediately we ensure that also the message exchange takes place immediately. Notice that a rule to separate the message exchange in two immediate behaviors is not necessary since the sometime behavior may also take place immediately (via subtyping).

Rule (S-BRK) separates a $\Diamond$ (sometime) distinguished slice of behavior from a communication prefixed type, provided this behavior can be split from the continuations in all branches. The rule thus captures the decomposition of a system in two parts, where one retains the (entire) interaction capability specified by the communication prefixed type while the other contributes to ensuing interactions—singled out by the $\Diamond$. Notice that (S-BRK) allows to split behaviors such that the same slice is shared between all branches, useful when addressing, e.g., a branching protocol where every branch terminates with an *ok* or *ack* message. Rule (S-BRKS) expresses the same principle as (S-BRK) but for $\Diamond$ prefixed types. Rule (S-SYM) closes the relation under symmetry and rule (S-EQU) closes the relation under type equivalence.

To simplify the presentation, we sometimes write $B_1 \circ B_2$ to represent a type B such that $B = B_1 \circ B_2$ (if any such B exists). Notice that $B_1 \circ B_2$ does not uniquely identify a type, as B_1 and B_2 may be the result of splitting distinct types. Notice also that a type may be split in several ways. In prior work on conversation types [4], we use "merge" instead of "split", in the sense that if $B = B_1 \circ B_2$ then we may see B as the result of merging the behaviors B_1 and B_2. The merge was originally inspired in the (non-algebraic) end-point projection introduced in [5]. We can show that split is an associative relation, which is a crucial property to our type system since we rely on the flexibility of the type decomposition to address the behavioral contributions of multiple parties.

We may now present the type system. A typing judgment is of the form $\Delta; \Gamma \vdash P$ where Δ is the typing environment that describes the interactions on linear channels, and Γ is the typing environment that describes the interactions on shared channels. We write $\Delta; \Gamma$ only when the domains of Δ and Γ are disjoint. A typing environment Δ is an assignment of identifiers to behavioral types ($\Delta \triangleq x_1 : B_1, \ldots, x_k : B_k$) and a typing environment Γ is an assignment of identifiers to shared types ($\Gamma \triangleq x_1 : T_1, \ldots, x_k : T_k$). We introduce some auxiliary notation to simplify presentation: we use $(x_1 : B'_1, \ldots, x_k : B'_k, \Delta_1) \circ (x_1 : B''_1, \ldots, x_k : B''_k, \Delta_2)$ to denote $x_1 : B_1, \ldots, x_k : B_k, \Delta_1, \Delta_2$ such that $B_i = B'_i \circ B''_i$, for all i in $1, \ldots, k$ and the domains of Δ_1 and Δ_2 are disjoint. Also, we use $x_1 : B_1, \ldots, x_k : B_k <: x_1 : B'_1, \ldots, x_k : B'_k$ when $B_i <: B'_i$, for all i in $1, \ldots, k$, and $\Delta_{\mathbf{end}}$ to denote $x_1 : \mathbf{end}, \ldots, x_k : \mathbf{end}$.

$$\Delta_{\mathbf{end}}; \Gamma \vdash \mathbf{0} \qquad \frac{\Delta_1; \Gamma \vdash P_1 \qquad \Delta_2; \Gamma \vdash P_2}{\Delta_1 \circ \Delta_2; \Gamma \vdash P_1 \mid P_2} \qquad \text{(T-END,T-PAR)}$$

$$\frac{\Delta, x : B; \Gamma \vdash P \qquad matched(B)}{\Delta; \Gamma \vdash (\mathbf{new}\ x)P} \qquad \frac{\Delta; \Gamma, x : l(B) \vdash P}{\Delta; \Gamma \vdash (\mathbf{new}\ x)P} \qquad \text{(T-NEW,T-SNEW)}$$

$$\frac{\Delta, y : B; \Gamma, x : l(B) \vdash P}{\Delta; \Gamma, x : l(B) \vdash x \triangleright_r \{l(y).P\}} \qquad \frac{\Delta; \Gamma, x : l(B) \vdash P}{\Delta \circ y : B; \Gamma, x : l(B) \vdash x \triangleleft_r l(y).P} \qquad \text{(T-SIN,T-SOUT)}$$

$$\frac{\forall i \in I \qquad \Delta \circ x : B_i, y_i : B'_i; \Gamma \vdash P_i \qquad ?r\{l_i(B'_i).B_i\}_{i \in I} <: B}{\Delta \circ x : B; \Gamma \vdash x \triangleright_r \{l_i(y_i).P_i\}_{i \in I}} \qquad \text{(T-IN)}$$

$$\frac{k \in I \qquad \Delta \circ x : B_k; \Gamma \vdash P \qquad !r\{l_i(B'_i).B_i\}_{i \in I} <: B}{\Delta \circ x : B \circ y : B'_k; \Gamma \vdash x \triangleleft_r l_k(y).P} \qquad \text{(T-OUT)}$$

$$\frac{\forall i \in I \qquad \Delta \circ x : B'_i; \Gamma, y_i : T_i \vdash P_i \qquad ?r\{l_i(T_i).B'_i\}_{i \in I} <: B}{\Delta \circ x : B; \Gamma \vdash x \triangleright_r \{l_i(y_i).P_i\}_{i \in I}} \qquad \text{(T-LSIN)}$$

$$\frac{\Delta \circ x : B'_k; \Gamma, y : T_k \vdash P \qquad !r\{l_i(T_i).B'_i\}_{i \in I} <: B}{\Delta \circ x : B; \Gamma, y : T_k \vdash x \triangleleft_r l_k(y).P} \qquad \text{(T-LSOUT)}$$

$$\frac{\Delta_1; \Gamma \vdash P \quad \Delta_1 <: \Delta_2}{\Delta_2; \Gamma \vdash P} \qquad \frac{\Delta_{\mathbf{end}}; \Gamma \vdash P}{\Delta_{\mathbf{end}}; \Gamma \vdash *P} \qquad \text{(T-SUB,T-REP)}$$

Fig. 9. Typing Rules

We say process P is well-typed if $\Delta; \Gamma \vdash P$ may be derived using the rules given in Fig. 9. We discuss the key features of the typing rules. Rule (T-END) says the inactive process has no linear behavior (but complies to any shared behavior specification). Rule (T-PAR) types the parallel composition process with the linear types that are split in the behaviors of the two parallel branches, while ensuring both branches comply to the same usage of shared types. Rule (T-NEW) types a restricted linear name provided its usage is matched, i.e., it has no outstanding unmatched (? or !) communications. Rule (T-SNEW) types a restricted shared name, if it is used according to a shared type.

Rules for communication prefixes are divided in three groups, depending on the shared or linear usage of both communication subject and object. Rules (T-SIN) and (T-SOUT) address the case when the communication subject has shared usage while the object has linear usage. Notice that the behavioral type B, specified in the argument type of the shared type $l(B)$ of x, captures the slice of behavior that is delegated in the communication. Type B describes the linear usage of the input parameter in the premise of (T-SIN). Argument type B is also used in the conclusion of (T-SOUT), singled out via splitting so as to take into account the usage of y (the sent name) by the continuation (crucial to type processes that delegate a name and continue to interact in it).

Rules (T-IN) and (T-OUT) address the cases when both the communication subject and object have linear usage, and follow the lines above. Both rules record the prefixed type $\rho\{l_i(B'_i).B_i\}_{i \in I}$ in the conclusions, where ρ is either $?r$ or $!r$ for input and output, respectively. A single output is typed with a communication menu (containing the label of the emitted message) so as to directly match input

$$\frac{k \in I}{s \to r\{l_i(M_i).B_i\}_{i\in I} \overset{s\to rl_k}{\longrightarrow} B_k} \qquad \frac{B_1 \overset{s\to rl}{\longrightarrow} B_2}{B_1 \mid B \overset{s\to rl}{\longrightarrow} B_2 \mid B} \qquad \frac{B_1 \overset{s\to rl}{\longrightarrow} B_2}{B \mid B_1 \overset{s\to rl}{\longrightarrow} B \mid B_2}$$

Fig. 10. Type Reduction

$$\Delta;\Gamma \overset{\tau}{\longrightarrow} \Delta;\Gamma \quad \Delta;\Gamma, x : l(B) \overset{x:s\to rl}{\longrightarrow} \Delta;\Gamma, x : l(B) \qquad \frac{B_1 \overset{s\to rl}{\longrightarrow} B_2}{\Delta, x : B_1;\Gamma \overset{x:s\to rl}{\longrightarrow} \Delta, x : B_2;\Gamma}$$

Fig. 11. Typing Environment Reduction

menus. Notice that the prefixed type is taken up to subtyping, so as to allow to introduce $\diamond$ types that may be necessary for the split in the conclusion. Notice also that the prefixed type is singled out via splitting, so as to take into account behaviors of x originally assigned to other threads (due to name delegation). Rules (T-LSIN) and (T-LSOUT) follow similar lines, addressing the case when the communication subject/object have linear/shared use. The last two rules are (T-REP), which types the replicated process, provided it uses no linear names, and the subsumption rule (T-SUB).

We can show that typing is preserved by substitution and by structural congruence. Given that our main result involves relating process actions and type specifications, we introduce type reduction, defined by the rules given in Fig. 10. In this way, we are able to precisely describe process reductions via the corresponding type reductions. Type reduction specifies how matched types reduce, explaining a message exchange that activates the respective continuation. Type reduction relies on reduction labels of the form $s \to rl$, identifying the roles involved in the communication and the label of the exchanged message.

Type reduction provides the expected semantics of behavioral types. Building on type reduction and in order to simplify the presentation of the results we introduce typing environment reduction, given by the rules in Fig. 11. Typing environment reduction specifies that environments seamlessly mimic internal τ (non public) reductions as well as synchronizations on shared channels. Also, typing environments exhibit linear reductions provided the reduction is observable at the level of the type of the respective channel. We may now state our main result that explains process reduction via typing environment reduction.

Theorem 1 (Type Preservation)
If $\Delta;\Gamma \vdash P$ *and* $P \overset{\lambda}{\longrightarrow} P'$ *then* $\Delta;\Gamma \overset{\lambda}{\longrightarrow} \Delta';\Gamma$ *and* $\Delta';\Gamma \vdash P'$.

The proof follows by induction on the length of the derivation of $P \overset{\lambda}{\longrightarrow} P'$ (full details can be found in the supporting technical report [1]). Theorem 1 states that any reduction of a well-typed process is explained by the corresponding type reduction, thus ensuring that processes interact according to the protocols prescribed by the types. Notice that this compliance entails that the protocols

are actually carried out by the roles accordingly to the type specifications. We provide a precise characterization of this property as follows.

We now define role-based protocol fidelity. Let P be a process and Δ, Γ typing environments. We say P follows the role-based protocols prescribed by Δ, Γ if for any reduction sequence of the process $P \xrightarrow{\lambda_1} P_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_k} P_k$ there is a matching reduction sequence of the typing environments $\Delta; \Gamma \xrightarrow{\lambda_1} \Delta_1; \Gamma \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_k} \Delta_k; \Gamma$. We have that well-typed processes satisfy role-based protocol fidelity as a direct consequence of Theorem 1.

Corollary 1 (Role-Based Protocol Fidelity)
If $\Delta; \Gamma \vdash P$ then P follows the role-based protocols prescribed by Δ, Γ.

In order to provide further intuition on our type system we show part of the typing of the running example (see Fig. 1). The type of name *chat*, as described in (1), is checked by successively splitting and matching resulting types with subprocesses. For example, in the typing of the `Buyer` process, after the first delegation, the type of *chat* can be decomposed by using rules (S-END), (S-BRK) and (S-BRK) (b abbreviates $\underline{\mathsf{B}}\mathsf{uyer}$).

$$\dfrac{\dfrac{\dfrac{}{\Diamond?\mathsf{b}\{details().\mathbf{end}\} = \mathbf{end} \circ \Diamond?\mathsf{b}\{details().\mathbf{end}\}}}{?\mathsf{b}\{price().\Diamond?\mathsf{b}\{details().\mathbf{end}\}\} = ?\mathsf{b}\{price().\mathbf{end}\} \circ \Diamond?\mathsf{b}\{details().\mathbf{end}\}}}{!\mathsf{b}\{buy().?\mathsf{b}\{price().\Diamond?\mathsf{b}\{details().\mathbf{end}\}\}\} = !\mathsf{b}\{buy().?\mathsf{b}\{price().\mathbf{end}\}\} \circ \Diamond?\mathsf{b}\{details().\mathbf{end}\}}$$

Now, the split given above appears when typing the subprocess

$$chat \lhd_{\mathsf{Buyer}} buy().(chat \rhd_{\mathsf{Buyer}} price() \mid mailBox \lhd_{\mathsf{Buyer}} storeService(chat))$$

Here, the delegation of name *chat*, on message *storeService*, requires that the behavior of *chat* is split between the two processes. Using (T-SUB), (T-SOUT) and (T-END) we have the following derivation.

$$\dfrac{\dfrac{\dfrac{}{chat : \mathbf{end}; mailBox : storeService(?\mathsf{b}\{details().\mathbf{end}\}) \vdash \mathbf{0}}}{chat : ?\mathsf{b}\{details().\mathbf{end}\}; mailBox : storeService(\cdot\cdot) \vdash mailBox \lhd_{\mathsf{Buyer}} storeService(chat)}}{chat : \Diamond?\mathsf{b}\{details().\mathbf{end}\}; mailBox : storeService(\cdot\cdot) \vdash mailBox \lhd_{\mathsf{Buyer}} storeService(chat)}$$

The example shows that the *sometime* operator behaves as a delayed choice between a *dot*, which expresses the sequentiality of behaviors, and a *parallel* composition, which types concurrent actions. These alternatives are introduced by rules (S-TAU), in only one of the branches of the split types, and in order to preserve, globally, the specified order of labels. Conceivably, the same flexibility would be achieved by a different (S-TAU) rule, which would immediately select between *dot* and *parallel*. Nevertheless, such rule would need to "look inside" the types and pull *parallels* to the top level. Therefore, this extension of session types with a new modality for breaking sequentiality, enriches the languages of types with an operator that enables us to perform choices locally and as needed. Such innovation supports a simple algebraic definition of the split operation.

4 Concluding Remarks

Our development is based on previous work on conversation types [4], extended so as to address assignment of dynamic roles to the several parties involved in a concurrent system. Technically, we identified a minimal set of ingredients to add to a core process specification language (the π-calculus [13,15], TyCO [16] more precisely) so as to address role-based protocol verification (labeled channels and role annotations) and extended the type analysis accordingly. Noticeably, the split relation defined in this paper is much more readable and also more expressive than the merge relation in [4] — in particular, it allows for splitting (the same) behavior out of the continuations of a branching behavior. Crucial to our development is the introduction of the $\diamond$ type to control behavior interleaving.

We discuss some possible extensions to our work. A necessary further development is the extension of the model to consider infinite behavior. An essential feature of any type analysis is a verification procedure. We are yet to implement such a procedure, but we may already assert there exists such a procedure in a setting where all bound names are type annotated. Another crucial property left out of this paper is progress. However, we expect that the progress analysis introduced in [4] for a labeled π-calculus, combined with our typing analysis, may be used to single-out systems that enjoy progress. An interesting further development to be addressed is the dynamic delegation of roles. In our setting roles are statically annotated in processes. Extending the language with role delegation would allow parties to dynamically assume unanticipated roles.

Several works address role-based type specifications to enforce security concerns (for example [8] introduces a type analysis to discipline role-based access control to data). We focus on communication protocol assignment and leave security to be handled orthogonally. Our approach builds on conversation type theory, introduced as a generalization of session types [10,12] to discipline multiparty interaction, including dynamically established conversations with an unanticipated number of participants. Other works share the goal to address multiparty interaction [2,3,6,11,14]. In particular with respect to the works more closely related to ours [2,11], we single out the approach of conversation types since it addresses multiparty interaction where the number of participants is not fixed a priori, while considering a simpler underlying model. We remark that in [2,6,11] a notion of role assignment is explicit, unlike in [4] where types do not mention identities of communicating partners. However, such role assignment is achieved via a structural projection, forcing single roles to be carried out by single threads. A different notion of dynamic roles is also considered in the approaches described in [7,9], allowing for several processes, much like a thread pool, to simultaneously carry out a single role.

In this work we have presented a type-based analysis that ensures that systems follow the prescribed role-based protocol specifications. Novel to our approach is the flexibility of role assignment, allowing us to address dynamic distributed implementations of role specifications, where a single role can be distributed among several processes and a single process can dynamically switch between roles. To the best of our knowledge, ours is the only (session-type like) approach

that addresses such configurations, that are actually found in, e.g., real world business protocols. Our development extends conversation types with role-based protocol specifications, retaining the simplicity of the approach, simplifying and generalizing the underlying technical framework, and contrasting with related approaches in the dynamic and flexible nature of roles.

Acknowledgments. Work partially supported by FCT Pest UI527 2011 and projects PTDC/EIA-CCO/113033/2009 ComFormCrypt, PTDC/EIA-CCO/104583/2008 StreamLine, PTDC/EIA-CCO/117513/2010 Liveness, Statically, PTDC/EIA-CCO/122547/2010 Multicore, and CMU-PT NGN-44A Interfaces.

References

1. Baltazar, P., Caires, L., Vasconcelos, V.T., Vieira, H.: Dynamic Roles in Multiparty Communicating Systems. UNL-DI-1–2012, Universidade Nova de Lisboa (2012)
2. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
3. Bravetti, M., Zavattaro, G.: A Foundational Theory of Contracts for Multi-party Service Composition. Fundamenta Informaticae 89(4), 451–478 (2008)
4. Caires, L., Vieira, H.: Conversation Types. Theoretical Computer Science 411(51-52), 4399–4440 (2010)
5. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
6. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On Global Types and Multiparty Sessions. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 1–28. Springer, Heidelberg (2011)
7. Deniélou, P.M., Yoshida, N.: Dynamic Multirole Session Types. In: POPL 2011, pp. 435–446. ACM (2011)
8. Ghilezan, S., Jaksic, S., Pantovic, J., Dezani-Ciancaglini, M.: Types and Roles for Web Security. Transactions on Advanced Research 8(2), 16–21 (2012)
9. Giachino, E., Sackman, M., Drossopoulou, S., Eisenbach, S.: Softly Safely Spoken: Role Playing for Session Types. In: PLACES 2009 (2009)
10. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I and II. Information and Computation 100(1), 1–77 (1992)
14. Padovani, L.: On Projecting Processes into Session Types. Mathematical Structures Computer Science 22, 237–289 (2012)
15. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
16. Vasconcelos, V.T., Tokoro, M.: A typing system for a calculus of objects. In: Nishio, S., Yonezawa, A. (eds.) ISOTAS 1993. LNCS, vol. 742, pp. 460–474. Springer, Heidelberg (1993)

A Multiparty Multi-session Logic*

Laura Bocchi[1], Romain Demangeon[2], and Nobuko Yoshida[3]

[1] University of Leicester
[2] Queen Mary, University of London
[3] Imperial College London

Abstract. Recent work on the enhancement of multiparty sessions types with logical annotations enables not only the validation of structural properties of the conversations and on the sorts of the messages, but also the validation of properties on the actual values exchanged. However, the specification and verification of the mutual effects of multiple cross-session interactions is still an open problem. We introduce a multiparty logical proof system with virtual states that enables the tractable specification and validation of fine-grained inter-session correctness properties of processes participating in several interleaved sessions. We present a sound and relatively complete static verification method.

1 Introduction

In extensively distributed computing environments, application scenarios often centre around structured conversations among multiple distributed participants. A fundamental challenge is to establish an effective specification and verification method to ensure safety in distributed software, where correctness depends on the state of individual participants and span over multiple conversations and applications. This requirement emerged from our ongoing collaboration with the Ocean Observation Initiative OOI [21], an NSF program to develop a long-term computational infrastructure for environmental ocean observation. The principals within the OOI infrastructure perform interactive activities involving distributed resources, e.g., remote instruments, off-shore sensors, data. It is important to: (1) ensure that the principals carry out each activity (session) in a way that conform a well-defined protocol, (2) express and ensure properties that span the single activities (e.g., associating a principal with a credit for resource usage, and ensuring that this credit will always be non negative across sessions[1]).

A promising direction is the logical elaboration of types for programming languages [16]. Types offer a stable linkage between the fundamental dynamics of programs and their mathematical abstractions, serving as a highly effective basis for safety assurance. In the context of process algebras, approaches like [5,13,18] allow tractable[2]

* This work has been partially funded by the project Leverhulme Trust Award "Tracing Networks", NSF Ocean Observatories Initiative, EPSRC EP/G015635/1 and EPSRC EP/G015481/1.

[1] This example is taken from the OOI Instrument Control case study and is illustrated in the Appendix of the online report [4].

[2] In [13, 18] verification is decidable and has linear complexity.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 97–111, 2013.
© Springer-Verlag Berlin Heidelberg 2013

(e.g., with respect to model checking techniques) validation of properties such as session fidelity, progress, and error freedom. Furthermore, they enable the specification of *global* properties of multiparty interactions, yet enabling modular *local* verification of each principal. The key idea is that conversations are built as the composition of units of design called *sessions* which are specified from a global perspective (i.e., a global session type). Each global type is then *projected*, making the responsibilities of each endpoint explicit. Validation guarantees that if each endpoint conforms to its projected specification(s), then the resulting conversation conforms to the corresponding global specification(s).

These approaches require to build applications starting from a set of global types that have to be agreed upon by the principals in the network. This assumption, which poses some limitations to the flexibility with which the single local processes are modelled, is reasonable in many scenarios, provided that local processes can be built as the composition of multiple, possibly interleaved, types of sessions. However, these approaches can only verify properties that are confined to the single multiparty sessions, and do not treat stateful specifications incorporating mutual effects of the multiple sessions run by a principal.

This paper presents a simple but powerful extension of multiparty session specifications, by enriching the assertion language studied in [5] with the capability to refer to *virtual states*, each local to one network principal. The resulting protocol specifications are called *multiparty stateful assertions* (*MPSA*s), and model the skeletal structure of the interactions of a session, the constraints on the exchanged messages and on the branches to be followed, and the *effects* of each interaction on the virtual state. We use *invariants* to express properties, on the state of each principal, that must hold even when several sessions are executed in parallel. Principals in a network hence serve as units of verification: static validation ensures that principals behave as prescribed by *MPSA*s and their invariants are satisfied.

To see the kind of properties we are interested in, consider the following fragment of specification for the dialogue between a ticket allocation server (`S`) and its client (`C`), where the server allocates numbered tickets of increasing value to each client in consecutive, *separate* sessions:

$$\mathtt{S} \to \mathtt{C} : (y{:}\mathtt{int})\{y = \mathtt{S.x}\}\langle \mathtt{S.x{+}{+}} \rangle$$

The protocol between the server and each client is the single message-passing action where `S` sends `C` a message of type `int`. The description of this simple distributed application implies behavioural constraints of greater depth than the basic communication actions. The (sender-side) *predicate and effect* for the interaction step, $\{y = \mathtt{S.x}\}\langle \mathtt{S.x{+}{+}} \rangle$, asserts that the message y sent to each client must equal the current value of `S.x`, a state variable x allocated to the *principal* serving as `S`; and that the local effect of sending this message is to increment `S.x`. In this way, `S` is specified to send incremental values across *consecutive* sessions.

The behaviour described above cannot be encoded by only using the primitives in [5]. In fact, in order to ensure inter-session properties one must discipline concurrent state updates with some mechanism of lock/unlock or atomic access/update, but lock/unlock and atomic access/update can only be described as properties that span over multiple sessions.

Contribution. We present a sound and relatively complete validation method for *MPSA*s, based on statically-verifiable proof rules. The most distinctive feature with respect to [5] is the possibility of expressing properties that span several sessions. The decidability/complexity of verification depends on the decidability/complexity of predicate evaluation in the logic that is chosen to express constraints and invariants (Proposition 10). We prove that our analysis is sound (Theorem 13) and complete (Theorem 14) w.r.t. the semantical satisfaction relation induced by the two labelled transition systems for processes and specifications.

Synopsis. In § 2 we present *MPSA*s, that is the language for (stateful) protocol specifications, and their consistency criteria (i.e., well-assertedness). In § 3 we present the calculus for networks of principals, each running a process. In § 4 we give the validation rules of networks against *MPSA*s; their properties are presented in § 5. Related work is discussed in § 6. Use cases from [21], auxiliary definitions, and full proofs can be found in the online report [4].

2 Multiparty Assertions with Virtual States

In the proposed framework, applications are built as the composition of units of design called *sessions*. Each type of session is specified as a *MPSA*, that is an abstract description of the interactions of the roles of a multiparty session.

The syntax of *MPSA*s is given in Figure 1. *Global assertions* ($\mathcal{G}, \mathcal{G}', \ldots$) describe a multiparty session from a global perspective; and *local assertions* ($\mathcal{L}, \mathcal{L}', \ldots$) describe it from the perspective of one role. U are types of the message contents, which can be sorts S or local assertions $\langle\mathcal{L}\rangle$ (i.e., for delegation).

$$A ::= \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid \neg A \mid A_1 \wedge A_2 \mid \exists x.A, \quad U ::= S \mid \langle\mathcal{L}\rangle, \quad S ::= \mathtt{bool} \mid \mathtt{int} \mid ..$$

$\mathcal{G} ::=$			$\mathcal{L} ::=$		
	$\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$	**(G-int)**		$\mathtt{p}!\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$	**(L-sel)**
	$\mathcal{G}_1 \mid \mathcal{G}_2$	**(G-par)**		$\mathtt{p}?\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$	**(L-bra)**
	$\mu\mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{G}$	**(G-def)**		$\mu\mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{L}$	**(L-def)**
	$\mathsf{t}\langle y : A'\rangle$	**(G-call)**		$\mathsf{t}\langle y : A'\rangle$	**(L-call)**
	end	**(G-end)**		end	**(L-end)**

Fig. 1. Global and local MPSAs

For expressing constraints we use *predicates* ($A, A', \ldots$) with the syntax illustrated in Figure 1, although we may use other predicates than equality in examples. Predicates are defined on *interaction variables*, modelling the content of a message exchanged by the roles in the session, and on *state variables*, which are variables of the virtual state local to one role.

Global Assertions. Interaction **(G-int)**, $\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$, models a message exchange where role $\mathtt{p}$ sends $\mathtt{q}$ one of the branch labels l_i and an interaction variable x_i, with x_i binding its occurrences in A_i, E_i, and $\mathcal{G}_i$. A_i is the predicate which

needs to hold for $\mathtt{p}$ to select l_i, and which may constrain the values to be sent for x_i. Note that A_i is at the same time an assumption for the receiver $\mathtt{q}$ and a constraint for the sender $\mathtt{p}$ (i.e., if A_i is violated then the blame is on $\mathtt{p}$). E_i is the update prescribed on the virtual states of $\mathtt{p}$ and $\mathtt{q}$, modelling the persistent effects (i.e., with respect to the lifetime of the single session) of that interaction. An update is a vector of assignments of the form $\mathtt{x} := e$, where $\mathtt{x}$ is updated by the result of evaluating e in the current state. We assume E does not contain two assignments to the same state variable, and is an atomic action. Assertion **(G-par)** is for parallel composition. The recursive definition **(G-def)** is the guarded recursion definition and defines a recursion parameter x initially set equal to a value satisfying the initialisation predicate A', with A being an invariant predicate. Global assertions are unfolded implicitly, following an equi-recursive view on types. **(G-call)** is the recursive instantiation and **(G-end)** is the termination.

Hereafter we omit true predicates, empty vectors of variables/updates, and labels of single branches.

Example 1. Consider a session with two roles, $\mathtt{C}$ and $\mathtt{S}$. $\mathtt{C}$ makes an offer x to $\mathtt{S}$ for buying a ticket; $\mathtt{S}$ either accepts or refuses the offer. In the former case $\mathtt{C}$ spends x credits and receives a ticket, and $\mathtt{S}$ earns x credits. Tickets are modelled as serial numbers; they must all be increasing numbers not exceeding 1000. $\mathcal{G}_T$ below specifies this scenario:

$$\begin{aligned}\mathcal{G}_T \;&= \mathtt{C} \to \mathtt{S} : (x : \mathtt{int})\{x \geqslant 0 \wedge \mathtt{C.credit} \geqslant x\}\langle \mathtt{C.credit} := \mathtt{C.credit} - x\rangle.\\ &\quad\; \mathtt{S} \to \mathtt{C} : \{\mathsf{ok}(y : \mathtt{int})\{\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}\}\langle E_{ok}\rangle.\mathsf{end},\\ &\qquad\qquad \mathsf{ko}\langle \mathtt{C.credit} := \mathtt{C.credit} + x\rangle.\mathsf{end}\;\}\\ E_{ok} \;&= \mathtt{S.credit} := \mathtt{S.credit} + x, \mathtt{S.count} := \mathtt{S.count} + 1\end{aligned}$$

$\mathtt{C}$ has state variable $\mathtt{credit}$, and $\mathtt{S}$ has state variables $\mathtt{credit}$ and $\mathtt{count}$ (a counter for serial numbers). The first interaction requires that the offer x does not exceed $\mathtt{C}$'s credit, and decrements the credit by x. $\mathtt{S}$ selects one of the two branches by either label ok or ko. The former branch can be selected only if $\mathtt{S.count} < 1000$.

We denote with $var(\mathcal{G})$ the set of (interaction/state) variables and recursion parameters in $\mathcal{G}$, and with $var(A)$ the free variables of A (same for e). The set of variables that $\mathtt{p} \in \mathcal{G}$ *knows*, written $var(\mathcal{G}) \upharpoonright \mathtt{p}$, consists of: (i) the state variables of the form $\mathtt{p}.x$ for some x, (ii) the interaction variables sent or received by $\mathtt{p}$ in $\mathcal{G}$, and (iii) the parameters of the recursive definitions $\mu \mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{G}'$ in $\mathcal{G}$ such that $\mathtt{p}$ knows all the free variables in initialisation A', and all free variables in A'' for all $\mathsf{t}\langle y : A''\rangle$ in $\mathcal{G}'$ (we assume each recursion parameter known by exactly two participants).

Well-assertedness. Our theory relies on two consistency principles: *history-sensitivity* and *temporal-satisfiability*. These principles were first introduced in [5]; we discuss them here as their adaptation to our stateful scenario requires non-trivial extensions.

By history-sensitivity each role must have enough information to fulfil the specified obligations, namely it requires that: (1) each role $\mathtt{p}$ knows all free variables in the predicates that $\mathtt{p}$ must guarantee, and (2) each role has enough information to perform the prescribed updates, that is (i) when to make an update, and (ii) which values to assign.

Definition 2 (History-sensitivity). $\mathcal{G}$ is history-sensitive if for each interaction, of the form $\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$, occurring in $\mathcal{G}$, for all $i \in I$:

1. $var(\mathcal{G}) \upharpoonright \mathtt{p} \supseteq var(A_i)$ (i.e., $\mathtt{p}$ knows all variables in $var(A_i)$),
2. for all $\mathtt{r}.x := e$ in E_i: (i) either $\mathtt{r} = \mathtt{p}$ or $\mathtt{r} = \mathtt{q}$, and (ii) $var(\mathcal{G}) \upharpoonright \mathtt{r} \supseteq var(e)$.

A checker for history-sensitivity can be found in the online report [4]. Consider the assertions:

$$\mathcal{G} = \mathtt{p} \to \mathtt{q} : (x{:}\mathtt{int}).\ \mathtt{q} \to \mathtt{r} : (y{:}\mathtt{int}).\ \mathtt{r} \to \mathtt{s} : (z{:}\mathtt{int})\{z > x\}$$
$$\mathcal{G}' = \mathtt{p} \to \mathtt{q} : (y{:}\mathtt{int}).\ \mathtt{q} \to \mathtt{r} : \{\mathsf{ok}(w{:}\mathtt{int})\langle \mathtt{r}.\mathtt{x}_1 := y, \mathtt{p}.\mathtt{x}_2 := y\rangle, \mathsf{ko}\}$$

$\mathcal{G}$ violates (1) because $\mathtt{r}$ has to send a value for z that is greater than x without knowing x. $\mathcal{G}'$ violates both clauses of (2): (i) because $\mathtt{p}$ must update $\mathtt{x}_2$ not knowing whether and when the update should be done, and (ii) because in the second interaction $\mathtt{r}$ has to update $\mathtt{x}_1$ with y without knowing y.[3]

By temporal-satisfiability, for each participant $\mathtt{p} \in \mathcal{G}$, whenever it is $\mathtt{p}$'s turn to send a value, $\mathtt{p}$ can find at least one selection branch and one value which satisfies the specified constraint. Temporal satisfiability is defined (and checked) using a function $\mathtt{ts}(\mathcal{G}, A)$ which returns true only if $\mathcal{G}$ always allows a path of interactions going through $\mathcal{G}$ in *any possible state*. Considering all possible states makes the specification robust with respect to arbitrary interactions the same principal may be engaged in through other sessions. Predicate A is incrementally built as a conjunction of the predicates that appear in $\mathcal{G}$ in all the recursive invocations and models the current set of assumptions.

Definition 3 (Temporal-satisfiability). Let $\mathcal{G}$ be a global specification, and A a predicate. $\mathtt{ts}(\mathcal{G}, A)$ is given by:

1. $\mathtt{ts}(\mathtt{p}\to\mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}, A) = \begin{cases} \bigwedge_{i\in I} \mathtt{ts}(\mathcal{G}_i, A \wedge \underline{A_i}) \text{ if } A \supset \bigvee_{i\in I} \exists x_i.A_i \\ \mathsf{false} \quad \text{otherwise} \end{cases}$
2. $\mathtt{ts}(\mathcal{G}_1 \mid \mathcal{G}_2, A) = \mathtt{ts}(\mathcal{G}_1, A) \wedge \mathtt{ts}(\mathcal{G}_2, A)$
3. $\mathtt{ts}(\mu \mathsf{t}\langle e\rangle(x{:}S)\{A'\}.\mathcal{G}', A) = \begin{cases} \mathtt{ts}(\mathcal{G}', A \wedge A') & \text{if } A \supset (A'[e/x]) \\ \mathsf{false} & \text{otherwise} \end{cases}$
4. $\mathtt{ts}(t_{A'(x)}\langle e\rangle, A) = \begin{cases} \mathsf{true} & \text{if } A \supset A'[e/x] \\ \mathsf{false} & \text{otherwise} \end{cases}$
5. $\mathtt{ts}(\mathsf{end}, A) = \mathsf{true}$

$\mathcal{G}$ satisfies temporal satisfiability if $\mathtt{ts}(\mathcal{G}, \mathsf{true}) = \mathsf{true}$.[4]

[3] [6] proposes algorithms to amend assertions that violate history-sensitivity and temporal-satisfiability as in [5]. No such algorithms have yet been investigated for the definitions introduced in this paper. Although relevant, the issue of amending inconsistent assertions is out of the scope of the current work.

[4] This property can be relaxed by starting from a stronger precondition A as long as A is then implied by the principal invariants (which are defined in § 4).

In (1) the first condition for "if" demands that there exists at least one branch for which it is possible to find a value for x_i that satisfies the current predicate A_i. The function is called recursively extending the set of preconditions A with with the closure $\underline{A_i}$ of predicate A_i (see Remark 4 below). (2) demands both parts of the composition are satisfiable. (3) and (4) check recursion, the latter relying on the annotation of recursive calls with the invariants of the corresponding recursive definitions.

Remark 4. The closure of a predicate A in $\mathcal{G}$, written $\underline{A}$, is the predicate obtained by closing the free state variables of $\mathcal{G}$ in A with existential quantifiers. Whereas the values of interaction variables in a session do not change after they are introduced[5], state variables can be updated a number of times. Hence a predicate on state variables may be true at a certain time, and become false at a later time. Hereafter we use $\underline{A}$ when we want to 'keep' only the persistent parts of a predicate A (i.e., those interaction variables), discarding those on state variables.

The following global specification violates temporal satisfiability

$$\mathtt{p} \to \mathtt{q} : (x : \mathtt{int})\{x > 0\}.\mathtt{q} \to \mathtt{p} : (y : \mathtt{int})\{y = x \wedge y > 100\}$$

In fact, in the first interaction $\mathtt{p}$ is allowed to choose any positive value for x, for instance 10. In this case, $\mathtt{q}$ cannot find any value for y such that $y = 10 \wedge y > 100$.

Proposition 5. *Given a global assertion $\mathcal{G}$, let m be the size of the syntactic tree of $\mathcal{G}$, n be the maximum number of variables occurring in each predicate in $\mathcal{G}$, and $eval(A)$ be the complexity of predicate evaluation (if decidable). History-sensitivity can be checked in $O(m \times n)$. Temporal-satisfiability is decidable if predicate evaluation is decidable and, if decidable, it can be checked in $O(m) \times eval(A)$.*

Hereafter, we assume assertions to be *well-asserted*.

Local Assertions. Each local assertion $\mathcal{L}$ refers to a specific role. Syntax is given in the right part of Figure 1. Selection **(L-sel)** $\mathtt{p}!\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$ models an interaction where the role sends $\mathtt{p}$ a branch label l_i and a message x_i. A_i and E_i are the predicate and update respectively. **(L-bra)** is the dual branching. The others are as in the global assertions, except that a local assertion cannot be multi-threaded.

Given a global assertion $\mathcal{G}$, we can automatically derive the local assertions for each role $\mathtt{p} \in \mathcal{G}$ by *projection*. The projection rules rely on a few auxiliary definitions: projection of a predicate, and projection of an update. The *projection of a predicate* A on $\mathtt{p}$ in $\mathcal{G}$, written $A \upharpoonright \mathtt{p}$, is defined as $\exists\tilde{x}.A$ where $\tilde{x} = var(A) \backslash (var(\mathcal{G}) \upharpoonright \mathtt{p})$ (i.e., the existential closure of the variables that $\mathtt{p}$ does not know). The *projection of an update* E on $\mathtt{p}$ in $\mathcal{G}$, written $E \upharpoonright \mathtt{p}$ is the update E' containing only the assignments $\mathtt{p}_j.\mathtt{x}_i := e_j$ such that $\mathtt{p}_j = \mathtt{p}$.

The projection rules for global assertions are as in [5], except that updates are now considered; their detailed presentation is not necessary to understand the results in this paper, hence we only give an illustration through Example 6. Henceforth, in $\mathcal{G} \upharpoonright \mathtt{p}$ we shall omit the $\mathtt{p}.\_$ prefix when referring to $\mathtt{p}$'s state variables.

[5] Actually, interaction variables in a recursion body are reused at each iteration. However, their values are due to follow the same constraints at each iteration.

Example 6. $\mathcal{L}_{\mathtt{C}}$ (resp. $\mathcal{L}_{\mathtt{S}}$) is the projection of $\mathcal{G}_T$ from Example 1 on $\mathtt{C}$ (resp. $\mathtt{S}$).

$$\begin{aligned}
\mathcal{L}_{\mathtt{C}} &= \mathtt{S}\,!\,(x : \mathtt{int})\{x \geqslant 0 \wedge \mathtt{credit} \geqslant \mathtt{x}\}\langle \mathtt{credit} := \mathtt{credit} - x\rangle.\mathcal{L}'_{\mathtt{C}}\\
\mathcal{L}'_{\mathtt{C}} &= \mathtt{S}\,?\{\mathsf{ok}(y : \mathtt{int})\{\exists \mathtt{S.count}.\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}\}.\mathsf{end},\\
&\qquad \mathsf{ko}\langle \mathtt{credit} := \mathtt{credit} + x\rangle.\mathsf{end}\}\\
\mathcal{L}_{\mathtt{S}} &= \mathtt{C}\,?(x : \mathtt{int})\{\exists \mathtt{C.credit}.x \geqslant 0 \wedge \mathtt{C.credit} \geqslant x\}.\mathcal{L}'_{\mathtt{S}}\\
\mathcal{L}'_{\mathtt{S}} &= \mathtt{C}\,!\,\{\mathsf{ok}(y : \mathtt{int})\{\mathtt{count} < 1000 \wedge y = \mathtt{count}\}\\
&\qquad\qquad \langle \mathtt{credit} := \mathtt{credit} + x, \mathtt{count} := \mathtt{count} + 1\rangle.\mathsf{end},\\
&\qquad \mathsf{ko}.\mathsf{end}\}
\end{aligned}$$

The projection of the first interaction of $\mathcal{G}_T$ on sender $\mathtt{C}$ (resp. receiver $\mathtt{S}$) is a send/select (resp. a receive/branch). The predicates/updates of the projections on a role are the projections of the predicates/updates on that role.[6] The continuation is projected similarly, proceeding point-wise for each branch. Sometimes the projected predicate includes information about constraints of interactions between third parties (without however revealing the actual values exchanged by the third parties), e.g., $\exists \mathtt{S.count}.\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}$ provides $\mathtt{C}$ with precondition $y < 1000$.

Well-assertedness is easily extended to local assertions.

3 Multiparty Networks with Local States

We consider networks of interactional entities called *principals* linked by a common global transport, modelled as queues. Each principal runs a *located process*, that is a process with multiparty session primitives [1, 18] (to enable rigorous representation of conversation structures) and with a *local state*.

Syntax. The syntax of networks and processes is given in Figure 2 and is a refined version of the multiparty session π-calculus from [1, 10] with local states. A local state σ maps a signature $[\tilde{\mathtt{x}} : \tilde{S}]$ of typed pairwise disjoint state variables $\tilde{\mathtt{x}}$ to their sorts. We use the injective function $\mathtt{id}(\sigma)$ to map each local state to an identifier.

A network can be an empty network $\varnothing$, a located process $[P]\sigma$, a parallel composition of networks $N_1 \mid N_2$, a new session name $(\nu s)N$ which binds s in N, or a queue $s : h$ where h are messages in transit through session channel s. A network is *initial* if it has no new session names and queues, otherwise it is *runtime*. We denote the free session channels in N with $\mathsf{fn}(N)$, similarly for P with $\mathsf{fn}([P]\sigma) = \mathsf{fn}(P)$.

Session request **(P-req)** multicasts a request to each session accept process **(P-acc)**, e.g., $a[\mathtt{i}](y).P$ with $i \in \{2, .., n\}$, by synchronisation through a shared name a and continuing as P. **(P-sel)** is Dijkstra's guarded command [15] and **(P-bra)** is the branching process; they represent communications through an established session k. **(P-sel)** acts as role $\mathtt{p}$ in session k and sends role $\mathtt{q}$ one of the labels l_i. The choice of the label is determined by boolean expressions e_i, assuming $\vee_{i\in I} e_i = \mathsf{true}$ and $i \neq j$ implies

[6] Note that by well-assertedness (clause 1) the projection of a predicate on the sender of an interaction is always the predicate itself.

$$
\begin{array}{lll}
\text{(network)} & N ::= \varnothing \mid [P]\sigma \mid N_1 | N_2 \mid (\nu s)N \mid s : h & \\
\text{(state/queue/value)} & \sigma ::= [\tilde{\mathtt{x}} : \tilde{S}] \mapsto \tilde{S} \quad h ::= \varnothing \mid (\mathtt{p}, \mathtt{q}, l\langle v\rangle) \cdot h \quad v ::= \mathtt{n} \mid s[\mathtt{p}] & \\
\text{(process)} & P ::= \overline{a}[\mathtt{n}](y).P & \textbf{(P-req)} \quad \mid P \mid Q \quad \textbf{(P-par)} \\
& \mid\ a[\mathtt{i}](y).P & \textbf{(P-acc)} \quad \mid (\mu X(x).P)\langle e\rangle \quad \textbf{(P-def)} \\
& \mid\ k[\mathtt{p}, \mathtt{q}]!\{e_i \mapsto l_i\langle e_i'\rangle(x_i)\langle E_i\rangle; P_i\}_{i\in I} & \textbf{(P-sel)} \quad \mid X\langle e\rangle \quad \textbf{(P-call)} \\
& \mid\ k[\mathtt{p}, \mathtt{q}]?\{l_i(x_i)\langle E_i\rangle.P_i\}_{i\in I} & \textbf{(P-bra)} \quad \mid \mathbf{0} \quad \textbf{(P-idle)} \\
\text{(channel/update/exp)} & k ::= y \mid s \quad E ::= \varnothing \mid E; \mathtt{x} := e \quad e ::= v \mid e\ \mathtt{op}\ e &
\end{array}
$$

$x, y, \ldots$ interaction variables $\quad \mathtt{x}, \mathtt{y}, \ldots$ state variables $\quad X, Y, \ldots$ process variables
$a, b, \ldots$ shared name $\quad s, s', \ldots$ session name $\quad \mathtt{n}, \mathtt{n}', \ldots$ constants

Fig. 2. Syntax of networks and processes

$e_i \wedge e_j = \mathsf{false}$. Each label l_i is sent with the corresponding expression e_i' which specifies the value for x_i, assuming e_i' and x_i have the same type.[7] **(P-bra)** plays role $\mathtt{q}$ in session k and is ready to receive from $\mathtt{p}$ one of the labels l_i and a value for the corresponding x_i, then behaves as P_i after instantiating x_i with the received value. In guarded command (resp. branching), the local state of the sender (resp. receiver) is updated according to update E_i; in both processes each x_i binds its occurrences in P_i and E_i.

Example 7. Processes $P_{\mathtt{S}}$ and $P_{\mathtt{C}}$ implement $\mathcal{L}_{\mathtt{S}}$ and $\mathcal{L}_{\mathtt{C}}$, respectively, from Example 6.

$$
\begin{array}{l}
P_{\mathtt{S}} = a[2](z).z[\mathtt{C}, \mathtt{S}]?(x); P_{\mathtt{S}}' \qquad E_{ok} = \mathtt{count} := \mathtt{count} + 1, \mathtt{credit} := \mathtt{credit} + x \\
P_{\mathtt{S}}' = z[\mathtt{S}, \mathtt{C}]!\{\{\mathtt{count} < 1000 \wedge x \geqslant 10\} \mapsto \mathsf{ok}\langle \mathtt{count}\rangle(y)\langle E_{ok}\rangle.\mathbf{0}, \\
\qquad\qquad \{\mathtt{count} \geqslant 1000 \vee x < 10\} \mapsto \mathsf{ko}.\mathbf{0}\} \\
P_{\mathtt{C}} = \overline{a}[2](w).w[\mathtt{C}, \mathtt{S}]!\langle 8\rangle(x)\langle \mathtt{credit} := \mathtt{credit} - x\rangle; P_{\mathtt{C}}' \\
P_{\mathtt{C}}' = w[\mathtt{S}, \mathtt{C}]?\{\mathsf{ok}(y).\mathbf{0},\ \mathsf{ko}\langle \mathtt{credit} := \mathtt{credit} + x\rangle.\mathbf{0}\}
\end{array}
$$

We let $\mathtt{C} = 1$ and $\mathtt{S} = 2$. $P_{\mathtt{S}}$ accepts a request to participate to a session specified by $\mathcal{G}_T$ (assuming a has type $\mathcal{G}_T$) on channel z as role 2. In the established session z, the principal receives an offer x from the co-party. It follows a guarded command with two cases; if $\mathtt{count}$ has not reached its maximum value for serial numbers and the offer is greater than 10 then the first branch (ok) is taken and $\mathtt{count}$ is sent as y, otherwise the second branch (ko) is taken. Dually, $P_{\mathtt{C}}$ sends a request to participate to one instance of session $\mathcal{G}_T$ as the role 1. A principal may repeatedly execute a process using recursion, or run concurrent instances of the same type of session (e.g., $[P_{\mathtt{S}} \mid P_{\mathtt{S}}]\sigma$) or different types of session (e.g., $[P_{\mathtt{S}} \mid P_{\mathtt{C}}]\sigma$) as discussed in Example 9.

Operational Semantics. The LTS is generated from the rules in Figure 3 using the following labels: $\ell ::= \overline{a}[\mathtt{n}]\langle s\rangle \mid a[\mathtt{i}]\langle s\rangle \mid s[\mathtt{p}, \mathtt{q}]!l\langle v\rangle \mid s[\mathtt{p}, \mathtt{q}]?l\langle v\rangle \mid \tau$. We denote

[7] Guarded command can be implemented using selection, if-then-else and lock-unlock. Although our theory is applicable to these primitives, we choose to make these low-level steps atomic for minimising the syntax.

$$[\overline{a}[\mathtt{n}](y).P]\sigma \xrightarrow{\overline{a}[\mathtt{n}]\langle s\rangle} [P[s/y]]\sigma \qquad [a[\mathtt{i}](y).P]\sigma \xrightarrow{a[i]\langle s\rangle} [P[s/y]]\sigma \quad (s \notin \mathsf{fn}(P))$$

$$[s[\mathtt{p},\mathtt{q}]!\{e_i \mapsto l_i\langle e_i'\rangle(x_i)\langle E_i\rangle; P_i\}_{i\in I}]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle v\rangle} [P[v/x_j]]\sigma'$$
$$(j \in I \quad \sigma \models e_j' \downarrow v \quad \sigma \models e_j \quad \sigma' = \sigma\,\mathtt{after}\,E_j[v/x_j])$$

$$[s[\mathtt{p},\mathtt{q}]?\{l_i(x_i)\langle E_i\rangle.P_i\}_{i\in I}]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]?l_j\langle v\rangle} [P_j[v/x_j]]\sigma' \qquad (j \in I \quad \sigma' = \sigma\,\mathtt{after}\,E_j[v/x_j])$$

$$\frac{[P_1]\sigma_1 \xrightarrow{\overline{a}[\mathtt{n}]\langle s\rangle} [P_1']\sigma_1 \quad [P_i]\sigma_i \xrightarrow{a[i]\langle s\rangle} [P_i']\sigma_i \quad (2 \leqslant i \leqslant n)}{[P_1]\sigma_1 \mid \cdots \mid [P_n]\sigma_n \xrightarrow{\tau} (\nu s)(s : \varnothing \mid [P_1']\sigma_1 \mid \cdots \mid [P_n']\sigma_n)}$$

$$\frac{[P]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle v\rangle} [P']\sigma'}{[P]\sigma \mid s : h \xrightarrow{\tau} [P']\sigma' \mid s : h \cdot (\mathtt{p},\mathtt{q},l_j\langle v\rangle)} \qquad \frac{[P]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]?l_j\langle v\rangle} [P']\sigma'}{[P]\sigma \mid s : (\mathtt{p},\mathtt{q},l_j\langle v\rangle)\cdot h \xrightarrow{\tau} [P']\sigma' \mid s : h}$$

Fig. 3. Labelled transition for networks

with $\sigma\,\mathtt{after}\,E$ the state σ after the update E. We write $\sigma \models e \downarrow v$ for a closed expression e when it evaluates to v in σ.
The first and second rule are for requesting and accepting a session initialisation. The guarded command checks if condition e_j is satisfied in the current state σ, and sends a message consisting of one of the labels l_j and an expression e_j' (which is evaluated to a value v in state σ), updates σ according to E_j, and behaves as $P[v/x_j]$. Branching is symmetric. The synchronous session initialisation creates a new queue. We omit the standard context/structural equivalence rules.

4 Proof System for Multiparty Session Logic with Virtual States

In this section we outline how to obtain the syntactic validation of networks, written $\Gamma \vdash N \rhd \Sigma$, assuming processes typable, following [5]. The proof rules rely on the following environments:

$$\Gamma ::= \varnothing \mid \Gamma, a : \mathcal{G} \mid \Gamma, X : (x : S)\mathcal{L}_1 @\,\mathtt{p}_1, \ldots, \mathcal{L}_n @\,\mathtt{p}_n, \qquad \Delta ::= \varnothing \mid \Delta, s[\mathtt{p}] : \mathcal{L},$$
$$\Sigma ::= \varnothing \mid \Sigma, [\Delta]\sigma$$

Γ maps shared names to global assertions and process variables to their parameters. If $\Gamma \vdash a : \mathcal{G}$ then a session specified by $\mathcal{G}$ can be initiated by processes (via session request or accept) using a. By the standard kinding rules, we check if the same free variable appears in different global types in Γ, then they have the same sort. The mapping of process variables is for the validation of recursive assertions. Δ maps session channels/roles to local assertions. If $\Delta \vdash s[\mathtt{p}] : \mathcal{L}$ then a session is active (i.e., it has been initialized) on channel s for role $\mathtt{p}$; $\mathcal{L}$ specifies the (part of the) session that has still to be executed. Σ is the specification of a network; each $[\Delta]_\sigma$ is the specification of a located process with the respective virtual state.

We also use an *assertion environment* $\mathcal{C}$, which is incrementally built by conjunction of the predicates and boolean expressions (i.e., the conditions of a guarded commands) occurring in the processes being validated, and models their assumptions. Hereafter,

given a predicate A and an update E, we define A `after` E to be the predicate obtained by substituting, for each assignment $x := e$ in E, each occurrence of x in A with e.

Modelling Cross-Session Properties: the Principal Invariant. Given a located process $[P]\sigma$ in a network, we want to allow the architect to model stable properties (i.e., invariant) over the variables in σ on across multiple sessions. We call these properties *principal invariant* of $[P]\sigma$, that is a predicate (following the syntax for A in Figure 1) over the state variables of σ. Hereafter we assume there exists a function $\mathcal{I}(\sigma)$ that given a local state σ returns the principal invariant for σ. Principal invariants depend from the application domain, and the architect should define them prior to the verification.

Example 8. Consider a located process $[P_{\mathtt{c}} \mid P_{\mathtt{s}}]\sigma_p$ with $P_{\mathtt{c}}$ and $P_{\mathtt{s}}$ from Example 7. Assume we want to require that the credit is always non-negative (i.e., the principal does not contracts debts) and that the counter does not exceed the maximum number of tickets which is 1000. We can enforce these constraints by setting the principal invariant $\mathcal{I}(\sigma_p)$ to be $\mathtt{credit} \geqslant 0 \wedge 0 \leqslant \mathtt{count} \leqslant 1000$.

Proof Rules. Figure 4 illustrates the proof rules for initial networks and processes.

[N1] decomposes the validation of a network into the validations of each located process against its corresponding specification Δ. The correspondence between principal and specification in checked by the clause $\mathtt{id}(\sigma_p) = \mathtt{id}(\sigma_a)$. Furthermore, local and

$$\frac{\mathtt{id}(\sigma_p) = \mathtt{id}(\sigma_a) \quad \sigma_p, \sigma_a \models \mathcal{I}(\sigma_p) \quad \mathcal{I}(\sigma_p) \wedge \mathcal{C}; \Gamma \vdash P \rhd \Delta}{\mathcal{C}; \Gamma \vdash [P]\sigma_p \rhd [\Delta]\sigma_a} \lfloor \text{N1} \rfloor$$

$$\frac{(\Gamma', \Delta', \sigma') \ni (\Gamma, \Delta, \sigma) \quad \mathcal{C} \supset \mathcal{C}' \quad \mathcal{C}'; \Gamma' \vdash N \rhd [\Delta']\sigma'}{\mathcal{C}; \Gamma \vdash N \rhd [\Delta]\sigma} \lfloor \text{N2} \rfloor$$

$$\frac{-}{\mathcal{C}; \Gamma \vdash \varnothing \rhd \varnothing} \qquad \frac{\mathcal{C}; \Gamma \vdash N \rhd \Sigma \qquad \mathcal{C}; \Gamma \vdash N' \rhd \Sigma'}{\mathcal{C}; \Gamma \vdash N \mid N' \rhd \Sigma, \Sigma'} \lfloor \text{N3/N4} \rfloor$$

$$\frac{\mathcal{C}; \Gamma, a : \mathcal{G} \vdash P \rhd y[\mathtt{i}] : \mathcal{G} \upharpoonright \mathtt{i}, \Delta}{\mathcal{C}; \Gamma, a : \mathcal{G} \vdash a[\mathtt{i}](y).P \rhd \Delta} \lfloor \text{MACC} \rfloor$$

$$\frac{\forall i \in I, \qquad \mathcal{C} \wedge A_i; \Gamma \vdash P_i \rhd \Delta, k[\mathtt{q}] : \mathcal{L}_i \qquad \mathcal{C} \wedge A_i \supset (\mathcal{C} \ \mathtt{after}\ E_i)}{\mathcal{C}; \Gamma \vdash k[\mathtt{p}, \mathtt{q}]?\{l_i(x_i)\langle E_i\rangle.P_i\}_{i \in I} \rhd \Delta, k[\mathtt{q}] : \mathtt{p}?\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i \in I}} \lfloor \text{BCH} \rfloor$$

$$\frac{\begin{array}{c}\forall i \in I \exists j \in J, \qquad l_i = l_j \qquad \mathcal{C} \wedge e_i; \Gamma \vdash P_i[e'_i/x_i] \rhd \Delta, k[\mathtt{p}] : \mathcal{L}_j[e'_i/x_j] \\ \mathcal{C} \wedge e_i \supset (A_j \wedge (E_i = E_j) \wedge (\mathcal{C} \ \mathtt{after}\ E_j))[e'_i/x_i]\end{array}}{\mathcal{C}; \Gamma \vdash k[\mathtt{p}, \mathtt{q}]!\{e_i \mapsto l_i\langle e'_i\rangle(x_i)\langle E_i\rangle; P_i\}_{i \in I} \rhd \Delta, k[\mathtt{p}] : \mathtt{q}!\{l_j(x_j : U_j)\{A_j\}\langle E_j\rangle.\mathcal{L}_j\}_{j \in J}} \lfloor \text{SEL} \rfloor$$

$$\frac{\mathcal{C}; \Gamma \vdash P_1 \rhd \Delta_1 \quad \mathcal{C}; \Gamma \vdash P_2 \rhd \Delta_2}{\mathcal{C}; \Gamma \vdash P_1 \mid P_2 \rhd \Delta_1, \Delta_2} \qquad \frac{\Delta \text{ end only}}{\mathcal{C}; \Gamma \vdash \mathbf{0} \rhd \Delta} \lfloor \text{PAR/END} \rfloor$$

$$\frac{\mathcal{L}_1[e/x], \ldots, \mathcal{L}_n[e/x] \ \text{ well-asserted}}{\mathcal{C}; \Gamma, X : (x)\mathcal{L}_1 @ \mathtt{p}_1, \ldots, \mathcal{L}_n @ \mathtt{p}_n \vdash X\langle e\rangle \rhd s[\mathtt{p}_1] : \mathcal{L}_1[e/x], \ldots, s[\mathtt{p}_n] : \mathcal{L}_n[e/x]} \lfloor \text{VAR} \rfloor$$

$$\frac{\mathcal{C}; \Gamma, X : (x)\mathcal{L}_1 @ \mathtt{p}_1, \ldots, \mathcal{L}_n @ \mathtt{p}_n \vdash P \rhd s[\mathtt{p}_1] : \mathcal{L}_1, \ldots, s[\mathtt{p}_n] : \mathcal{L}_n}{\mathcal{C}; \Gamma \vdash (\mu X(x).P)\langle e\rangle \rhd s[\mathtt{p}_1] : \mathcal{L}_1[e/x], \ldots, s[\mathtt{p}_n] : \mathcal{L}_n[e/x]} \lfloor \text{REC} \rfloor$$

Fig. 4. Proof rules for networks (top) and proof rules for processes (bottom)

virtual states must satisfy the principal invariant $\mathcal{I}(\sigma_p)$. P is then validated in the assertion environment extended (i.e., in conjunction with) the principal invariant.

[N2] is the rule for refinement. This rule is useful to validate processes even if they do not match exactly a given assertion as long as they implement a behaviour that is 'more refined' than the one prescribed. Refinement is also necessary to prove completeness of these rules (Theorem 14). We use the following refinement relation between specifications: $(\Gamma', \Delta', \sigma') \Rrightarrow (\Gamma, \Delta, \sigma)$ if $(\Gamma', \Delta', \sigma')$ specifies a more refined behaviour than (Γ, Δ, σ), in that it poses more restrictions on the output actions and poses less restrictions on the input actions. [N2] allows to refine the assertion environment $\mathcal{C}$ by considering, in the premise, a weaker set of assumptions $\mathcal{C}'$.

[N3] is for empty networks and [N4] is for decomposing the validation of networks.

[Macc] validates a session accept on a shared channel a as role $\mathtt{i}$ provided that a is in the domain of Γ, and that the continuation P is validated against the specification Δ extended with the new session $y[\mathtt{i}]$. In the (now active) session $y[\mathtt{i}]$, P must behave as $\Gamma(a)$ projected on role $\mathtt{i}$. The rule for session request is similar hence omitted.

[Bch] validates the branching process. Δ must include an active session $k[\mathtt{q}]$ on session channel k for the receiver role $\mathtt{q}$. In the premise, the continuation for each branch i is required to be still valid in the assertion environment extended with A_i. In the second clause of the premise, for each branch i the update E_i must not invalidate $\mathcal{C}$; this ensures that the update does not invalidate the principal invariant. The invariant is not mentioned explicitly (to keep the proof rules concise), but it is implied by $\mathcal{C}$. In fact, $\mathcal{C}$ is the conjunction of (1) the principal invariant (by [N1]), (2) possibly some interaction predicates (by [Bch]), and (3) possibly some boolean expressions (by [Sel]). Since predicates (2), (3) and A_i do not contain free state variables[8], then E_i can only invalidate the principal invariant (1); on the other hand (2), (3) and A_i are necessary premises (i.e., $\mathcal{C} \wedge A_i$ before the implication) as they may constrain interaction variables used by E_i.

In [Sel] each branch i of the process must correspond to a branch j of the specification ($l_i = l_j$). The continuation must be validated in assertion environment $\mathcal{C}$ extended with the closure $\underline{e_i}$ of the condition of the branch e_i. The closure of boolean expression e_i is defined as the closure for predicates (see Remark 4). The clause at the bottom of the premise requires that, under the assumption $\mathcal{C} \wedge e_i$: (1) expression e_i' satisfies A_j, (2) assertion and process have the same effects/updates on the states, (3) update E_j does not invalidate the principal invariant. [9]

[Par] is similar to [N2] but for parallel processes. [End] validates the idle process provided that each active session in the specification Δ is of the form $y[\mathtt{p}] : \mathsf{end}$.

[Var] validates recursive call given that the active sessions in Δ correspond to the roles and local assertions associated to process variable X in Γ and that each $\mathcal{L}_i$ is still well-asserted when the recursion parameter is substituted with e. [Rec] is the standard rule for recursion definition. The validation of recursive processes is handled in a similar way to [5]; it uses a refinement rule for processes, similar to [N2] and omitted for

[8] By history-sensitivity A_i does not include any free state variable.

[9] [Bch]/[Sel] can be extended to delegation adding the following clause for $U_i = \langle \mathcal{L} \rangle$: ([Bch]) $\mathcal{C} \wedge A_i; \Gamma \vdash P_i \rhd \Delta, k[\mathtt{q}] : \mathcal{L}_i, x_i : \mathcal{L}$, and ([Sel]) $\mathcal{C} \wedge \underline{e_i}; \Gamma \vdash P[e_i'/x_i] \rhd \Delta', k[\mathtt{p}] : \mathcal{L}_j[e_i'/x_j]$ with $\Delta = \Delta'', e_i : \mathcal{L}_i'$ and $\Delta' = \Delta''$.

simplicity, and the fact that assertions are refined by their unfolding. See [4] for more details.

Example 9. Consider the located process $\lceil P_{\mathtt{S}} \mid P_{\mathtt{C}} \rceil \sigma_p$ from Example 8 that executes two parallel threads: one selling a ticket and the other one buying another kind of ticket from another principal (the other principal is not modelled here). We show the validation of $\mathsf{true}; \Gamma \vdash \lceil P_{\mathtt{S}} \mid P_{\mathtt{C}} \rceil \sigma_p \rhd \lceil \varnothing \rceil \sigma_a$ proceeding top-down using the rules in Figure 4.

The global specification $\lceil \varnothing \rceil \sigma_a$ is initially empty since there are no active sessions. The active sessions will be added upon session request/accept by $P_{\mathtt{S}}$ and $P_{\mathtt{C}}$. We assume $\sigma_p = \sigma_a = \{\mathtt{count} : \mathtt{int}, \mathtt{credit} : \mathtt{int}\} \mapsto \{10, 500\}$ and initially $\mathcal{C} = \mathsf{true}$.

We first apply $\lfloor \text{N1} \rfloor$ with $\mathcal{I}(\sigma_p) = \mathtt{credit} \geqslant 0 \wedge 0 \leqslant \mathtt{count} \leqslant 1000$ from Example 8. For readability we will write $\mathcal{I}$ instead of $\mathcal{I}(\sigma_p)$ in this example. Note that $\mathcal{I}$ is satisfied by the local and virtual state. Next we apply rule $\lfloor \text{PAR} \rfloor$ that decomposes the derivation of two threads for $P_{\mathtt{S}}$ and $P_{\mathtt{C}}$. We omit the illustration of the latter thread.

Below we illustrate the application of rule $\lfloor \text{MACC} \rfloor$ and $\lfloor \text{BCH} \rfloor$ to the former thread:

$$\dfrac{\dfrac{\mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}; \Gamma \vdash P'_{\mathtt{S}} \rhd z[\mathtt{S}] : \mathcal{L}'_{\mathtt{S}} \quad \mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\} \supset (\mathcal{I}\ \mathtt{after}\ \varnothing)}{\mathcal{I}; \Gamma \vdash z[\mathtt{C}, \mathtt{S}]?(x).P'_{\mathtt{S}} \rhd z[\mathtt{S}] : \mathtt{C}?(x : \mathtt{Nat})\{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}.\mathcal{L}'_{\mathtt{S}}}\ \lfloor \text{BCH} \rfloor}{\mathcal{I}; \Gamma \vdash P_{\mathtt{S}} \rhd \varnothing}\ \lfloor \text{MACC} \rfloor$$

For readability we will simplify $\mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}$ with the equivalent predicate $\mathcal{I}$. Next, by $\lfloor \text{SEL} \rfloor$, setting $e = \mathtt{count} < 1000 \wedge x \geqslant 10$, and $E_{ok} = \mathtt{count} := \mathtt{count} + 1,\ \mathtt{credit} := \mathtt{credit} + x$:

$$\dfrac{\begin{array}{c}\mathcal{I} \wedge e \supset (\mathtt{count} < 1000 \wedge y = \mathtt{count} \wedge E_{ok} = E_{ok} \wedge \mathcal{I}\ \mathtt{after}\ E_{ok})[\mathtt{count}/y] \quad \mathcal{I}; \Gamma \vdash \mathbf{0} \rhd z[\mathtt{S}] : \mathsf{end} \\ \mathcal{I} \wedge \neg e \supset \mathsf{true} \quad \mathcal{I}; \Gamma \vdash \mathbf{0} \rhd z[\mathtt{S}] : \mathsf{end}\end{array}}{\begin{array}{l}\mathcal{I}; \Gamma \vdash z[\mathtt{S}, \mathtt{C}]!\{e \mapsto \mathsf{ok}\langle \mathtt{count} \rangle (y) \langle E_{ok} \rangle.\mathbf{0}, \neg e \mapsto \mathsf{ko}.\mathbf{0}\} \\ \qquad \vdash z[\mathtt{S}] : \mathtt{C}!\{\mathsf{ok}(y : \mathtt{Nat})\{\mathtt{count} < 1000 \wedge y = \mathtt{count}\}\langle E_{ok} \rangle.\mathsf{end},\ \mathsf{ko}.\mathsf{end}\}\end{array}}$$

where each line in the premise refers to a branch (i.e., ok and ko). The most delicate clause is $\mathcal{I} \wedge e \supset (\mathtt{count} < 1000 \wedge y = \mathtt{count} \wedge E_{ok} = E_{ok} \wedge \mathcal{I}\ \mathtt{after}\ E_{ok})[\mathtt{count}/y]$ which requires: (1) the interaction predicate to be satisfied under the current assumptions, and in fact $(\mathtt{count} < 1000 \wedge y = \mathtt{count})[\mathtt{count}/y]$ is implied by e, (2) the updates to be consistent, and in fact trivially $E_{ok} = E_{ok}$, and (3) the update to not invalidate the invariant, and in fact $\mathtt{credit} + x \geqslant 0 \wedge 0 \leqslant \mathtt{count} + 1 \leqslant 1000$ is true under the assumptions $\mathtt{credit} \geqslant 0$, $x \geqslant 10$ and $0 \leqslant \mathtt{count}$. Finally we apply $\lfloor \text{END} \rfloor$ to the second premise of each branch.

The effectiveness of the proof rules depends on the logic chosen for the predicates, which depends on the application scenario. An example which fits these criteria is the Presburger arithmetic, which is often sufficiently expressive: practical uses of multiplication are encodable [17], and formulae with quantifiers may be calculated efficiently [20, 22].

Proposition 10. *The proof of $\mathcal{C}; \Gamma \vdash N \rhd \Sigma$ is decidable if predicate evaluation is decidable.*

5 Soundness and Completeness of the Validation Rules

We define a labelled transition relation for specifications $\langle \Gamma, \Sigma \rangle$ using the same labels as for networks. The main difference with the rules for networks is that predicates must be satisfied for the transition to occur. We illustrate below the most remarkable rules (see [4] for the other rules). The rule for session request:

$$\langle (a : \mathcal{G}, \Gamma); [\Delta]\sigma \rangle \xrightarrow{\overline{a}[\mathtt{n}]\langle s \rangle} \langle (a : \mathcal{G}, \Gamma); [\Delta, s[1] : \mathcal{G} \upharpoonright 1]\sigma \rangle$$

extends Δ with the new session, given that $a : \mathcal{G}$ in Γ and the current state satisfies assertion invariant A. The rule for session accept is dual. The rule for selection/send:

$$\frac{j \in I \quad \sigma \models A_j[\mathtt{n}/x_j] \quad \sigma' = \sigma \,\mathtt{after}\, E_j[\mathtt{n}/x_j]}{\langle \Gamma; [\Delta, s[\mathtt{p}] : \mathtt{q}!\{l_i(x_i : U_i)\{A_i\}\langle E_i \rangle.\mathcal{L}_i\}_{i \in I}]\sigma \rangle \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle \mathtt{n} \rangle} \langle \Gamma; [\Delta, s[\mathtt{p}] : \mathcal{L}_j[\mathtt{n}/x_j]]\sigma' \rangle}$$

moves to the continuation $\mathcal{L}_j$ of the selected branch with the updated state σ', given that the sent value $\mathtt{n}$ satisfies predicate A_j for branch j in the current state σ.

Semantic conformance is defined using *conditional simulation* [5] to relate networks N to specifications $\langle \Gamma; \Sigma \rangle$.

Definition 11 (Conditional Simulation). A binary relation $\mathcal{R}$ over N and $\langle \Gamma; \Sigma \rangle$ is a *conditional simulation* if, for each $(N, \langle \Gamma; \Sigma \rangle) \in \mathcal{R}$, if $N \xrightarrow{\ell} N'$ with ℓ being:

(1) a branching then $\langle \Gamma; \Sigma \rangle$ is capable to move at the subject of ℓ, and if $\langle \Gamma; \Sigma \rangle \xrightarrow{\ell} \langle \Gamma; \Sigma' \rangle$ then $(N', \langle \Gamma; \Sigma' \rangle) \in \mathcal{R}$;

(2) a select, session request/accept, τ then $\langle \Gamma; \Sigma \rangle \xrightarrow{\ell} \langle \Gamma; \Sigma' \rangle$ and $(N', \langle \Gamma; \Sigma' \rangle) \in \mathcal{R}$.

We write $N \lesssim \langle \Gamma; \Sigma \rangle$ if there exists a conditional simulation $\mathcal{R}$ s.t. $(N, \langle \Gamma; \Sigma \rangle) \in \mathcal{R}$.

Conditional simulation is like standard simulation for all types of actions except for branching, for which it requires N to be simulated only for legal values/labels (i.e., a process must conform to a given specification as long as its environment does so).

Definition 12 (Satisfaction). N satisfies Σ in Γ and $\mathcal{C}$, written $\mathcal{C}; \Gamma \models N \rhd \Sigma$, if for all closing substitutions $\tilde{\sigma}$ over N and Σ respecting Γ and $\mathcal{C}$, $N\tilde{\sigma} \lesssim \langle \Gamma; \Sigma\tilde{\sigma} \rangle$.

We write $\Gamma \models N \rhd \Sigma$ when $\mathcal{C}$ is true (e.g., for initial networks). Soundness and completeness for initial networks are stated below.

Theorem 13 (Soundness of Proof Rules). *Let N be an initial network. Then $\Gamma \vdash N \rhd \Sigma$ implies $\Gamma \models N \rhd \Sigma$.*

Theorem 14 (Completeness of Proof Rules). *Let $N \equiv \prod_{i \in I}[P_i]\sigma_{pi}$ be an initial network and $\Sigma = \prod_{i \in I}[\Delta_i]\sigma_{ai}$ be a specification. Assume that for all $i \in I$:* (1) $\mathtt{id}(\sigma_{pi}) = \mathtt{id}(\sigma_{ai})$, (2) $dom(\sigma_{pi}) = dom(\sigma_{ai})$, *and* (3) $\mathcal{I}(\sigma_{pi})$ *equivalent to* true. *If $\Gamma \models N \rhd \Sigma$ then $\Gamma \vdash N \rhd \Sigma$.*

(1-2) are for symmetry between N and Σ. (3) is necessary since the principals in N can make updates that differ from those made by the corresponding specifications in Σ; this may not compromise the observable behaviour of N with respect to Σ, but N may invalidate some principal invariant which would make the thesis false.

6 Related Work and Further Topics

The preceding integrations of session types with logical constraints include [13], based on concurrent constraints ensuring bi-linear usage of channels, and [5], based on logical annotations on interactions, do not treat stateful properties. The combination of types and logical assertions referring to local state newly proposed in this paper enable fine-grained specifications and validation, which are not possible in [5, 13].

The expressiveness of the session type-based analyses has been greatly extended these past few years. On one side, the conversation calculus [8], contracts [11] and dynamic multirole session types [14] have opened the way to the modelling of protocols complex in their shapes, by describing more accurately how sessions can be joined or left, who is allowed participate. On the other side, works such as [5, 9] improved the way interactions inside a session are described: in [5], an assertion framework ensures logical properties on the communicated values, in [9], a security analysis guarantees that the coherence of the information flow is preserved. Our work improves the session type analyses in both directions: by proposing a division of the process being tested into separate principals that can join one or several sessions independently when conditions are matched and manage their own state, and by giving a description, inside each session, of the internal state of each participant and the property it should satisfy. A recent work [12] examines conditions to ensure that a stateful specification is robust w.r.t. asynchronous communications. Our work provides a complete proof system ensuring soundness for processes, whereas [12] only addresses properties of types.

The refinement types for channels (e.g. [3]) specify value dependency with logical constraints. For example, one might write $?(x:\mathsf{int}, !\{y : \mathsf{int} \mid y > x\})$ (using the notation from [16]). It specifies a dependency at a *single point* (channel). Our theory, based on multiparty sessions, can verify processes against a contract globally agreed by multiple distributed peers. [2] uses refinement types for channels to verify authentication in multiparty session protocols, but does not consider multi-session properties.

The work [7] investigates a relationship between a dual intuitionistic linear logic and binary session types, and shows that the former defines a proof system for a session calculus which can automatically characterise and guarantee a session fidelity and global progress. None of the above works treat either virtual states or logical specifications for interleaved multiparty sessions.

The use of Rely-Guarantee conditions or other related methods [19] instead of a single invariant does not increase the expressiveness of our system, but could ease proofs for parallel composition.

A future direction is to link between our static analysis and a dynamic monitor-based approach. Using our local specification as a monitor at each end-point, incoming and outgoing messages can be verified and filtered. We are currently working on this topic with [21] based on the logic developed in this paper.

References

1. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)

2. Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C., Leifer, J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: CSF, pp. 124–140 (2009)
3. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: POPL, pp. 445–456 (2010)
4. Bocchi, L., Demangeon, R., Yoshida, N.: A multiparty multi-session logic (extended report), http://www.cs.le.ac.uk/people/lb148/statefulassertions.html
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
6. Bocchi, L., Lange, J., Tuosto, E.: Three algorithms and a methodology for amending contracts for choreographies. Scientific Annals of Computer Science 22(1), 61–104 (2012)
7. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010)
8. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
9. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. In: EXPRESS. EPTCS, vol. 64, pp. 16–30 (2011)
10. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
11. Castagna, G., Padovani, L.: Contracts for mobile processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)
12. Chen, T.-C., Honda, K.: Specifying stateful asynchronous properties for distributed programs. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 209–224. Springer, Heidelberg (2012)
13. Coppo, M., Dezani-Ciancaglini, M.: Structured communications with concurrent constraints. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 104–125. Springer, Heidelberg (2009)
14. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: POPL, pp. 435–446. ACM (2011)
15. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18, 453–457 (1975)
16. Freeman, T., Pfenning, F.: Refinement types for ML. SIGPLAN Not. 26(6), 268–277 (1991)
17. Ganai, M.K.: Efficient decision procedure for bounded integer non-linear operations using SMT($\mathcal{LIA}$). In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 68–83. Springer, Heidelberg (2009)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL, pp. 273–284. ACM (2008)
19. Jones, C.B.: Abstraction as a unifying link for formal approaches to concurrency. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 1–15. Springer, Heidelberg (2012)
20. Nelson, G., Oppen, D.C.: A simplifier based on efficient decision algorithms. In: POPL, pp. 141–150. ACM (1978)
21. Ocean Observatories Initiative (OOI), http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/
22. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing 1991, pp. 4–13. ACM, New York (1991)

LTS Semantics for Compensation-Based Processes*

Roberto Bruni[1] and Anne Kersten Kauer[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] IMT Institute for Advanced Studies, Lucca, Italy

Abstract. Business processes design is an error-prone task often relying on long-running transactions with compensations. Unambiguous formal semantics and flexible verification tools should be used for early validation of processes. To this aim, we define a small-step semantics for the Sagas calculus according to the so-called "coordinated interruption" policy. We show that it can be tuned via small changes to deal with other compensation policies and discuss possible enhancements.

1 Introduction

Long-running transactions (LRTs) in business processes are composed by services taken off-the-shelf. One important problem is failure recovery, i.e., the ability to bring a faulty process back to a consistent state. Processes may grow large and complex and when a fault occurs the designer has to take several constraints into account: all sibling activities that run unaware of the fault should be stopped and all the activities that were executed before the fault need to be undone in a suitable order. Moreover, in many cases, an action, like a service invocation, cannot simply be undone: it can be an ACID transaction on its own, or it may involve asynchronous messaging (e.g., via SMTP).

A *compensation* is the means of reversing the effects of an activity in case a later fault occurs in the business process. Compensations were introduced in [17] to implement (non-ACID) database LRTs as a sequence of short, ACID sub-transactions $t_1 ... t_n$. Each t_i had an associated activity c_i, its compensation, to be installed when t_i committed, and to be executed if a fault occurred before the whole LRT was committed. Compensations are executed in the reverse order of installation. For example, if t_3 fails, then the observed activities are $t_1 t_2 c_2 c_1$. Service-oriented computing is a particularly favourable setting for the concept of compensation, because services are designed without knowing in advance the context where they will be used. For example, take a process that receives an order form with multiple items and delegates each request to a different supplier. If some request fails while others already succeeded, the process may cancel the successful requests and inform the customer about the failure. Still, certain cancellations may involve fees and others may not be possible at all. Moreover,

* Research supported by the EU Integrated Project 257414 ASCENS and by the Italian MIUR Project IPODS (PRIN 2008).

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 112–128, 2013.
© Springer-Verlag Berlin Heidelberg 2013

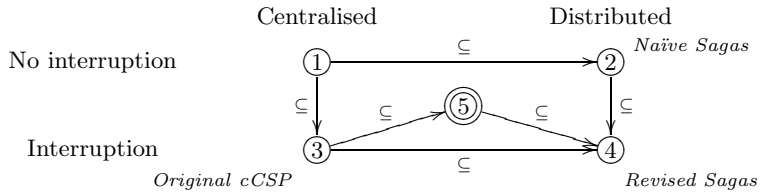


Fig. 1. Compensation policies (arrows stand for trace inclusion)

when the fault occurs one would like to interrupt non-issued requests. Thus, it is natural to demand that a service comes with one (or more) companion service(s) for compensation. An action and its compensation form a *compensation pair*.

Concurrency makes process design an error-prone activity: processes must be assigned with unambiguous semantics and early validated to detect unwanted behaviour and to suppress as many inconsistencies as possible. We focus on the semantics of the Sagas calculus [7]. The core fragment of *parallel* Sagas has been sufficient to characterise different compensation policies for parallel processes. A thorough analysis is presented in [3] by comparing the Sagas calculus with *compensating* CSP [10] (cCSP) along two axes of classification: i) interruption of siblings in case of an abort (*interruption* vs *no interruption*); ii) whether compensations are started at the same time or siblings can start their compensation on their own (*centralised* vs *distributed*). The relation between the four different policies is displayed in Fig. 1. The fifth policy (double lined in Fig. 1) has been formalised in [6] and proved more satisfactory than #1–4, and all semantics #1–5 coincide on the sequential fragment of Sagas. A key contribution in [6] is the definition of a concurrent semantics for policy #5, obtained by encoding Sagas processes in (safe) Petri nets. The Petri net model is more informative than trace semantics, because it accounts for the branching of processes arising from the propagation of interrupts, but the sophisticated mechanism needed for handling interrupts introduces many auxiliary places and transitions that make the Petri net model quite intricate to parse (§ 5.1) and difficult to extend (§ 7).

Our aim is to provide an operational semantics for Sagas whose main requirements are: i) it must follow the small-step style of operational semantics, so to account for the branching caused by the propagation of interrupts; ii) other policies can be implemented without radical redesign; iii) it must be easy to introduce other features, like choice, iteration, and faulty compensations (crashes).

In this paper we propose an LTS semantics that meets all the above requirements. The main result consists of the correspondence theorems with the existing semantics. We started by considering the "optimal" policy #5 and were guided by the correspondence with the Petri net semantics to correct many wrong design choices in our first attempts. The main result is the proof that our LTS semantics matches the Petri nets semantics in [6] up to weak bisimilarity. This gives a way to read markings as (weak bisimilar) terms of a process algebra that describes the run-time status of the process.

$$\begin{array}{llll} (\text{ACT}) & A, B ::= a \mid \mathit{skip} \mid \mathit{throw} & (\text{PROCESS}) & P, Q ::= X \mid P; Q \mid P|Q \\ (\text{STEP}) & X \quad ::= A \div B & (\text{SAGA}) & S, T ::= A \mid S; T \mid S|T \mid \{\!|P|\!\} \end{array}$$

Fig. 2. Core fragment of Sagas

Synopsis. In § 2 we recall the denotational semantics of Sagas. In § 3 we define the LTS semantics for the sequential fragment only, and extend it to the parallel case in § 4. In § 5 we sketch the Petri net semantics from [6] and outline the technique used for proving the main result. In § 6 and § 7 we show the flexibility of our LTS semantics in accommodating other policies and advanced features. Related work, concluding remarks and future work are collected in § 8.

2 Background

The syntax of the parallel Sagas calculus is in Fig. 2. Atomic actions A include generic activities $a \in \mathcal{A}$, the vacuous activity *skip* and the faulty activity *throw*. In a compensation pair $A \div B$, the activity B compensates A. We write *throww* for *throw* $\div$ *skip*. Beside the ordinary sequential and parallel composition, we use $\{\!|P|\!\}$ to enclose a compensable process within a saga. Below, we outline the denotational semantics of policies #1–5, while the Petri net semantics for policy #5 is recalled in § 5.1. The Petri net semantics and our novel LTS semantics are parametric to the context of execution that fixes the success or failure of activities. Let $\Omega = \{\boxdot, \boxtimes\}$. A *context* Γ is a function $\Gamma : \mathcal{A} \to \Omega$ that maps a basic activity to $\boxdot$ or $\boxtimes$ depending on whether it commits or aborts, with $\Gamma(\mathit{skip}) = \boxdot$ and $\Gamma(\mathit{throw}) = \boxtimes$. We assume a compensation activity cannot fail. Dealing with faulty compensations is discussed in § 7. The denotational semantics does not use Γ: only *throww* is used for failures.

Notation. A trace for a saga is a string $s\langle\omega\rangle$, where $s \in \mathcal{A}^*$ is said the *observable flow* and $\omega \in \mathcal{R}$ is the *final event*, with $\mathcal{R} = \{\checkmark, !, ?\}$ and $\mathcal{A} \cap \mathcal{R} = \emptyset$ ($\checkmark$ stands for success, ! for fail, and ? for yield to an interrupt). Note that ? appears only in traces of compensable processes. We let ϵ denote the empty observable flow. Slightly abusing the notation, we let $p, q, \ldots$ range over traces and also observable flows. We denote by $p|||q$ the set of all possible interleavings of the observable flows p and q, with final event $\omega\&\omega'$, where & is associative and commutative. A trace of a compensable process P is a pair (p, q), where p is the *forward trace* and q is a *compensation trace* for p. We find it convenient to define policy #3 first (see Fig. 3) and then explain the other ones by difference. We use policy numbers as subscripts of the symbol $\triangleq$ when the defining equation may not be valid for other policies (e.g., $\triangleq_{3,4}$ means the definition is valid for policies #3 and #4). Later, we write $[\![\cdot]\!]_i$ to denote the trace semantics w.r.t. policy #i.

Sagas (policies #1–5). For sagas, the most interesting case is the one of $\{\!|P|\!\}$: it selects all successful forward traces $s\langle\checkmark\rangle$ of P, and the traces sq, corresponding to failed forward flows $s\langle!\rangle$ followed by their compensations q.

Interruption and Centralized Compensation (Policy #3). When composing compensable traces in series, the forward trace corresponds to the sequential

Traces of sagas

$S;T \triangleq \{p;q \mid p \in S \wedge q \in T\}$ $\quad a \triangleq \{a\langle\checkmark\rangle\}$

$S|T \triangleq \{r \mid r \in (p||q) \wedge p \in S \wedge q \in T\}$ $\quad skip \triangleq \{\langle\checkmark\rangle\}$

$\{\![P]\!\} \triangleq \{s\langle\checkmark\rangle \mid (s\langle\checkmark\rangle, q) \in P\} \cup \{sq \mid (s\langle!\rangle, q) \in P\}$ $\quad throw \triangleq \{\langle!\rangle\}$

Composition of standard traces

Sequential $\begin{cases} p\langle\checkmark\rangle;q \triangleq pq \\ p\langle\omega\rangle;q \triangleq p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $p\langle\omega\rangle||q\langle\omega'\rangle \triangleq \{r\langle\omega\&\omega'\rangle \mid r \in (p|||q)\}$, where

ω \ ω'	!	!	!	?	?	✓
!	!	!	!	?	?	✓
?	?	✓	?	✓	✓	
!	!	!	!	?	?	✓

and $\begin{cases} p|||\epsilon \triangleq \epsilon|||p \triangleq \{p\} \\ a\ p|||b\ q \triangleq \{a\ r \mid r \in (p|||b\ q)\} \cup \{b\ r \mid r \in (a\ p|||q)\} \end{cases}$

Traces of compensable processes

$A \div B \triangleq_{3,4} \{(p,q) \mid p \in A \wedge q \in B\} \cup \{(\langle?\rangle, \langle\checkmark\rangle)\}$

$P;Q \triangleq \{pp;qq \mid pp \in P \wedge qq \in Q\}$

$P|Q \triangleq \{rr \mid rr \in (pp||qq) \wedge pp \in P \wedge qq \in Q\}$

Composition of compensable traces

Sequential $\begin{cases} (p\langle\checkmark\rangle, p');(q,q') \triangleq (pq, q';p') \\ (p\langle\omega\rangle, p');(q,q') \triangleq (p\langle\omega\rangle, p') \text{ when } \omega \neq \checkmark \end{cases}$

Parallel $(p,p')|||(q,q') \triangleq_{1,3} \{(r,r') \mid r \in (p||q) \wedge r' \in (p'||q')\}$

Fig. 3. Denotational semantics (policy #3)

composition of the original forward traces, while the compensation trace starts by the second compensation followed by the first one. The parallel composition is defined (pairwise) interleaving the forward flows and the backward flows.

No Interruption and Centralized Compensation (Policy #1). Policy #1 differs from policy #3 just by ruling out interruption.

$$A \div B \triangleq_{1,2} \{(p,q) \mid p \in A \wedge q \in B\}$$

Interruption and Distributed Compensation (Policy #4). Policy #4 differs from policy #3 only by the following definition of parallel composition of compensable traces. Note that compensations can be triggered by "guessing" that a fault will be issued.

$$(p\langle\checkmark\rangle, p')|||(q\langle\checkmark\rangle, q') \triangleq_{2,4} \{(r\langle\checkmark\rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \cup \{(r\langle?\rangle, \langle\omega\rangle) \mid r\langle\omega\rangle \in (pp'||qq')\}$$

$$(p\langle\omega\rangle, p')|||(q\langle\omega'\rangle, q') \triangleq_{2,4} \{(r\langle\omega\&\omega'\rangle, \langle\omega''\rangle) \mid r\langle\omega''\rangle \in (pp'||qq')\} \quad \text{if } \omega\&\omega' \neq \checkmark$$

No Interruption and Distributed Compensation (Policy #2). Policy #2 differs from policy #3 by combining together the two changes above.

Coordinated Compensation (Policy #5). Policy #5 differs from policy #3 by slightly changing the semantics of compensation pairs, to allow a successfully completed activity to yield, and the semantics of parallel composition, to account for distributed compensation without the guessing mechanism.

$$
\begin{array}{rcl}
A \div B & \triangleq_5 & \{(p,q) \mid p \in A \wedge q \in B\} \cup \{(\langle ?\rangle, \langle \checkmark \rangle)\} \cup \\
 & & \{(p\langle ?\rangle, q) \mid p\langle \checkmark \rangle \in A \wedge q \in B\} \\
(p\langle \checkmark \rangle, p') || (q\langle \checkmark \rangle, q') & \triangleq_5 & \{(r\langle \checkmark \rangle, r') \mid r \in (p|||q) \wedge r' \in (p'||q')\} \\
(p\langle \omega \rangle, p') || (q\langle \omega' \rangle, q') & \triangleq_5 & \begin{cases} itp((p\langle \omega \rangle, p'), (q\langle \omega' \rangle, q')) \cup & \\ itp((q\langle \omega' \rangle, q'), (p\langle \omega \rangle, p')) & \text{when } \omega, \omega' \neq \checkmark \\ \emptyset & \text{otherwise} \end{cases} \\
itp((p\langle \omega \rangle, p'), (q\langle \omega' \rangle, q')) & \triangleq & \{((p|||q_1)\langle \omega \rangle, (p'||q_2 q')) \mid q = q_1 q_2\}
\end{array}
$$

In Summary: in #1 and #2 all sibling processes will finish their execution before compensating; in #3, aborted and interrupted processes cannot start the compensation before all their siblings are ready to compensate; #2 and #4 rely on a "guessing" mechanism for which a process may start its compensation when a sibling will fail in the future; #5 is "optimal" in the sense that distributed compensations can start as soon as needed, but only after an actual fault occurred.

Example 1. Consider the processing of an order in an eStore. First the order is accepted, then, in parallel, the customer's credit card is processed and the order is packed and the courier is booked. If something goes wrong each activity can be compensated, the courier will be cancelled, the order unpacked, for the credit card an error message will be sent and the order can be deleted. Assume that the booking of the courier will always fail and is thus replaced with *throww*.

$$eStore \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; (\mathsf{pC} \div \overline{\mathsf{pC}} | \mathsf{pO} \div \overline{\mathsf{pO}}; throww)$$

In policies #1 and #2, pC and its compensation will always be executed, while policies #3 and #4 admit, e.g., the trace $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \overline{\mathsf{aO}}$. Policies #1 and #3 are centralized and no compensation activity can precede a forward activity. Policies #2 and #4 admit the trace $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \mathsf{pC}\overline{\mathsf{pC}}\ \overline{\mathsf{aO}}$ (distributed case). They also admit the less realistic trace $\mathsf{aO}\ \mathsf{pC}\overline{\mathsf{pC}}\ \mathsf{pO}\overline{\mathsf{pO}}\ \overline{\mathsf{aO}}$ where the compensation $\overline{\mathsf{pC}}$ is executed before the actual *throww* could have issued a fault. This trace is forbidden in policy #5 (where $\mathsf{aO}\ \mathsf{pO}\overline{\mathsf{pO}}\ \mathsf{pC}\overline{\mathsf{pC}}\ \overline{\mathsf{aO}}$ is still allowed).

3 A Small-Step Semantics for Sequential Sagas

In this section we define a small-step LTS semantics for the sequential fragment of the Sagas calculus. (w.r.t. the syntax in Fig. 2, we ignore parallel composition). To be able to reason on intermediate states in the execution of a process we introduce a runtime syntax.

$$
\begin{array}{lll}
(\textsc{comp})\ C ::= A \mid C;C \mid \mathbf{nil} & (\textsc{process})\ P ::= \ldots \mid P\$C \mid [C] \\
 & (\textsc{saga})\ \ S ::= \ldots \mid \mathbf{nil}
\end{array}
$$

First we add a distinct type for compensations. They can either be basic activities A, the sequential composition of compensations $C;C$ or **nil**. With **nil** we denote completion of a compensation, in the sense that, e.g., the compensation $\mathbf{nil};C$ can never execute activities in C. For compensable processes, $P\$C$ denotes a process P running with the already installed compensation C. Compensations

$$\text{(C-ACT)}\ \frac{}{\Gamma \vdash A \xrightarrow{A} \mathbf{nil}} \qquad \text{(C-SEQ1)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge \neg dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} C'; D} \qquad \text{(C-SEQ2)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C' \wedge dn(C')}{\Gamma \vdash C; D \xrightarrow{\lambda} D}$$

Fig. 4. LTS for sequential compensations

$$\text{(S-ACT)}\ \frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \div B \xrightarrow{A} \boxdot, [B]} \qquad \text{(SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, P'; Q}$$

$$\text{(F-ACT)}\ \frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \div B \xrightarrow{\tau} \boxtimes, [\mathbf{nil}]} \qquad \text{(S-SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxdot, Q\$cmp(P')} \qquad \text{(A-SEQ)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P; Q \xrightarrow{\lambda} \boxtimes, P'}$$

$$\text{(STEP)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', P'\$C} \qquad \text{(AS-STEP1)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge tocmp(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [cmp(P'); C]}$$

$$\text{(AS-STEP2)}\ \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge dn_{\sigma'}(P') \wedge \neg tocmp(P')}{\Gamma \vdash \sigma, P\$C \xrightarrow{\lambda} \sigma', [C]} \qquad \text{(COMP)}\ \frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash \boxtimes, [C] \xrightarrow{\lambda} \boxtimes, [C']}$$

Fig. 5. LTS for sequential compensable processes

of P will be installed on top of C once P is finished. The completion of forward activities is denoted by $[C]$ instead of **nil**, because we need to consider the installed compensation C (informally, $[C]$ can be read as $\mathbf{nil}\$C$). We also add **nil** for marking the completion of a saga.

The small-step semantics is defined by three LTSs, one for each syntax category. Given the set of compensations $\mathcal{C}$, the set of compensable processes $\mathcal{P}$ and the set of sagas $\mathcal{S}$, we let $\mathcal{S}_C = \mathcal{C}$, $\mathcal{S}_P = \Omega \times \mathcal{P}$, $\mathcal{S}_S = \Omega \times \mathcal{S}$.

Definition 1. *The LTS semantics of (sequential) sagas is the least LTS* $(\mathcal{S}, L, \mathcal{T})$ *generated by the rules in Fig. 4–6, whose set of states is* $\mathcal{S} = \mathcal{S}_C \cup \mathcal{S}_P \cup \mathcal{S}_S$*, whose set of labels is* $L = \mathcal{A} \cup \{\tau\}$.

We will write transitions $t \in \mathcal{T}$ as $t : \Gamma \vdash s \xrightarrow{\lambda} s'$ for states s, s', a label $\lambda \in L$ and a context Γ. The component Ω in a state describes whether the process can still commit (it can still move forward) or must abort (a fault was issued that needs to be compensated). Note that states of the LTS for compensations have clearly no Ω component. Sagas initially start executing in a commit state.

The semantics exploits some auxiliary notation. The predicate dn_σ checks the completion of (the forward execution of) a compensable process. The subscript σ stands for $\boxdot$ or $\boxtimes$ and means that the process is either evaluated in a commit or an abort context. The predicate dn_σ is inductively defined as:

$$dn_\sigma([C]) \triangleq \mathtt{tt} \quad dn_\sigma(A \div B) \triangleq \mathtt{ff} \quad dn_\sigma(P\$C) \triangleq dn_\sigma(P; Q) \triangleq dn_\sigma(P)$$

$$\frac{A \mapsto_\Gamma \boxdot}{\Gamma \vdash \boxdot, A \xrightarrow{A} \boxdot, \mathbf{nil}}\ (\text{S-SACT}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \sigma', S' \wedge \neg dn(S')}{\Gamma \vdash \sigma, S;T \xrightarrow{\lambda} \sigma', S';T}\ (\text{SSEQ})$$

$$\frac{A \mapsto_\Gamma \boxtimes}{\Gamma \vdash \boxdot, A \xrightarrow{\tau} \boxtimes, \mathbf{nil}}\ (\text{F-SACT}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxdot, S' \wedge dn(S')}{\Gamma \vdash \sigma, S;T \xrightarrow{\lambda} \boxdot, T}\ (\text{S-SSEQ}) \qquad \frac{\Gamma \vdash \sigma, S \xrightarrow{\lambda} \boxtimes, S' \wedge dn(S')}{\Gamma \vdash \sigma, S;T \xrightarrow{\lambda} \boxtimes, S'}\ (\text{A-SSEQ})$$

$$\frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \sigma', P' \wedge \neg dn_{\sigma'}(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \sigma', \{\!|P'|\!\}}\ (\text{SAGA}) \qquad \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge tocmp(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxtimes, \{\!|P'|\!\}}\ (\text{A-SAGA1})$$

$$\frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn_{\boxdot}(P')}{\Gamma \vdash \boxdot, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}\ (\text{S-SAGA}) \qquad \frac{\Gamma \vdash \sigma, P \xrightarrow{\lambda} \boxtimes, P' \wedge dn_{\boxtimes}(P') \wedge \neg tocmp(P')}{\Gamma \vdash \sigma, \{\!|P|\!\} \xrightarrow{\lambda} \boxdot, \mathbf{nil}}\ (\text{A-SAGA2})$$

Fig. 6. LTS for sequential sagas

Note that for sequential processes dn is independent of the subscript; this will change when introducing parallel composition. Analogously, we define a predicate dn on compensations and sagas, together with a function $cmp(P)$ that extracts the installed compensation from a process P that is "done". When P is done, we use the shorthand $tocmp(P) \triangleq \neg dn(cmp(P))$ (i.e., $tocmp(P)$ holds when there is some compensation to run).

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq \mathtt{ff} \qquad dn(C;C') \triangleq dn(C)$$

$$dn(\mathbf{nil}) \triangleq \mathtt{tt} \qquad dn(A) \triangleq dn(\{\!|P|\!\}) \triangleq \mathtt{ff} \qquad dn(S;T) \triangleq dn(S)$$

$$cmp([C]) \triangleq C \qquad cmp(P;Q) \triangleq cmp(P) \qquad cmp(P\$C) \triangleq \begin{cases} C & \text{if } dn(cmp(P)) \\ cmp(P);C & \text{if } \neg dn(cmp(P)) \end{cases}$$

The rules in Fig. 4 handle compensations. As we assume a compensation is always successful, only rule C-ACT is needed for basic activities. Rules C-SEQ1 and C-SEQ2 exploit the "done" predicate to avoid reaching states such as $\mathbf{nil};C$.

For processes (Fig. 5), a basic activity A of $A \div B$ can either commit or abort, depending on the context: if A commits then B is installed (S-ACT); if A fails, then there is nothing to be compensated (F-ACT). A sequential composition $P;Q$ acts according to how P acts (SEQ and A-SEQ). If P finishes successfully (S-SEQ), then Q will run under the installed compensation $cmp(P')$. The process $P\$C$ acts according to P. When P finishes its compensation is installed on top of C (AS-STEP1). The rule AS-STEP2 ensures that a **nil** is not installed on top of a compensation. Compensations are executed via COMP.

The rules for sagas A and $S;T$ are as expected (Fig. 6). A saga $\{\!|P|\!\}$ can be executed as long as either it is still running forward (SAGA and S-SAGA) or it has already aborted and compensates (SAGA and A-SAGA1). If the saga aborts but is able to compensate, then it reaches a good state (A-SAGA2).

The formal correspondence between the LTS semantics and the denotational semantics of policies #1–5 is an immediate consequence of our main result and will be deferred to § 4 (see Corollary 1).

Example 2. Let $eS \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; \mathsf{pC} \div \overline{\mathsf{pC}}; \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}}$. Assume that the packing of the order fails, and let Γ map pO to $\boxtimes$ and the other actions to $\boxdot$. We have e.g. $\boxdot, \{\!|eS|\!\} \xrightarrow{\mathsf{aO}} \xrightarrow{\mathsf{pC}} \xrightarrow{\tau} \xrightarrow{\overline{\mathsf{pC}}} \xrightarrow{\overline{\mathsf{aO}}} \boxdot, \mathbf{nil}$, because

$$\boxdot, eS \xrightarrow{\mathsf{aO}} \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}}; \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{aO}} \xrightarrow{\mathsf{pC}}$$
$$\boxdot, ((\mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pC}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \boxtimes, [\overline{\mathsf{pC}}; \overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{pC}}} \boxtimes, [\overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{aO}}} \boxtimes, [\mathbf{nil}]$$

4 Extension to Concurrency

In this section we extend the LTS semantics to handle parallel Sagas. First, we extend the runtime syntax as follows:

$$\begin{array}{lll} (\textsc{comp}) & C ::= A \mid C;C \mid \mathbf{nil} \mid C|C \\ (\textsc{process}) & P ::= X \mid P;P \mid P\$C \mid [C] \mid P_\sigma|_{\sigma'}P \\ (\textsc{saga}) & S ::= A \mid S;S \mid \{\!|P|\!\} \mid \mathbf{nil} \mid S_\sigma|_{\sigma'}S \end{array}$$

We add parallel composition to compensations. We use subscripts for the parallel composition of processes or sagas ${}_\sigma|_{\sigma'}$ such that $\sigma, \sigma' \in \Omega$. If a thread is denoted with $\boxdot$, it can still move forward and commit. A thread denoted with $\boxtimes$ either aborted or was interrupted, so it can compensate. If a thread is denoted with a $\boxdot$ then also every parallel composition contained as a subprocess in this thread must have a $\boxdot$. Similarly if the global state is a $\boxdot$, any parallel composition in this state has subscripts $\boxdot$. We consider $P_\boxdot|_\boxdot Q$ part of the normal syntax, not just of the runtime syntax, and usually write just $P|Q$. We sometimes use $\|$ instead of ${}_\sigma|_{\sigma'}$ if the values of σ, σ' are irrelevant.

Definition 2. *The LTS semantics of parallel sagas is the least LTS $(\mathcal{S}, L, \mathcal{T})$ generated by the rules in Fig. 4–6 together with the rules in Fig. 7 (symmetric rules* C-PAR-R, PAR-R, INT-L *and* SPAR-R *are omitted).*

The semantics exploits some auxiliary notation. First, the binary function $\sqcap : \Omega \times \Omega \to \Omega$ is defined such that $\sigma \sqcap \sigma' = \boxdot$ iff $\sigma = \sigma' = \boxdot$. It is easy to check that $\sqcap$ is associative and commutative. Then, the predicates dn_σ, dn and the function cmp are extended to parallel composition:

$$dn(C|C') \triangleq dn(C) \wedge dn(C') \quad dn_\sigma(P_{\sigma_1}|_{\sigma_2}Q) \triangleq dn_\sigma(P) \wedge dn_\sigma(Q) \wedge (\sigma = \sigma_1 = \sigma_2)$$
$$dn(S_{\sigma_1}|_{\sigma_2}T) \triangleq dn(S) \wedge dn(T) \quad cmp(P \parallel Q) \triangleq cmp(P)|cmp(Q)$$

The process $P_{\sigma_1}|_{\sigma_2}Q$ is done when both P and Q are done and both subscripts are the same and coincide with the global state σ.

The rules C-PAR-L/C-PAR-R define just the ordinary interleaving of compensations. The rules PAR-L/PAR-R are analogous, but the subscript determines the

$$\frac{\Gamma \vdash C \xrightarrow{\lambda} C'}{\Gamma \vdash C|D \xrightarrow{\lambda} C'|D}\ (\text{C-PAR-L}) \qquad \frac{\Gamma \vdash \sigma_1, S \xrightarrow{\lambda} \sigma_1', S'}{\Gamma \vdash \sigma, S_{\sigma_1}|_{\sigma_2} T \xrightarrow{\lambda} \sigma_1' \sqcap \sigma_2, S'_{\sigma_1'}|_{\sigma_2} T}\ (\text{SPAR-L})$$

$$\frac{\Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2} Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{\sigma_1'}|_{\sigma_2} Q}\ (\text{PAR-L}) \qquad \frac{Q \rightsquigarrow Q'}{\Gamma \vdash \boxtimes, P_\sigma|_\boxdot Q \xrightarrow{\tau} \boxtimes, P_\sigma|_\boxtimes Q'}\ (\text{INT-R})$$

Fig. 7. LTS rules for parallel Sagas (symmetric rules omitted for brevity)

$$\frac{}{[C] \rightsquigarrow [C]} \quad \frac{}{A \div B \rightsquigarrow [\mathbf{nil}]} \quad \frac{P \rightsquigarrow P' \wedge \neg par(P)}{P; Q \rightsquigarrow P'} \quad \frac{par(P)}{P; Q \rightsquigarrow P} \quad \frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'|_{\boxtimes|\boxdot} Q} \quad \frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_{\boxdot}|_{\boxtimes} Q'}$$

$$\frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge tocmp(P')}{P\$C \rightsquigarrow [cmp(P'); C]} \quad \frac{P \rightsquigarrow P' \wedge \neg dn_{\boxtimes}(P')}{P\$C \rightsquigarrow P'\$C} \quad \frac{P \rightsquigarrow P' \wedge dn_{\boxtimes}(P') \wedge \neg tocmp(P')}{P\$C \rightsquigarrow [C]}$$

Fig. 8. Predicate $P \rightsquigarrow P'$ for interrupting a process

modality of execution. A thread can move forward when it is in a commit state. If a thread aborts, the failure is annotated also in the global state by taking $\sigma \sqcap \sigma_1'$. A commit thread can still move forward even if the global mode is abort.

The rules INT-L/INT-R use an "extract" predicate $P \rightsquigarrow P'$ to interrupt a commit thread if the global process is in abort mode. In $P \rightsquigarrow P'$, the process P' is a possible result of interrupting P (see Fig. 8). As a special case, note the interrupt of a sequential composition: we distinguish whether P is a parallel composition (predicate $par(P)$ is true) or not. This is motivated by the intention to adhere to the Petri net semantics, where $(P|Q); R$ can be interrupted discarding R but without necessarily interrupting P or Q.

For the parallel composition of sagas (SPAR-L/SPAR-R) we just remark that in the case of fault of one thread we let the other threads execute as much as possible and just record the global effect in the σ component of the state.

Example 3. Let $eS' \triangleq \mathsf{aO} \div \overline{\mathsf{aO}}; (\mathsf{pC} \div \overline{\mathsf{pC}} | \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})$, and assume that the processing of the card fails while the other actions are successful.

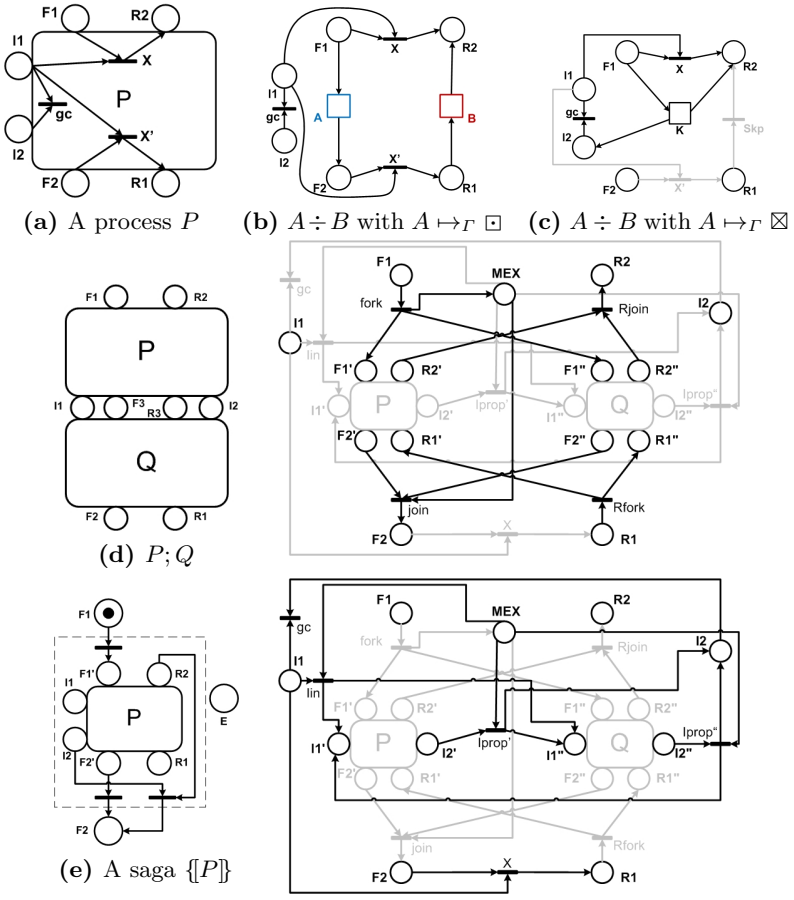
$$\begin{aligned} &\boxdot, eS' \xrightarrow{\mathsf{aO}} \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}} \mid \mathsf{pO} \div \overline{\mathsf{pO}}; \mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{aO}} \xrightarrow{\mathsf{pO}} \\ &\quad \boxdot, (\mathsf{pC} \div \overline{\mathsf{pC}} \mid (\mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pO}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \boxtimes, ([\mathbf{nil}]_{\boxtimes|\boxdot} (\mathsf{bC} \div \overline{\mathsf{bC}})\$\overline{\mathsf{pO}})\$\overline{\mathsf{aO}} \xrightarrow{\tau} \\ &\quad \boxtimes, [(\mathbf{nil} \mid \overline{\mathsf{pO}}); \overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{pO}}} \boxtimes, [\overline{\mathsf{aO}}] \xrightarrow{\overline{\mathsf{aO}}} \boxtimes, [\mathbf{nil}] \end{aligned}$$

5 Operational Correspondence

In this section we will show a weak bisimilarity between our novel LTS semantics and the Petri net semantics of [6].

5.1 Petri Net Semantics (for Policy #5)

In [6] Sagas processes are encoded in safe Petri nets by structural induction (see Fig. 9). A saga has just three places to interact with the environment: F_1 starts its flow, F_2 signals successful termination, and E raises a fault. Each compensable process has six places to interact with the environment: a token in F_1 triggers the forward flow, to end in F_2; a token in R_1 starts the compensation, to end in R_2; a token in I_1 indicates the arrival of an interrupt from the outside; a token in I_2 informs the environment that a fault occurred. For sagas, a computation starting in F_1 will lead either to F_2 or to E, while for compensable processes we expect to have the following kinds of computations:



(a) A process P **(b)** $A \div B$ with $A \mapsto_\Gamma \boxdot$ **(c)** $A \div B$ with $A \mapsto_\Gamma \boxtimes$

(d) $P;Q$

(e) A saga $\{\!|P|\!\}$

(f) Parallel composition $P|Q$

Fig. 9. Encoding of compensable processes as (safe) Petri nets

Successful (forward) computation: from marking F_1 the net reaches F_2
Compensating (backward) computation: from R_1 the net reaches R_2.
Aborted computation: from F_1 the net reaches $R_2 + I_2$.
Interrupted computation: from $F_1 + I_1$ the net reaches R_2.

The nets for compensable processes are depicted in Fig. 9. The encoding introduces several auxiliary transitions (thinner and black filled), e.g., to fork and join the control flow, to catch an interrupt and reverse the flow.

Depending on the context, for a successful compensation pair $A \div B$ (Fig. 9b) we have the obvious transitions modelling activities A and B together with auxiliary transitions for handling interruption. The net for a failing compensation pair (Fig. 9c) has a transition K that models the abort of the transaction. The net for the sequential composition $P; Q$ (Fig. 9d) is obtained by merging the forward output place F_3 of P with the forward input place of Q and the backward output place R_3 of Q with the backward input place of P. Moreover, P and Q share also the places for I_1 and I_2.

The encoding of parallel composition $P|Q$ (Fig. 9f) is more complex. We use two subnets for the two processes, with places $F'_1, F'_2, \ldots$ and $F''_1, F''_2, \ldots$ resp. The upper part of the figure highlights the transitions used in absence of interruptions and the lower part focuses on transitions exploited by interruption.

5.2 Weak Bisimilarity Result

In the following, we write $p \stackrel{\hat{\tau}}{\Rightarrow} q$ if $(p, q) \in (\stackrel{\tau}{\rightarrow})^*$. Moreover, for $\mu \neq \tau$ we write $p \stackrel{\hat{\mu}}{\Rightarrow} q$ if there exists p', q' such that $p' \stackrel{\mu}{\rightarrow} q'$ and $(p, p'), (q, q') \in (\stackrel{\tau}{\rightarrow})^*$.

Definition 3. *Let (S_1, L, T_1) and (S_2, L, T_2) be two LTSs. A relation $\mathbf{R} \subseteq S_1 \times S_2$ is a* weak bisimulation *if whenever $(s_1, s_2) \in \mathbf{R}$, then:*

1. *if $s_1 \stackrel{\mu}{\rightarrow}_1 s'_1$ then there exists s'_2 such that $s_2 \stackrel{\hat{\mu}}{\Rightarrow}_2 s'_2$ and $(s'_1, s'_2) \in \mathbf{R}$; and*
2. *if $s_2 \stackrel{\mu}{\rightarrow}_2 s'_2$ then there exists s'_1 such that $s_1 \stackrel{\hat{\mu}}{\Rightarrow}_1 s'_1$ and $(s'_1, s'_2) \in \mathbf{R}$.*

The largest weak bisimulation is called weak bisimilarity *and denoted by $\approx$.*

We shall let the marking graph of the net N_P play the role of (S_1, L, T_1) and (the fragment of) the LTS reachable from process P play the role of (S_2, L, T_2), so that $\approx$ relates markings of N_P with processes P' reachable from P. More precisely, we assume the only observable actions in the marking graph are those corresponding to activities $a \in \mathcal{A}$; all the other transitions are labelled with τ.

We have seen that the Petri net semantics associates to a compensable process P a corresponding net N_P that exchanges tokens with the context via six places. The places F_1, R_1, I_1 are used to receive tokens in input from the environment, while the places F_2, R_2, I_2 are used to output tokens to the environment. Nets are usually considered up-to isomorphism, therefore the names of their places and transitions are not important, as long as the same structure is maintained. However, to establish the behavioural correspondence between our LTS for P

and the marking graph of the net N_P we need to fix a particular naming of the elements in N_P. Moreover, the same activity can occur many times in a process and every instance corresponds to a different element of the net. One way to eliminate any ambiguity is to annotate processes with the names of the places to be used for building the interface of the corresponding net (before the translation takes place). The proof of the main theorem requires some ingenuity to fix the correspondence between net markings and process terms. Here, we just mention that we write $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$ meaning that process P (and all its sub-processes) has been annotated in such a way that the names of the places in the "public interface" of the net N_P are $F_1, F_2, R_1, R_2, I_1, I_2$.

Theorem 1. *Let N_P be the Petri net associated with the tagged compensable process $P@\langle F_1, F_2, R_1, R_2, I_1, I_2\rangle$. Then, $F_1 \approx (\boxdot, P)$.*

As an immediate consequence of the theorem and the main result of [6] the correspondence to the denotational semantics given in § 2 follows.

For any sagas S we let $(\!|S|\!)$ denote the set of *weak traces* in our LTS semantics:

$$(\!|S|\!) \triangleq \{\, a_1 ... a_n\langle\checkmark\rangle \mid \exists S_1, ..., S_n, \sigma_1, ..., \sigma_{n-1}.\ \boxdot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxdot, S_n \not\rightarrow \} \cup$$
$$\{\, a_1 ... a_n\langle!\rangle \mid \exists S_1, ..., S_n, \sigma_1, ..., \sigma_{n-1}.\ \boxdot, S \stackrel{\hat{a_1}}{\Rightarrow} \sigma_1, S_1 \stackrel{\hat{a_2}}{\Rightarrow} \cdots \stackrel{\hat{a_n}}{\Rightarrow} \boxtimes, S_n \not\rightarrow \}$$

Actually, under the assumption that compensation cannot fail, only successful traces are present in $(\!|S|\!)$ (as well as in $[\![S]\!]_i$ for any $i \in [1,5]$). This is not necessarily the case for the last extension in § 7.

Corollary 1. *For any sagas $S = \{\!|P|\!\}$ we have $[\![S]\!]_5 = (\!|S|\!)$. Moreover, if P is sequential then $[\![S]\!]_i = (\!|S|\!)$ for $i \in [1,5]$.*

6 Dealing with Other Compensation Policies

In this section we show that we can tune the LTS semantics to match and improve other compensation policies discussed in the literature.

Notification and Distributed Compensation. To remove the possibility to interrupt a sibling process before it ends its execution we just redefine the "extract" predicate by removing most cases, so that the interrupt is possible only when the process is "done".

$$\frac{}{[C] \rightsquigarrow [C]} \qquad \frac{P \rightsquigarrow P'}{P\$C \rightsquigarrow P'\$C} \qquad \frac{P \rightsquigarrow P'}{P|Q \rightsquigarrow P'_{\boxtimes}|_{\boxdot}Q} \qquad \frac{Q \rightsquigarrow Q'}{P|Q \rightsquigarrow P_{\boxdot}|_{\boxtimes}Q'}$$

Now, the rule INT is only applicable if the interrupted process consists of an installed compensation $[C]$. The new extract predicate only changes the subscripts, not the process: since any interrupted thread is "done" we never inhibit sibling forward activities upon a fault.

We call this strategy *Notification and distributed compensation* (policy #6) to emphasize the fact that siblings are notified about the fault, not really interrupted. Since compensations are distributed and the fault is not observable, it

can happen that a notified thread starts compensating even before the sibling that actually aborted. However, contrary to policy #2, a thread cannot guess the presence of faulty siblings, so it is not possible to observe a forward activity of the only faulty thread after a compensation activity of a notified thread. Thus policy #6 defines a variant of policy #2 where unrealistic traces are discarded.

Proposition 1. *Let $(\!|\cdot|\!)_6$ denote the set of weak traces generated by policy #6 above. Then, for any sagas $S = \{\![P]\!\}$ we have $[\![S]\!]_1 \subseteq (\!|S|\!)_6 \subseteq [\![S]\!]_2$. Moreover, for some P the inclusion is strict, while for sequential processes $[\![P]\!]_1 = (\!|P|\!)_6 = [\![P]\!]_2$.*

Interruption and Centralized Compensation. To move from distributed to centralized execution we simply strengthen the premise in PAR-L (and PAR-R):

$$\frac{(\sigma_1 = \boxdot \vee dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2} Q) \vee \neg dn_{\boxtimes}(P)) \qquad \Gamma \vdash \sigma_1, P \xrightarrow{\lambda} \sigma_1', P'}{\Gamma \vdash \sigma, P_{\sigma_1}|_{\sigma_2} Q \xrightarrow{\lambda} \sigma \sqcap \sigma_1', P'_{\sigma_1'}|_{\sigma_2} Q} \text{(PAR-L)}$$

Thus a process can only be executed if it is either moving forward ($\sigma_1 = \boxdot$) or the complete parallel composition finished in a failing case ($dn_{\boxtimes}(P_{\sigma_1}|_{\sigma_2} Q)$) or the thread has not yet finished its execution in a failing case ($\neg dn_{\boxtimes}(P)$).

Proposition 2. *Let $(\!|\cdot|\!)_3$ denote the set of weak traces generated by policy #3 above. Then, for any sagas $S = \{\![P]\!\}$ we have $[\![S]\!]_3 = (\!|S|\!)_3$.*

No Interruption and Centralized Compensation. By combining the above changes we recover policy #1.

7 Possible Extensions

Choice and Iteration. Our first extension adds choice $P + P$ and iteration P^* operators to the syntax for processes The corresponding rules are in Fig. 10. In a process $P + Q$ one option is nondeterministically executed while the alternative is dropped. For iteration, a process P^* either executes a τ and finishes or acts as the sequential composition $P; P^*$. Note that, while it is easy to account for choice and iteration in the denotational semantics, the extension is harder for the Petri net semantics. For example, let us consider the sequential process $(A \div A' + B \div B')^*; \mathit{throww}$. At any iteration, either A or B is executed and thus either A' or B' is installed. When the iteration is closed, the installed compensation may be any arbitrary sequence of A' or B', an information that cannot be recorded in the state of a finite (safe) Petri net.

Failing Compensations. One important contribution of [7] was the ability to account for the failure of compensations. Here we discuss how to extend our LTS semantics accordingly.

For compensations, we extend the states with $\Omega = \{\boxdot, \boxtimes\}$, modify the sources / targets from C to $\boxdot, C$ in the rules we have presented, change the rule C-ACT as

$$dn_\sigma(P+Q) \triangleq dn_\sigma(P) \wedge dn_\sigma(Q) \quad dn_\sigma(P^*) \triangleq dn_\sigma(P) \quad \frac{}{P+Q \rightsquigarrow [\mathbf{nil}]} \quad \frac{}{P^* \rightsquigarrow [\mathbf{nil}]}$$

$$\text{(CHOICE-L)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \sigma, P'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, P'} \qquad \text{(E-ITER)}\ \frac{}{\Gamma \vdash \boxdot, P^* \xrightarrow{\tau} \boxdot, [\mathbf{nil}]} \qquad \text{(S-ITER)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge \neg dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P'; P^*}$$

$$\text{(CHOICE-R)}\ \frac{\Gamma \vdash \boxdot, Q \xrightarrow{\lambda} \sigma, Q'}{\Gamma \vdash \boxdot, P+Q \xrightarrow{\lambda} \sigma, Q'} \qquad \text{(A-ITER)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxtimes, P'}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxtimes, P'} \qquad \text{(S-ITER2)}\ \frac{\Gamma \vdash \boxdot, P \xrightarrow{\lambda} \boxdot, P' \wedge dn(P')}{\Gamma \vdash \boxdot, P^* \xrightarrow{\lambda} \boxdot, P^*\$cmp(P')}$$

Fig. 10. LTS for choice and iteration

below and add the rules F-C-SEQ, C-PAR-L and C-PAR-R that record the execution of a faulty compensation in the target of the transition:

$$\text{(C-ACT)}\ \frac{A \mapsto_\Gamma \sigma}{\Gamma \vdash \boxdot, A \xrightarrow{A} \sigma, \mathbf{nil}} \qquad \text{(F-C-SEQ)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \boxdot, C; D \xrightarrow{\lambda} \boxtimes, C'} \qquad \text{(C-PAR-L)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \sigma, C'}{\Gamma \vdash \boxdot, C|D \xrightarrow{\lambda} \sigma, C'|D}$$

For compensable processes, we extend the state in LTS to $\Omega^{\circledast} = \{\boxdot, \boxtimes, \circledast\}$, where the symbol $\circledast$ denotes the fault of a compensation, i.e., a non recoverable crash. As a matter of notation for meta-variables, we let $\sigma, \ldots \in \Omega$ and $\delta \in \{\boxtimes, \circledast\}$. When executing a compensation $[C]$, we must take into account the possibility of a crash (COMP-1 and COMP-2). Moreover, if we generate a crash, previously installed local compensations will not be executed (C-STEP):

$$\text{(COMP-1)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxdot, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \delta, [C']} \qquad \text{(COMP-2)}\ \frac{\Gamma \vdash \boxdot, C \xrightarrow{\lambda} \boxtimes, C'}{\Gamma \vdash \delta, [C] \xrightarrow{\lambda} \circledast, [C']} \qquad \text{(C-STEP)}\ \frac{\Gamma \vdash \boxtimes, P \xrightarrow{\lambda} \circledast, P'}{\Gamma \vdash \boxtimes, P\$C \xrightarrow{\lambda} \circledast, P'}$$

(Note that in the premises of rules COMP-1 and COMP-2 we intentionally put $\boxdot$ in the source of the transition, because the LTS for compensations has only such states as sources of transitions.) The other rules for sequential Sagas stay as before. For parallel composition we redefine the predicate dn such that

$$dn_\boxdot(P_\boxdot|_\boxdot Q) \triangleq dn_\boxdot(P) \wedge dn_\boxdot(Q) \quad dn_\delta(P_{\delta_1}|_{\delta_2} Q) \triangleq dn_\delta(P) \wedge dn_\delta(Q)$$

where $\delta, \delta_1, \delta_2 \in \{\boxtimes, \circledast\}$. The rules PAR-L/PAR-R are as before however for any meta-variable we allow also $\circledast$ as a possible value, i.e., $\sigma, \sigma_1, \sigma_2, \sigma_1', \sigma_2' \in \Omega^{\circledast}$. Thus we have to extend the operation $\sqcap$ such that $\circledast \sqcap \sigma = \circledast$. The rules INT-L/INT-R are also applicable in a global $\circledast$ state.

The rules guarantee that in case of a crash parallel branches can execute their compensations as far as possible, only previously installed compensation (i.e., before the parallel composition) are not reachable anymore.

8 Concluding Remarks

We presented an LTS semantics for the Sagas calculus. Using a weak bisimulation we investigated the correspondence with previously defined Petri net and denotational semantics. Moreover, with small changes we can deploy a different policy for the execution of concurrent compensable processes. We have shown suitable semantics extensions enriching first the syntax and then the LTS itself.

This work is a first step towards a flexible tool for specifying and verifying LRTs. While previous semantic definitions for Sagas gave a formal model for LRTs, the LTS semantics is more suitable for custom property verification, like model-checking. The LTS has been implemented in Maude, a language based on rewriting logic and including tools like an inductive theorem prover or an LTL model-checker (`http://maude.cs.uiuc.edu`). In the end we would like to integrate the specified extensions as well as the option to choose which compensation policy should be used together with a high-level dynamic logic for validation and verification. Some promising steps in this direction are described in [5].

Related Work. One of the first attempt to a process algebraic formalization of LRTs is StAC [9], from which both Sagas [7] and cCSP [10] later originated. A small-step semantics for cCSP was defined in [11]. It relies on the centralized compensation policy, but is otherwise similar to our approach. Using a synchronizing step at the end of the forward flow the success or failure of the transaction is published, in case of a failure the compensations are executed as normal saga processes (outside the transaction scope). In our approach the information about a failure is kept in the state and compensations are executed inside the saga.

Compensation for a simple class of nets, called workflow nets, has been studied in [1]. It is simpler than the net semantics of [6] as it does not account for interruption after a fault, but it is less elegant because a compensated run may end with some remaining tokens. A *dynamic* policy for compensation is defined in [20], where the compensation of a concurrent process depends on the order of the interleaving of the forward actions, i.e., there is a unique compensation stack that is updated by each action. The above studies have been applied to provide formal support to standard technologies for web services [12,25,15,2] and to develop provably correct engines for transactional workflows [4,18,22,19].

Finally, we mention other approaches that focus on the interaction between processes. Notable examples are: webπ [23] and dcπ [26] that extend the π-calculus, cJoin [8] that extends the Join calculus, CommTrans [14] for CCS and the reversible process calculi in [13,24,21], where the special case of perfect rollback is investigated. We refer to [16] for some conceptual comparisons.

References

1. Acu, B., Reisig, W.: Compensation in workflow nets. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 65–83. Springer, Heidelberg (2006)
2. Bocchi, L., Guanciale, R., Strollo, D., Tuosto, E.: BPMN modelling of services with dynamically reconfigurable transactions. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 396–410. Springer, Heidelberg (2010)

3. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
4. Bruni, R., Ferrari, G., Melgratti, H., Montanari, U., Strollo, D., Tuosto, E.: From theory to practice in transactional composition of web services. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW 2005 and WS-FM 2005. LNCS, vol. 3670, pp. 272–286. Springer, Heidelberg (2005)
5. Bruni, R., Ferreira, C., Kauer, A.K.: First-order dynamic logic for compensable processes. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 104–121. Springer, Heidelberg (2012)
6. Bruni, R., Kersten, A., Lanese, I., Spagnolo, G.: A new strategy for distributed compensations with interruption in long-running transactions. In: Mossakowski, T., Kreowski, H.-J. (eds.) WADT 2010. LNCS, vol. 7137, pp. 42–60. Springer, Heidelberg (2012)
7. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL 2005, pp. 209–220. ACM (2005)
8. Bruni, R., Melgratti, H.C., Montanari, U.: Nested Commits for Mobile Calculi: Extending Join. In: Levy, J.-J., Mayr, E.W., Mitchell, J.C. (eds.) TCS 2004. IFIP, vol. 155, pp. 563–576. Springer, Boston (2004)
9. Butler, M., Ferreira, C.: A process compensation language. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 61–76. Springer, Heidelberg (2000)
10. Butler, M., Hoare, T., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
11. Butler, M., Ripon, S.: Executable Semantics for Compensating CSP. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW 2005 and WS-FM 2005. LNCS, vol. 3670, pp. 243–256. Springer, Heidelberg (2005)
12. Chen, Z., Liu, Z., Wang, J.: Failure-divergence refinement of compensating communicating processes. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 262–277. Springer, Heidelberg (2011)
13. Danos, V., Krivine, J., Sobocinski, P.: General reversibility. Electr. Notes Theor. Comput. Sci. 175(3), 75–86 (2007)
14. de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010)
15. Eisentraut, C., Spieler, D.: Fault, compensation and termination in WS-BPEL 2.0 — A comparative analysis. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 107–126. Springer, Heidelberg (2009)
16. Ferreira, C., Lanese, I., Ravara, A., Vieira, H.T., Zavattaro, G.: Advanced mechanisms for service combination and transactions. In: Wirsing, M., Hölzl, M. (eds.) Sensoria Project. LNCS, vol. 6582, pp. 302–325. Springer, Heidelberg (2011)
17. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD, pp. 249–259. ACM Press (1987)
18. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundam. Inform. 95(1), 73–102 (2009)
19. Johnsen, E.B., Lanese, I., Zavattaro, G.: Fault in the future. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 1–15. Springer, Heidelberg (2011)
20. Lanese, I.: Static vs dynamic SAGAs. In: ICE 2010. EPTCS, vol. 38, pp. 51–65 (2010)

21. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011)
22. Lanese, I., Zavattaro, G.: Programming sagas in SOCK. In: SEFM 2009, pp. 189–198. IEEE Computer Society (2009)
23. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
24. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. 73(1-2), 70–96 (2007)
25. Qiu, Z., Wang, S., Pu, G., Zhao, X.: Semantics of BPEL4WS-like fault and compensation handling. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 350–365. Springer, Heidelberg (2005)
26. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)

Linking Unlinkability

Mayla Brusó[1], Konstantinos Chatzikokolakis[2], Sandro Etalle[1,3], and Jerry den Hartog[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] CNRS & École Polytechnique, Paris, France
[3] University of Twente, Enschede, The Netherlands

Abstract. Unlinkability is a privacy property of crucial importance for several systems (such as RFID or voting systems). Informally, unlinkability states that, given two events/items in a system, an attacker is not able to infer whether they are related to each other. However, in the literature we find several definitions for this notion, which are apparently unrelated and shows a potentially problematic lack of agreement. This paper sheds new light on unlinkability by comparing different ways of defining it and showing that in many practical situations the various definitions coincide. It does so by (a) expressing in a unifying framework four definitions of unlinkability from the literature (b) demonstrating how these definitions are different yet related to each other and to their dual notion of "inseparability" and (c) by identifying conditions under which all these definitions become equivalent. We argue that the conditions are reasonable to expect in identification systems, and we prove that they hold for a generic class of protocols.

1 Introduction

Unlinkability is a privacy property which holds when an attacker cannot identify the link between two or more items in a system. This property is fundamental in the context of identification systems. For instance, a person who buys an item with an EPC tag at a shop may expect that the protocol used to identify tags prevents his/her tracking.

In this paper we use Radio Frequency Identification (RFID) systems as a case study for our protocol analysis. RFID systems are a wireless technology for automatic identification consisting of a set of tags, readers and a backend. Tags usually offer very limited resources, while backends and readers have standard computational resources. An identification protocol allows tags to authenticate to a backend exchanging information through a reader. One of the main issues raised by the widespread use of RFID is that of privacy. The problem is that anyone in the neighbourhood of a tag may access it wirelessly, and the resource limitation of RFID tags makes it difficult to use full-fledged cryptographic algorithms. The ease of access paves the way to misuse: an attacker could exploit tags to follow the movements of people or goods. The attacker does not even need to break anonymity, since a tag sending the expiry date of a product already allows a certain degree of tracking.

These privacy concerns lead to the definition of *unlinkability* (sometimes called *untraceability* or *privacy*). In the case of RFID systems, unlinkability [17,21,4,5,22,8,26] is satisfied if an attacker is not able to infer whether two sessions have been executed by the same agent. In the RFID literature, unlinkability is usually defined either in a computational setting in terms of games [10,17,21,4,22] or in a symbolic setting [12,13,2,3,7].

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 129–144, 2013.
© Springer-Verlag Berlin Heidelberg 2013

[12] proposes a definition of untraceability in a trace-based model. [2,3] formalize protocols in the applied pi calculus and define weak and strong unlinkability in terms of trace and observational equivalence respectively. Finally, [7] proposes a definition of unlinkability in the applied pi calculus, inspired by the unlinkability games of the computational setting.

As most definitions are different in model and strength, there is no agreement in the literature on the concept of unlinkability. The goal of this paper is to create a better understanding of this notion by comparing the strength of different definitions and determining whether the differences have a practical impact on real world systems. Our contribution is threefold. First, we express four trace-based definitions of unlinkability from the literature in a unifying model. We start with the one of weak unlinkability from [12,3]. By strengthening it, we obtain the definition of strong unlinkability from [3]. Then, we express two game-based notions [10,17,4,20,7], and we give a definition that capture them both. We also investigate *inseparability*, a notion dual to unlinkability, which requires that the attacker cannot infer that two messages are *not* linked. Second, we identify a set of conditions and demonstrate that, when they hold, all the above forms of unlinkability and inseparability coincide. Last, we prove that these conditions are satisfied by a generic class of simple identification protocols from [7].

These results help us to understand the essence of these privacy properties. Working in an abstract setting, we can concentrate on their inherent nature – the inability to distinguish certain traces – without dealing with the complications of a concrete model. As a result, the definitions and the conditions under which they coincide become intuitive, while the results can be transferred to a concrete trace model as we do in Section 6.

Plan of the paper. Section 2 briefly introduces epistemic logic. Section 3 presents our abstract trace model. Section 4 states several definitions of privacy using epistemic logic. Section 5 presents several conditions and shows that all the privacy properties coincide under them. Section 6 presents the class of RFID single-step protocols and shows that they satisfy all the conditions stated in Section 5. Section 7 lists the related work. Section 8 and provides conclusions.

2 Preliminaries

In this section we briefly introduce epistemic logic with public announcements, a logic modeling agent knowledge, that we later use to formalize privacy properties. Only the basic concepts are stated here, we refer the reader to [18] for more details.

Let P be a set of propositional constants (atoms). The set $\mathfrak{L}(P)$ of epistemic formulas $\varphi, \psi, \ldots$ over A is given by:

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid K\varphi \mid [\varphi]\psi$$

with $p \in P$. The formula $K\varphi$ means "the attacker knows φ", while $[\varphi]\psi$ means "after φ is revealed, ψ holds". The semantics is given in terms of Kripke structures. A Kripke structure M is a tuple $(S, f, \sim)$ where S is a set of possible states, $f : S \to 2^P$ is a function assigning to each state a set of atoms that hold in that state, and $\sim$ is an equivalence relation on S. Intuitively, $s_1 \sim s_2$ means that from the attacker's point of view, the two states are indistinguishable. The semantics of the logic is given by:

$M, s \vDash p$ iff $p \in f(s)$
$M, s \vDash \varphi \wedge \psi$ iff $M, s \vDash \varphi$ and $M, s \vDash \psi$
$M, s \vDash \varphi \vee \psi$ iff $M, s \vDash \varphi$ or $M, s \vDash \psi$
$M, s \vDash \neg\varphi$ iff $s \nvDash \varphi$
$M, s \vDash K\varphi$ iff $s' \vDash \varphi$ for all s' such that $s' \sim s$
$M, s \vDash [\varphi]\psi$ iff ($M, s \vDash \varphi$ implies $M|\varphi, s \vDash \psi$)

Intuitively, $M, s \vDash \varphi$ means that M satisfies φ at state s (we write $s \vDash \varphi$ when M is clear from the context). The interesting case is the knowledge operator K: the attacker knows φ at state s iff φ is satisfied in all states that are indistinguishable from s from the attacker's point of view. For the $[\varphi]$ operator, let $M|\varphi$ be a Kripke structure obtained by M by restricting only to states satisfying φ, i.e. having state space $S' = \{s \in S \mid M, s \vDash \varphi\}$. Intuitively, revealing φ (in a state where φ holds) restricts the model to a smaller one where φ always holds. Then $[\varphi]\psi$ is true if ψ holds in the restricted model.

3 A Trace-Based Model

In a system, agents exchange messages according to a protocol with a specific purpose. To capture one or more protocol runs in our model, we introduce the concept of *transactions*. A transaction starts when the attacker gains access to an agent and lasts until the attacker loses it. During the transaction the attacker can passively eavesdrop or actively forge messages. We allow the attacker to execute an arbitrary number of protocol sessions within a transaction, while knowing that the agent participating in the transaction does not change. However, when a new transaction starts, the agent involved can be either the same as before, or a different one, and the attacker's goal is to distinguish these two cases. The attacker's intentions are captured by the concept of a *strategy*. For example, the attacker might passively eavesdrop a session, then build a new message and send it to the agent executing another transaction, and so on. An attacker strategy and a mapping of transactions to agent identities, completely defines a trace.

Definition 1. *A system is a tuple* $(A, \Sigma, \mathrm{T}, \sim)$ *where:*

- $A = \{a_1, a_2, \ldots\}$ *is a (possibly infinite) set of agents; we assume an ordered set of transactions* $\{p_1, p_2, \ldots\}$*, and we denote by* $\Pi_n = \{p_1, \ldots, p_n\} \to A$ *the set of assignments of* n *transactions to agents;*
- Σ *is a set of strategies; each strategy* $\sigma \in \Sigma$ *has a length* $|\sigma|$*; we denote the set of transactions involved in the strategy by* $Dom_\sigma = \{p_1, \ldots, p_{|\sigma|}\}$*;*
- $\mathrm{T} = \{(\pi, \sigma) \mid \sigma \in \Sigma, \pi \in \Pi_{|\sigma|}\}$ *is a set of traces; each trace* $\tau \in \mathrm{T}$ *is a tuple* (π, σ) *where* σ *is a strategy and* π *is a mapping of transactions to agents;*
- $\sim$ *is an equivalence relation on* T *such that* $(\pi_1, \sigma_1) \sim (\pi_2, \sigma_2) \Rightarrow \sigma_1 = \sigma_2$.

A trace $\tau \in \mathrm{T}$ is a complete execution of the system, and is determined by a strategy σ, chosen by the attacker, and a mapping π, which the attacker does not control, that defines which agent participates in each transaction. A protocol is an abstract object that describes the behaviour of the agents in a system. Each protocol generates the set T of all the possible traces that can be obtained under any attacker strategy $\sigma \in \Sigma$. The relation $\sim$ is crucial for defining privacy properties. Consider two traces $\tau_1 = (\pi_1, \sigma)$

Table 1. Unlinkability and Inseparability

Property	*Epistemic formula*	*The attacker cannot infer*
WU	$\neg Klink(p, p')$	that two messages p, p' are linked
SU	$\neg K(anyLink)$	the existence of linked messages
GB-1	$[\pi_a \vee \pi_{a'}]\neg K\pi_a$	the mapping π_a even when $\pi_a \vee \pi_{a'}$ is revealed
GB-2	$[\pi_{a,a} \vee \pi_{a_1,a_2}]\neg K\pi_{a,a}$	the mapping $\pi_{a,a}$ even when $\pi_{a,a} \vee \pi_{a_1,a_2}$ is revealed
WI	$\neg K(unlink(p, p'))$	that two messages p, p' are unlinked
SI	$\neg KanyUnlink$	the existence of unlinked messages

and $\tau_2 = (\pi_2, \sigma)$, produced when the attacker chooses the strategy σ and interacts with two different sets of agents. If $\tau_1 \sim \tau_2$, it means that when using the strategy σ, the attacker cannot distinguish between the sets of the agents.

We sometimes use π_τ to emphasize that it belongs to the trace τ. For a trace $\tau = (\pi, \sigma)$, we write Dom_τ for Dom_σ, $|\tau|$ for $|\sigma|$ and A_τ, A_π for the image of π (i.e. the set of agents involved in the trace). For a mapping $\pi \in \Pi_n$ we define $|\pi| = n$ and we denote $\Pi = \cup_{n\geq 1}\Pi_n$. Since transactions are ordered, we write mappings as sequences of agents, e.g. $\pi = (a_4, a_1, a_2)$ assigns agent a_4 to the first transaction, a_1 to the second and a_2 to the third ones. We extend $\sim$ to mappings as follows:

$$\pi \sim \pi' \quad \text{iff} \quad |\pi| = |\pi'| \text{ and } (\pi, \sigma) \sim (\pi', \sigma)\ \forall \sigma \in \Sigma \text{ s.t. } |\sigma| = |\pi|$$

Note that we keep our model abstract and do not explicitly define the messages in the protocol, the exact strategies Σ and the relation $\sim$. We assume that these are produced by a concrete protocol model (such as the one given in Section 6).

4 Unlinkability Definitions

In this section we express several definitions of unlinkability from the literature in our trace-based model: the weak unlinkability of [12,3], the strong unlinkability of [3], and two game-based definitions [10,17,4,20,7]. Finally, we introduce the notion of inseparability, which does not appear in the literature, but arises as a natural dual notion to unlinkability. Table 1 summarizes all the resulting notions. Note that our purpose is not a technical comparison between definitions expressed in different models, but a comparison between the ideas behind each definition, expressed in a common unifying model.

4.1 Kripke Structure

To express unlinkability using epistemic logic, first we have to define a Kripke structure $M = (\mathrm{T}, f, \sim)$ corresponding to a system $(A, \Sigma, \mathrm{T}, \sim)$. T is the set of states and the

attacker's indistinguishability relation $\sim$ is provided directly by the system. The set of atomic propositions P and the assignment function $f : \mathrm{T} \to P$ are built as follows:

$$P = \Pi \cup \{link(p, p') \mid p, p' \in Dom_\tau, p \neq p', \tau \in \mathrm{T}\}$$
$$f((\pi, \sigma)) = \{\pi\} \cup \{link(p, p') \mid \pi(p) = \pi(p')\}$$

We use two types of propositions: $\pi \in \Pi$ denotes that the mapping of the trace is π, while $link(p, p')$ denotes that the transactions p, p' are linked, i.e. they are mapped to the same agent in a given trace. Note that $link(p, p')$ holds iff $p \neq p'$, which we implicitly assume in all the definitions.

4.2 Weak Unlinkability

The first definition is the one of weak unlinkability of [12,3]. Although presented in different models, they are similar in nature, both requiring that, given a trace where two messages are linked, an equivalent trace must exist where those messages are unlinked.

Definition 2 (Weak unlinkability). *A protocol generating the set of traces* T *guarantees weak unlinkability iff*

$$\forall \tau \in \mathrm{T}, p, p' \in Dom_\tau : \tau \vDash \neg K(link(p, p'))$$

This definition imposes that the attacker does not know whether any two given transactions are linked to each other. This implies that for all traces τ and all pairs of distinct transactions, there must exist an equivalent trace $\tau' \sim \tau$ in which the transactions are mapped to two different agents. So the above definition can be written as:

$$\forall \tau \in \mathrm{T}, p, p' \in Dom_\tau : \exists \tau' \in \mathrm{T}, \tau' \sim \tau : \tau' \vDash \neg link(p, p')$$

which corresponds exactly to the one of [12,3]. The weakness of this definition lies in the fact that it does not completely prevent the attacker from obtaining knowledge about linked transactions. For example, in a system satisfying weak unlinkability, the attacker could still know that p_1 is linked to either p_2 or p_3, but without knowing which one.

4.3 Strong Unlinkability

[3] also defines a strong version of unlinkability by requiring that a system is equivalent to one where each agent executes one session only. Their definition, in a simplified form and without entering into details, requires that $!\,T \approx !\,T_s$ where $T = \nu m.\ init.\ !main$ and $T_s = \nu m.\ init.\ main.\ T$ represents an agent executing an initialization phase ($init$) and an unbounded number (denoted by !) of protocol sessions ($main$), while T_s is an agent executing one session. $\approx$ denotes observational equivalence in [3], while we use trace equivalence here because it is directly expressable in our model. To capture this definition in our framework, we not only require that the attacker is not able to infer the link between two given transactions, but also the *existence* of linked transactions.

Definition 3 (Strong unlinkability). *We say that a protocol generating the set of traces* T *guarantees strong unlinkability iff*

$$\forall \tau \in \mathrm{T} : \tau \vDash \neg K(anyLink) \forall p, p' \in Dom_\tau$$

where $anyLink = \bigvee_p \bigvee_{p'} link(p, p')$.

$anyLink$ holds for a trace if there exists at least a linked transaction. Thus, strong unlinkability holds iff the attacker does not know whether there exists a link at all. For this to hold, each trace must be equivalent to one with no linked transactions:

$$\forall \tau \in \mathrm{T} : \exists \tau' \in \mathrm{T}, \tau' \sim \tau : \forall p, p' \in Dom_\tau : \tau' \vDash \neg link(p, p')$$

This formulation corresponds exactly to the definition of Arapinis et al. since τ' is a trace that can be produced by the process $!T_s$, where no agent executes more than one transaction.

4.4 Game-Based Definitions of Privacy

In the game-based definitions, privacy is defined as the result of a game between an attacker (whose goal is to distinguish between the actions of different agents) and a challenger. We refer to two different types of game-based definition of privacy: the first is related to the definition of [21], variations of which can be found also in [10,17,4,22], while the second corresponds to the definition given by [10,20]. We demonstrate that in our model, these two classes of definitions are equivalent to each other and to a third *simpler* definition of game-base unlinkability based on trace equivalence.

Both types of games consist of three phases. In the first game, which we call two-agents game unlinkability, during the first phase the attacker can interact with all the agents of the system. In the second phase, the attacker is asked to select two agents a, a'. The challenger selects an agent $x \in \{a, a'\}$, and gives x to the attacker, hiding its identity. The attacker can interact with all the agents, including a and a', and, in the final phase, she wins the game if she can infer whether x is a or a' with non-negligible probability. We use π_x to denote a partial mapping from transactions to agents, where some transactions are mapped to a variable x, while all the others are known to the attacker; Π_x is the set of all the partial mappings. π_a is a mapping obtained from π_x by mapping to an agent a all transactions previously mapped to the variable x. We formalize this game by requiring that the attacker cannot infer whether she is given a mapping π_a or $\pi_{a'}$. Thus, a protocol generating the set of traces T guarantees *two-agents game unlinkability* iff

$$\forall \tau \in \mathrm{T}, a, a' \in A, \pi_x \in \Pi_x : \tau \vDash [\pi_a \vee \pi_{a'}] \neg K \pi_a \tag{1}$$

Although the only forbidden knowledge concerns π_a, (1) is equivalent to $\tau \vDash [\pi_a \vee \pi_{a'}] \neg K \pi_{a'}$, thus the attacker cannot know $\pi_{a'}$ either. This property implies the equivalence of the mappings π_a and $\pi_{a'}$ under all strategies, thus we can express (1) as:

$$\forall a, a' \in A, \pi_x \in \Pi_x : \pi_a \sim \pi_{a'}$$

The second game, which we call three-agents game unlinkability, can be found in a computational [10,20] and a formal setting [7]. Only the second phase differs from the

previous game: the attacker selects three agents a, a_1, a_2 and the challenger gives her two agents x, y, that are set to either $x = y = a$ or $x = a_1, y = a_2$; the attacker wins if she can infer whether x and y are linked. We now use $\pi_{x,y}$ as a partial mapping, $\Pi_{x,y}$ as a set of all the partial mappings and $\pi_{a,b}$ as a complete mapping obtained from $\pi_{x,y}$. We require that the attacker cannot infer whether she is given a mapping $\pi_{a,a}$ or π_{a_1,a_2}. A protocol generating the set of traces T guarantees three-agents game unlinkability iff

$$\forall \tau \in \mathrm{T}, a, a_1, a_2 \in A, \pi_{x,y} \in \Pi_{x,y} : \tau \vDash [\pi_{a,a} \vee \pi_{a_1,a_2}] \neg K \pi_{a,a} \tag{2}$$

In terms of equivalence of traces, (2) can be restated as follows:

$$\forall \tau \in \mathrm{T}, a, a_1, a_2 \in A, \pi_{x,y} \in \Pi_{x,y} : \pi_{a,a} \sim \pi_{a_1,a_2}$$

It is easy to see that both the games require all mappings to be equivalent. Thus, we give a definition of game-based unlinkability which unifies these two notions.

Definition 4 (Game-based unlinkability). *We say that a protocol generating the set of traces* T *guarantees game-based unlinkability iff*

$$\forall \pi, \pi' \in \Pi, |\pi| = |\pi'| : \pi \sim \pi' \tag{3}$$

Each of the referenced works uses a variant of either (1) or (2), while [10] mentions both, referring to the first as untraceability and to the second as unlinkability, but does not explore the relation between them. Instead, we can prove that they reduce to Def. 4:

Theorem 1. *A protocol satisfies game-based unlinkability iff it satisfies two-agents game unlinkability, which it does iff it satisfies three-agents game unlinkability.*

From now on we will only use the definition of game-based unlinkability. However, by Theorem 1, all the results that hold for game-based unlinkability hold also for the other game-based notions. Finally, as one may expect, we can show that strong unlinkability and game-based unlinkability are both stronger than weak unlinkability.

Theorem 2. *Strong unlinkability and game-based unlinkability imply both weak unlinkability.*

Note that strong unlinkability has already been proven to imply weak unlinkability by [3] in their model in the applied pi calculus.

4.5 Inseparability

In some situations we want to hide the existence of *unlinked* transactions. In fact, an attacker might be interested in changes in the system rather than in tracking agents. For example, consider a high security location protected by a guard who authenticates himself using an RFID tag. The attacker might want to be alerted when a *new* guard appears. The definitions of weak and strong unlinkability impose no condition when two messages are unlinked, thus we introduce the concept of *inseparability*, which requires that the attacker does not know that two transactions are *not* linked.

Definition 5 (Inseparability). *We say that a protocol generating the set of traces* T *guarantees weak inseparability iff* $\forall \tau \in \mathrm{T}, p, p' \in Dom_\tau : \tau \vDash \neg K(unlink(p, p'))$ *and strong inseparability iff* $\forall \tau \in \mathrm{T} : \tau \vDash \neg K(anyUnlink)$ *where* $unlink(p, p') = \neg link(p, p')$ *and* $anyUnlink = \bigvee_p \bigvee_{p' \neq p} unlink(p, p')$.

As expected, strong inseparability is stronger than weak inseparability. On the other hand, somewhat surprisingly, game-based unlinkability is stronger than strong inseparability, although game-based unlinkability is incomparable to strong unlinkability, which is incomparable to strong inseparability. The reason is that game-based unlinkability enforces all traces to be equivalent to one performed by a single agent.

Theorem 3. *Game-based unlinkability implies strong inseparability, which in turn implies weak inseparability.*

The above properties are in general different (the implications not covered here do not hold in general), as shown by the following examples; in the next section we investigate conditions under which some or all of the properties become equivalent.

4.6 RFID Systems: Protocols Where the Properties Do Not Coincide

In this section we list some examples of RFID protocols that guarantee only some of the properties described in Section 4. They are variations of the OSK protocol (see Section 6, although understanding the protocol is not needed to follow the examples) that satisfies all the properties. Here we introduce features that cause some properties to fail. These features are artificial and unlikely to be present in realistic protocols. In fact, in the next section, we identify some conditions under which all properties coincide.

Example 1 (System with a bounded number of tags). Consider a system with a bounded number of tags. If the attacker observes a number of sessions greater than the number of tags, she knows that there exist some linked sessions, although she cannot point to any specific message, i.e. weak unlinkability holds, but strong unlinkability does not. Still, all the traces of equal length produced by the OSK protocol are equivalent, thus game-based unlinkability holds. As a consequence, also strong inseparability holds. □

Example 2 (System with several "types" of tags). Consider a system with two distinguishable types of tags $Type_1$ and $Type_2$, for example because they differ in technical characteristics. Weak inseparability and game-based unlinkability are violated since the attacker can distinguish tags of different type. Instead, strong unlinkability holds: the adversary cannot know the existence of linked transactions since all transactions of the same type could come from different tags. Together with the previous example, this shows that strong unlinkability and the game-based definition are incomparable. However, if the number of tags of $Type_2$ is bounded, we have a situation similar to the previous example (although the total number of tags might still be unbounded); again, strong unlinkability is violated while weak unlinkability holds. □

Example 3 (Protocol outputs depending on past sessions I). Consider a variation of the OSK protocol where the reader beeps when the session it is executing is linked to a previous session, but only if at least two tags appeared before it. This protocol satisfies weak unlinkability: the beep does not allow the attacker to point to any two specific

linked sessions. Despite this, the attacker knows that the session that made the reader beep must be linked to a past session. Consider the observation where the reader beeps at the third session. The beep tells him that the third session is either linked to the first or the second one, and the first two sessions are not linked, violating strong unlinkability and weak inseparability. Since not all mappings are equivalent to each other due to the beep, game-based unlinkability is also violated. □

Example 4 (Protocol outputs depending on past sessions II). Consider a variation of the OSK protocol where the reader beeps when the third tag of a trace first appears. The protocol satisfies weak unlinkability, but violates strong unlinkability: if the reader beeps after four or more sessions, there must be a link. Game-based unlinkability is violated, since not all the traces are equivalent. It also breaks weak inseparability, since a beep during the third session means that the first three sessions are unlinked. □

Example 5 (Protocol outputs depending on past sessions III). Consider a variation of the OSK protocol where the reader beeps when it sees at least two tags and one link. The protocol satisfies weak unlinkability but violates strong unlinkability and game-based unlinkability. Strong inseparability is also violated, because a beep means that at least two tags run some session. However, the attacker cannot link two specific sessions, thus weak inseparability holds. □

5 Conditions under Which the Properties Coincide

In this section we identify a (large) class of protocols C and we demonstrate for these protocols that all the notions of unlinkability and inseparability are equivalent: if a protocol in C satisfies any of them, then it satisfies all of them. The class C is given by all the protocols satisfying the five conditions that we identify in the next section, where we argue that most RFID protocols actually satisfy them, at least in their abstract form.

5.1 Conditions

Condition Unbounded Number of Agents. As we showed in Example 1, a system with a bounded number of agents cannot satisfy strong unlinkability, since observing a greater number of transactions reveals that at least two transactions are linked. Thus, protocols in C contain an unbounded number of agents.

Definition 6. *A protocol has an unbounded number of agents iff* $\forall n > 0 \exists \tau \in \mathrm{T} : |A_\tau| = n$.

Clearly, an unbounded number of agents is not realistic. However, identification systems have usually a wide number of agents, thus an attacker cannot usually communicate with all of them in a limited amount of time, and does not know the total number of agents. This is why, at an abstract level, this condition is often assumed in the literature.

Condition Renaming. As shown in Example 2, having distinguishable types of agents can be problematic. However, in real systems agents are usually identical in functionality, differing only in the secret information. As a result, we can expect that replacing *all* transactions of an agent with a new one will not have a visible effect.

Definition 7. *Let π be a mapping, $a \in A_\pi$ and $a' \notin A_\pi$. The renaming of a to a' in π, denoted by $\pi[a'/a]$, is a mapping such that*

$$\pi[a'/a](p) = \begin{cases} a' & \text{if } \pi(p) = a \\ \pi(p) & \text{otherwise} \end{cases}$$

A protocol satisfies the condition Renaming iff $\pi \sim \pi[a'/a] \forall \pi \in \Pi, a' \notin A_\pi$.

In other words, only the positions in which an agent appears in the trace matters. In the rest of the paper, we assume that this condition holds and we write all mappings in a normalized form, sorting the agents by their order of appearance: the first agent is always a_1, the next agent different from a_1 will be a_2 and so on. For example, we write $(a_1, a_1, a_2, a_3, a_1, a_2)$ instead of $(a_5, a_5, a_3, a_1, a_5, a_3)$. Thus, the number of possible traces is always finite for any given length, even when the number of agents is infinite.

Condition Swapping. Consider $\pi_1 = (\ldots, a_1, a_2, \ldots)$ and $\pi_2 = (\ldots, a_3, a_4, \ldots)$, two mappings where the k-th and $k+1$-st transactions involve different agents. Now assume that $\pi_1 \sim \pi_2$ and consider the mappings $\pi_1' = (\ldots, a_2, a_1, \ldots)$ and $\pi_2' = (\ldots, a_4, a_3, \ldots)$ that differ from π_1 and π_2 only for the k-th and $k+1$-st agents. We require that different agents act in an independent way and the execution of the one should not affect the execution of the other. Thus, π_1' and π_2' should also be indistinguishable.

Definition 8. *Let π be a mapping. The swapping of π at position $k < |\pi|$, denoted by $sw_k(\pi)$ is a new mapping such that:*

$$sw_k(\pi)(p_i) = \begin{cases} \pi(p_{k+1}) & \text{if } p_i = p_k \\ \pi(p_k) & \text{if } p_i = p_{k+1} \\ \pi(p_i) & \text{otherwise} \end{cases}$$

A protocol satisfies the condition Swapping iff $\pi \sim \pi' \Rightarrow sw_k(\pi) \sim sw_k(\pi')$ for all π, π', k such that $\pi(p_k) \neq \pi(p_{k+1})$ and $\pi'(p_k) \neq \pi'(p_{k+1})$.

In practice, the condition Swapping simply implies that the agents are independent of each other. As a consequence, the order of appearance of agents does not affect the knowledge of the attacker. Note that this condition is violated in the Example 3. The mappings (a_1, a_1, a_2) and (a_1, a_2, a_3) produce the same observations $(\_, \_, \_)$. However, by swapping the second and third transactions, we obtain the mappings (a_1, a_2, a_1) and (a_1, a_2, a_3), which produce different observations: $(\_, \_, beep)$ and $(\_, \_, \_)$. This happens because the execution of one agent depends on the previous executions of other agents.

Conditions: Extension I and II. We now introduce two last conditions. The first states that two equivalent mappings should preserve their equivalence when extended with a new transaction mapped to a new agent. The underlying idea is that adding a new agent should not make two traces distinguishable, if they could not be distinguished before.

Definition 9. *Let π be a mapping. The extension of π with a new agent $a \notin A_\pi$, denoted by $extn(\pi)$, is a mapping of length $|\pi| + 1$ such that*

$$extn(\pi)(p_i) = \begin{cases} \pi(p_i) & i \leq |\pi| \\ a & i = |\pi| + 1 \end{cases}$$

A protocol satisfies the condition Extension I if $\pi \sim \pi' \Rightarrow extn(\pi) \sim extn(\pi')$ for all mappings π, π'.

Note that this condition is violated by the protocol in the Example 4. The mappings (a_1, a_1, a_1), (a_1, a_1, a_2) are equivalent (produce no beep), while their extensions are not, since (a_1, a_1, a_1, a_2) does not make the reader beep while (a_1, a_1, a_2, a_3) does.

For the second extension condition, consider two equivalent mappings π_1, π_2 of length n. We extend these mappings with a new transaction p_{n+1}, which is mapped to the last agent appearing in each mapping. Since the attacker had access to these agents during the transaction p_n, and she could not distinguish the two mappings, she does not gain any new knowledge by querying the same agents in the new transactions.

Definition 10. *Let π be a mapping. The extension of π with the last appearing agent, denoted by $extl(\pi)$ is a mapping of length $|\pi| + 1$ such that*

$$extl(\pi)(p_i) = \begin{cases} \pi(p_i) & i \leq |\pi| \\ \pi(p_{|\pi|}) & i = |\pi| + 1 \end{cases}$$

A protocol satisfies the condition Extension II if $\pi \sim \pi' \Rightarrow extl(\pi) \sim extl(\pi')$ for all mappings π, π'.

This condition is violated by the protocol in the Example 5. In fact, the traces with mappings (a_1, a_1) and (a_1, a_2) are not distinguishable, while their extensions are distinguishable because the first trace produces no beep while the second beeps.

5.2 Equivalence Results

In this section we demonstrate that under the conditions stated in Section 5.1, all the definitions of unlinkability coincide. Moreover, we prove that, under a smaller set of conditions, the definitions of inseparability coincide as well as all the strong definitions.

Theorem 4 (Unification of unlinkability). *If a protocol guarantees all the conditions of Section 5.1 then all the unlinkability properties (weak unlinkability, strong unlinkability, game-based unlinkability) coincide.*

The intuition is that, under these conditions, all the definitions require all the equal length mappings to be equivalent in particular to a mapping where all the transactions are not linked.

Theorem 5 (Unification of inseparability). *Under the conditions Renaming and Extension II, a protocol satisfies weak inseparability iff it satisfies strong inseparability.*

Again, under these conditions, both properties require all the mappings of the same length to be equivalent, in particular to one where all the transactions are linked.

Theorem 6 (Unification of strong properties). *Under the conditions Unbounded number of agents and Renaming, all the strong properties (strong unlinkability, game-based unlinkability, strong inseparability) coincide.*

It is worth noting that this result uses weaker assumptions than the previous ones; indeed, the conditions Swapping and Extension I and II are not needed. An unbounded number of agents is required to guarantee the existence of completely unlinked traces (for strong unlinkability), while the condition Renaming implies that agents are not observationally different (for strong inseparability and game-based unlinkability).

Finally, we can state the result we were aiming at.

Corollary 1. *If a protocol guarantees all the conditions of Section 5.1 then all the forms of unlinkability and inseparability coincide.*

This result shows that all the privacy definitions of Section 4 coincide under a set of conditions.

6 RFID Systems: Single-Step Protocols

In this section, we show that the conditions stated in the previous section are satisfied by a generic class of "single-step" protocols [7]. To this end, we express such protocols in the applied pi calculus, using the model of [3], which defines an instantiation of our model, i.e. it provides a concrete set of traces and an equivalence relation between them.

Single-step protocols consist of a single message from a tag to a reader. A protocol of this class is shown in Figure 1. Each tag is initialized with an internal state S_0, which contains a secret s that is unique to that tag. At each run of the protocol, the tag computes an output function $O(S)$ on its current state S, and sends the result to the reader. Then, the tag computes an update function $U(S)$ on its current state S. O and U can be arbitrary functions, using any cryptographic operation that can be modelled by an equational theory in the applied pi calculus. As discussed in [7], two well-known protocols from the literature, namely the OSK protocol [21] and the basic hash protocol of [27], fall in this class.

6.1 Modelling Single-Step Protocols

We model single-step protocols in the applied pi calculus [1], a language for describing concurrent processes and their interaction. It extends the pi calculus [19] adding the possibility to model cryptographic primitives using a signature and an equational theory. A detailed description of the calculus is available in [1]; here, we only assume a basic understanding of the calculus.

Tags are modelled as processes in the calculus, using a public channel c to communicate with the reader. To model the state of a tag, we use an internal channel w. The content of the state is available to the tag by a sub-process $\overline{w}\langle S\rangle$ running in parallel to

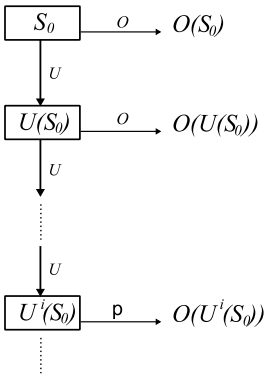


Fig. 1. A single-step protocol

it, so the tag can read the state by an input on w. A tag execution can be modelled as follows:

$$TagExec \triangleq c(\_).\ w(x).\ \nu\tilde{\rho}.\ \overline{c}\langle O(x)\rangle.\ \overline{w}\langle U(x)\rangle$$

The tag is first triggered by an input on the public channel c (without reading a value). Then, it reads the current state x by an input on w and outputs $O(x)$ on a public channel. $\nu\tilde{\rho}$ denotes the possibility of generating fresh nonces for the output. Finally, the tag outputs $U(x)$ on w, updating its state with the new value.

A complete tag starts with an initial state S_0, containing the tag secret s, and can execute an unbounded number of sessions.

$$Tag \triangleq \nu s.\nu w.(\overline{w}\langle S_0\rangle \mid !TagExec)$$

Note that the secret is private to the tag, thus we use a freshly generated name s. We also restrict w so that only the tag can access its state. Finally, the complete system P consists of an unbounded number of tags: $P \triangleq !Tag$. Here the reader only triggers tags. Since c is public, the tag can be triggered by any external process, so we can simply omit the reader altogether.

6.2 Instantiating Our Trace Model

The system P can perform labelled transitions, according to the semantics of the applied pi calculus. We denote by $\stackrel{\alpha}{\Longrightarrow}$ a sequence of internal transitions, followed by the visible transition α, followed again by internal transitions. A trace tr is a sequence

$$tr = P \stackrel{\alpha_1}{\Longrightarrow} P_1 \stackrel{\alpha_2}{\Longrightarrow} P_2 \stackrel{\alpha_3}{\Longrightarrow} \ldots \stackrel{\alpha_q}{\Longrightarrow} P_q$$

Two traces are equivalent, denoted by $tr_1 \sim_{tr} tr_2$ if they contain the same transitions, and all intermediate processes are *statically* equivalent according to the definition of [1], which states that the processes provide the attacker with the same static knowledge. We refer to [3] for the formal definition of $\sim_{tr}$.

To instantiate our trace model, we need do define a set of agents A, a set of strategies Σ such that a strategy σ together with a mapping π give rise to a trace $\tau = (\pi, \sigma)$, and an equivalence relation $\sim$ between traces. The agents $A = \{a_i | i \in \mathbb{N}\}$ correspond to the tags of the system. In the applied pi calculus model, we use replication to denote an unbounded number of tags. We identify the tags by their secret s, which is unique for each tag. When a_i is spawned we denote its secret by s_i.

Since tags in single-step protocols have no input, the only thing that the attacker can decide is how many transactions she will run, and how many protocol executions she will trigger in each transaction. Thus, a strategy σ is a sequence $\sigma = (\sigma_1, \ldots, \sigma_k)$ such that k is the number of transactions the attacker performs, and σ_i the number of executions that she triggers in the transaction p_i. A mapping π determines which tag will participate in each transaction. Given a strategy σ and a mapping π, we can define a unique trace $tr(\pi, \sigma)$ starting from P. We also define trace equivalence as follows:

$$(\pi, \sigma) \sim (\pi, \sigma') \quad \text{iff} \quad tr(\pi, \sigma) \sim_{tr} tr(\pi, \sigma')$$

Now that we have a concrete trace model for single-step protocols, we can show that they satisfy all the conditions of Section 5.1.

Theorem 7. *Single-step protocols satisfy all the conditions of Section 5.1, thus all the unlinkability and inseparability properties coincide.*

We can conclude that all the forms of unlinkability and inseparability defined in Section 4 coincide for the class of single-step protocols. As a consequence, if any of these properties is proven to hold for a single-step protocol, by Theorem 7 all the unlinkability and inseparability properties should be satisfied.

7 Related Work

Our work makes direct use of several definitions of unlinkability from the literature. As explained in detail in Section 4, we express the notion of weak unlinkability of [12,3], strong unlinkability of [2,3], and game-based definitions of [10,17,21,4,22,20]. While all these works have given their own definitions of privacy properties in a very specific context, ours provides a more general and abstract framework where all other definitions can be captured.

Epistemic models have been used in the past to formalize privacy. Similarly to our work, [15] gives general privacy definitions for a multiagent system using a modal logic of knowledge. The paper considers different levels of strength for unlinkability, providing some probabilistic definition as well. In [9], epistemic logic is used to give intuitive definitions of privacy in voting systems, with the applied pi calculus as the underlying model. Similarly, [11] proposes a framework in which protocols are expressed in a process language while security properties in a logic with both temporal and epistemic operators. The properties considered in the above works are quite different than the unlinkability properties that we consider in this paper. Moreover, the above works are involved with the mechanics of the corresponding formalisms, while we try to completely abstract from the concrete model, viewing a system as an abstract set of traces.

Several other definitions of unlinkability have also been studied in the literature. A logic approach has been followed in [25], where an axiom system is defined to reason about anonymity, and in [16] that expresses privacy properties using logic and models the system through other formalisms, like CSP, combining two different techniques. As in our work, logic is used to define in a natural way privacy properties, while having an abstract model applicable to any real system. However, while our work focuses on unlinkability, [15] and [25] study anonymity, namely a property that ensures that the identity of the agent which executes some action remains hidden from other observers. [23] proposes a terminology for anonymity, unlinkability, unobservability and pseudonimity. While it aims at clarifying terminology at an informal level, our work aims instead at comparing definitions of unlinkability in a unifying formal model.

Finally, other papers introduce the notion of unlinkability using approaches based on information theory. Examples are [14], [24], and [6] that give probabilistic descriptions of unlinkability, quantifying the linkability of items in the system. Our work does not provide any probabilistic definition, but this would be possible following the approach used in [15], that we leave as future work.

8 Conclusion and Future Work

In this paper we studied the privacy notion of unlinkability. We captured several definitions from the literature into a simple abstract model based on epistemic logic, obtaining natural and intuitive definitions in terms of the attacker's knowledge. We also identified inseparability, a notion dual to unlinkability, in weak and strong forms. Moreover, we showed that these privacy definitions are different in general, but do coincide in systems satisfying a set of conditions. Finally, we proved that the conditions hold for a class of identification protocols.

As future work, we plan at investigating probabilistic descriptions of unlinkability. We also plan at developing a more concrete model in the style of [11]. This work bridges the gap between operational semantics and epistemic logic, offering a combined framework where it is possible to easily model the behavior of a protocol in a process language with an operational semantics and reason about properties expressed in a rich epistemic temporal logic. This would allow to automatically verify privacy properties.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. of POPL, pp. 104–115 (2001)
2. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Untraceability in the applied pi-calculus. In: Proc. of ICITST, pp. 1–6. IEEE (2009)
3. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: Proc. of CSF, pp. 107–121. IEEE Computer Society (2010)
4. Avoine, G.: Adversary model for radio frequency identification. Technical Report LASEC-REPORT-2005-001, Swiss Federal Institute of Technology, Lausanne, Switzerland (2005)
5. Avoine, G.: Cryptography in Radio Frequency Identification and Fair Exchange Protocols. Ph.D. thesis, EPFL, Lausanne, Switzerland (2005)
6. Berman, R., Fiat, A., Ta-Shma, A.: Provable unlinkability against traffic analysis. In: Juels, A. (ed.) FC 2004. LNCS, vol. 3110, pp. 266–280. Springer, Heidelberg (2004)

7. Brusó, M., Chatzikokolakis, K., den Hartog, J.: Formal verification of privacy for rfid systems. In: Proc. of CSF, pp. 75–88. IEEE Computer Society (2010)
8. Burmester, M., van Le, T., de Medeiros, B.: Provably secure ubiquitous systems: Universally composable rfid authentication protocols. In: Proc. of Securecomm, pp. 1–9 (2006)
9. Chadha, R., Delaune, S., Kremer, S.: Epistemic logic for the applied pi calculus. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 182–197. Springer, Heidelberg (2009)
10. Chatmon, C., van Le, T., Burmester, M.: Secure anonymous RFID authentication protocols. Technical Report TR-060112, Florida State University, Tallahassee, Florida, USA (2006)
11. Dechesne, F., Mousavi, M.R., Orzan, S.: Operational and epistemic approaches to protocol analysis: Bridging the gap. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 226–241. Springer, Heidelberg (2007)
12. van Deursen, T., Mauw, S., Radomirović, S.: Untraceability of RFID protocols. In: Onieva, J.A., Sauveron, D., Chaumette, S., Gollmann, D., Markantonakis, K. (eds.) WISTP 2008. LNCS, vol. 5019, pp. 1–15. Springer, Heidelberg (2008)
13. van Deursen, T., Radomirović, S.: Algebraic attacks on RFID protocols. In: Markowitch, O., Bilas, A., Hoepman, J.-H., Mitchell, C.J., Quisquater, J.-J. (eds.) WISTP 2009. LNCS, vol. 5746, pp. 38–51. Springer, Heidelberg (2009)
14. Franz, M., Meyer, B., Pashalidis, A.: Attacking unlinkability: The importance of context. In: Borisov, N., Golle, P. (eds.) PET 2007. LNCS, vol. 4776, pp. 1–16. Springer, Heidelberg (2007)
15. Halpern, J.Y., O'Neill, K.R.: Anonymity and information hiding in multiagent systems. In: Proc. of CSFW, pp. 75–88. IEEE Computer Society (2003)
16. Hughes, D., Shmatikov, V.: Information hiding, anonymity and privacy: A modular approach. Journal of Computer Security 12, 3–36 (2004)
17. Juels, A., Weis, S.A.: Defining strong privacy for rfid. In: Proc. of PerCom Workshops, pp. 342–347. IEEE Computer Society (2007)
18. Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Cambridge University Press, New York (2004)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts i and ii. I and II. Information and Computation 100, 1–77 (1989)
20. Nohl, K., Evans, D.: Privacy through noise: a design space for private identification. In: Annual Computer Security Applications Conference, ACSAC 2009 (2009)
21. Ohkubo, M., Suzuki, K., Kinoshita, S.: Cryptographic approach to "privacy-friendly" tags. In: Proc. of RFID Privacy Workshop (2003)
22. Ouafi, K., Phan, R.C.-W.: Privacy of recent RFID authentication protocols. In: Chen, L., Mu, Y., Susilo, W. (eds.) ISPEC 2008. LNCS, vol. 4991, pp. 263–277. Springer, Heidelberg (2008)
23. Pfitzmann, A., Köhntopp, M.: Anonymity, unobservability, and pseudonymity - A proposal for terminology. In: Federrath, H. (ed.) Anonymity 2000. LNCS, vol. 2009, pp. 1–9. Springer, Heidelberg (2001)
24. Steinbrecher, S., Köpsell, S.: Modelling unlinkability. In: Dingledine, R. (ed.) PET 2003. LNCS, vol. 2760, pp. 32–47. Springer, Heidelberg (2003)
25. Syverson, P.F., Stubblebine, S.G.: Group principals and the formalization of anonymity. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 814–833. Springer, Heidelberg (1999)
26. Vaudenay, S.: On privacy models for RFID. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 68–87. Springer, Heidelberg (2007)
27. Weis, S.A., Sarma, S.E., Rivest, R.L., Engels, D.W.: Security and privacy aspects of low-cost radio frequency identification systems. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) Security in Pervasive Computing 2004. LNCS, vol. 2802, pp. 201–212. Springer, Heidelberg (2004)

Towards Quantitative Analysis of Opacity

Jeremy W. Bryans, Maciej Koutny, and Chunyan Mu

School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, UK
{jeremy.bryans,maciej.koutny,chunyan.mu}@ncl.ac.uk

Abstract. Opacity is a general approach for describing and unifying security properties expressed as predicates. A predicate is opaque if an observer of the system is unable to determine the satisfaction of the predicate in a given run of the system. The meaning of opacity is straightforward when considering the standard (qualitative) operational semantics, but there are a number of possible interpretations in a context where quantitative information about system evolutions is available. We propose four variants of quantitative opacity defined for probabilistic labelled transition systems, with each variant capturing a different aspect of quantifying the opacity of a predicate. Moreover, we present results showing how these four properties can be checked or approximated for specific classes of probabilistic labelled transition systems, observation functions, and system predicates.

Keywords: Probabilistic opacity, Probabilistic labelled transition systems, Observations.

1 Introduction

Opacity has been shown to be a promising technique for describing and unifying security properties [6]. For a given observer of a system (or adversary), a predicate capturing a system property is opaque if the observer will never be able to determine the truth of that predicate.

The definition of [6] is based on a qualitative operational semantics. In it, observation functions are used in order to give fine-grained control over the capabilities of an observer. Through such observations, an observer can establish certain properties of the system. Informally, an observer cannot establish the predicate (and hence the predicate is opaque) if for any run of the system in which the predicate is true, there is a run for which the predicate is false, and the two runs are observationally equivalent under the defined observation function. However, in the case where the probability of the first run is significantly higher than the probability of the second, the observer (although not able to be certain) may have good reason to believe that the predicate (although opaque) is none the less true. The formalism of probabilistic opacity can play an important role with regard to the representation and unification of security properties of concurrent system behaviours. This paper presents the results of our initial investigations into this probabilistic case.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 145–163, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Quantified security properties have recently turned into one of the key research topics in the computer security community. A main reason for this is that access control systems designed to regulate access to sensitive information can no longer fully control the propagation of information which has already been accessed. However, possible applications of the quantitative approach to security properties goes far beyond access control systems, as explained below. Absolute (or qualitative) security properties, such as non-interference [16] and opacity [7], abstract away to large extent the non-predictability is a system's behaviour. They are therefore attractive from the theoretical and analytical point of view, but the drawback is that they are rarely satisfied by the actual computing systems. A promising approach to relaxing the absolute nature of security properties, and bringing them closer to the real life application domain, is to quantify them, allowing one to tolerate 'small' violations. The paper focuses on quantifying *opacity* security property in transition systems. Our early work developed a method in modelling *opacity* property using Petri Nets [7], and in generalising the *opacity* property to transition systems [5,6]. The contribution of this paper is to show how the work referenced above extends to the more general case when the information given about the system is qualitative. We therefore consider the general theory of probabilistic opacity in the context of *probabilistic labelled transition systems* which allows one to reason about the quantitative security properties of systems. Based on the probabilistic model of opacity, we introduced four alternative definitions of probabilistic opacity, and investigated the efficiency with which they can be verified or approximated. We relate the definitions to the existing work on qualitative opacity. The obtained results can be used in a quantified security analysis of a computing system.

This paper is organised as follows. In Section 2 we recall some definitions from the literature in particular relating to probability distributions, and in Section 3 we give the definition of probabilistic labelled transition systems and prove a property which is then needed to estimate the efficiency of our approximations of probabilistic opacity. Section 4 contains our main contribution, i.e., the definitions of four variants of opacity together with an investigation of their basic properties. Section 5 contains a brief comparison with other work, and in Section 6 we present our concluding remarks.

2 Preliminaries

We use the standard mathematical notation. In particular, ϵ denotes the empty sequence, $|\lambda|$ denotes the length of a finite sequence λ, and λ^k denotes the concatenation of k copies of λ.

A *probability distribution* on a countable set X is a function $f : X \to [0, 1]$ such that $\sum_{x \in X} f(x) = 1$. To measure difference between probability distributions on the same set, we will use Jensen-Shannon divergence [20] which is related to information-theoretical functionals. It is symmetric, always well-defined and bounded by 1.

Let $P = \{p_x\}_{x \in X}$ and $P' = \{p'_x\}_{x \in X}$ be two probability distributions on a countable set X with associated weights w and w', respectively ($0 \leq w, w' \leq 1$

and $w + w' = 1$). Then the weighted Jensen-Shannon divergence between P and P' is given by:

$$D_{JS}(w \cdot P, w' \cdot P') = \mathcal{H}\left(\{w \cdot p_x + w' \cdot p'_x\}_{x \in X}\right) - w \cdot \mathcal{H}\left(\{p_x\}_{x \in X}\right) - w' \cdot \mathcal{H}\left(\{p'_x\}_{x \in X}\right),$$

where $\mathcal{H}(\{q_x\}_{x \in X}) = -\sum_{x \in X} q_x \log_2 q_x$ denotes Shannon entropy [26] (note that if $q_x = 0$ then $q_x \log_2 q_x$ is taken to be 0 which is justified by $\lim_{q \to 0^+} q \log_2 q = 0$).

If, in the above formula, we denote by d_x the 'contribution' made by a single element $x \in X$, then:

$$D_{JS}(w \cdot P, w' \cdot P') = \sum_{x \in X} d_x \ ,$$

where:

$$d_x = -(w \cdot p_x + w' \cdot p'_x) \cdot \log_2(w \cdot p_x + w' \cdot p'_x) + w \cdot p_x \cdot \log_2 p_x + w' \cdot p'_x \cdot \log_2 p'_x \ . \quad (1)$$

An individual contribution is minimal ($d_x = 0$) if $p_x = p'_x$, i.e., when P and P' do not diverge at x. It is maximal if one of the probabilities is 0, which gives $d_x = -w \cdot p_x \cdot \log_2 w$ or $d_x = -w' \cdot p'_x \cdot \log_2 w'$, and so:

$$d_x \leq -w \cdot p_x \cdot \log_2 w - w' \cdot p'_x \cdot \log_2 w' \leq c \cdot (p_x + p'_x) \ ,$$

where $c > 0$ is a constant depending on w and w'. As a consequence, if we take $Y \subset X$ then the contribution d_Y of the elements of Y to the overall divergence satisfies:

$$d_Y \leq c \cdot (P(Y) + P'(Y)) \ . \quad (2)$$

3 Probabilistic Labelled Transition Systems

In order to consider probabilistic behaviour and quantitative analysis of opacity, we use probabilistic labelled transition systems which adapts the well-known model introduced in [19].

A *labelled transition system* is a tuple: $LTS = (S, L, \Delta, s_0)$, where S is a countable set of states, L is a countable set of labels, $\Delta \subseteq S \times L \times S$ is the transition set, and $s_0 \in S$ is the initial state. We consider deterministic labelled transition systems,[1] and so for any transitions $(s, l, s'), (s, l, s'') \in \Delta$, it is the case that $s' = s''$. For every state $s \in S$, we will denote by Δ_s the set of all transitions outgoing from s, i.e., $\Delta_s = \{(s', l, s'') \in \Delta \mid s = s'\}$.

A *run* of LTS is a finite sequence of labels $\lambda = l_1 \ldots l_n$ $(n \geq 0)$[2] such that there are states $s_1, \ldots, s_n$ satisfying (s_{i-1}, l_i, s_i), for $i = 1, \ldots, n$. We will denote the state s_n by s_λ and call it *reachable*. Note that s_λ is well-defined since LTS is deterministic. Moreover, $s_\epsilon = s_0$, where ϵ denotes the empty run. The set of all runs of LTS will be denoted by $runs(LTS)$.

Probabilistic labelled transition systems are labelled transition systems with probability distributions attached to all the states.

[1] Nondeterminism is introduced in the next section, through the notion of an observation function.

[2] If $n = 0$ then $\lambda = \epsilon$.

Definition 1. *A* probabilistic labelled transition system *is a tuple:*

$$PLTS = (S, L, \Delta, s_0, \mu) \ ,$$

such that $LTS = (S, L, \Delta, s_0)$ *is a labelled transition system and* $\mu : S \cup \Delta \to [0, 1]$ *is a mapping satisfying the following:*

(i) for every $s \in S$, μ *restricted to* $\{s\} \cup \Delta_s$ *is a probability distribution:*

$$\mu(s) + \sum_{d \in \Delta_s} \mu(d) = 1 \ ,$$

and $\inf\{\mu(s) \mid s \in S \wedge \mu(s) \neq 0\} > 0$.

(ii) there is an integer $k \geq 1$ *such that there is no sequence of transitions in* Δ*:*

$$(s, l_1, s_2), (s_2, l_2, s_3), \ldots, (s_m, l_m, s_{m+1})$$

such that $\mu(s_2) = \mu(s_3) = \ldots = \mu(s_m) = 0$ *and* $m > k$.

The set of runs of PLTS, denoted runs(PLTS), is the same as that of the underlying labelled transition system. Other notations are also inherited.

Definition 1(i) ensures that for every state s, the probability $\mu(s)$ of remaining in that state together with the probabilities associated with all transitions out of that state form a probability distribution. We also require that non-empty probabilities $\mu(s)$ cannot be arbitrarily small (similarly as in [19] it was assumed that non-empty probabilities $\mu(s, l, s')$ cannot be arbitrarily small). Note that this is always the case if there are finitely many states.

We extend the mapping μ to each run $\lambda = l_1 \ldots l_k$ of *PLTS*, in the following way. Let $s_0, s_1, \ldots, s_k$ be states such that $(s_{i-1}, l_i, s_i) \in \Delta$, for $i = 1, \ldots, k$. Then:

$$\mu(\lambda) = \mu(s_k) \cdot \prod_{i=1}^{k} \mu(s_{i-1}, l_i, s_i) \ .$$

Note that $\mu(\lambda)$ is well-defined as the underlying labelled transition system is deterministic. We also denote:

$$\widetilde{\mu}(\lambda) = \prod_{i=1}^{k} \mu(s_{i-1}, l_i, s_i)$$

(i.e., $\mu(\lambda) = \mu(s_k) \cdot \widetilde{\mu}(\lambda)$) and, for every set of runs $\Lambda \subseteq runs(PLTS)$:

$$\mu(\Lambda) = \sum_{\lambda \in \Lambda} \mu(\lambda) \quad \text{and} \quad \widetilde{\mu}(\Lambda) = \sum_{\lambda \in \Lambda} \widetilde{\mu}(\lambda) \ .$$

Proposition 1. $\mu(runs(PLTS)) \leq 1$.

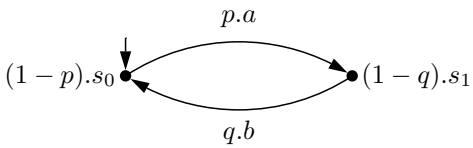
Proof. Follows from the fact that, for every $i \geq 0$:

$$\mu(\{\lambda \in runs(PLTS) \mid |\lambda| \leq i\}) + \widetilde{\mu}(\{\lambda \in runs(PLTS) \mid |\lambda| = i + 1\}) = 1 \ .$$

The above can be shown by a straightforward induction on i, using Definition 1(i). □

The formalisation of a probabilistic labelled transition system is tailored to reflect our understanding of observation of a computing system. In a nutshell, we treat $\prod_{i=1}^{k} \mu(s_{i-1}, l_i, s_i)$ in the standard way as the probability of executing a sequence of transitions making up the run λ. In addition to that, the factor $\mu(s_k)$ gives the probability that the observation is concluded after the state s_k has been reached. For instance, it may be the probability that the process terminates after reaching s_k, similarly as it was done in [2]. It therefore follows that Definition 1(ii) captures our intuition that the system cannot be indefinitely 'unobserved' (i.e., probability of a conclusive observation cannot be zero forever).

Example 1. Consider the following probabilistic labelled transition system:



where $0 \leq p, q \leq 1$ and the notation $x.y$ indicates that $x = \mu(y)$. According to Definition 1(ii), we must have $p \cdot q \neq 1$. We can then show that μ defines a probability distribution, as follows:

$$\begin{aligned}
\mu(runs(PLTS)) &= \sum_{k=0}^{\infty} \mu((ab)^k) + \sum_{k=0}^{\infty} \mu(a(ba)^k) \\
&= \sum_{k=0}^{\infty} (p \cdot q)^k \cdot (1-p) + \sum_{k=0}^{\infty} p \cdot (q \cdot p)^k \cdot (1-q) \\
&= (1-p) \cdot \sum_{k=0}^{\infty} (p \cdot q)^k + p \cdot (1-q) \cdot \sum_{k=0}^{\infty} (q \cdot p)^k \\
&= (1-p) \cdot \frac{1}{1-p \cdot q} + p \cdot (1-q) \cdot \frac{1}{1-p \cdot q} \\
&= 1 \,.
\end{aligned}$$

Note that if in the above example we assumed that $p = q = 1$, and hence $\mu(s_0) = \mu(s_1) = 0$, then we would have $\mu(runs(PLTS)) = 0$, and so μ would not be a probability distribution on the runs of *PLTS*. To avoid this, we introduced condition (ii) in Definition 1 which rules out this case. Note also that the condition captured through Definition 1(ii) is easy to verify by checking that in the graph of *PLTS* there are no cycles passing only through reachable states s satisfying $\mu(s) = 0$.

Since the set of runs is in general infinite, we will be approximating various quantities defined on the basis of the set of runs, by looking only at runs up to certain length. We therefore define, for every $m \geq 0$:

$$runs_m(PLTS) = \{\lambda \in runs(PLTS) \mid |\lambda| \leq m\} .$$

The next result is crucial for the soundness of such approximations.

Proposition 2. *There is an integer $\kappa \geq 1$ and a real number $0 \leq \delta < 1$ such that, for every $i \geq 0$:*

$$\mu(runs_{\kappa \cdot i}(PLTS)) \geq 1 - \delta^i .$$

Proof. See Appendix. □

In other words, we know how far to 'unfold' the transition system to approximate with arbitrary accuracy 'almost all' the runs (in probabilistic terms).

As a corollary of our previous results, μ always defines a probability distribution for the set of runs of the probabilistic labelled transition system.

Theorem 1. $\mu(runs(PLTS)) = 1$.

Proof. Follows directly from Propositions 1 and 2. □

4 Probabilistic Opacity

In this section, we introduce concepts relating to the definitions of probabilistic opacity, and prove our main results.

In what follows, $PLTS = (S, L, \Delta, S_0, \mu)$ is a probabilistic labelled transition system, and *Obs* is a set of elements called *observables*. To model the different capabilities for observing the system modelled by *PLTS*, we use an *observation function*:

$$obs : runs(PLTS) \to Obs^* .$$

We will, in particular, use the *static* observation function *obs* for which there is a map $obs' : L \to Obs \cup \{\epsilon\}$ such that, for every run $\lambda = l_1 \ldots l_n$ of *PLTS*:

$$obs(\lambda) = obs'(l_1)obs'(l_2)\ldots obs'(l_n) .$$

Consider now an observation function *obs*. We are interested in whether an observer (or attacker) can establish a *property* ϕ (a predicate over system runs) for a run of *PLTS* having only access to the result of the observation function. We will identify ϕ with its characteristic set, i.e., the set of all those runs for which it holds. Now, given an observed execution of the system, we would want to find out whether the fact that the underlying run belongs to ϕ can be deduced by the observer. We will, in particular, be interested in the *final* opacity predicates, ϕ_Z, where $Z \subseteq S$, defined as the set of all the runs λ of *PLTS* satisfying $s_\lambda \in Z$. Intuitively, this means that we are interested in finding out whether an observed run of the system represented by *PLTS* ended in one of secret (or sensitive) states belonging to Z. (Note that we are not interested in establishing whether the underlying run does not belong to ϕ; to do this, we would consider the property $runs(PLTS) \setminus \phi$.)

We will now introduce a series of notions relating to the idea of opacity, recalling first its standard non-probabilistic (or qualitative) definition. In what follows, *obs* is an observation function for *PLTS*, ϕ is a subset of $runs(PLTS)$, and $\overline{\phi} = runs(PLTS) \setminus \phi$. Intuitively, ϕ captures a property which we want to declare opaque.

4.1 Qualitative (Non-probabilistic) Opacity of [6]

When no probabilistic information is supplied, or if one is simply not interested in probabilistic aspects of the system, we say that ϕ is *opaque* w.r.t. *obs* if, for every run $\lambda \in \phi$, there is another run $\lambda' \notin \phi$ such that $obs(\lambda) = obs(\lambda')$, i.e., λ' *covers* λ. In other words, all runs in ϕ are covered by runs in $\overline{\phi}$:

$$obs(\phi) \subseteq obs(\overline{\phi}) \ . \tag{3}$$

Different variants of qualitative opacity have been discussed in, for example, [6].

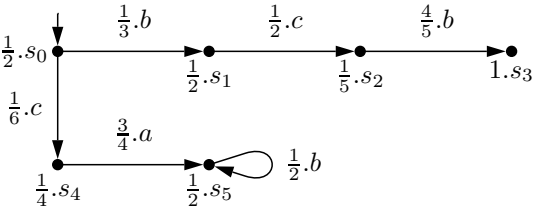
4.2 Quantitative (Probabilistic) Opacity

What it means to deduce (or satisfactorily cover) a property expressed as ϕ in the probabilistic case may mean different things, depending on what is relevant or important from the point of view of a real application. In particular, one may consider different ways of quantifying the degree to which runs contained in ϕ are covered by the runs in $\overline{\phi}$ (c.f. the inclusion (3)), leading to different variants of quantitative opacity.

π-Opacity. A straightforward approach to defining probabilistic opacity might be to require that the likelihood of ever witnessing an uncovered run of ϕ is negligible. That is, we say that ϕ is π-*opaque* w.r.t. *obs* if the probability of having a run in ϕ which is not covered by a run in $\overline{\phi}$ is zero:

$$\mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))) = 0 \ . \tag{4}$$

Example 2. Consider the following probabilistic labelled transition system:



where $obs(a) = a$, $obs(b) = \epsilon$ and $obs(c) = c$, as well as the property $\phi = \phi_{\{s_2, s_3\}}$. We then have:

$$\phi = \{bc, bcb\} \ \text{ and } \ \overline{\phi} = \{\epsilon, b, c, ca, ca(b)^*\} \ .$$

In this case, we have

$$obs(runs(PLTS)) = obs(\overline{\phi}) = \{\epsilon, c, ca\}, \ \text{ and } \ obs(\phi) = \{c\},$$

and so, obviously, π-opacity is satisfied:

$$\mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))) = \mu(\varnothing) = 0.$$

Checking pi-opacity is decidable, as shown by the next result and its proof.

Theorem 2. *For a finite PLTS and a static observation function obs, it is decidable whether ϕ_Z (where $Z \subset S$) is π-opaque.*

Proof. We first observe that, in this case, $\mu(\phi_Z \setminus obs^{-1}(obs(\overline{\phi}_Z))) = 0$ is equivalent to $obs(L) \subseteq obs(L')$, where L is the regular language obtained from *PLTS* by changing each label l to $obs(l)$ and setting as the final states all those $s \in Z$ for which $\mu(s) > 0$; and L' is the regular language obtained from *PLTS* by changing each label l to $obs(l)$ and setting $S \setminus Z$ as the final states. Since the inclusion of two regular languages is decidable, the result follows. □

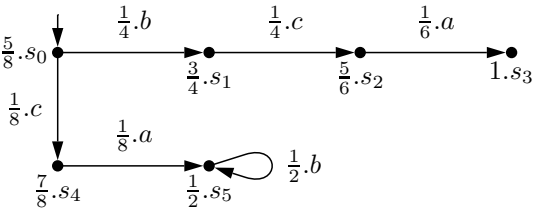
π_ξ-opacity One could argue that the probabilistic opacity captured by (4) is too demanding, and one might require only that the probability of witnessing an uncovered run of ϕ is low. To capture this, we say that ϕ is *π_ξ-opaque* w.r.t. *obs* if $0 \leq \xi \leq 1$ is the probability of having a run in ϕ which is not covered:

$$\mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))) = \xi \ . \tag{5}$$

One would then declare ϕ opaque if ξ was sufficiently small number. Note that π_0-opacity coincides with π-opacity.

In practice, knowing the value of ξ with high accuracy (see Theorem 3) would allow a designer or verifier to compare it with a given required opacity level, ξ_{req}. The system represented by *PLTS* would then satisfy the opacity w.r.t. ϕ if $\xi \leq \xi_{req}$. Similar comment applies to other opacity notions introduced in the rest of this paper.

Example 3. Consider the following probabilistic labelled transition system:



where $obs(a) = a$, $obs(b) = \epsilon$ and $obs(c) = c$, as well as the property $\phi = \phi_{\{s_2,s_3,s_5\}}$. We then have:

$$\phi = \{bc, bca, ca(b)^*\} \ \text{ and } \ \overline{\phi} = \{\epsilon, b, c\} \ .$$

Hence:

$$\phi \setminus obs^{-1}(obs(\overline{\phi})) = \{bca, ca(b)^*\} \ .$$

and we obtain:

$$\mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))) = \frac{1}{4}\cdot\frac{1}{4}\cdot\frac{1}{6} + \frac{1}{8}\cdot\frac{1}{8}\cdot\frac{1}{2} + \frac{1}{8}\cdot\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{2} + \frac{1}{8}\cdot\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2} + \ldots$$
$$= \frac{5}{192} \approx 0.026.$$

The property $\phi_{\{s_2,s_3,s_5\}}$ is therefore $\pi_{0.026}$-opaque.

Although determining the π_ξ-opacity may in general be difficult, in several important cases it is possible to approximate the value of ξ with a desired accuracy.

The next result requires that the observation function is such that one does not have to wait for 'too long' in order to find a run in $\overline{\phi}$ covering $\lambda \in \phi$. More precisely, we say that *obs* is *N-efficient* for *PLTS* and ϕ, if N is a positive integer such that, for every run $\lambda \in \phi$ which is covered by runs in $\overline{\phi}$, there exists a run $\lambda' \in \overline{\phi}$ covering λ and satisfying $|\lambda'| \leq N \cdot |\lambda|$. Note that being efficient is not too demanding a requirement; in particular, each static observation function *obs* is N-efficient provided that *PLTS* is finite and $\phi = \phi_Z$ for some $Z \subset S$ (N can then be taken to be the number of states of *PLTS*). Below, $\phi_k = \phi \cap runs_k(PLTS)$ and $\overline{\phi}_k = \overline{\phi} \cap runs_k(PLTS)$, for every $k \geq 0$.

Theorem 3. *If obs is N-efficient for PLTS and ϕ, then there is an integer $\zeta \geq 1$ and a real number $0 \leq \eta < 1$ such that, for every $i \geq 0$:*

$$|\mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))) - \mu(\phi_{\zeta\cdot i} \setminus obs^{-1}(obs(\overline{\phi}_{\zeta\cdot i})))| \leq \eta^i \ .$$

Proof. By Proposition 2, there exists a positive integer κ and a real number $0 \leq \delta < 1$ such that $\mu(runs_{\kappa\cdot i}(PLTS)) \geq 1 - \delta^i$, for every $i \geq 0$. In other words, for every $i \geq 0$:

$$\mu(runs(PLTS) \setminus runs_{\kappa\cdot i}(PLTS)) \leq \delta^i \ . \tag{6}$$

Let us now take any $i \geq 0$, and consider:

$$x = \mu(\phi \setminus obs^{-1}(obs(\overline{\phi}))), \quad y = \mu(\phi_{N\cdot\kappa\cdot i} \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i}))) \ .$$

We then observe that, by *obs* being N-efficient, we have:

$$y = \mu(\phi_{\kappa\cdot i} \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i}))) + \mu((\phi_{N\cdot\kappa\cdot i} \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i})))$$
$$= \mu(\phi_{\kappa\cdot i} \setminus obs^{-1}(obs(\overline{\phi}))) + \mu((\phi_{N\cdot\kappa\cdot i} \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i}))) \ .$$

We therefore obtain:

$$x - y = \mu((\phi \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi}))) - \mu((\phi_{N\cdot\kappa\cdot i} \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i})) \ .$$

Now, since

$$(\phi \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi})) \subseteq runs(PLTS) \setminus runs_{\kappa\cdot i}(PLTS)$$
$$(\phi_{N\cdot\kappa\cdot i} \setminus \phi_{\kappa\cdot i}) \setminus obs^{-1}(obs(\overline{\phi}_{N\cdot\kappa\cdot i}) \subseteq runs(PLTS) \setminus runs_{\kappa\cdot i}(PLTS)$$

and $x, y \geq 0$, we obtain from (6) that $|x - y| \leq \delta^i$. Hence the result holds with $\zeta = N \cdot \kappa$ and $\xi = \delta$. □

In other words, finite unfoldings of a probabilistic labelled transition system can approximate the probability of the uncovered runs in ϕ with a desired precision, providing a natural way of estimating π_ξ-opacity.

$\overline{\pi}_\gamma$-Opacity. Let us consider the set ϕ^{cov} of runs of $\overline{\phi}$ which cover at least one run in ϕ, i.e., $\phi^{cov} = \overline{\phi} \cap obs^{-1}(obs(\phi))$. The first two notions of quantitative opacity retained the flavour of the original (qualitative) opacity. In particular, so far we have accepted that a set of runs ϕ with non-zero occurrence probability, $\mu(\phi) > 0$, can be covered by a set of runs with occurrence probability much lower than that of ϕ, $\mu(\phi^{cov}) \ll \mu(\phi)$, or indeed with a negligible chance of ever occurring, $\mu(\phi^{cov}) = 0$. That is, we were basically demanding very low occurrence probability of totally uncovered runs in ϕ. In our next definition, we remedy this by intuitively requiring that each run in ϕ is covered by γ runs, where $\gamma > 0$ would normally be expected to be (much) greater than 1. More precisely, for every $\gamma \geq 0$, we say that ϕ is $\overline{\pi}_\gamma$-*opaque* if:

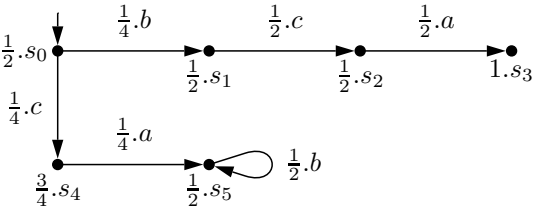
$$\mu(\phi) > 0 \quad \text{and} \quad \frac{\mu(\phi^{cov})}{\mu(\phi)} = \gamma \,, \tag{7}$$

or, slightly more generally (as we do not have to assume that $\mu(\phi) > 0$), if the following holds:

$$\mu(\phi^{cov}) - \gamma \cdot \mu(\phi) = 0 \,. \tag{8}$$

In combination with π_ξ-opacity for small ξ, $\overline{\pi}_\gamma$-opacity for large γ would clearly increase our confidence in declaring ϕ opaque.

Example 4. Consider the following probabilistic labelled transition system:



where $obs(a) = \epsilon$, $obs(b) = b$ and $obs(c) = c$, as well as the property $\phi = \phi_{\{s_3, s_5\}}$. We then have:

$$\phi = \{bca, ca, ca(b)^*\} \quad \text{and} \quad \phi^{cov} = \{bc, c\} \,.$$

and so we obtain:

$$\mu(\phi) = \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{2} + \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} + \ldots = \frac{1}{8}$$

$$\mu(\phi^{cov}) = \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{4} \cdot \frac{3}{4} = \frac{1}{4}$$

which leads to:

$$\frac{\mu(\phi^{cov})}{\mu(\phi)} = 2.$$

The property $\phi_{\{s_3,s_5\}}$ is therefore $\overline{\pi}_2$-opaque.

As before, we will now investigate how one could approximate $\overline{\pi}_\gamma$-opacity. Below we assume that *obs* is *inversely M-efficient* for *PLTS* and ϕ, by which we mean that M is a positive integer such that, for every run $\lambda \in \phi^{cov}$, there exists a run $\lambda' \in \phi$ covered by λ and satisfying $|\lambda'| \leq M \cdot |\lambda|$. Again, being inversely efficient is not too demanding a requirement; in particular, each static observation function *obs* is inversely M-efficient provided that *PLTS* is finite and $\phi = \phi_Z$ for some $Z \subset S$ (M can then be taken to be the number of states of *PLTS*).

Theorem 4. *Let ϕ be $\overline{\pi}_\gamma$-opaque. If obs is inversely M-efficient for PLTS and ϕ, then there is an integer $\rho \geq 1$ and a real number $0 \leq \nu < 1$ such that, for every $i \geq 0$:*

$$|(\mu(\phi^{cov}) - \gamma \cdot \mu(\phi)) - (\mu(\phi^{cov}_{\rho \cdot i}) - \gamma \cdot \mu(\phi_{\rho \cdot i}))| \leq (1+\gamma) \cdot \nu^i \ .$$

Proof. See Appendix. □

In practice, one could approximate the value of γ using the inequalities $|x| \leq \delta^i$ and $y \leq \delta^i$ from the proof in the Appendix. More precisely, we approximate this value provided that $\mu(\phi^{cov}_{M \cdot \kappa \cdot i}) > 0$ as then we also have $\mu(\phi^{cov}) \geq \mu(\phi^{cov}_{M \cdot \kappa \cdot i}) > 0$. From $|x| \leq \delta^i$ we further obtain: $\mu(\phi^{cov}) - \mu(\phi^{cov}_{M \cdot \kappa \cdot i}) \leq \delta^i$. Hence:

$$\mu(\phi^{cov}_{M \cdot \kappa \cdot i}) \leq \mu(\phi^{cov}) \leq \mu(\phi^{cov}_{M \cdot \kappa \cdot i}) + \delta^i \ .$$

Moreover, from $y \leq \delta^i$ and $\mu(\phi) \geq \mu(\phi_{M \cdot \kappa \cdot i})$ we obtain:

$$\mu(\phi_{M \cdot \kappa \cdot i}) \leq \mu(\phi) \leq \mu(\phi_{M \cdot \kappa \cdot i}) + \delta^i \ .$$

Hence we obtain:

$$\frac{\mu(\phi^{cov}_{M \cdot \kappa \cdot i})}{\mu(\phi_{M \cdot \kappa \cdot i}) + \delta^i} \leq \gamma \leq \frac{\mu(\phi^{cov}_{M \cdot \kappa \cdot i}) + \delta^i}{\mu(\phi_{M \cdot \kappa \cdot i})} \ .$$

$\widetilde{\pi}_\psi$-Opacity. The above notions of defining probabilistic opacity may still find it difficult to distinguish between subtle differences in which *obs* acts upon ϕ and $\overline{\phi}$. A possible way to assess such differences could be, e.g., to look at the probability distributions induced by $obs(\phi)$ and $obs(\phi^{cov})$ and conclude that they are rather similar.

In our last attempt at a notion of quantitative opacity, we define $\widetilde{\pi}_\psi$-opacity which uses Jensen-Shannon divergence as a way to measure the differences in which *obs* acts upon ϕ and ϕ^{cov}. Below we assume that $\mu(\phi) > 0$ and $\mu(\phi^{cov}) > 0$.

The reason we choose the Jensen-Shannon divergence as a measures is that it is related to information-theoretical functionals, such as Kullback-Leibler distance

(the relative entropy). It therefore shares some of their properties as well as their intuitive interpretation, and measures the difference in information bits. Specifically, unlike the Kullback-Leibler distance, it is symmetric, always well-defined and bounded by 1.

Since $runs(PLTS)$ with μ is a probabilistic space, $obs(runs(PLTS))$ can also be turned into a probabilistic space by defining

$$\pi(o) = \mu(obs^{-1}(o) \cap runs(PLTS)) \ ,$$

for every $o \in \mathcal{O} = obs(runs(PLTS))$. Moreover, any subset Λ of $runs(PLTS)$ with $\mu(\Lambda) \geq 0$ gives rise to a probability distribution Π_Λ on $\mathcal{O}$. More precisely, for every $o \in \mathcal{O}$:

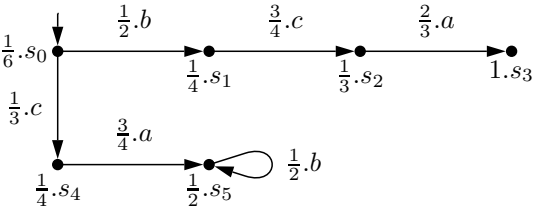
$$\Pi_\Lambda(o) = \frac{\mu(obs^{-1}(o) \cap \Lambda)}{\mu(\Lambda)}.$$

Then, for a property ϕ, we can define Π_ϕ and $\Pi_{\phi^{cov}}$ and say that ϕ is $\widetilde{\pi}_\psi$*-opaque* if $0 \leq \psi \leq 1$ is their weighted Jensen-Shannon divergence:

$$D_{JS}(w \cdot \Pi_\phi, w' \cdot \Pi_{\phi^{cov}}) = \psi \ ,$$

where $w = \frac{\mu(\phi)}{\mu(\phi)+\mu(\phi^{cov})}$ and $w' = \frac{\mu(\phi^{cov})}{\mu(\phi)+\mu(\phi^{cov})}$.

Example 5. Consider the following probabilistic labelled transition system:



where $obs(a) = a$, $obs(b) = \epsilon$ and $obs(c) = c$, as well as the property $\phi = \phi_{\{s_2,s_3\}}$. We then have:

$$\phi = \{bc, bca\} \ \text{ and } \ \phi^{cov} = \{c, ca, ca(b)^*\} \ .$$

and so:

$$\mu(\phi) = \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{2}{3} \cdot 1 = \frac{3}{8}$$

$$\mu(\phi^{cov}) = \frac{1}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{3}{4} \cdot \frac{1}{2} + \frac{1}{3} \cdot \frac{3}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} + \cdots = \frac{1}{3}$$

In this case, $\mathcal{O} = \{\epsilon, c, ca\}$, and we obtain:

$$\Pi_\phi = \left\{\epsilon \mapsto 0, \ c \mapsto \frac{\frac{1}{8}}{\frac{3}{8}}, \ ca \mapsto \frac{\frac{1}{4}}{\frac{3}{8}}\right\} = \left\{\epsilon \mapsto 0, \ c \mapsto \frac{1}{3}, \ ca \mapsto \frac{2}{3}\right\}$$

$$\Pi_{\phi^{cov}} = \left\{\epsilon \mapsto 0, \ c \mapsto \frac{\frac{1}{12}}{\frac{1}{3}}, \ ca \mapsto \frac{\frac{1}{4}}{\frac{1}{3}}\right\} = \left\{\epsilon \mapsto 0, \ c \mapsto \frac{1}{4}, \ ca \mapsto \frac{3}{4}\right\}$$

We calculate the weights of Π_ϕ and $\Pi_{\phi^{cov}}$ as:

$$w_{\Pi_\phi} = \frac{\frac{3}{8}}{\frac{3}{8}+\frac{1}{3}} = \frac{9}{17} \text{ and } w_{\Pi_{\phi^{cov}}} = \frac{\frac{1}{3}}{\frac{3}{8}+\frac{1}{3}} = \frac{8}{17}$$

and finally calculate:

$$\begin{aligned} D_{JS}(w_{\Pi_\phi}\cdot\Pi_\phi, w_{\Pi_{\phi^{cov}}}\cdot\Pi_{\phi^{cov}}) &= \mathcal{H}\left(0\ ,\ \frac{9}{17}\cdot\frac{1}{3}+\frac{8}{17}\cdot\frac{1}{4}\ ,\ \frac{9}{17}\cdot\frac{2}{3}+\frac{8}{17}\cdot\frac{3}{4}\right) \\ &\quad - \left(\frac{9}{17}\cdot\mathcal{H}\left(0,\frac{1}{3},\frac{2}{3}\right)+\frac{8}{17}\cdot\mathcal{H}\left(0,\frac{1}{4},\frac{3}{4}\right)\right) \\ &\approx 0.006\ . \end{aligned}$$

The property $\phi_{\{s_2,s_3\}}$ is therefore $\widetilde{\pi}_{0.006}$-opaque.

Similarly as in the previous cases, it may be possible to approximate the value of ψ in $\widetilde{\pi}_\psi$-opacity with a desired accuracy, using finite unfoldings of the probabilistic transition system. Below we assume that *obs* is K, L-*bounded* for *PLTS* and ϕ, by which we mean that K and L are positive integers such that:

- for every observation $o \in \mathcal{O}$ and $\lambda \in \phi \cup \phi^{cov}$, if $obs(\lambda) = o$ then $|\lambda| \leq K \cdot |o|$.
- for every run $\lambda \in \phi \cup \phi^{cov}$, $|obs(\lambda)| \leq L \cdot |\lambda|$.

Note that each static observation function *obs* is $K, 1$-bounded provided that *PLTS* is finite and *obs* does not induce ϵ-loops in the part of *PLTS* which is covered by the runs in $\phi \cup \phi^{cov}$ (K can then be taken to be the length of the longest ϵ-path in such a part of *PLTS* plus 1).

In the next result, we attempt to approximate the value of:

$$D_{JS}\left(w\cdot\left\{\frac{\mu(obs^{-1}(o)\cap\phi)}{\mu(\phi)}\right\}_{o\in\mathcal{O}}\ ,\ w'\cdot\left\{\frac{\mu(obs^{-1}(o)\cap\phi^{cov})}{\mu(\phi^{cov})}\right\}_{o\in\mathcal{O}}\right)\ .$$

To simplify the discussion, we assume that we are given the values of w, w', $\mu(\phi)$ and $\mu(\phi^{cov})$ (note that we can calculate them with a desired accuracy using Proposition 2).

Below $\mathcal{O}_k$ denotes $\{o \in \mathcal{O} \mid |o| \leq k\}$, for every $k \geq 0$. Moreover, for every $o \in \mathcal{O}$ and $m \geq 0$:

$$d_o^m = -(w\cdot p + w'\cdot p')\cdot\log_2(w\cdot p + w'\cdot p') + w\cdot p\cdot\log_2 p + w'\cdot p'\cdot\log_2 p'\ ,$$

where:

$$p = \frac{\mu(obs^{-1}(o)\cap\phi_m)}{\mu(\phi)} \quad\text{and}\quad p' = \frac{\mu(obs^{-1}(o)\cap\phi_m^{cov})}{\mu(\phi^{cov})}\ .$$

Theorem 5. *Let obs be K, L-bounded for PLTS and ϕ, and κ and $0 \leq \delta < 1$ be as in Proposition 2. Then there is $\alpha > 0$ such that, for every $i \geq 0$:*

$$0 \quad\leq\quad D_{JS}(w\cdot\Pi_\phi, w'\cdot\Pi_{\phi^{cov}}) - \sum_{o\in\mathcal{O}_{\kappa\cdot L\cdot i}} d_o^{\kappa\cdot L\cdot K\cdot i} \quad\leq\quad \alpha\cdot\delta^i\ . \tag{9}$$

Proof. See Appendix. □

5 Related Work

Opacity and related concepts were first studied and related to information flow properties in a qualitative context in [6,7,5]. In the probabilistic context, opacity has been studied in [18,2]. [18] studied the notion of opacity in the probabilistic computational world. There opacity was based on the probabilities of observer's pre-beliefs on the truth of the predicate. The work in [2] presents a quantitative information leakage analysis concerning probabilistic opacity, and there is a clear relationship between that work and the work in this paper. Indeed, although the setting in [2] is based on finite probabilistic automata, our probabilistic labelled transition system could be viewed as a generalisation of the fully probabilistic automaton (FPA) considered there. Note, however, that the automata in [2] are always finite and the notion of opacity is symmetric, while our system model allows infinite state spaces and we consider asymmetric opacity. The asymmetry here comes from the weights in the Jensen-Shannon divergence. We introduce approximation approaches to deal with the infinite transition systems. Our work can also be related to quantitative analysis for secure information flow conceptually. In general, the following measures have been investigated and applied quantified secure flow analysis w.r.t. non-interference properties: including Shannon's information entropy [15,9,11,10,3,12,22,17,24,23,8,4], min-entropy [27], guessing entropy [17], and belief-based measures [13,14]. Furthermore, [25] gave a definition of probabilistic measures on flows in a specific probabilistic declarative language: Probabilistic Concurrent Constraint Programming (PCCP). [21] measured information flow in CSP by counting refusals. [1] measured the process similarity relation with regard to an approximation of weak bisimulation of CCS. Most of these works relate to the property of *non-interference* from the security literature, and focus on *flow* measurement rather than the more general property *opacity* as studied in this paper.

Opacity has already provided a promising technique for describing and unifying more general security properties. This paper has extended the notion of opacity to the model of probabilistic labelled transition systems. The results presented allow one to investigate and represent concepts from the literature on secure flow analysis.

6 Conclusions and Further Work

We have presented a formal model for the description of probabilistic opacity based on probabilistic labelled transition systems. We extend and generalise the notion of qualitative opacity and show how it applies to probabilistic and quantitative systems. We have investigated four alternative definitions of probabilistic opacity and given initial efficiency and approximation results. We believe that these results are promising and merit further consideration.

There is a clear link between the work presented here and the work on quantified information flow within the security community. Information flow security aims to ensure that information propagates throughout the execution environment without security violations such that only controlled secure information

is deducible from observations of the system. The information we require to be confidential can be described as a predicate which we require to be opaque. By studying opacity in a quantified context we can relax the strict qualitative security policies, and tolerate a low probability that a quantitatively 'small' amount of secure information is leaked. We therefore believe our general model and results will be useful for quantified flow analysis in the security community.

More generally, we believe our work can provide a framework for the measurement of system security, by quantifying the opacity of key predicates with respect to the system. In future work, we plan to develop and extend the initial results presented here, as well as investigate and establish links between our work and the other work in the security community on the measurement of quantified information flow.

References

1. Aldini, A., Pierro, A.D.: A quantitative approach to noninterference for probabilistic systems (2004)
2. Bérard, B., Mullins, J., Sassolas, M.: Quantifying opacity. In: QEST, pp. 263–272 (2010)
3. Boreale, M.: Quantifying information leakage in process calculi. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006, Part II. LNCS, vol. 4052, pp. 119–131. Springer, Heidelberg (2006)
4. Boreale, M., Pampaloni, F., Paolini, M.: Asymptotic information leakage under one-try attacks. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 396–410. Springer, Heidelberg (2011)
5. Bryans, J.W., Koutny, M., Mazaré, L., Ryan, P.Y.A.: Opacity generalised to transition systems. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2005. LNCS, vol. 3866, pp. 81–95. Springer, Heidelberg (2006)
6. Bryans, J., Koutny, M., Mazaré, L., Ryan, P.Y.A.: Opacity generalised to transition systems. Int. J. Inf. Sec. 7(6), 421–435 (2008)
7. Bryans, J., Koutny, M., Ryan, P.Y.A.: Modelling dynamic opacity using petri nets with silent actions. In: Dimitrakos, T., Martinelli, F. (eds.) FAST 2004. IFIP, vol. 173, pp. 159–172. Springer, Boston (2004)
8. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical measurement of information leakage. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 390–404. Springer, Heidelberg (2010)
9. Clark, D., Hunt, S., Malacaria, P.: Quantitative analysis of the leakage of confidential data. Electronic Notes in Theoretical Computer Science 59 (2002)
10. Clark, D., Hunt, S., Malacaria, P.: Quantified interference for a while language. Electronic Notes in Theoretical Computer Science 112, 149–166 (2005)
11. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. J. Log. and Comput. 15(2), 181–199 (2005)
12. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15, 321–371 (2007)
13. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Belief in information flow. In: 18th IEEE Computer Security Foundations Workshop, Aix-en-Provence, France, pp. 31–45 (June 2005)

14. Clarkson, M.R., Myers, A.C., Schneider, F.B.: Quantifying information flow with beliefs. Journal of Computer Security (2007)
15. Denning, D.E.R.: Cryptography and Data Security. Addison-Wesley (1982)
16. Goguen, J., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20. IEEE Computer Society Press (1982)
17. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 286–296. ACM SIGSAC, ACM Press, New York, NY (2007)
18. Lakhnech, Y., Mazaré, L.: Probabilistic Opacity for a Passive Adversary and its Application to Chaum's Voting Scheme. Technical Report 4, Verimag (2005)
19. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing (preliminary report). In: POPL, pp. 344–352. ACM (1989)
20. Lin, J.: Divergence measures based on the shannon entropy. IEEE Transactions on Information Theory 37, 145–151 (1991)
21. Lowe, G.: Defining information flow quantity. Journal of Computer Security 12(3-4), 619–653 (2004)
22. Malacaria, P.: Assessing security threats of looping constructs. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 225–235. ACM Press, Nice (2007)
23. Mu, C., Clark, D.: An interval-baseed abstraction for quantifying information flow. ENTCS 59, 119–141 (2009)
24. Mu, C., Clark, D.: Quantitative analysis of secure information flow via probabilistic semantics. In: ARES, pp. 49–57 (2009)
25. Pierro, A.D., Hankin, C., Wiklicky, H.: Approximate non-interference. In: CSFW, pp. 3–17 (2002)
26. Shannon, C.E.: A mathematical theory of communication. SIGMOBILE Mob. Comput. Commun. Rev. 5(1), 3–55 (1948)
27. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)

Appendix: Proofs of Proposition 2, Theorem 4 and 5

Proposition 2. There is an integer $\kappa \geq 1$ and a real number $0 \leq \delta < 1$ such that, for every $i \geq 0$:

$$\mu(runs_{\kappa \cdot i}(PLTS)) \geq 1 - \delta^i \ .$$

Proof. In what follows, for every state $s \in S$, we denote by $PLTS_s$ the probabilistic labelled transition system obtained from $PLTS$ by setting the initial state to s.

In the first part of the proof, we assume that $PLTS$ satisfies the following two properties:

(i) $\mu(s) > 0$, for all $s \in S \setminus \{s_0\}$.
(ii) If $\mu(s_0) = 0$ then there is no transition $(s, l, s') \in \Delta$ such that $s' = s_0$.

We also define:

$$\delta = \begin{cases} \sup\{1 - \mu(s) \mid s \in S\} & \text{if } \mu(s_0) > 0 \\ \sup\{1 - \mu(s) \mid s \in S \setminus \{s_0\}\} & \text{otherwise .} \end{cases}$$

Note that $0 \leq \delta < 1$ is well-defined by Definition 1(i). Proceeding by induction on $i \geq 0$, will now show that, for every $i \geq 0$ and every $s \in S$:

$$\mu(runs_i(PLTS_s)) \geq \begin{cases} 1-\delta^i & \text{if } s = s_0 \text{ and } \mu(s_0) = 0 \\ 1-\delta^{i+1} & \text{otherwise .} \end{cases} \tag{10}$$

In the base case:

$$\mu(runs_0(PLTS_s)) = \mu(\epsilon) = \mu(s) \geq \begin{cases} 1-1 = 1-\delta^0 & \text{if } s = s_0 \text{ and } \mu(s_0) = 0 \\ 1-\delta = 1-\delta^1 & \text{otherwise .} \end{cases}$$

In the induction step, we assume that the thesis holds for i, and proceed as follows:

$$\begin{aligned} \mu(runs_{i+1}(PLTS_s)) &= \mu(s) \ + \sum_{(s,l_j,s_j)\in\Delta_s} \mu(s,l_j,s_j) \cdot \mu(runs_i(PLTS_{s_j})) \\ &= 1-\mu(\Delta_s) \ + \sum_{(s,l_j,s_j)\in\Delta_s} \mu(s,l_j,s_j) \cdot \mu(runs_i(PLTS_{s_j})) \ . \end{aligned}$$

By the induction hypothesis, we obtain the following (note that $s_j \neq s_0$, for every $(s,l_j,s_j) \in \Delta_s$):

$$\mu(runs_{i+1}(PLTS_s)) \geq 1-\mu(\Delta_s) \ + \sum_{(s,l_j,s_j)\in\Delta_s} \mu(s,l_j,s_j) \cdot (1-\delta^{i+1}) = 1-\delta^{i+1} \cdot \mu(\Delta_s) \ .$$

Now, if $s = s_0$ and $\mu(s_0) = 0$, then $\mu(\Delta_s) = 1$ and we get $\mu(runs_{i+1}(PLTS_s)) \geq 1-\delta^{i+1}$; otherwise, $\delta \geq \mu(\Delta_s)$ and we obtain: $\mu(runs_{i+1}(PLTS_s)) \geq 1-\delta^{i+2}$. Hence (10) holds.

In the second part of the proof, we transform $PLTS$ into a probabilistic labelled transition system $PLTS'$ satisfying (i) and (ii) above, in the following way:

- For every $s \in S$, we set $\mu'(s) = \mu(s)$.
- We create a fresh initial state s'_0 with $\mu'(s'_0) = \mu(s_0)$ and, for every sequence of transitions of $PLTS$ of the form:

  $$(s_0,l_1,s_1),(s_1,l_2,s_2)\ldots(s_{k-1},l_k,s_k)$$

  such that $0 = \mu(s_1) = \ldots = \mu(s_{k-1}) \neq \mu(s_k)$, we introduce a transition $(s'_0, l_1l_2\ldots l_k, s_k)$ and set:

  $$\mu'(s'_0, l_1l_2\ldots l_k, s_k) = \mu(s_0,l_1,s_1) \cdot \mu(s_1,l_2,s_2) \cdot \ldots \cdot \mu(s_{k-1},l_k,s_k) \ .$$

- For every sequence of transitions of $PLTS$ of the form:

  $$(s_1,l_1,s_2),(s_2,l_2,s_3)\ldots(s_k,l_k,s_{k+1})$$

  such that $\mu(s_1) \neq 0 = \mu(s_2) = \ldots = \mu(s_k) \neq \mu(s_{k+1})$, we introduce a transition $(s_1, l_1l_2\ldots l_k, s_{k+1})$ and set:

  $$\mu'(s_1, l_1l_2\ldots l_k, s_{k+1}) = \mu(s_1,l_1,s_2) \cdot \mu(s_2,l_2,s_3) \cdot \ldots \cdot \mu(s_k,l_k,s_{k+1}) \ .$$

Note that by Definition 1(ii), there is the largest k as above, denoted by k_{max}. We then delete all the states $s \in S$ with $\mu(s) = 0$ together with the adjacent arcs. Thanks to Definition 1(ii), $PLTS'$ is a well-defined probabilistic labelled transition system whose labels are finite sequences of labels from $PLTS$,

$$runs(PLTS') = \{\lambda \in runs(PLTS) \mid \mu(\lambda) > 0\} ,$$

and $\mu(\lambda) = \mu'(\lambda)$, for all $\lambda \in runs(PLTS')$. Moreover, we can apply (10) to $PLTS'$ and conclude that, for every $i \geq$:

$$\mu(runs_i(PLTS')) \geq 1 - \delta^i .$$

Thus, by:

$$runs_i(PLTS') \subseteq \{\lambda \in runs_{(k_{max}+1)\cdot i}(PLTS) \mid \mu(\lambda) > 0\} ,$$

we have that $\mu(runs_{\kappa\cdot i}(PLTS)) \geq 1 - \delta^i$ for $\kappa = k_{max} + 1$. □

Theorem 4. Let ϕ be $\overline{\pi}_\gamma$-opaque. If obs is inversely M-efficient for $PLTS$ and ϕ, then there is an integer $\rho \geq 1$ and a real number $0 \leq \nu < 1$ such that, for every $i \geq 0$:

$$|(\mu(\phi^{cov}) - \gamma \cdot \mu(\phi)) - (\mu(\phi^{cov}_{\rho\cdot i}) - \gamma \cdot \mu(\phi_{\rho\cdot i}))| \leq (1+\gamma) \cdot \nu^i .$$

Proof. By Proposition 2, there exists a positive integer κ and a real number $0 \leq \delta < 1$ such that, for every $i \geq 0$:

$$\mu(runs(PLTS) \setminus runs_{\kappa\cdot i}(PLTS)) \leq \delta^i . \tag{11}$$

Let us now take any $i \geq 0$, and consider:

$$x = \mu(\phi^{cov}) - \mu(\phi^{cov}_{M\cdot\kappa\cdot i}), \quad y = \mu(\phi) - \mu(\phi_{M\cdot\kappa\cdot i}) .$$

We then observe that, by obs being inversely M-efficient, we have:

$$\begin{aligned}
x &= \mu(\overline{\phi} \cap obs^{-1}(obs(\phi))) - (\mu(\overline{\phi}_{\kappa\cdot i} \cap obs^{-1}(obs(\phi_{M\cdot\kappa\cdot i}))) \\
&\quad + \ \mu((\overline{\phi}_{M\cdot\kappa\cdot i} \setminus \overline{\phi}_{\kappa\cdot i}) \cap obs^{-1}(obs(\phi_{M\cdot\kappa\cdot i})))) \\
&= \mu(\overline{\phi} \cap obs^{-1}(obs(\phi))) - (\mu(\overline{\phi}_{\kappa\cdot i} \cap obs^{-1}(obs(\phi))) \\
&\quad + \ \mu((\overline{\phi}_{M\cdot\kappa\cdot i} \setminus \overline{\phi}_{\kappa\cdot i}) \cap obs^{-1}(obs(\phi_{M\cdot\kappa\cdot i})))) \\
&= \mu((\overline{\phi} \setminus \overline{\phi}_{\kappa\cdot i}) \cap obs^{-1}(obs(\phi))) - \mu((\overline{\phi}_{M\cdot\kappa\cdot i} \setminus \overline{\phi}_{\kappa\cdot i}) \cap obs^{-1}(obs(\phi_{M\cdot\kappa\cdot i})))
\end{aligned}$$

$$y = \mu(\phi \setminus \phi_{M\cdot\kappa\cdot i}) \ \leq \ \mu(\phi \setminus \phi_{\kappa\cdot i}) .$$

We therefore obtain from (11) that $|x| \leq \delta^i$ and $y \leq \delta^i$. Consequently, we obtain:

$$|x - \gamma \cdot y| \leq (\gamma + 1) \cdot \delta^i .$$

Hence the result holds with $\rho = M \cdot \kappa$ and $\nu = \delta$. □

Theorem 5. Let *obs* be K, L-bounded for *PLTS* and ϕ, and κ and $0 \leq \delta < 1$ be as in Proposition 2. Then there is $\alpha > 0$ such that, for every $i \geq 0$:

$$0 \quad \leq \quad D_{JS}(w \cdot \Pi_\phi, w' \cdot \Pi_{\phi^{cov}}) - \sum_{o \in \mathcal{O}_{\kappa \cdot L \cdot i}} d_o^{\kappa \cdot L \cdot K \cdot i} \quad \leq \quad \alpha \cdot \delta^i \,. \tag{12}$$

Proof. Let d_o be the individual contribution of each $o \in \mathcal{O}$ to $D_{JS}(w \cdot \Pi_\phi, w' \cdot \Pi_{\phi^{cov}})$ as defined in (1). By the first part of K, L-boundedness, we obtain $d_o = d_o^{\kappa \cdot L \cdot K \cdot i}$, for every $o \in \mathcal{O}_{\kappa \cdot L \cdot i}$. This and (2) yields:

$$\begin{aligned} 0 \leq D_{JS}(w \cdot \Pi_\phi, w' \cdot \Pi_{\phi^{cov}}) - \sum_{o \in \mathcal{O}_{\kappa \cdot L \cdot i}} d_o^{\kappa \cdot L \cdot K \cdot i} \quad &= \sum_{o \in \mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i}} d_o \\ \leq c \cdot (\Pi_\phi(\mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i}) + \Pi_{\phi^{cov}}(\mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i})) \,. \end{aligned}$$

Now, by the second part of K, L-boundedness, we have that

$$obs^{-1}(\mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i}) \subseteq runs_{\kappa \cdot i}(PLTS).$$

Hence, by Proposition 2, we obtain

$$\Pi_\phi(\mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i}) \leq \frac{\delta^i}{\mu(\phi)} \quad \text{and} \quad \Pi_{\phi^{cov}}(\mathcal{O} \setminus \mathcal{O}_{\kappa \cdot L \cdot i}) \leq \frac{\delta^i}{\mu(\phi^{cov})}.$$

As a result, (12) holds with $\alpha = c \cdot \left(\frac{1}{\mu(\phi)} + \frac{1}{\mu(\phi^{cov})}\right)$. □

An Algebra for Symbolic Diffie-Hellman Protocol Analysis

Daniel J. Dougherty and Joshua D. Guttman*

Worcester Polytechnic Institute
{dd,guttman}@wpi.edu

Abstract. We study the algebra underlying symbolic protocol analysis for protocols using Diffie-Hellman operations. Diffie-Hellman operations act on a cyclic group of prime order, together with an exponentiation operator. The exponents form a finite field: this rich algebraic structure has resisted previous symbolic approaches.

We define an algebra that validates precisely the equations that hold almost always as the order of the cyclic group varies. We realize this algebra as the set of normal forms of a particular rewriting theory.

The normal forms allow us to define our crucial notion of *indicator*, a vector of integers that summarizes how many times each secret exponent appears in a message. We prove that the adversary can never construct a message with a new indicator in our adversary model. Using this invariant, we prove the main security goals achieved by UM, a protocol using Diffie-Hellman for implicit authentication.

Despite vigorous research in symbolic analysis of security protocols, many limitations remain. While systems such as NPA-Maude [21], ProVerif [8], AVISPA [3, 5], CPSA [36], and Scyther [16] are extremely useful, great ingenuity is still needed—as for instance in [31]—for the analysis of protocols that use fundamental cryptographic ideas such as Diffie-Hellman key agreement [17], henceforth, DH. Moreover, important protocols, such as the implicitly authenticated key-agreement protocol MQV [7], appear to be out of reach of known symbolic techniques. Indeed, for these protocols, computational techniques have led to arduous proofs after which controversy remains [27, 29, 30, 33]. In this paper, we develop algebraic ideas that allow us to give rigorous proofs of security goals such as authentication and confidentiality in a symbolic model. Moreover, our techniques also help identify the security goals that the protocol does not achieve.

DH protocols work in a cyclic group of prime order q, which we will write multiplicatively, using an agreed-upon generator g. For a particular session, A and B choose random values x, y respectively, raising a base g to these scalar powers:

$$A, x \qquad \bullet \xrightarrow{g^x} \qquad \xleftarrow{g^y} \bullet \qquad B, y \tag{1}$$

* We gratefully acknowledge support by the National Science Foundation under grant CNS-0952287.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 164–181, 2013.
© Springer-Verlag Berlin Heidelberg 2013

They can then each compute the value $(g^y)^x = g^{xy} = (g^x)^y$ as a new shared secret for A, B. The Decisional Diffie-Hellman assumption (DDH) says that, in suitable groups, any observer who has observed neither x nor y, cannot distinguish g^{xy} from the g^z we would get from a randomly chosen z.

This basic protocol—while secure against a passive adversary, who observes messages, but can neither create them nor alter (or misdirect) messages of compliant principals—is, however, vulnerable to an active attacker. The adversary chooses his own values w, g^w, substituting g^w for the values each participant should receive. Then the two participants will end up with different keys, g^{xw} and g^{yw}, unfortunately each shared with the attacker.

One idea to avoid this man-in-the-middle attack is for each of the principals A and B to maintain a long-term secret value. We will write A's long term secret as a, and B's as b. They publish the long term public values $Y_A = g^a, Y_B = g^b$, having a certificate authority certify the bindings to A and B. Now any pair of participants may each use the long term public value of the other—and their own long term secrets—to compute the same fresh secret, in such a way that no principal other than A or B can. The "Unified Model" UM of Ankney, Johnson, and Matyas [2] is an example. A and B send only the messages shown in Eqn. 1. For clarity, the value B receives, purportedly from A, will be called R_A. A receives the value R_B, purportedly from B. Without adversary interference, $R_A = g^x$ and $R_B = g^y$. Letting $\mathsf{h}(x)$ be a hash function, A and B compute their keys:

$$A: \; k = \mathsf{h}({Y_B}^a \parallel {R_B}^x) \quad B: \; k = \mathsf{h}({Y_A}^b \parallel {R_A}^y), \tag{2}$$

obtaining the shared value $\mathsf{h}(g^{ab} \parallel g^{xy})$ if $R_A = g^x$ and $R_B = g^y$. We will present a technique for proving authentication and confidentiality results about protocols such as this.

The heart of this paper develops a well-behaved rewriting theory for DH values, which yields a powerful tool for symbolic analysis. The challenge for such a theory derives from the fact that, since we are operating in a cyclic group of prime order, the exponents form a *field*. Although UM uses only the field multiplication, some protocols (including MQV) also use the field addition. This is challenging for rewriting-based approaches to protocol analysis since the theory of fields does not admit an axiomatization using equations, or even conditional equations. The standard axiomatization uses negation to say that 0 has no multiplicative inverse; to see that there can be no conditional-equational axiomatization, note that the category of fields is not closed under products. This paper makes the following contributions:

1. We define an order-sorted equational theory $\mathsf{AG}\hat{}$ whose models include all fields. We equip $\mathsf{AG}\hat{}$ with a rewrite system modulo associativity and commutativity (AC), and show that this system is terminating and confluent modulo AC: an equation $s = t$ is derivable in $\mathsf{AG}\hat{}$ if and only if s and t rewrite to the same normal form modulo AC. The free algebra over this rewrite system offers a natural DH message algebra. (Section 1.)
2. We show, via a model-theoretic argument using ultraproducts, that $\mathsf{AG}\hat{}$ captures uniform equality in the theory of finite fields. Namely, if $s = t$ is an

equation that is valid in the field $\mathbb{F}_q$ of characteristic q for infinitely many q, then $\mathsf{AG}\hat{}$ proves $s = t$. In particular, $\mathsf{AG}\hat{}$ proves every equation that is valid in $\mathbb{F}_q$ aymptotically as q increases. (Section 2.)

3. We use $\mathsf{AG}\hat{}$ to prove Thm. 12, the *indicator theorem*, a symbolic analogue to the computational Diffie-Hellman assumption (CDH). It states that the adversary cannot obtain a new exponentiated value t^{xy} without access either to x, or to y, or to some value that already included t^{xy}. Thm. 12 gives a proof method in $\mathsf{AG}\hat{}$ that avoids unification. (Section 3.)
4. We apply the indicator theorem within the strand space framework (introduced in Section 4) to prove that UM meets its authentication and confidentiality goals (construed as trace properties). We also explain why it does not meet another goal, resisting impersonation attacks. (Section 5.)

Elsewhere, we apply our method to more challenging protocols, e.g. MQV [18].

Related Work. Within the symbolic model, there has been substantial work on some aspects of DH, starting with Boreale and Buscemi [9], which provides a symbolic semantics [1, 22, 34] for a process calculus with algebraic operations for DH. Their symbolic semantics is based on unification.

Indeed, symbolic approaches to protocol analysis have relied on unification as a central part of their reasoning. Goubault-Larrecq, Roger, and Verma [24] use a method based on Horn clauses and resolution modulo AC, providing automated proofs of passive security. Maude-NPA [20, 21] is also usable to analyze many protocols involving DH, again depending heavily on unification. Tamarin [15] offers a new approach to analysis, also relying on unification.

All of these approaches model the multiplication in the exponents, but do not explicitly model the addition. This suffices for many protocols, but not for protocols such as Menezes-Qu-Vanstone MQV [7] and Cremers-Feltz CF [14], in which the ring structure in the exponents is used in the protocol definition. Indeed, even in protocols which use only the multiplicative structure, the adversary may choose to use the ring or field properties. The richer theory is needed to prove no new attacks can arise.

This field structure combines poorly with the heavy reliance of previous approaches on unification. Unifiability is undecidable in the theory of rings, by the unsolvability of Hilbert's tenth problem. There are, however, many related theories for which undecidability is not known, for instance the diophantine theory of the rationals [6]; see the beautiful paper by Kapur, Narendran, and Wang [28].

Küsters and Truderung [31] finesse this issue by rewriting protocol analysis problems. The original problems use an AC theory involving exponentiation. They transform it into a corresponding problem that does not require the AC property, and so can work using standard ProVerif resolution [8]. Their approach covers a surprising range of protocols, although, like [13], not Implicitly Authenticated Diffie-Hellman protocols such as MQV.

Another contrast between this paper and previous work is our uniform treatment of security goals (see Figs. 2–3). Our methods are applicable to confidentiality, authentication, and further properties such as forward secrecy.

Meadows and Pavlovic [35], cf. [11], do not explicitly represent the algebra. Instead, they offer a family of authentication axioms. Each axiom in the family expresses a limitation on the adversary by saying that some receptions can be only explained only by actions of regular principals. Such an axiom may be justified by a computational principle such as CDH. While this method leads to illuminating results, it appears to sidestep a foundational question about the algebraic structures in which these axioms are satisfied. our paper is a complementary attempt to fill in information about these models.

Our adversary model is active. For passive attacks, there has been some work on computational soundness for Diffie-Hellman, with Bresson et al. [10] giving an excellent treatment.

1 An Equational Theory of Messages

By *DH-structure* we mean a cyclic group G of prime order q, together with an exponentiation operator. The exponents E are integers modulo the prime q, which form a field of characteristic q. In cryptographic applications G is often taken to be a subgroup of the multiplicative group of integers modulo a prime p, where q divides $p-1$; sometimes G is a prime-order subgroup of the group of points over an elliptic curve.

Our challenge is to define an equational theory that captures the relevant algebra of DH structures, with a notion of reduction that supports modeling messages as normal forms. By the Decisional Diffie-Hellman assumption, an adversary *cannot* retrieve the exponent x from a value g^x that a regular participant has constructed. Our formalism reflects this limitation by not including a logarithm function in the signature of DH-structures.

Our strategy for handling the fact that the field of exponents in a DH structure cannot be axiomatized by equations is as follows. We work with a sort G for base-group elements and a sort E for exponents. The novelty is that we enrich E by adding a subsort NZE. Its intended interpretation is the non-0 elements of E, and it does not include 0 in any interpretation.

The device of approximating "non-zero" reflects a philosophy of capturing uniform capabilities algebraically. For instance no term which is a sum $e_1 + e_2$ is syntactically of sort NZE because each finite field has finite characteristic and so there are instantiations of the variables in $e_1 + e_2$ driving the term to 0. On the other hand, we will want to ensure that NZE is closed under multiplication; this is the role of the operator $**$ below.

We show in this section that $\mathsf{AG}^{\wedge}$ admits a confluent and terminating notion of reduction. In section 2 we prove Thm. 9 that describes the sense in which $\mathsf{AG}^{\wedge}$ captures the equalities that hold in almost all finite prime fields.

Definition 1. *The order-sorted signature $\Sigma(\mathsf{AG}^{\wedge})$ has the sorts G, E, and NZE, with NZE a subsort of E with operators:*

$$
\begin{array}{lccc}
 & \cdot : G \times G \to G & id :\to G & inv : G \to G \\
+,\ -,\ * & : E \times E \to E & 0 :\to E & exp : G \times E \to E \\
 & i : NZE \to NZE & 1 :\to NZE & ** : NZE \to NZE
\end{array}
$$

and axioms (writing $exp(t, e)$ as t^e):

1. $(G, \cdot, inv, id)$ *is an abelian group;*
2. $(E, +, 0, -, *, \mathbf{1})$ *is a commutative ring with identity;*
3. *Exponentiation makes G a right E-module with identity, i.e.*
$$(a^x)^y = a^{x*y} \qquad a^1 = a \qquad id^x = id$$
$$(a \cdot b)^x = a^x \cdot b^x \qquad a^{(x+y)} = a^x \cdot a^y$$
4. *Multiplicative inverse, closure at sort NZE:*
$$u ** v = u * v \qquad u * i(u) = \mathbf{1} \qquad i(-u) = -i(u)$$
$$i(u * v) = i(u) * i(v) \qquad i(1) = 1 \qquad i(i(w)) = w$$

We extract an AC rewrite system from $\mathsf{AG}^{\hat{}}$ by orienting the non-AC equations, using additional equations derivable from $\mathsf{AG}^{\hat{}}$ to join critical pairs:

Definition 2. *Let R be the set of rewrite rules given by the natural orientation of the equations in Definition 1, other than associativity and commutativity, together with the additional rules presented in Table 1. The rewrite relation $\rightarrow_{\mathsf{AG}^{\hat{}}}$ is rewriting with R modulo the associativity and commutativity equations.*

Theorem 3. *The reduction $\rightarrow_{\mathsf{AG}^{\hat{}}}$ is terminating and confluent modulo AC.*

Proof. Termination can be established using the AC-recursive path order defined by Rubio [37] with a precedence in which exponentiation is greater than inverse, which is in turn greater than multiplication (and 1). This has been verified with the Aprove termination tool [23].

Then confluence follows from local confluence, which is established via a verification that all critical pairs are joinable. This result has been confirmed with the Maude Church-Rosser Checker [19]. □

Terms that are irreducible with respect to $\rightarrow_{\mathsf{AG}^{\hat{}}}$ are called *normal forms*. The following taxonomy of the normal forms will be crucial in what follows, most of all in the definition of indicators, Definition 10. The proof is a routine simultaneous induction over the size of e and t. By G-variables and E-variables, we mean variables of those types.

Table 1. Additional rewrite rules for $\rightarrow_{\mathsf{AG}^{\hat{}}}$

At sort G			At sort E		
$inv(id)$	$\rightarrow$	id	$-(0)$	$\rightarrow$	0
$inv(a \cdot b)$	$\rightarrow$	$inv(a) \cdot inv(b)$	$-(x + y)$	$\rightarrow$	$-(x) + (-(y))$
$inv(inv(b))$	$\rightarrow$	b	$-(-(x))$	$\rightarrow$	x
$(inv(a))^x$	$\rightarrow$	$inv(a^x)$	$0 * x$	$\rightarrow$	0
a^0	$\rightarrow$	id	$-(x) * y$	$\rightarrow$	$-(x * y)$
$a^{-(x)}$	$\rightarrow$	$inv(a^x)$			

Lemma 4. *1. If $e : E$ is a normal form then e is a sum $m_1 + \ldots + m_n$ where (i) each m_i is of the form $\pm(e_1 * \ldots * e_k)$ where $k \geq 0$, (ii) no e_i is of the form $i(e_j)$, and (iii) each e_i is one of x and $i(x)$, with x an E-variable. When $n = 0$, e is the ring element 0; when $k = 0$, m_i is the ring element 1. We call terms of the form $\pm m_i$* irreducible monomials.

2. If $t : G$ is a normal form then t is a product $t_1 \cdot \ldots \cdot t_n$, for $n \geq 0$ where (i) no t_i is of the form $inv(t_j)$, and (ii) each t_i is one of: v, $inv(v)$, v^e, $inv(v^e)$, with v a G-variable, and $e : E$ an irreducible monomial. When $n = 0$, $t = id$.

2 Uniform Equality and the Completeness of AGˆ

In this section we justify the use of AGˆ, specifically the use of AGˆ-normal forms to model messages. Since the axioms of AGˆ are clearly true in all DH-structures, any theorem of AGˆ holds in all DH-structures. Theorem 9 gives us a strong converse, namely that every equation that holds in infinitely many DH-structures is a theorem of AGˆ. If fact we show how to construct a single structure $\mathcal{M}_D$ that is "generic" for all DH-structures: An equation $s = t$ is holds in $\mathcal{M}_D$ if and only if it holds in infinitely many DH-structures.

Algebraically isomorphic DH-structures can have very different *computational* properties. Indeed, the prime field $\mathbb{F}_q$ presented as the group of integers mod q can be viewed as a DH-structure where the base group is the *additive* group of $\mathbb{F}_q$ and exponentiation is multiplication. The discrete log problem in this structure is computationally tractable. However, $\mathbb{F}_q$ is isomorphic to a subgroup of order q of the *multiplicative* group of integers modulo some prime p. There, the discrete log problem may be intractable. We focus on algebraic equations between terms in DH-structures; the absence of the log operator in our signature models the fact that our intended models are those in which discrete log is intractable.

First, we observe that the field of scalars, i.e. the exponents, carries all the algebraic information in a model of AGˆ.

Definition 5. *Let F be a field. We define the model $\mathcal{M}_F$ of theory* AGˆ *to be as follows. The sorts E and G are each interpreted as the domain of F; the sort NZE is interpreted as the set of non-0 elements of E. The operations of E are interpreted just as in F itself. The group operation $\cdot$ in G is taken to be $+$ from E, thus id and inv are taken to be 0 and $-$. Exponentiation is multiplication: a^e is interpreted as $a * e$.*

For each field F, $\mathcal{M}_F$ satisfies all of the equations in AGˆ. It is easy to check the following.

Lemma 6. *Every DH-structure is isomorphic to some $\mathcal{M}_{\mathbb{F}_q}$, where F is the prime field of order q.*

The key device for reasoning about uniform equality across DH-structures is the notion of *ultraproduct*, cf. e.g. [12]. We let the variable D range over non-principal ultrafilters over the set of prime numbers. The crucial facts about

ultraproducts for our purposes are: (i) a first-order sentence is true in an ultraproduct if and only if the set of indices at which it is true is a set in D; (ii) every infinite set belongs to some non-principal ultrafilter; (iii) when D is non-principal, every set whose complement is finite is in D.

Definition 7. *Let D be a non-principal ultrafilter over the set of prime numbers and let $\mathbb{F}_D$ be the ultraproduct structure $\prod_D\{\mathbb{F}_q \mid q \text{ prime}\}$. $\mathcal{M}_{\mathbb{F}_D}$ is the DH structure obtained from $\mathbb{F}_D$ via Definition 5. For brevity we write $\mathcal{M}_D$ for $\mathcal{M}_{\mathbb{F}_D}$.*

$\mathbb{F}_D$ is a field, since each $\mathbb{F}_q$ satisfies the first-order axioms for fields, and has characteristic 0, since each equation $1 + \ldots + 1 = 0$ is false in all but finitely many $\mathbb{F}_q$.

When F is the additive group of rational numbers then $\mathcal{M}_F = \mathcal{M}_{\mathbb{Q}}$ is of special interest to us. The proof of the following lemma is in Appendix A.

Lemma 8. *1. The structure $\mathcal{M}_{\mathbb{Q}}$ can be embedded as a submodel in any $\mathcal{M}_D$.*
2. If s and t are distinct normal forms then it is not the case that $\mathcal{M}_{\mathbb{Q}} \models s = t$.

Our main result is that $\mathsf{AG}\hat{}$ is complete for uniform equality, in the following sense:

Theorem 9. *For each pair of G-terms s and t, the following are equivalent*

1. $\mathsf{AG}\hat{} \vdash s = t$
2. *For all q, $\mathcal{M}_{\mathbb{F}_q} \models s = t$*
3. *For all non-principal D, $\mathcal{M}_D \models s = t$*
4. *For infinitely many q, $\mathcal{M}_{\mathbb{F}_q} \models s = t$*
5. *For some non-principal D, $\mathcal{M}_D \models s = t$*
6. $\mathcal{M}_{\mathbb{Q}} \models s = t$
7. *If s reduces to s' and t reduces to t', with s', t' irreducible, then s' and t' are identical modulo associativity and commutativity of $\cdot$, $+$, and $*$.*

Proof. It suffices to establish the cycle of entailments 1 implies 2 ... implies 7 implies 1. The first four of these steps are immediate, as is the fact that 7 implies 1. The fact that 5 implies 6 follows from Lemma 8, item 1. To conclude 7 from 6, use Lemma 8, item 2. □

The results of Theorem 9 hold as well for equations between E-terms. Given terms e and e', form the equation $g^e = g^{e'}$. It is provable iff $e = e'$ is provable, and is true in a given model $\mathcal{M}$ iff $e = e'$ is.

The model $\mathcal{M}_{\mathbb{Q}}$ is convenient: this single model, based on a familiar structure, witnesses uniform equality faithfully. The models $\mathcal{M}_D$ satisfy another striking property. It follows from results of Ax [4] that a first-order sentence in the language of rings/fields is true in a given $\mathcal{M}_D$ if it is in the set of sentences true in all but a finite set of finite fields. Moreover this theory is decidable. So the structures $\mathcal{M}_D$ are attractive for closer study of the "uniform" properties of DH-structures.

3 Indicators

We turn now to a formal definition of indicators and the proof of a key invariant that all adversary actions preserve. For intuition about the following definition, think of N as being a set of secret values in a protocol run (such as A's x) not transmitted by any participant (although a related value such as g^x may be transmitted). Say that a monomial m is a *maximal-monomial* of t if t has a subterm of the form b^m.

Definition 10 (Indicators). *Let $N = \langle v_1, \ldots, v_d \rangle$ be a vector of NZE-variables. If m is an irreducible monomial, the N-*vector *for m is $\langle z_1, \ldots, z_k \rangle$ where z_i is the multiplicity of v_i in m, counting occurrences of $i(v_i)$ negatively.*

*An E-term $e = m_1 + \ldots + m_k$ is N-*free *if each m_i has N-vector $\langle 0, \ldots, 0 \rangle$.*

If t is irreducible, then $\mathrm{Ind}_N(t)$ is the set of all vectors $\boldsymbol{z}$ such that $\boldsymbol{z}$ is the N-vector of m, where m is a maximal-monomial subterm of t.

Example: For $N = \langle x, y \rangle$, $\mathrm{Ind}_N(\, g^{x\, i(y)} \cdot g^{zxy} \cdot g^{xx}\,) = \{\langle 1, -1 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle\}$.

If e is N-free, then $\mathrm{Ind}_N(t^e) = \mathrm{Ind}_N(t)$, because no new occurrences of N-variables are created in passing from t to t^e.

Definition 11. *Let $T = \{t_1, \ldots, t_k\}$ be a set of terms. The set* $\mathrm{Gen}(T)$ generated *by T is the least set of terms including T and closed under applications of function symbols.*

Functions cannot cancel to reveal a $v_i \in N$, which leads to our main theorem.

Theorem 12 (Indicator Theorem). *Let N be a vector of NZE-variables and let T be a set of terms where each $e : E \in T$ is N-free. Then*

1. *every $e \in \mathrm{Gen}(T)$ of sort E is N-free, and*
2. *if $u \in \mathrm{Gen}(T)$ is of sort G and $\boldsymbol{z} \in \mathrm{Ind}_N(u)$, then for some $t \in T$, $\boldsymbol{z} \in \mathrm{Ind}_N(t)$.*

Proof. By induction on operations used to construct terms from elements of T.

The main cases are for 2., when (i) $u = u_1 \cdot u_2$ or (ii) $u = t^e$, where t, u_1, u_2 and e are irreducible terms in $\mathrm{Gen}(T)$. First, if $u = u_1 \cdot u_2$, then u is a product $t_1 \cdot \ldots \cdot t_n$, and each factor t_i is of the form $v, \mathit{inv}(v), v^e$, or $\mathit{inv}(v^e)$ and comes from u_1 or u_2. Thus, the normal form of this term results by canceling any pair of factors, one from u_1 and one from u_2 that are inverses of each other. No new E-subterms are created, so no new indicator vectors are created, and our assertion holds.

Otherwise $u = t^e$. Since e is in $\mathrm{Gen}(T)$, we know inductively that e is N-free. It suffices to show that $\mathrm{Ind}_N(t^e) = \mathrm{Ind}_N(t)$. Letting t be in normal form, t^e is $(t_1)^e \cdot \ldots \cdot (t_n)^e$. However, as we just observed, $\mathrm{Ind}_N(t_i^e) = \mathrm{Ind}_N(t_i)$. □

This "conservation of indicators" principle essentially restricts adversary behavior; Theorem 15 below makes this precise in the strand-space setting.

4 Strands and Indicators

We will now adapt the strand space theory [25,38] to the case where the messages include a free algebra over $\mathsf{AG}^\wedge$. A *strand* is a sequence of local actions called *nodes*, each of which is:

- a message *transmission*, written $\bullet \rightarrow$;
- a message *reception*, written $\bullet \leftarrow$; or
- a *neutral* node $\circ$. Neutral nodes are local events in which a principal consults or updates its local state [26].

If n is a node, and the message t is transmitted, received, or coordinated with the state on n, then we write $t = \mathsf{msg}(n)$. We sometimes write $+t = \mathsf{msg}(n)$ and $-t = \mathsf{msg}(n)$ when n is respectively a transmission or reception node. Double arrows indicate successive events on the same strand, e.g. $\circ \Rightarrow \bullet \Rightarrow \bullet$.

A *protocol* Π is a set of strands, called the *roles* of the protocol. We assume every protocol contains a specific role, called the *listener* role, consisting of a single reception node $n = \!\!\rightarrow \bullet$. Listener strands provide "witnesses" when $\mathsf{msg}(n)$ has been disclosed, aiding in specifying confidentiality properties. A *regular* strand for Π means an instance of one of the roles of Π.

Adversary strands consist of zero or more reception nodes followed by one transmission node. The adversary obtains the transmitted value as a function of the values received; or creates it, if there are no reception nodes. All values that the adversary handles are received or transmitted; none are silently obtained from long-term state. Allowing the adversary to use neutral nodes—or strands of other forms—provides no additional power. (See Defn. 13.)

Messages. The messages transmitted and received on $\bullet$ nodes, and obtained from long-term state on neutral nodes $\circ$, form an abstract algebra. The message algebra MA includes as basic values:

- Elements of the free algebra over $\mathsf{AG}^\wedge$ built from the infinite sets of E-variables $\mathcal{V}^E$ and G-variables $\mathcal{V}^G$; we denote this algebra by $\mathsf{Free}(\mathsf{AG}^\wedge)$,
- Disjoint infinite sets of *names*, *symmetric* and *asymmetric keys*, and *texts*.

The elements of the algebra $\mathsf{Free}(\mathsf{AG}^\wedge)$ are equivalence classes of terms. However, the results in Section 1 say that each class has a canonical representative, namely an AC normal form modulo $\rightarrow_{\mathsf{AG}^\wedge}$. This justifies a syntactic approach, particularly in our treatment of indicators in Thm. 15.

We assume that some of the asymmetric keys are of the form $\mathsf{pk}(A)$ and $\mathsf{vk}(A)$, where A ranges over names, denoting the public encryption and signature verification key of A. We also assume that asymmetric keys are equipped with an inverse operation; for instance, $\mathsf{pk}(A)^{-1}$ is A's private decryption key.

The *parameters* of an $\mathsf{AG}^\wedge$ normal form are the $\mathcal{V}^E$ and $\mathcal{V}^G$ variables occurring in it. The parameter of a value $\mathsf{pk}(A)$ or $\mathsf{vk}(A)$ is A. For all other basic values a, the parameter of a is a. MA is closed under the constructors:

- Pairing, where the pair of t_1 and t_2 is written $t_0 \parallel t_1$;
- Encryption, where the encryption of t_0 using t_1 as key is written $\{\!|t_0|\!\}_{t_1}$.

As constructors, the operations are free, yielding equal results only when the arguments are equal: $\{|t_0|\}_{t_1} = \{|t_2|\}_{t_3}$ implies $t_0 = t_2$ and $t_1 = t_3$, etc. We regard hashes and digital signatures as coded using (deterministic) encryption: the hash $\mathsf{h}(t) = \{|t|\}_{K_0}$, where K_0 is an asymmetric encryption key to which no one knows the inverse. We will always assume that K_0^{-1} is uncompromised. The digital signature $[\![t_0]\!]_{t_1}$ can be encoded as $t_0 \parallel \{|t_0|\}_{t_1}$.

The parameters of a pair, encryption, digital signature, or hash are the union of the parameters of its immediate subterms.

A parameter represents a "degree of freedom" in describing executions, which can be instantiated or restricted. It may also represent an independent choice, as A's choice of a group element x to build g^x is independent of B's choice of y.

Ingredients and Origination. A value t_1 is an *ingredient* of another value t_2, written $t_1 \sqsubseteq t_2$, if t_1 contributes to t_2 via concatenation or as the plaintext of encryptions: $\sqsubseteq$ is the least reflexive, transitive relation such that:

$$t_1 \sqsubseteq t_1 \parallel t_2, \qquad t_2 \sqsubseteq t_1 \parallel t_2, \qquad t_1 \sqsubseteq \{|t_1|\}_{t_2}.$$

By this definition, $t_2 \sqsubseteq \{|t_1|\}_{t_2}$ implies that (anomalously) $t_2 \sqsubseteq t_1$. For basic values a, b, we have $a \sqsubseteq b$ iff $a = b$. Thus, the ingredient relation is much coarser than the "occurs in" relation.

A value t *originates* on a transmission node n if $t \sqsubseteq \mathsf{msg}(n)$, so that it is an ingredient of the message sent on n, but it was not an ingredient of any message earlier on the same strand. That is, $m \Rightarrow^+ n$ implies $t \not\sqsubseteq \mathsf{msg}(m)$.

A basic value is *uniquely originating* in a bundle $\mathcal{B}$ if there is exactly one $n \in \mathsf{node}(\mathcal{B})$ at which it originates. Freshly chosen nonces or DH values g^x are typically assumed to be uniquely originating. A basic value is *non-originating* if there is no $n \in \mathsf{node}(\mathcal{B})$ at which it originates. An uncompromised long term secret (e.g. a private decryption key) is assumed to be non-originating. Because adversary strands receive their arguments as incoming messages, an adversary strand that decrypts a message receives its key as a message, which must originate somewhere. The set of non-originating values is denoted non; the set of uniquely originating values is denoted unique.

In DH protocols unique origination and non-origination are used in tandem. When a compliant principal generates a random x and transmits g^x, the former will be non-originating and the latter uniquely originating. A probabilistic implementation of the (non-probabilistic) unique- and non-origination randomly chooses values from large sets, with overwhelming probability of faithfulness.

Adversary Model. The adversary strands are defined:

Definition 13. *1. A strand $+a$, having one transmission node, is an adversary strand if a is a parameter or a constant $id, 1, 0$.*

2. A strand $-t \Rightarrow +f(t)$, having a reception node and a transmission node, is an adversary strand if f is any of the unary functions $inv, i, -, \mathsf{pk}, \mathsf{sk}, \mathsf{h}$.

3. A strand $-t_1 \Rightarrow -t_2 \Rightarrow +g(t_1, t_2)$, having two reception nodes and a transmission node, is an adversary strand if g is any of the binary functions $\cdot, *, +, \cdot \parallel \cdot, \{|\cdot|\}_\cdot, [\![\cdot]\!]_\cdot$.

4. *A strand* $-\{|t_1|\}_K \Rightarrow -K^{-1} \Rightarrow +t_1$ *is an adversary strand.*

Importantly, there is no adversary strand executing the asymmetric key inverse function K^{-1}, nor any logarithm operation.

This adversary model suggests a game between adversary and system:

1. The system chooses a security goal Φ, involving secrecy, authentication, key compromise, etc., as in Figs. 2–3.
2. The adversary proposes a potential counterexample $\mathbb{A}$ consisting of regular strands with equations between values on the nodes, e.g. an equation between session keys as computed by two participants.
3. For each message reception node in $\mathbb{A}$, the adversary chooses a recipe, intended to produce an acceptable message, using the strands of Def. 13. The adversary may use earlier transmission events on regular strands to build messages for subsequent reception events.
   These recipes determine a set of equalities between the values computed by the adversary and the values t "expected" by the recipient (i.e. acceptable to the recipient). They are the *adversary's proposed equations.*
4. The adversary wins if his proposed equations are valid in $\mathcal{M}_{\mathbb{F}_q}$, for infinitely many primes q; or equivalently, by Theorem 9, valid for all primes q.

This game may seem too challenging for the adversary. First, it wins only if the equations are valid, i.e. true for all instances of the variables. Second, the adversary must choose how to generate all the messages, its adversary strategy, before seeing any concrete bitstrings, or indeed learning the prime q.

These objections motivate work on *computational soundness.* The hardness of DDH suggests that, when an equation is not valid, it is hard to obtain a satisfying instance. Moreover, the adversary should acquire no advantage from seeing the values g^x etc. However, precise results will require reduction arguments.

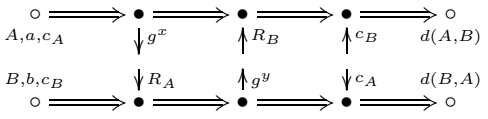
Executions Are Bundles. We formalize protocol executions by *bundles.* A bundle is a directed, acyclic graph. Its vertices are nodes on some strands (which may include both regular and adversary strands). Its edges include the succession edges $n_1 \Rightarrow n_2$, as well as *communication edges* written $n_1 \rightarrow n_2$. Such a dag $\mathcal{B} = (V, E_{\Rightarrow} \cup E_{\rightarrow})$ is a *bundle* if it is causally self-contained, meaning:

- If $n_2 \in V$ and $n_1 \Rightarrow n_2$, then $n_1 \in V$ and $(n_1, n_2) \in E_{\Rightarrow}$;
- If $n_2 \in V$ is a reception node, then there is a unique transmission node $n_1 \in V$ such that $\mathsf{msg}(n_2) = \mathsf{msg}(n_1)$ and $(n_1, n_2) \in E_{\rightarrow}$;
- Precedence $\preceq_{\mathcal{B}}$ for $\mathcal{B}$, defined to be $(E_{\Rightarrow} \cup E_{\rightarrow})^*$, is a well-founded relation.

Indicators and the Adversary. We justify now our central technique, that the adversary cannot generate messages with new indicators. We will write $\mathbf{0}$ for the all zero vector, i.e. the origin. We will also write $\mathbf{1}_v$ for the v^{th} basis vector $\langle \ldots, 0, \ldots, 1, \ldots, 0, \ldots \rangle$.

Definition 14. *Let N be a vector of NZE-variables. If a is a name, symmetric key, asymmetric key, or text, then its indicator set* $\mathrm{Ind}_N(a) = \{\mathbf{0}\}$*, the singleton of the origin.* $\mathrm{Ind}_N(t_0 \parallel t_1) = \mathrm{Ind}_N(t_0) \cup \mathrm{Ind}_N(t_1)$.

$$\mathrm{Ind}_N(\{|t_0|\}_{t_1}) = \mathrm{Ind}_N(\llbracket\, t_0 \,\rrbracket_{t_1}) = \mathrm{Ind}_N(\mathsf{h}(t_0)) = \mathrm{Ind}_N(t_0).$$



$$c_P = [\![\mathsf{cert}\ Y_P \parallel P]\!]_{\mathsf{sk(CA)}} \qquad d(P, P') = \mathsf{keyrec}\ P \parallel P' \parallel K$$

Fig. 1. UM Initiator and Responder Strands

A basic value a is non-originating before *n in bundle $\mathcal{B}$ if, for all $n' \preceq_{\mathcal{B}} n$, a does not originate at n'. The* indicator basis $\mathrm{IB}_{\mathcal{B}}(n)$ *of node n, where n is a node of $\mathcal{B}$, is the set (ordered in some conventional way):*

$$\{a \in \mathsf{Params}(\mathcal{B}) \colon a \text{ of sort } E \text{ is non-originating before } n\}.$$

Theorem 15 (Indicator Theorem for Strands). *Let n be an adversary transmission node of $\mathcal{B}$, and let N be a sequence of elements drawn from $\mathrm{IB}_{\mathcal{B}}(n)$. If $v \in \mathrm{Ind}_N(\mathsf{msg}(n))$ and $v \neq \mathbf{0}$, then there is a regular transmission node $n' \prec_{\mathcal{B}} n$ in $\mathcal{B}$ such that $v \in \mathrm{Ind}_N(\mathsf{msg}(n'))$.*

Proof. Let T_R be the set of messages transmitted on a regular node $m \prec n$, and let T_M be the set of parameters and constants transmitted on one-node adversary strands $\prec n$. By induction on adversary actions, $\mathsf{msg}(n) \in \mathrm{Gen}(T_R \cup T_M)$. T_R and T_M are N-free, by the definition of IB. So Theorem 12 applies.

Since $t : G \in T_M$ implies $\mathrm{Ind}_N(t) = \{\mathbf{0}\}$, we conclude that every non-zero indicator in u comes from a message in T_R, as desired. □

5 Analyzing the Unified Model

Regular participants in the UM protocol [2] act as *initiators* and *responders* as shown in Figure 1. We specify, for the initiator A:

1. A retrieves from its secure storage its principal name A, its long term secret a, and its public certificate c_A.
2. A chooses an ephemeral parameter $z \in \mathcal{V}^E$ to instantiate x, sending $R_A = g^z$.
3. A receives some R_B, which it checks to be a non-trivial group element, i.e. a value of the form g^y for some $y \neq 0, 1 \bmod q$.
4. It receives a certificate c_B associating Y_B with B's identity. How the participant determines what name B to require in this certificate, or how it determines which CAs to accept, is implementation-dependent.
5. A computes $K = \mathsf{h}({Y_B}^a \parallel {R_B}^z)$, depositing a *key record* into its local database, so that K may be used as a session key between A and B.

In clause 2, A chooses z freshly. A never sends z as an ingredient in any message, only g^z, and the adversary cannot find a strategy to guess the same value z, we model z as non-originating, and g^z as uniquely originating. In other Implicitly

Authenticated Diffie-Hellman protocols, other key computations may be used instead of Eqn. 2. A responder B behaves correspondingly. The syntax of Fig. 1 entails that no regular node n ever transmits a product $t_1 \cdot t_2$ as a (normal form) ingredient of any message, $t_1 \cdot t_2 \not\sqsubseteq \mathsf{msg}(n)$.

Regular initiator and responder strands that choose that parameters x, y transmit only messages g^x, g^y, where

$$\mathrm{Ind}_{\langle a,b,x,y\rangle} g^x = \{\mathbf{1}_x\} \text{ and } \mathrm{Ind}_{\langle a,b,x,y\rangle} g^y = \{\mathbf{1}_y\}.$$

Strands with other choices transmit the zero vector $\mathbf{0}$ relative to this x, y basis. In case 2, $\mathrm{Ind}_{\langle a,b,x,y\rangle}(Y) = \{\mathbf{1}_a\}$. However, the key K has indicators

$$\mathrm{Ind}_{\langle a,b,x,y\rangle} = \{\langle 1,1,0,0\rangle, \langle 0,0,1,1\rangle\}.$$

Here, the regular principals transmit only messages with basis vectors $\mathbf{1}_v$ or $\mathbf{0}$ as indicators, but the key has two non-zero entries in its two indicators.

Cryptographically, DH ensures that the choices of the principals always contribute in a non-cancellable way to the result. An analogue is:

Lemma 16 (Contributive Parameters). *Let $\mathcal{B}$ be a* UM*-bundle, and s be an initiator or responder strand with long term secret a and ephemeral value x:*

1. *If $x \in \mathsf{non}_{\mathcal{B}}$, then for $K = \mathsf{h}({Y_B}^a \parallel {R_B}^x)$, we have $\mathbf{1}_x \in \mathrm{Ind}_{\langle x\rangle}(K)$.*
2. *If $a \in \mathsf{non}_{\mathcal{B}}$, then $\mathbf{1}_a \in \mathrm{Ind}_{\langle a\rangle}(K)$.*

Proof. Since $\mathsf{h}(\cdot)$ and $\parallel$ are constructors, a or x can cancel only if s receives a value R_B or Y_b with indicator $\langle -1\rangle$ for a or x, resp. Hence there is some earlier node m on which some message with indicator $\langle -1\rangle$ was transmitted, and let m_0 be a minimal such node.

However, by the definitions, m_0 is not a regular node, which transmit only values with non-negative indicators. By Thm. 15, m_0 cannot be an adversary node either, when $x \in \mathsf{non}_{\mathcal{B}}$ or $a \in \mathsf{non}_{\mathcal{B}}$ resp. □

Key Secrecy and Impersonation. In Fig. 2 we present the core idea of key secrecy. Suppose that the upper strand s is an initiator or responder run that ends by computing session key K. Moreover, suppose that a listener strand is present, which receives K. Then, if the long term secrets $a, b \in \mathsf{non}$, this diagram cannot be completed to a bundle $\mathcal{B}$. This holds even without the freshness assumptions on regular initiator and responder strands. It includes bundles in which we add any number of regular strands, so long as these particular long-term secrets $a, b \in \mathsf{non}$. Other principals' long term keys may be freely compromised or not.

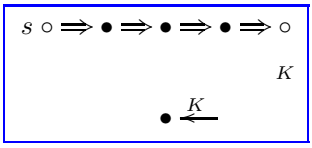


Fig. 2. Key secrecy: This diagram cannot occur

Security Goal 17 (Key Secrecy). *Suppose $\mathcal{B}$ is a bundle with $a, b \in \mathsf{non}_{\mathcal{B}}$, and s is an initiator or responder strand with long term secret parameter a and long term peer public value $Y = g^b$. Then $\mathcal{B}$ does not contain a listener $\bullet \leftarrow K$.*

Theorem 18. UM *achieves the security goal of key secrecy.*

Proof. Suppose instead that $\bullet \leftarrow K$ is in $\mathcal{B}$, so some node transmits K.

Computing indicators using the basis $\langle a, b\rangle$ by applying Lemma 16 to both a and b, K has indicator $\langle 1, 1\rangle$. By Thm. 15, some regular node transmits a message with indicator $\langle 1, 1\rangle$. But regular strands transmit only values with indicators $\mathbf{0}$ and, in certificates, $\mathbf{1}_a$, $\mathbf{1}_b$, relative to basis $\langle a, b\rangle$. □

Curiously, resistance to impersonation attacks concerns the same diagram, Fig. 2, although with different assumptions. An impersonation attack is a case in which the adversary, having compromised B's long term secret b, uses it to obtain a session key K, while causing B to have a session yielding K as session key. If B's session uses $Y_A = g^a$, where a is the uncompromised long term secret of A, then the adversary has succeeded in *impersonating* A to B. By contrast, it is hopeless—when b is compromised—to try to prevent the adversary from impersonating B to others.

Security Goal 19 (Impersonation Resistance). *Suppose $\mathcal{B}$ is a bundle with $a, x \in \mathsf{non}_{\mathcal{B}}$, and s is an initiator or responder strand with long term secret parameter a ephemeral value x. Then $\mathcal{B}$ does not contain a listener $\bullet \leftarrow K$.*

This goal trades off a long term secret for an ephemeral value. UM does not achieve it. Its key $K = \mathsf{h}(g^{ab} \parallel g^{xy})$ has indicators $\{\langle 1, 0\rangle, \langle 0, 1\rangle\}$ in the basis $\langle a, x\rangle$, suggested by our assumptions. Thus, Theorem 15 buys us nothing.

Example 20. *The adversary can impersonate A to B by supplying its own g^z, as B supplies g^y; it computes $K = \mathsf{h}(g^{ab} \parallel g^{zy})$ by raising A's public g^a to the compromised value b, and raising g^y to its own ephemeral value z.*

Implicit Authentication. Implicit authentication takes two forms [7, 27, 32].

The essential common idea is expressed in Figure 3. It shows two strands that compute the same session key K. One has parameters $[A, B', \ldots]$ and the other has parameters $[A', B, \ldots]$, where we assume that the parameter for the initiator's name appears first (A, A') and parameter for the responder's name appears second (B', B). The authentication property is that the participants agree on each other's identities, so that the responder has the correct opinion about the initiator's identity and *vice versa*. That is, we want $A = A'$ and $B = B'$ whenever the computed keys agree. Stronger and weaker implicit key authentication properties differ in what non-compromise assumptions they make. The stronger property is that $A = A'$ and $B = B'$ whenever $a, b \in \mathsf{non}$. A weaker assertion is that $A = A'$ and $B = B'$ whenever $a, b, a' \in \mathsf{non}$. The additional non-compromise assumption is about a', the long term secret of the principal E that B *thinks* he is communicating with [7, 18, 32]. MQV satisfies only this weaker form [27]. We focus on the stronger property here.

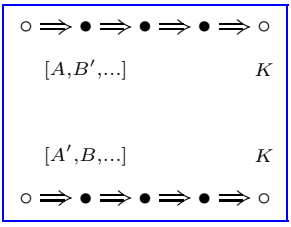


Fig. 3. Implicit authentication: In this diagram, $A = A'$ and $B = B'$

Authentication depends on the certification protocol, which ensures proof of possession. Rather than representing it, we characterize it by an assumption:

Assumption 21. *If* $c_P \sqsubseteq \mathsf{msg}(n)$ *for* $n \in \mathsf{node}(\mathcal{B})$*, then* $c_P = [\![\, \mathsf{cert}\ g^e \parallel P \,]\!]_{\mathsf{sk(CA)}}$ *for some* E*-value* $e \neq 0, 1$*, and either:*

1. *there exists* $n \in \mathcal{B}$ *with* $e \sqsubseteq \mathsf{msg}(n)$*, or else*
2. *(i)* $e \in \mathcal{V}^E$ *is a parameter, and*
   (ii) if $[\![\, \mathsf{cert}\ g^e \parallel P' \,]\!]_{\mathsf{sk(CA)}} \sqsubseteq \mathsf{msg}(n')$ *for any* $n' \in \mathsf{node}(\mathcal{B})$*, then* $P = P'$*.*

Clause (1) holds when e is generated by the adversary; clause (2) applies when e is chosen by a compliant principal.

Security Goal 22 (Implicit Authentication). *Suppose that* $\mathcal{B}$ *is a* Π*-bundle with* $a, b \in \mathsf{non}_{\mathcal{B}}$*, and strands* s_1, s_2 *are* Π *initiator and responder strands with parameters* $[A, B', a, x, Y_{B'}, R_{B'}]$ *and* $[A', B, b, y, Y_{A'}, R_{A'}]$ *resp. If* s_1, s_2 *both yield session key* K*, then* $A = A'$ *and* $B = B'$*.*

Theorem 23. UM *achieves implicit authentication.*

Proof. Let s_1, s_2 be strands in $\mathcal{B}$ as in the implicit authentication goal, where also $a, b \in \mathsf{non}_{\mathcal{B}}$. Since s_1 receives a certificate $[\![\, \mathsf{cert}\ Y_{B'} \parallel B' \,]\!]_{\mathsf{sk(CA)}}$, by Assumption 21, $Y_{B'} = g^e$ for some $e \neq 0, 1$. By symmetry, $Y_{A'} = g^d$.

The key computation ensures $g^{db} = g^{ae}$; by injectiveness, $db = ae$. Thus, there is some c such that $d = ca$ and $e = cb$. Thus, by Assumption 21 either:

1. there exists $n_d \in \mathsf{node}(\mathcal{B})$ such that $cb \sqsubseteq \mathsf{msg}(n_d)$, or else
2. cb's normal form is a parameter, i.e. $c = 1$ and $e = b$.

In the latter case, we also have that $B' = B$. In the former case, n_d lies on an adversary strand. It must result from multiplying the values b and c, since no regular strand transmits a message with any product as an ingredient. But this contradicts $b \in \mathsf{non}(\mathcal{B})$. Symmetrically, $A' = A$. □

Future Work. We will apply these methods to more challenging protocols [18]. We will also study their computational soundness. A tool implementation approach is to represent $\mathsf{AG}^{\hat{}}$ and protocols using it in geometric logic; model-finding can generate counterexamples or establish their absence. An alternative approach is integration with Tamarin [15]. $\mathsf{AG}^{\hat{}}$ appears to extend to represent bilinear pairings.

Acknowledgments. We have benefited from discussions with Shriram Krishnamurthi, Moses Liskov, Cathy Meadows, Paliath Narendran, John Ramsdell, Paul Rowe, Paul Timmel, and Ed Zieglar.

References

1. Amadio, R.M., Lugiez, D., Vanackère, V.: On the symbolic reduction of processes with cryptographic functions. Theoretical Computer Science 290(1), 695–740 (2003)

2. Ankney, R., Johnson, D., Matyas, M.: The Unified Model. Contribution to ANSI X9F1. Standards Projects (Financial Crypto Tools), ANSI X, 42 (1995)
3. Armando, A., et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
4. Ax, J.: The elementary theory of finite fields. The Annals of Mathematics 88(2), 239–271 (1968)
5. Basin, D.A., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. Int. J. Inf. Sec. 4(3), 181–208 (2005)
6. Bergstra, J.A., Tucker, J.V.: The rational numbers as an abstract data type. Journal of the ACM 54 (2007)
7. Blake-Wilson, S., Menezes, A.: Authenticated Diffie-Hellman Key agreement protocols. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 339–361. Springer, Heidelberg (1999)
8. Blanchet, B.: An efficient protocol verifier based on Prolog rules. In: 14th Computer Security Foundations Workshop, pp. 82–96. IEEE CS Press (June 2001)
9. Boreale, M., Buscemi, M.G.: Symbolic analysis of crypto-protocols based on modular exponentiation. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 269–278. Springer, Heidelberg (2003)
10. Bresson, E., Lakhnech, Y., Mazaré, L., Warinschi, B.: Computational soundness: The case of Diffie-Hellman keys. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols. Cryptology and Information Security Series. IOS Press (2011)
11. Cervesato, I., Meadows, C., Pavlovic, D.: An encapsulated authentication logic for reasoning about key distribution protocols. In: 18th IEEE Workshop on Computer Security Foundations, CSFW-18 2005, pp. 48–61. IEEE (2005)
12. Chang, C.C., Keisler, H.J.: Model Theory. Studies in Logic and the Foundations of Mathematics, vol. 73 (1990)
13. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 124–135. Springer, Heidelberg (2003)
14. Cremers, C., Feltz, M.: One-round strongly secure key exchange with perfect forward secrecy and deniability. Cryptology ePrint Archive, Report 2011/300 (2011), `http://eprint.iacr.org/2011/300`
15. Cremers, C., Schmidt, B., Meier, S., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Computer Security Foundations (CSF) (2012)
16. Cremers, C.J.F.: Scyther - Semantics and Verification of Security Protocols. Ph.D. dissertation, Eindhoven University of Technology (2006)
17. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
18. Dougherty, D.J., Guttman, J.D.: Symbolic protocol analysis for Diffie-Hellman, Arxiv preprint arXiv:1202.2168 (2012), `http://arxiv.org/abs/1202.2168v1`
19. Durán, F., Meseguer, J.: A church-Rosser checker tool for conditional order-sorted equational maude specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010), Version 3j, available at `http://maude.lcc.uma.es/CRChC`
20. Escobar, S., Meadows, C., Meseguer, J.: State space reduction in the Maude-NRL protocol analyzer. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 548–562. Springer, Heidelberg (2008)

21. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: Cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009)
22. Fiore, M., Abadi, M.: Computing symbolic models for verifying cryptographic protocols. In: Computer Security Foundations Workshop (June 2001)
23. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
24. Goubault-Larrecq, J., Roger, M., Verma, K.: Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. Journal of Logic and Algebraic Programming 64(2), 219–251 (2005)
25. Guttman, J.D.: Shapes: Surveying crypto protocol runs. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols. Cryptology and Information Security Series. IOS Press (2011)
26. Guttman, J.D.: State and progress in strand spaces: Proving fair exchange. Journal of Automated Reasoning (March 2010) (accepted), doi:10.1007/s10817-010-9202-1
27. Kaliski, B.S.: An unknown key-share attack on the MQV key agreement protocol. ACM Transactions on Information and System Security 4(3), 275–288 (2001)
28. Kapur, D., Narendran, P., Wang, L.: An E-unification algorithm for analyzing protocols that use modular exponentiation. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 165–179. Springer, Heidelberg (2003)
29. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer, Heidelberg (2005)
30. Kunz-Jacques, S., Pointcheval, D.: About the Security of MTI/C0 and MQV. In: De Prisco, R., Yung, M. (eds.) SCN 2006. LNCS, vol. 4116, pp. 156–172. Springer, Heidelberg (2006)
31. Küsters, R., Truderung, T.: Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In: IEEE Computer Security Foundations Symposium, pp. 157–171. IEEE (2009)
32. Law, L., Menezes, A., Qu, M., Solinas, J., Vanstone, S.: An efficient protocol for authenticated key agreement. Designs, Codes and Cryptography 28(2), 119–134 (2003)
33. Menezes, A.: Another look at HMQV. Journal of Mathematical Cryptology 1, 47–64 (2007)
34. Millen, J.K., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: 8th ACM Conference on Computer and Communications Security (CCS 2001), pp. 166–175. ACM (2001)
35. Pavlovic, D., Meadows, C.: Deriving secrecy in key establishment protocols. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 384–403. Springer, Heidelberg (2006)
36. Ramsdell, J.D., Guttman, J.D.: CPSA: A cryptographic protocol shapes analyzer. In: Hackage. The MITRE Corporation (2009), `http://hackage.haskell.org/package/cpsa`; see esp. doc subdirectory
37. Rubio, A.: A fully syntactic AC-RPO. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 133–147. Springer, Heidelberg (1999)
38. Javier Thayer, F., Herzog, J.C., Guttman, J.D.: Strand spaces: Proving security protocols correct. Journal of Computer Security 7(2/3), 191–230 (1999)

A Appendix

Lemma 8

1. *The structure* $\mathcal{M}_{\mathbb{Q}}$ *can be embedded as a submodel in any* $\mathcal{M}_D$.
2. *If* s *and* t *are distinct normal forms then it is not the case that* $\mathcal{M}_{\mathbb{Q}} \models s = t$.

1. Since $\mathbb{F}_D$ has characteristic 0, and $\mathbb{Q}$ is the prime field of characteristic 0, $\mathbb{Q}$ is embeddable in $\mathbb{F}_D$. The models $\mathcal{M}_D$ and $\mathcal{M}_{\mathbb{Q}}$ are definitional expansions of $\mathbb{F}_D$ and $\mathbb{Q}$, so the embedding of $\mathbb{Q}$ into $\mathbb{F}_D$ extends to embed $\mathcal{M}_{\mathbb{Q}}$ into $\mathcal{M}_D$.
2. If s and t are distinct normal forms, the term $u \equiv s \cdot inv(t)$ is in normal form and not identically id. With this observation we see that our result follows if we establish the following fact: if u is a normal form not identically id then it is not the case that $\mathcal{M}_{\mathbb{Q}} \models u = id$.
   To see this, note that in the structure $\mathcal{M}_{\mathbb{Q}}$, the group operation is interpreted as addition, inverse by additive inverse, and exponentiation as multiplication, so it suffices to consider the expression obtained from u by replacing $\cdot$ and inv by $+$ and $-$, and the exponentiation operator by $*$. In this way we may view u as an ordinary rational expression in the variables $x_1, \ldots, x_k$ occurring in u. So u determines a real function $f_u : \mathbb{R}^k \to \mathbb{R}$ not identically 0. We can find a rational point $\boldsymbol{r} = (r_1, ..., r_k)$ such that $f_u(\boldsymbol{r}) \neq 0$. Then the environment $\eta : Vars \to \mathbb{Q}$ with $\eta(x_i) = r_i$ witnesses the fact that $\mathcal{M}_{\mathbb{Q}} \not\models u = id$. □

Security Analysis in Probabilistic Distributed Protocols via Bounded Reachability*

Silvia S. Pelozo and Pedro R. D'Argenio

CONICET - FaMAF - Universidad Nacional de Córdoba
{spelozo,dargenio}@famaf.unc.edu.ar

Abstract. We present a framework to analyze security properties in distributed protocols. The framework is constructed on top of the so called (strongly) distributed schedulers where secrecy is also considered. Secrecy is presented as an equivalence class on actions to those components that do not have access to such secrets; however these actions can be distinguished by those with appropriate clearance. We also present an algorithm to solve bounded reachability analysis on this kind of models. The algorithm appropriately encodes the nondeterministic model by interpreting the decisions of the schedulers as parameters. The problem is then reduced to a polynomial optimization problem.

1 Introduction

Model-based verification has proven very useful in the verification of a handful of systems. It is particularly fit for the verification of distributed systems, in which the model of the system is obtained by composing simpler models describing the behaviour of each component. A particular class of these systems are distributed algorithms that aim to provide information hiding, i.e., they try to prevent an adversary to infer confidential information from the observables [2]. Many of these algorithms propose a solution by adding randomization to their decisions (e.g. [8,15]). In addition, the security property is best understood quantitatively by contrasting the likelihood of producing a secret w.r.t. the likelihood of guessing such secret after reading the observables (see e.g. [3,16,2,1]). Notice that, due to the nature of distributed systems, the model needs to consider both probabilistic and nondeterministic behaviour: probabilities allow to model randomization (including the likelihood of secrets) while nondeterminism expresses the interleaving of different processes, abstraction of decision mechanisms, and model underspecification among other things.

In this paper, we focus in quantitative reachability properties, i.e. those that assert about the probability of reaching some states of particular interest, be it because it is desirable or undesirable. In particular, we are interested in accurately verifying security properties of distributed system where private actions and interactions between some components are effectively hidden from others. In this setting, we consider a system secure if the probability of reaching states

* Supported by Project ANPCYT PAE-PICT 02272, SeCyT-UNC, and EU 7FP grant agreement 295261 (MEALS).

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 182–197, 2013.
© Springer-Verlag Berlin Heidelberg 2013

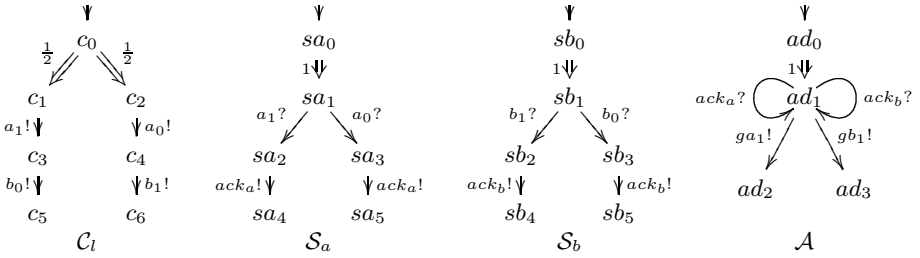


Fig. 1. Output actions are suffixed with "!" and input actions are suffixed with "?". The set of actions of each component are only those depicted in the component. To ensure input enabledness, all states are assumed to have self-loops labeled with the inputs others than those already depicted leaving the state. For every final state s we also omitted the transition $s \Rightarrow \mu$ with $\mu(s) = 1$.

where these properties are violated is between certain known bounds. For example, in an anonymous message exchanging protocol, we would like to verify that the probability of guessing the anonymous sender of a message is not greater than the probability of guessing the sender by chance alone.

Probabilistic model checking is –in principle– adequate to achieve this. However, traditional analysis techniques do not necessarily provide results reflecting the security guarantees of the modeled systems: they only provide pessimistic over-approximations for the actual probability of the property. This is due to the fact that traditional techniques consider *all* possible resolutions of nondeterminism [4]. To understand the problem we present an example that we will use along the paper. Consider a protocol where a client $\mathcal{C}_l$ chooses randomly one out of two available service providers $\mathcal{S}_a$ and $\mathcal{S}_b$ (see Fig. 1). This component asks for service with an encrypted message (a_1 or b_1, depending on the chosen provider). To misguide possible attackers it also interchanges a dummy (encrypted) message with the other provider (a_0 or b_0). Once the provider receives any of the notifications it acknowledges reception of the message. If the encryption scheme is secure, the selection and its outcome should be known only by the client and the providers. Any other component of the system (in particular, the adversary $\mathcal{A}$) should not be able to infer it with certainty.

Resolution of nondeterminism is done by the so called *schedulers* which are functions that select the next step based on the past execution of the system. Traditional probabilistic model checking will consider the scheduler that, whenever enabled, selects action ga_1 (for "guess $\mathcal{S}_a$ got 1") if action a_1 appears in the execution, and it selects gb_1 ("guess $\mathcal{S}_b$ got 1") if b_1 appears in the execution. This is a valid scheduler that lets the adversary $\mathcal{A}$ guess with probability 1.

One of the reasons this happens is because the resolution of the local nondeterminism in $\mathcal{A}$ is made using global knowledge. This problem has been recently observed by several authors [9,13,14,11] who proposed to restrict to the so called *distributed schedulers*. Distributed schedulers enforce that *local* decisions of the processes are based only on local knowledge [14]. However, nondeterminism originated by interleaving in parallel composition is still resolved using global knowledge in this new framework. Since distributed schedulers were not defined

in a context were secrecy was important, actions a_1 and b_1 are considered different from a_0 and b_0. As they are part of the global knowledge, the resolution of the interleaving nondeterminism can be different in each case. This may lead to some unrealistic leakage of information. We present an example in detail in Sec. 3.

In this paper we adapt distributed schedulers to deal also with secrecy, presented as an equivalence class on local states and actions. Therefore, two actions in the same equivalence class can only be distinguished locally in components with the appropriate clearance but cannot be distinguished globally.

In general, reachability under distributed schedulers is an undecidable problem [13]. However, if restricted to reachability properties where the goal states should be reached within a given number of steps (i.e. *(time-)bounded* reachability properties), the problem becomes decidable [5]. This is done by reducing the bounded reachability problem to a polynomial optimization problem. In this paper we adapt and improve the algorithm of [5] to work on the class of distributed schedulers under secrecy. As an example of the application to information hiding, we automatically verify the anonymity property in a case study.

2 Modeling Probabilistic Distributed System

To assert or refute a quantitative reachability property about a distributed system, we first construct a model of each component of the system. The full model is constructed by composing these submodels, considering possible interactions between the components and all possible interleaving. Each component is described with a state-based model that combines discrete-time Markov chains and labeled transition systems. In this work we use a restricted variant of interactive probabilistic chains (IPCs) [10,5], a formalism where probabilistic transitions and action-labelled transitions are handled orthogonally.

Definition 1. *A* basic I/O-IPC *is a tuple* $\langle S, A, \to, \Rightarrow, \hat{s}\rangle$ *where* S *is a finite set of states with initial state* $\hat{s} \in S$, $A = A^I \cup A^O$ *is a finite set of actions consisting of disjoint sets of input actions* (A^I) *and output actions* (A^O), $\to \subseteq S \times A \times S$ *is the set of interactive transitions, and* $\Rightarrow : S \rightharpoonup Dist(S)$ *is a partial function representing probabilistic transitions. We require that the I/O-IPC is (i)* input enabled*, i.e. for all* $s \in S$ *and* $a \in A^I$*, there exists* $s' \in S$ *s.t.* $s \xrightarrow{a} s'$*; and (ii)* action deterministic*, i.e. for all* $s \in S$ *and* $a \in A$, $s \xrightarrow{a} s'$ *and* $s \xrightarrow{a} s''$ *implies* $s' = s''$.

We use the shorthand notation $s \xrightarrow{a} s'$ for an interactive transition $(s, a, s') \in \to$, $s \Rightarrow \mu$ for $(s, \mu) \in \Rightarrow$ and $s \not\Rightarrow$ if there is no $a \in A^O$ and s' such that $s \xrightarrow{a} s'$.

The requirement of input action determinism is present in this work to simplify presentation. The general model can be treated just like in [14].

Parallel Composition. Distributed I/O-IPC are obtained by composing basic ones through the parallel composition. We define it simultaneously on a finite set of basic I/O-IPC. To avoid unnecessary technicalities, we require *output isolation*, i.e. no output action can be performed by more than one basic process.

Similarly to action determinism, the framework may be extended to models without output isolation (see [14,12]).

Definition 2. *A finite set of basic I/O-IPCs* $\mathcal{P}_i = \langle S_{\mathcal{P}_i}, A_{\mathcal{P}_i}, \rightarrow_{\mathcal{P}_i}, \Rightarrow_{\mathcal{P}_i}, \hat{s}_{\mathcal{P}_i} \rangle$, $1 \leq i \leq n$, *is* composable *if* $A^O_{\mathcal{P}_i} \cap A^O_{\mathcal{P}_j} = \emptyset, \forall i \neq j$. *Provided that* $\{\mathcal{P}_i\}_{i=1}^n$ *is composable, the* parallel composition $\mathcal{C} = \mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_n$ *is defined by the I/O-IPC* $\langle S_{\mathcal{C}}, A^I_{\mathcal{C}} \cup A^O_{\mathcal{C}}, \rightarrow_{\mathcal{C}}, \Rightarrow_{\mathcal{C}}, \hat{s}_{\mathcal{C}} \rangle$, *where* $S_{\mathcal{C}} = S_{\mathcal{P}_1} \times \cdots \times S_{\mathcal{P}_n}$ *and* $\hat{s}_{\mathcal{C}} = (\hat{s}_{\mathcal{P}_1}, \ldots, \hat{s}_{\mathcal{P}_n})$ *is the initial state;* $A^O_{\mathcal{C}} := \bigcup_{i=1}^n A^O_{\mathcal{P}_i}$, $A^I_{\mathcal{C}} := \bigcup_{i=1}^n A^I_{\mathcal{P}_i} \setminus A^O_{\mathcal{C}}$ *and the transition relations are defined according to the following rules:*

$$\frac{\{s_i \xrightarrow{a}_{\mathcal{P}_i} s'_i \mid a \in A_{\mathcal{P}_i}\} \qquad t_i = \textbf{ if } (a \in A_{\mathcal{P}_i}) \textbf{ then } s'_i \textbf{ else } s_i}{(s_1, \ldots, s_n) \xrightarrow{a}_{\mathcal{C}} (t_1, \ldots, t_n)} \tag{1}$$

$$\frac{\{s_i \Rightarrow_{\mathcal{P}_i} \mu_i \ \wedge \ s_i \not\rightarrow \mid 1 \leq i \leq n\}}{(s_1, \ldots, s_n) \Rightarrow_{\mathcal{C}} \mu_1 \times \cdots \times \mu_n} \tag{2}$$

with $\mu_1 \times \cdots \times \mu_n$ *denoting the product distribution on* $S_{\mathcal{P}_1} \times \cdots \times S_{\mathcal{P}_n}$ *defined by* $(\mu_1 \times \cdots \times \mu_n)(s_1, \ldots, s_n) = \prod_{i=1}^n \mu_i(s_i)$ *for all* $s_i \in S_{\mathcal{P}_i}$.

Notice that composability guarantees that at most one processes performs an output action when synchronizing according to rule (1). Moreover, every other process that *knows* the action will perform the step because of input enabledness.

We consider that output transitions are immediate and probabilistic transitions are timed. We assume that immediate transitions always take precedence over timed transitions. This assumption is known as *maximal progress* [17] and it is reflected on rule (2). Following this criteria, a state is called *vanishing* if at least one output-labeled transition is enabled on it. If only probabilistic or input-labeled transitions are enabled in a state then it is called *tangible*. The introduction of maximal progress may induce an infinite sequence of consecutive output-labeled transitions. We consider this as *Zeno behaviour* and require that composed models *do not* exhibit this type of behaviour. We will also assume that composed systems are *closed*, that is, its set of input actions is empty. Dealing with open systems in our context demands some assumption on the environment behaviour. In any case, such assumption (even the most general) can be encoded in one or more extra components such that the whole system is finally closed.

Resolving Nondeterminism through Schedulers. To obtain the probability of reaching some states, nondeterministic choices between enabled transitions have to be resolved. This is achieved by the so called *schedulers* (also adversaries or policies). A scheduler is a function mapping each partial execution (or finite path) to a distribution on actions enabled in the last state of the execution.

Let $\mathcal{P} = \langle S, A = A^I \cup A^O, \rightarrow, \Rightarrow, \hat{s} \rangle$ be an I/O-IPC. We define $enab(s) = \{a \in A^O \mid \exists s'.\, s \xrightarrow{a} s'\}$, the set of output actions enabled in s.

A *finite path* of $\mathcal{P}$ is a sequence $\sigma = s_0 a_0 s_1 a_1 \cdots a_{n-1} s_n$ with $s_0 = \hat{s}$, where states and either actions or distributions alternate, that is, for $i = 0, \ldots, n-1$: (i) $a_i \in enab(s_i)$ and $s_i \xrightarrow{a_i} s_{i+1}$, or (ii) $a_i \in Dist(S)$, $s_i \Rightarrow a_i$, $a_i(s_{i+1}) > 0$, and $enab(s_i) = \emptyset$. The last state s_n is denoted by $last(\sigma)$, and $length(\sigma) = n$ is the total number of transitions (interactive or probabilistic) along the path.

Moreover, we denote $enab\,(last\,(\sigma))$ as $enab\,(\sigma)$ for finite path σ. An *infinite path* of $\mathcal{P}$ is an infinite sequence $s_0 a_0 s_1 a_1 \cdots$ alternating states and actions or distributions satisfying the previous condition. We denote with $Paths\,(\mathcal{P})$ and $Paths_{fin}\,(\mathcal{P})$ the set of paths and finite paths of $\mathcal{P}$, respectively.

Definition 3. *A* scheduler *for $\mathcal{P}$ is a function $\eta_{\mathcal{P}} : Paths_{fin}\,(\mathcal{P}) \to Dist\,(A)$ such that $\eta_{\mathcal{P}}(\sigma)(enab(\sigma)) = 1$ (that is $\eta_{\mathcal{P}}(\sigma)(a) > 0$ implies $a \in enab(\sigma)$).*

Probability Measure Induced by a Scheduler. When all the nondeterministic choices in an I/O-IPC are resolved by a scheduler the resulting system is a (possibly infinite) Markov chain. Hence it is possible to define a probability measure over the sets of infinite paths of the model.

For the sake of simplicity we will assume that the I/O-IPC does not contain tangible states which do not have an outgoing probabilistic transition. That is, for every tangible state s, $\Rightarrow(s)$ is defined. If this is not the case, we just complete it by extending $\Rightarrow$ with the tuple (s, δ_s) where $\delta_s(s) = 1$.

We first define the σ-algebra on the set of infinite paths of the I/O-IPC and then the probability measure on this σ-algebra induced by a given scheduler. The *cylinder* induced by the finite path σ is the set of infinite paths $\sigma^{\uparrow} = \{\sigma' \in Paths\,(\mathcal{P}) \mid \sigma'$ is infinite and σ is a prefix of $\sigma'\}$. Define $\mathcal{F}$ to be the σ-algebra on the set infinite paths of $\mathcal{P}$ generated by the set of cylinders.

Definition 4. *Let η be a scheduler for $\mathcal{P}$. The* probability measure induced by *η on $\mathcal{F}$ is the unique probability measure $\Pr_\eta$ such that, for any state $s \in S$, any action $a \in A$ and any distribution $\mu \in Dist(S)$:*

$$\begin{array}{lll} \Pr_\eta(s^{\uparrow}) & = 1 & \text{if } s = \hat{s} \\ \Pr_\eta(\sigma a s^{\uparrow}) & = \Pr_\eta(\sigma^{\uparrow}) \cdot \eta(\sigma)(a) & \text{if } enab(\sigma) \neq \emptyset \text{ and } last(\sigma) \xrightarrow{a} s \\ \Pr_\eta(\sigma \mu s^{\uparrow}) & = \Pr_\eta(\sigma^{\uparrow}) \cdot \mu(s) & \text{if } enab(\sigma) = \emptyset \text{ and } last(\sigma) \Rightarrow \mu \\ \Pr_\eta(\sigma^{\uparrow}) & = 0 & \text{in any other case} \end{array}$$

By the assumption that tangible states are in the domain of $\Rightarrow$, $enab\,(\sigma) = \emptyset$ implies that $last\,(\sigma) \Rightarrow \mu$ for some μ. Hence, $\Pr_\eta$ is indeed a probability measure.

Time-bounded reachability properties demand that the system reaches a goal state from a given set G within a given time t. In our case, the notion of time is given by each probability step. So, for any finite path σ, we let $time(\sigma)$ be the number of probability steps appearing on σ. Then, given a scheduler η for $\mathcal{P}$, the probability of reaching a goal state in G within time t, can be computed by $\Pr_\eta(\Diamond^{\leq t} G) = \Pr_\eta(\bigcup\{\sigma^{\uparrow} \mid time(\sigma) \leq t \wedge last(\sigma) \in G\})$. Let $\bar{G}$ be the set of paths $\sigma \in Paths_{fin}\,(\mathcal{P})$ such that $last(\sigma) \in G$ and for any proper prefix $\hat{\sigma}$ of σ, $last(\hat{\sigma}) \notin G$. Notice that for every $\sigma' \in Paths\,(\mathcal{P})$ reaching a state in G, there exists a unique $\sigma \in \bar{G}$ such that $\sigma' \in \sigma^{\uparrow}$. Then, we have that

$$\Pr_\eta\left(\Diamond^{\leq t} G\right) = \Pr_\eta\left(\biguplus\left\{\sigma^{\uparrow} \mid time(\sigma) \leq t \wedge \sigma \in \bar{G}\right\}\right) = \sum_{\substack{\sigma \in \bar{G} \\ time(\sigma) \leq t}} \Pr_\eta(\sigma^{\uparrow}). \quad (3)$$

The model checking problem on nondeterministic probabilistic systems is focused on finding worst case scenarios. Therefore, it aims to find the maximum or minimum probability of reaching a set of goal states ranging over a

particular class of schedulers. That is, if $\mathcal{K}$ is a class of schedulers (i.e. a set of all schedulers satisfying some given condition), then we are interested on finding $\sup_{\eta\in\mathcal{K}} \Pr_\eta(\Diamond^{\leq t} G)$ or $\inf_{\eta\in\mathcal{K}} \Pr_\eta(\Diamond^{\leq t} G)$.

Distributed Schedulers. Not all resolutions of nondeterminism are appropriate in a distributed setting. There are "almighty schedulers" that allow a component to guess the outcome of a probabilistic choice of a second component even when they have no communication at all (not even indirectly). The reader is referred to, e.g., [14,12] for a discussion. Therefore, we restrict to the class of *distributed schedulers* [14]. This leads to a more realistic resolution of nondeterminism, but unfortunately it also renders the model checking problem undecidable in general [13,12].

Distributed schedulers consider a notion of local knowledge of each component which is obtained by partially observing the global system behaviour. Thus, a component can only see the global execution through its local states and the actions it performs. Hence, two different global execution may appear the same to a single component. We implement this with a *projection function*.

From now on we will consider a composed model $\mathcal{C} = \mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_n = \langle S_\mathcal{C}, A_\mathcal{C}, \rightarrow_\mathcal{C}, \Rightarrow_\mathcal{C}, \hat{s}_\mathcal{C}\rangle$, where each $\mathcal{P}_i = \langle S_i, A_i = A_i^I \cup A_i^O, \rightarrow_i, \Rightarrow_i, \hat{s}_i\rangle$. For every path $\sigma \in \mathit{Paths}\,(\mathcal{C})$, the *projection* $\sigma\,[\mathcal{P}_i] \in \mathit{Paths}\,(\mathcal{P}_i)$ *of* σ *over* $\mathcal{P}_i$ is defined as:

$$
\begin{aligned}
(s_1,\ldots,s_n)\,[\mathcal{P}_i] &= s_i \\
\sigma a(s_1,\ldots,s_n)\,[\mathcal{P}_i] &= \textbf{ if } a \in A_i \textbf{ then } (\sigma\,[\mathcal{P}_i])as_i \textbf{ else } \sigma\,[\mathcal{P}_i] \\
\sigma(\mu_1 \times \cdots \times \mu_n)(s_1,\ldots,s_n)\,[\mathcal{P}_i] &= (\sigma\,[\mathcal{P}_i])\mu_i s_i
\end{aligned}
$$

In a (composed) state with enabled interactive transitions, the nondeterminism is resolved in two phases. First, a component is chosen among all enabled components. This choice may be probabilistic, and it is performed by the so called *interleaving scheduler*. Afterwards, the chosen component decides which transition to perform among all its enabled output transitions. This local choice is resolved by a *local scheduler* taking into account only local knowledge. Hence they are functions on *local* executions which are obtained by properly projecting the global executions. Therefore, a local scheduler is a scheduler as defined in Def. 3, only that its domain is the set of all finite paths of the local component. An interleaving scheduler is defined on finite paths of the composed system (hence, *global* executions) as follows.

Definition 5. *A function* $\mathcal{I} : \mathit{Paths}_{\mathit{fin}}\,(\mathcal{C}) \rightarrow \mathit{Dist}\,(\{\mathcal{P}_1,\ldots,\mathcal{P}_n\})$ *is an* interleaving scheduler *if for all* $\sigma \in \mathit{Paths}_{\mathit{fin}}\,(\mathcal{C})$, $\mathcal{I}\,(\sigma)\,(\mathcal{P}_i) > 0 \;\Rightarrow\; \mathit{enab}\,(\sigma\,[\mathcal{P}_i]) \neq \emptyset$.

A *distributed scheduler* is obtained by properly composing the interleaving schedulers with all local schedulers.

Definition 6. *A function* $\eta_\mathcal{C} : \mathit{Paths}_{\mathit{fin}}\,(\mathcal{C}) \rightarrow \mathit{Dist}(A_\mathcal{C})$ *is a* distributed scheduler *if there are local schedulers* $\eta_{\mathcal{P}_1},\ldots,\eta_{\mathcal{P}_n}$ *and an interleaving scheduler* $\mathcal{I}$*, such that, for all* $\sigma \in \mathit{Paths}_{\mathit{fin}}\,(\mathcal{C})$ *with* $\mathit{enab}(\sigma) \neq \emptyset$ *and for all* $a \in A_\mathcal{C}$*:* $\eta_\mathcal{C}(\sigma)(a) = \sum_{i=1}^n \mathcal{I}(\sigma)(\mathcal{P}_i) \cdot \eta_{\mathcal{P}_i}(\sigma\,[\mathcal{P}_i])(a)$.

Since at most one component can output action a, last equation reduces to: $\eta_\mathcal{C}(\sigma)(a) = \mathcal{I}(\sigma)(\mathcal{P}_i) \cdot \eta_{\mathcal{P}_i}(\sigma\,[\mathcal{P}_i])(a)$ whenever $a \in A_{\mathcal{P}_i}^O$.

It has been shown that distributed scheduler as defined above still permeates information from one component to others with no apparent reason (see [14,12]). In essence, the problem is that the interleaving scheduler may use information from a component $\mathcal{P}_1$ to decide how to pick between components $\mathcal{P}_2$ and $\mathcal{P}_3$. Therefore, we need to restrict the set of possible interleaving schedulers. We will require that, if neither components $\mathcal{P}_2$ and $\mathcal{P}_3$ can distinguish execution σ from σ', the interleaving scheduler has to consider the same relative (i.e. conditional) probabilities for choosing $\mathcal{P}_2$ or $\mathcal{P}_3$ after both paths.

Definition 7. *A scheduler η of $\mathcal{C}$ is said to be* strongly distributed *if it is distributed and for every two components $\mathcal{P}_i, \mathcal{P}_j$, and $\sigma, \sigma' \in Paths_{fin}(\mathcal{C})$ the interleaving scheduler $\mathcal{I}$ of η satisfies that, whenever* (i) $\sigma[\mathcal{P}_i] = \sigma[\mathcal{P}_j]$ *and* $\sigma'[\mathcal{P}_i] = \sigma'[\mathcal{P}_j]$, *and* (ii) $\mathcal{I}(\sigma)(\mathcal{P}_i) + \mathcal{I}(\sigma)(\mathcal{P}_j) \neq 0$ *and* $\mathcal{I}(\sigma')(\mathcal{P}_i) + \mathcal{I}(\sigma')(\mathcal{P}_j) \neq 0$, *it holds that* $\frac{\mathcal{I}(\sigma)(\mathcal{P}_i)}{\mathcal{I}(\sigma)(\mathcal{P}_i)+\mathcal{I}(\sigma)(\mathcal{P}_j)} = \frac{\mathcal{I}(\sigma')(\mathcal{P}_i)}{\mathcal{I}(\sigma')(\mathcal{P}_i)+\mathcal{I}(\sigma')(\mathcal{P}_j)}$.

The previous restriction generalizes to $\frac{\mathcal{I}(\sigma)(\mathcal{P}_j)}{\sum_{i\in I}\mathcal{I}(\sigma)(\mathcal{P}_i)} = \frac{\mathcal{I}(\sigma')(\mathcal{P}_j)}{\sum_{i\in I}\mathcal{I}(\sigma')(\mathcal{P}_i)}$, where $I \subseteq \{1,\ldots,n\}$ and $j \in I$ [14, Theorem 3.4].

3 Distributed Schedulers in Systems with Secrecy

Recall the example of Fig. 1 and consider the following paths:

$$\sigma_a = \langle c_0, sa_0, sb_0, ad_0\rangle\, \mu\, \langle c_1, sa_1, sb_1, ad_1\rangle\, a_1\, \langle c_3, sa_2, sb_1, ad_1\rangle\, b_0\, \langle c_5, sa_2, sb_3, ad_1\rangle$$
$$\sigma_b = \langle c_0, sa_0, sb_0, ad_0\rangle\, \mu\, \langle c_2, sa_1, sb_1, ad_1\rangle\, a_0\, \langle c_4, sa_3, sb_1, ad_1\rangle\, b_1\, \langle c_6, sa_3, sb_2, ad_1\rangle$$
$$\sigma'_a = \sigma_a\, ack_a\, \langle c_5, sa_4, sb_3, ad_1\rangle \qquad \sigma'_b = \sigma_b\, ack_b\, \langle c_6, sa_3, sb_4, ad_1\rangle$$

with $\mu(\langle c_1, sa_1, sb_1, ad_1\rangle) = \mu(\langle c_2, sa_1, sb_1, ad_1\rangle) = \frac{1}{2}$.

σ_a and σ_b are the only two possible paths of the system in which $\mathcal{C}_l$ executes all its outputs before any other component. σ'_a (resp., σ'_b) is the path in which server $\mathcal{S}_a$ (resp., $\mathcal{S}_b$) acknowledge reception after both servers were contacted by the client $\mathcal{C}_l$. Define the interleaving scheduler $\mathcal{I}$ by $\mathcal{I}(\sigma_a)(\mathcal{S}_a) = \mathcal{I}(\sigma_b)(\mathcal{S}_b) = \mathcal{I}(\sigma'_a)(\mathcal{A}) = \mathcal{I}(\sigma'_b)(\mathcal{A}) = 1$, and $\mathcal{I}(\sigma)(\mathcal{C}_l) = 1$ if σ is a prefix of σ_a or σ_b. The definition of $\mathcal{I}$ in any other case is not relevant as long as it satisfies the condition of Def. 7. Then $\mathcal{I}$ satisfies in general the condition of Def. 7. (Notice, in particular, that $\sigma_a[\mathcal{S}_a] \neq \sigma_b[\mathcal{S}_a]$ and $\sigma_a[\mathcal{S}_b] \neq \sigma_b[\mathcal{S}_b]$.)

Define the local scheduler for the adversary $\mathcal{A}$ by $\eta_{\mathcal{A}}(\sigma\, ack_a\, ad_1)(ga_1) = \eta_{\mathcal{A}}(\sigma\, ack_b\, ad_1)(gb_1) = 1$. I.e., the scheduler chooses with probability 1 to guess that server $\mathcal{S}_a$ has being selected (action ga_1) if the last acknowledgement it observes comes from $\mathcal{S}_a$ (action ack_a). Instead, if $\mathcal{S}_b$ is the last to acknowledge, the scheduler choses to guess $\mathcal{S}_b$. Note that all other local schedulers are trivial.

Let η be the strongly distributed scheduler obtained by properly combining the previous interleaving and local schedulers. It is not difficult to verify that

$$\begin{aligned}&\Pr_\eta\left((\sigma'_a\, ga_1\, \langle c_5, sa_4, sb_3, ad_2\rangle)^\uparrow \cup (\sigma'_b\, gb_1\, \langle c_6, sa_3, sb_4, ad_3\rangle)^\uparrow\right) = \\ &\quad = \Pr_\eta(\sigma'^\uparrow_a) \cdot \mathcal{I}(\sigma'_a)(\mathcal{A}) \cdot \eta_{\mathcal{A}}(\sigma'_a[\mathcal{A}])(ga_1) + \Pr_\eta(\sigma'^\uparrow_b) \cdot \mathcal{I}(\sigma'_b)(\mathcal{A}) \cdot \eta_{\mathcal{A}}(\sigma'_b[\mathcal{A}])(gb_1) \\ &\quad = \Pr_\eta(\sigma'^\uparrow_a) + \Pr_\eta(\sigma'^\uparrow_b) = \tfrac{1}{2} + \tfrac{1}{2} = 1.\end{aligned}$$

Notice that the adversary $\mathcal{A}$ guesses right whenever the system reaches a state of the form $\langle c_5, *, *, ad_2\rangle$ or $\langle c_6, *, *, ad_3\rangle$. Therefore, the previous calculation states that the adversary guesses the server chosen by the client with probability 1.

It is nonetheless clear that there is no apparent reason for the adversary $\mathcal{A}$ to guess right all the time. In fact, scheduler $\eta_{\mathcal{A}}$ is defined in a pretty arbitrary manner. The correct guessing is actually a consequence of the interleaving scheduler $\mathcal{I}$ that let the chosen server acknowledge first and immediately after it passes the control to the adversary $\mathcal{A}$ (hence making $\mathcal{A}$ guess through the arbitrary scheduler $\eta_{\mathcal{A}}$).

Observe that the interleaving scheduler of the previous example decides which is the next enabled component based on the outcome of a secret. However, a secret should not be directly observed by the environment. Hence, the interleaving should not be able to distinguish action a_1 from a_0 and neither b_1 from b_0. Similarly, internal states of a component that only differ on the value of confidential information should not influence the decision of the interleaving scheduler.

Therefore the notion of a valid interleaving scheduler of Def. 7 needs to be strengthen. In the new definition, the interleaving scheduler has to consider the same relative (i.e. conditional) probabilities for two components if both of them show the same behaviour to the environment. By "showing the same behaviour" we mean the projected traces are the same *after hiding* secret information.

The way in which we have chosen to hide information is through an equivalence relation. Thus, two actions that are equivalent share a secret and should not be distinguished by the environment, and similarly for states. This idea of indistinguishability is local to each component. So for each component $\mathcal{P}_i$ we consider an equivalence relation $\sim_{A_i}$ on actions and another equivalence relation $\sim_{S_i}$ on states. In our example, we need to define equivalence relations for $\mathcal{C}_l$, $\mathcal{S}_a$, and $\mathcal{S}_b$ such that:

$$a_1 \sim_{A_{\mathcal{C}_l}} a_0 \qquad a_1 \sim_{A_{\mathcal{S}_a}} a_0 \qquad c_1 \sim_{S_{\mathcal{C}_l}} c_2 \qquad sa_2 \sim_{S_{\mathcal{S}_a}} sa_3 \qquad sb_2 \sim_{S_{\mathcal{S}_b}} sb_3$$
$$b_1 \sim_{A_{\mathcal{C}_l}} b_0 \qquad\qquad c_3 \sim_{S_{\mathcal{C}_l}} c_4 \qquad sa_4 \sim_{S_{\mathcal{S}_a}} sa_5 \qquad sb_4 \sim_{S_{\mathcal{S}_b}} sb_5$$
$$b_1 \sim_{A_{\mathcal{S}_b}} b_0 \qquad c_5 \sim_{S_{\mathcal{C}_l}} c_6$$

This relations can be lifted to an equivalence relation on paths as expected: if $\sim_{S_i} \subseteq S_i \times S_i$ and $\sim_{A_i} \subseteq A_i \times A_i$ are equivalence relations, we define $\sim_{\mathcal{P}_i} \subseteq \mathit{Paths}\,(\mathcal{P}_i) \times \mathit{Paths}\,(\mathcal{P}_i)$ recursively by

$$\begin{aligned} s \sim_{\mathcal{P}_i} s' &\Leftrightarrow s \sim_{S_i} s' \\ \sigma a s \sim_{\mathcal{P}_i} \sigma' a' s' &\Leftrightarrow \sigma \sim_{\mathcal{P}_i} \sigma' \wedge a \sim_{A_i} a' \wedge s \sim_{S_i} s' \\ \sigma \mu s \sim_{\mathcal{P}_i} \sigma' \mu' s' &\Leftrightarrow \sigma \sim_{\mathcal{P}_i} \sigma' \wedge s \sim_{S_i} s' \end{aligned}$$

Notice that in our example, $\sigma_a\,[\mathcal{S}_a] \sim_{\mathcal{S}_a} \sigma_b\,[\mathcal{S}_a]$, $\sigma_a\,[\mathcal{S}_b] \sim_{\mathcal{S}_b} \sigma_b\,[\mathcal{S}_b]$, and $\sigma_a\,[\mathcal{C}_l] \sim_{\mathcal{C}_l} \sigma_b\,[\mathcal{C}_l]$. Therefore, under some secrecy assumptions, the environment is not able to distinguish the local execution of $\mathcal{S}_a$ under σ_a from the local execution under σ_b (and similarly $\mathcal{S}_b$ and $\mathcal{C}_l$). Hence, the interleaving scheduler should not make a difference on the relative probabilities of choosing $\mathcal{S}_a$ or $\mathcal{S}_b$. Therefore, we define distributed scheduler under secrecy as follows.

Definition 8. *Let $\mathcal{C} = \mathcal{P}_1 \parallel \cdots \parallel \mathcal{P}_n$ and let $\sim_{S_i} \subseteq S_i \times S_i, \sim_{A_i} \subseteq A_i \times A_i$, with $i = 1, \ldots, n$, be equivalence relations. A scheduler η of $\mathcal{C}$ is a* distributed

scheduler under secrecy *if it is distributed and, for every* $I \subseteq \{1, \ldots, n\}$*, and* $\sigma, \sigma' \in Paths_{fin}(\mathcal{C})$*, the interleaving scheduler* $\mathcal{I}$ *of* η *satisfies that, whenever*

(i) $\sigma[\mathcal{P}_i] \sim_{\mathcal{P}_i} \sigma'[\mathcal{P}_i]$ *for all* $i \in I$,
(ii) $\sum_{i \in I} \mathcal{I}(\sigma)(\mathcal{P}_i) \neq 0$ *and* $\sum_{i \in I} \mathcal{I}(\sigma')(\mathcal{P}_i) \neq 0$, *and*
(iii) $enab(\sigma[\mathcal{P}_i]) \neq \emptyset \Leftrightarrow enab(\sigma'[\mathcal{P}_i]) \neq \emptyset$, *for all* $i \in I$,

it holds that, for every $j \in I$, $\frac{\mathcal{I}(\sigma)(\mathcal{P}_j)}{\sum_{i \in I} \mathcal{I}(\sigma)(\mathcal{P}_i)} = \frac{\mathcal{I}(\sigma')(\mathcal{P}_j)}{\sum_{i \in I} \mathcal{I}(\sigma')(\mathcal{P}_i)}$. *We let* $\mathcal{DSS}$ *denote the set of all* distributed scheduler under secrecy.

Just like for the case of strongly distributed schedulers, it suffices to consider a pair of components for the restriction of $\mathcal{I}$ (i.e., sets I s.t. $\#I = 2$). We choose instead this more general definition because it plays an important role later.

Conditions (i) and (ii) already appear in the definition of strongly distributed schedulers (see Def. 7) only that (i) is consider under equality rather than the secrecy equivalences $\sim_{\mathcal{P}_i}$. Condition (iii) is new here and not needed in Def. 7. This has to do with the fact that, though $\sigma[\mathcal{P}_i]$ and $\sigma'[\mathcal{P}_i]$ may appear the same to the environment due to hidden secrets, they may be different executions of $\mathcal{P}_i$ and hence enable different sets of output actions. This is not the case for Def. 7 in which item (i) requires that $\sigma[\mathcal{P}_i] = \sigma'[\mathcal{P}_i]$. In fact, strongly distributed schedulers are a particular case of distributed schedulers under secrecy in which there is no secret (i.e. $\sim_{A_i}$ and $\sim_{S_i}$ are the identity relation).

Returning to our running example, notice that the interleaving scheduler defined at the beginning of this section is *not* a valid interleaving scheduler for a distributed scheduler under secrecy. In effect, notice that (i) $\sigma_a[\mathcal{S}_a] \sim_{\mathcal{S}_a} \sigma_b[\mathcal{S}_a]$ and $\sigma_a[\mathcal{S}_b] \sim_{\mathcal{S}_b} \sigma_b[\mathcal{S}_b]$, (ii) $\mathcal{I}(\sigma_a)(\mathcal{S}_a) + \mathcal{I}(\sigma_a)(\mathcal{S}_b) = 1$ and $\mathcal{I}(\sigma_b)(\mathcal{S}_a) + \mathcal{I}(\sigma_b)(\mathcal{S}_b) = 1$ (since $\mathcal{I}(\sigma_a)(\mathcal{S}_a) = \mathcal{I}(\sigma_b)(\mathcal{S}_b) = 1$), and (iii) $enab(\sigma_a[\mathcal{S}_a]) = enab(\sigma_b[\mathcal{S}_a]) = \{ack_a\}$ and $enab(\sigma_a[\mathcal{S}_b]) = enab(\sigma_b[\mathcal{S}_b]) = \{ack_b\}$; hence conditions (i), (ii), and (iii) from Def. 8 hold. However, $\frac{\mathcal{I}(\sigma_a)(\mathcal{S}_a)}{\mathcal{I}(\sigma_a)(\mathcal{S}_a)+\mathcal{I}(\sigma_a)(\mathcal{S}_b)} = 1 \neq 0 = \frac{\mathcal{I}(\sigma_b)(\mathcal{S}_a)}{\mathcal{I}(\sigma_b)(\mathcal{S}_a)+\mathcal{I}(\sigma_b)(\mathcal{S}_b)}$, contradicting the requirement on $\mathcal{I}$ in Def. 8.

4 Parametric Characterization

A scheduler resolves all nondeterministic choices of an I/O-IPC thorough probabilistic choices. Therefore, it defines an (infinite) Markov chain which is a particular instance of the original I/O-IPC where all nondeterminism has been replaced by a probabilistic transition.

To illustrate this, consider our example of Fig. 1 with component $\mathcal{A}$ replaced by the one in Fig. 2 (which we adopt for simplicity). The resolution through a scheduler of the composed system would look very much like the tree of Fig. 3, except that variables x_i should be omitted and variables y_i should be interpreted as probability values in the interval $[0, 1]$ properly defining a probabilistic distribution (e.g., $y_1 = 1$ and $y_3 + y_4 = 1$). However, if variables y_i are not interpreted we

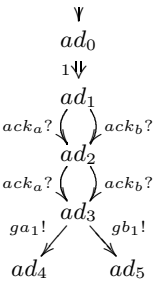


Fig. 2. $\mathcal{A}$

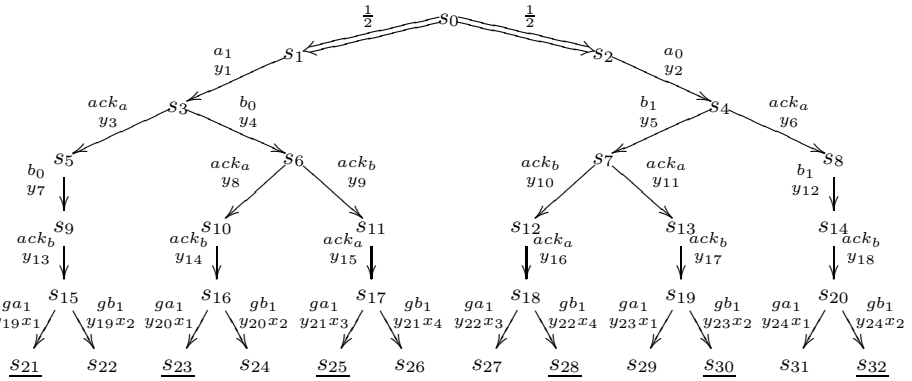


Fig. 3. Parametric scheduler η. States are the obvious tuples. Underlined states are guessing states.

could think of such parametric tree as a symbolic representation of *any possible* scheduler. (Keep ignoring x_i's by the time being.)

The probability of path $\sigma = s_0\,\mu\,s_1\,a_1\,s_3\,b_0\,s_6\,ack_a\,s_{10}\,ack_b\,s_{16}$ can be calculated using Def. 4 for the *symbolic* scheduler: $\Pr(\sigma^{\uparrow}) = \frac{1}{2} \cdot y_1 \cdot y_4 \cdot y_8 \cdot y_{14}$.

To construct a distributed scheduler under secrecy, we need to consider the interleaving scheduler and the local schedulers. Notice that in our example, only the local scheduler of $\mathcal{A}$ is relevant. In the parametric scheduler η of Fig. 3, variables y_i's correspond to the probabilistic choices of the interleaving scheduler, while variables x_j's correspond to the probabilistic choices of the local scheduler of $\mathcal{A}$ (all other local schedulers only have trivial choices and hence we omit them). The multiplication of variables y_i's and x_j's in the last step corresponds to the composition of the interleaving scheduler with the local scheduler in order to define the distributed scheduler η as it is in Def. 6 (which extends to Def. 8). Notice that, contrarily to the fact that variables y_i's are all different, x_1, x_2, x_3, and x_4 repeat in several branches. This has to do with the fact that some local paths of $\mathcal{A}$ are the same for different paths of the composed system. For example, the choice of the local scheduler of $\mathcal{A}$ at states s_{15}, s_{16}, s_{19}, and s_{20} is determined by the same local path $ad_0\,\mu'\,ad_1\,ack_a\,ad_2\,ack_b\,ad_3$, with $\mu'(ad_1) = 1$. (Notice that, e.g. $(s_0\,\mu\,s_1\,a_1\,s_3\,ack_a\,s_5\,b_0\,s_9\,ack_b\,s_{15})\,[\mathcal{A}] = ad_0\,\mu'\,ad_1\,ack_a\,ad_2\,ack_b\,ad_3$.)

Following Def. 4 and equation (3), the *parametric* probability of reaching a guessing state (which are underlined in Fig. 3), is given by the polynomial

$$\frac{1}{2} \cdot y_1 \cdot y_3 \cdot y_7 \cdot y_{13} \cdot y_{19} \cdot x_1 + \frac{1}{2} \cdot y_1 \cdot y_4 \cdot y_8 \cdot y_{14} \cdot y_{20} \cdot x_1 + \frac{1}{2} \cdot y_1 \cdot y_4 \cdot y_9 \cdot y_{15} \cdot y_{21} \cdot x_3 +$$
$$\frac{1}{2} \cdot y_2 \cdot y_5 \cdot y_{10} \cdot y_{16} \cdot y_{22} \cdot x_4 + \frac{1}{2} \cdot y_2 \cdot y_5 \cdot y_{11} \cdot y_{17} \cdot y_{23} \cdot x_2 + \frac{1}{2} \cdot y_2 \cdot y_6 \cdot y_{12} \cdot y_{18} \cdot y_{24} \cdot x_2\,.$$

Maximizing (resp. minimizing) the previous polynomial under the obvious constraints (each variable takes a value within $[0, 1]$, and they define proper probability distribution, e.g. $y_3 + y_4 = 1$), yields to the maximum (resp. minimum) probability under distributed schedulers. This was presented in [5].

However, this is not sufficient to characterize distributed schedulers under secrecy. Notice that, for $\sigma_a = s_0\mu s_1 a_1 s_3 b_0 s_6$ and $\sigma_b = s_0\mu s_2 a_0 s_4 b_1 s_7$ (which are

the same σ_a and σ_b of Sec. 3), and components $\mathcal{S}_a$ and $\mathcal{S}_b$, we are under conditions (i), (ii), and (iii) of Def. 8. Therefore, it should hold that $\frac{\mathcal{I}(\sigma_a)(\mathcal{S}_a)}{\mathcal{I}(\sigma_a)(\mathcal{S}_a)+\mathcal{I}(\sigma_a)(\mathcal{S}_b)} = \frac{\mathcal{I}(\sigma_b)(\mathcal{S}_a)}{\mathcal{I}(\sigma_b)(\mathcal{S}_a)+\mathcal{I}(\sigma_b)(\mathcal{S}_b)}$. Since $\mathcal{I}(\sigma_a)(\mathcal{S}_a) = y_8$, $\mathcal{I}(\sigma_a)(\mathcal{S}_b) = y_9$, $\mathcal{I}(\sigma_b)(\mathcal{S}_a) = y_{11}$ and $\mathcal{I}(\sigma_b)(\mathcal{S}_b) = y_{10}$, this means that constraint $\frac{y_8}{y_8+y_9} = \frac{y_{11}}{y_{10}+y_{11}}$ needs to be considered in the optimization problem. Similarly, constraints $\frac{y_9}{y_8+y_9} = \frac{y_{10}}{y_{10}+y_{11}}$, $\frac{y_3}{y_3+y_4} = \frac{y_6}{y_5+y_6}$, and $\frac{y_4}{y_3+y_4} = \frac{y_5}{y_5+y_6}$ are also needed.

In the following, we give the formal construction of the optimization problem. Let $Paths^{\leq t}(\mathcal{C}) = \{\sigma \in Paths_{fin}(\mathcal{C}) \mid time(\sigma) \leq t\}$. We consider the following set of variables

$$\begin{aligned} V = \big\{ & y^i_\sigma \mid \sigma \in Paths^{\leq t}(\mathcal{C}) \wedge 1 \leq i \leq \#\mathcal{C} \wedge enab(\sigma[\mathcal{P}_i]) \neq \emptyset \big\} \cup \\ & \big\{ x^a_{\sigma[\mathcal{P}_i]} \mid \sigma \in Paths^{\leq t}(\mathcal{C}) \wedge 1 \leq i \leq \#\mathcal{C} \wedge a \in enab(\sigma[\mathcal{P}_i]) \big\} \cup \\ & \big\{ w^{j,I}_f \mid I \subseteq \{1,\ldots,\#\mathcal{C}\} \wedge j \in I \wedge f : I \to Paths_{fin} \wedge \\ & \qquad \exists \sigma \in Paths^{\leq t}(\mathcal{C}) : \forall i \in I : enab(\sigma[\mathcal{P}_i]) \neq \emptyset \wedge f(i) = [\sigma[\mathcal{P}_i]]_{\sim_i} \big\} \end{aligned} \tag{4}$$

Variables y^i_σ and $x^a_{\sigma[\mathcal{P}_i]}$ are associated to the interleaving and local schedulers respectively, and we expect that $\mathcal{I}(\sigma)(i) = y^i_\sigma$ and $\eta_{\mathcal{P}_i}(\sigma[\mathcal{P}_i])(a) = x^a_{\sigma[\mathcal{P}_i]}$. Variables $w^{j,I}_f$ are associated to the restriction of the interleaving scheduler in Def. 8. We expect that $w^{j,I}_f = \frac{\mathcal{I}(\sigma)(\mathcal{P}_j)}{\sum_{i\in I}\mathcal{I}(\sigma)(\mathcal{P}_i)} = \frac{y^j_\sigma}{\sum_{i\in I} y^i_\sigma}$ whenever $f(i) = [\sigma[\mathcal{P}_i]]_{\sim_i}$ for all $i \in I$. Notice that, if there is another σ' such that $f(i) = [\sigma'[\mathcal{P}_i]]_{\sim_i}$ for all $i \in I$, then also $w^{j,I}_f = \frac{\mathcal{I}(\sigma')(\mathcal{P}_j)}{\sum_{i\in I}\mathcal{I}(\sigma')(\mathcal{P}_i)}$. This ensure the desired equality.

In our example of Fig. 3, if $\sigma = s_0\,\mu\,s_1\,a_1\,s_3\,b_0\,s_6$, then y_8 and y_9 correspond respectively to $y^{\mathcal{S}_a}_\sigma$ and $y^{\mathcal{S}_b}_\sigma$. If $\hat{\sigma} = ad_0\,\mu'\,ad_1\,ack_a\,ad_2\,ack_b\,ad_3$, then x_1 and x_2 correspond respectively to $x^{ga_1}_{\hat{\sigma}}$ and $x^{gb_1}_{\hat{\sigma}}$. Moreover, notice that if $\sigma' = s_0\,\mu\,s_1\,a_1\,s_3\,b_0\,s_6\,ack_a\,s_{10}\,ack_b\,s_{16}$ and $\sigma'' = s_0\,\mu\,s_1\,a_1\,s_3\,ack_a\,s_5\,b_0\,s_9\,ack_b\,s_{15}$, then $x^{ga_1}_{\hat{\sigma}}$, $x^{ga_1}_{\sigma'[\mathcal{A}]}$, and $x^{ga_1}_{\sigma''[\mathcal{A}]}$ are the same variable.

Let G be the set of goal states and let t be the time bound of the time-bounded reachability property under study. Let $Paths^{\leq t}_G(\mathcal{C}) = Paths^{\leq t}(\mathcal{C}) \cap \bar{G}$. The function $\mathbf{P}$ that assigns a polynomial term with variables in V to each path in $Paths^{\leq t}_G(\mathcal{C})$, is defined by

$$\begin{aligned} \mathbf{P}(\hat{s}_\mathcal{C}) &= 1 \\ \mathbf{P}(\sigma\alpha s) &= \begin{cases} \mathbf{P}(\sigma)\cdot y^i_\sigma \cdot x^\alpha_{\sigma[\mathcal{P}_i]} & \text{if } enab(\sigma) \neq \emptyset \wedge \alpha \in A^O_{\mathcal{P}_i} \wedge last(\sigma) \xrightarrow{\alpha} s \\ \mathbf{P}(\sigma)\cdot\alpha(s) & \text{if } enab(\sigma) = \emptyset \wedge last(\sigma) \Rightarrow \alpha \end{cases} \\ \mathbf{P}(\sigma) &= 0 \qquad \text{in any other case} \end{aligned} \tag{5}$$

Then, following (3), the objective polynomial of the optimization problem is

$$\sum_{\sigma \in Paths^{\leq t}_G(\mathcal{C})} \mathbf{P}(\sigma) \tag{6}$$

and it is subject to the following constraints:

$$0 \leq v \leq 1 \qquad \text{if } v \in V \tag{7a}$$

$$\textstyle\sum_{a \in A} x^a_{\sigma[\mathcal{P}_i]} = 1 \qquad \text{if } \sigma \in \mathit{Paths}^{\leq t}(\mathcal{C}),\ 1 \leq i \leq \#\mathcal{C},\ A = \mathit{enab}(\sigma[\mathcal{P}_i]) \tag{7b}$$

$$\textstyle\sum_{i \in I} y^i_\sigma = 1 \qquad \text{if } \sigma \in \mathit{Paths}^{\leq t}(\mathcal{C}),\ I = \{i \mid 1 \leq i \leq \#\mathcal{C}, \mathit{enab}(\sigma[\mathcal{P}_i]) \neq \emptyset\} \tag{7c}$$

$$w^{j,I}_f\left(\textstyle\sum_{i \in I} y^i_\sigma\right) = y^j_\sigma \ \text{ if } \begin{cases} \sigma \in \mathit{Paths}^{\leq t}(\mathcal{C}),\ I \subseteq \{i \mid 1 \leq i \leq \#\mathcal{C}, \mathit{enab}(\sigma[\mathcal{P}_i]) \neq \emptyset\}, \\ j \in I,\ \forall i \in I : \mathit{enab}(\sigma[\mathcal{P}_i]) \neq \emptyset \wedge f(i) = [\sigma[\mathcal{P}_i]]_{\sim_i} \end{cases} \tag{7d}$$

Equations (7a–7c) ensure that the probabilisitic choices of the local and interleaving schedulers are well defined. Equation (7d) is a rewriting of $w^{j,I}_f = \frac{y^j_\sigma}{\sum_{i \in I} y^i_\sigma}$ to avoid possible division by 0. (Notice that, when $\sum_{i \in I} y^i_\sigma = 0$, the constraint becomes trivially valid.) These constraints encode the restriction on the interleaving scheduler. In effect, let σ and σ' be such that, for all $i \in I$, $\sigma[\mathcal{P}_i] \sim_{\mathcal{P}_i} \sigma'[\mathcal{P}_i]$, $\mathit{enab}(\sigma[\mathcal{P}_i]) \neq \emptyset$ and $\mathit{enab}(\sigma'[\mathcal{P}_i]) \neq \emptyset$. Then, equations $w^{j,I}_f(\sum_{i \in I} y^i_\sigma) = y^j_\sigma$ and $w^{j,I}_f(\sum_{i \in I} y^i_{\sigma'}) = y^j_{\sigma'}$, with $f(i) = [\sigma[\mathcal{P}_i]]_{\sim_i} = [\sigma'[\mathcal{P}_i]]_{\sim_i}$, are present in the constraints and hence $\frac{y^j_\sigma}{\sum_{i \in I} y^i_\sigma} = \frac{y^j_{\sigma'}}{\sum_{i \in I} y^i_{\sigma'}}$.

We have the following theorem, whose proof we omit as it follows closely the proof of [5, Theorem 2].

Theorem 1. *Time-bounded reachability for a distributed I/O-IPC $\mathcal{C}$ under the class $\mathcal{DSS}$ is equivalent to solve the polynomial optimization problem with objective function in (6) under constraints (7). The result of maximizing (resp. minimizing) polynomial (6) is* $\sup_{\eta \in \mathcal{DSS}} \Pr_\eta(\Diamond^{\leq t} G)$ *(resp.* $\inf_{\eta \in \mathcal{DSS}} \Pr_\eta(\Diamond^{\leq t} G)$*).*

5 Implementation

We developed a prototypical tool to produce the optimization problem. It takes as input the model of each component of the system (as transitions between states with the initial state and equivalence classes indicated); a list of goal states G and a time-bound t.

The tool computes elements of $\mathit{Paths}^{\leq t}(\mathcal{C})$ from the composed initial state following rules (1) and (2). While generating paths, new variables and constraints are defined if the conditions of Eq. (7) hold. Also the expression $\mathbf{P}$ of the path is determined according to Eq. (5). This process is iterated for each generated path as long as its last state is not in G and it has successors in $\mathit{Paths}^{\leq t}(\mathcal{C})$, i.e., as long as the enabled transitions also lead to a finite path within the requested time-bound. When all the elements of $\mathit{Paths}^{\leq t}_G(\mathcal{C})$ are identified, the tool exports the constrained optimization problem. For an exact solution, we set a quantifier elimination problem over the real domain as an input for Redlog[1] (within the Reduce computer algebra system) or QEPCAD[2]. For a numeric solution, we generate source code for compiling against the IPOPT[3] library.

[1] `http://redlog.dolzmann.de`
[2] `http://www.usna.edu/cs/~qepcad`
[3] `https://projects.coin-or.org/Ipopt`

The complexity of the algorithm is clear: the optimization problem grows exponentially with the number of components in the system and the degree of local nondeterminism. Therefore it is essential to find ways to reduce it to a smaller equivalent optimization problem. Curiously enough, the fact of considering arbitrary summations in the constraints for the interleaving scheduler (see Def. 8) rather than only binary, permits a drastic reduction on the number of variables and constraints as well as the size and degree of the objective polynomial.

Notice that if $I = \{i \mid enab(\sigma\,[\mathcal{P}_i])\}$ in (7d), we know by (7c) that $\sum_{i\in I} y^i_\sigma = 1$. As a consequence, $w^{j,I}_f = y^j_\sigma$ for all $j \in I$. If there is another σ' such that $I = \{i \mid enab(\sigma'\,[\mathcal{P}_i])\}$ and $f(i) = [\sigma\,[\mathcal{P}_i]]_{\sim_i}$ for all $i \in I$, then $w^{j,I}_f = y^j_{\sigma'} = y^j_\sigma$ for all $j \in I$. This only simplification has allowed us to eliminate a large number of variables and, more importantly, non-linear constraint introduced by (7d).

Moreover, this unification of variables reduces also the size and degree of the polynomial. Often, after substitution, we find out that by factorizing, we can single out $\sum_{i\in I} y^i_\sigma$ which, when $I = \{i \mid enab(\sigma\,[\mathcal{P}_i])\}$, equals to 1. This reduces the number of terms in $I - 1$ and the degree of these terms by 1.

Case Study: The Dining Cryptographers Protocol. As a case study we automated the verification of sender untraceability in dc-net, a protocol inspired on a solution for the *dining cryptographers* problem [8]: Three cryptographers are having dinner at a restaurant while the waiter informs that the bill has been paid anonymously. If one of the cryptographers is paying they want to respect his anonymity, but they like to know if their boss is paying. Briefly, the protocol goes as follows. Each participant toss a fair coin and share the outcome with his neighbour at the left, then he publicly announces if his outcome and the one of the neighbour are the same, but if the participant is paying, he announces the opposite. If the number of "different" announces is odd, one of the participants is paying but the others cannot distinguish which one.

This protocol has been analyzed a large number of times with different techniques. It is noticeable that most of the proofs, if not all (in particular model-based fully automated proofs), fix the order in which the participants make their announcements. Most generally, this has to do precisely with the inability of the techniques to control the arbitrariness of the scheduler. The model we verify does not impose such restrictions. We anyway prove that the protocol preserves anonymity when the participants' announcement do not follow a fixed order.

We consider the case of three cryptographers C_1, C_2 and C_3. We fix the probability of the boss paying in $\frac{1}{2}$. Otherwise, the probability of any of the participants paying is uniformly distributed ($\frac{1}{6}$ each). We want to know what is the probability P_{guess} that participant C_3 correctly guesses if C_1 or C_2 have paid, knowing that one of them has actually paid. We will then calculate the maximum and minimum probability of reaching the set of states $G = \{s \mid C_3 \text{ guesses } C_i \text{ and } C_i \text{ paid}, i = 1, 2\}$, say $P^+_{\Diamond G}$ and $P^-_{\Diamond G}$, respectively. Knowing that the probability that C_1 or C_2 have paid is $\frac{1}{3}$, if it turns out that $P^+_{\Diamond G} = P^-_{\Diamond G} = p$, the conditional probability that we are looking for is $P_{\text{guess}} = p/\frac{1}{3} = 3p$.

Indeed, after running our tool, we verified that $P^{+}_{\Diamond G} = P^{-}_{\Diamond G} = \frac{1}{6}$. Hence $P_{\text{guess}} = \frac{1}{2}$, proving that C_3 has no better knowledge than previously known (i.e. that any of C_1 or C_2 pay with probability $\frac{1}{2}$). The original system contains 9606 variables, 1348 linear constraints, 16176 nonlinear constraints and the polynomial has 3456 terms and degree 7. After unification of variables the numbers are 774, 200, 36, 3456, and 7, resp., and after factorization and elimination of irrelevant variables, numbers reduces to 351, 17, 0, 544, and 5, resp. Given the complexity of the optimization problem, we were unable to solve it exactly, but the numerical computation was almost immediate. On an aside note, a similar verification of our running example was solved using only the simplifications.

6 Concluding Remarks

Related Work. The interest on understanding and verifying probabilistic distributed systems under the assumption that not all information is shared by all components has appeared several times before in the literature (e.g. [9,14,11,1]). We base this work on extending the framework of [12] and the algorithm of [5].

Our ideas about equivalence relations and limiting the interleaving scheduler based on projections under equivalences are inspired in task-PIOAs [6]. Task-PIOAs are a variation of probabilistic I/O automata with an equivalence relation over the set of "controllable actions" of the composed system. The model restrict to output isolation and action determinism. The set of schedulers they considered are a combination of a *task schedule* with a regular total-information scheduler. The resulting scheduler maps equivalent execution fragments to probability measures that ensure that equivalent actions receive the same probability value. This roughly corresponds to our interleaving scheduling under secrecy assumptions. Because of output isolation, no other scheduler is needed. Despite that [6] provides techniques to analyze time-bounded attacks in Task-PIOA, to our knowledge, no fully automatic algorithm is provided.

[2] proposes another restricted family of schedulers over *tagged probabilistic automata*, a formalism with similar semantics to ours. The actions in composed systems are "tagged" with the components engaged in the action (or components, in case of synchronization). The set of enabled tags in a state is part of the observable behavior. Then, the scheduler is defined as a function from *observable* traces to tags. In this sense, the schedulers are quite like our interleaving schedulers, and no local scheduler is needed. Another restriction is that they only consider deterministic schedulers (which are strictly less expressive than randomised schedulers, see [14]). [2] also provides an automatic technique based on automorphism that check for sufficient conditions to ensure anonymity in systems whose components do not have internal nondeterminism (comparable to our output nondeterminism).

A somewhat similar approach is presented in [7] where labels are also used for the scheduler to resolve the nondeterminism. In this case the scheduler is provided explicitly as a deterministic component that only synchronizes with the system through labels. Again randomized schedulers are not considered. A particularity

of this work is that the "equivalence classes" (which are actually defined by how labels are associated to actions) can change dynamically.

Conclusion and Further Work. We refined the schedulers of [14] to deal with information hiding and adapted the technique of [5] to the new setting. Moreover, the generation of the polynomial optimization problem has been significantly improved, first by avoiding the generation of the intermediate parametric Markov chain, and then by adding simplification rules that drastically reduced the size of the optimization problem. In addition, the connection to numerical tools allow for effective and rapid calculations. In particular, our technique allowed for the verification of a more nondeterministic version of the dining cryptographers, without constraining the ordering of independent actions.

In the future, we propose to revise the synchronisation mechanism. Notice that, in the running example, adversary $\mathcal{A}$ does not observe the communication of the secret at all. However, it is reasonable that $\mathcal{A}$ observes the transmission of the *encrypted* secret, that is, $\mathcal{A}$ should synchronize with the equivalence class without knowing which particular action was executed. Another further work is to study the use of our framework to analize timing attacks, after all it is already prepared for it. Moreover, by properly bounding the numbers of steps of the attackers, we may explore the possibility of restricting the computational complexity of the attack.

References

1. Andrés, M.: Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems. Ph.D. thesis, Radboud Universiteit Nijmegen (2011)
2. Andrés, M.E., Palamidessi, C., van Rossum, P., Sokolova, A.: Information hiding in probabilistic concurrent systems. Theor. Comput. Sci. 412(28), 3072–3089 (2011)
3. Bhargava, M., Palamidessi, C.: Probabilistic anonymity. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 171–185. Springer, Heidelberg (2005)
4. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
5. Calin, G., Crouzen, P., D'Argenio, P.R., Hahn, E., Zhang, L.: Time-bounded reachability in distributed input/output interactive probabilistic chains. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 193–211. Springer, Heidelberg (2010); extended version: AVACS Technical Report No. 64 (June 2010)
6. Canetti, R., Cheung, L., Kaynar, D., Liskov, M., Lynch, N., Pereira, O., Segala, R.: Analyzing security protocols using time-bounded Task-PIOAs. Discrete Event Dynamic Systems 18, 111–159 (2008)
7. Chatzikokolakis, K., Palamidessi, C.: Making random choices invisible to the scheduler. Information and Computation 208(6), 694–715 (2010)
8. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
9. Cheung, L., Lynch, N., Segala, R., Vaandrager, F.: Switched PIOA: Parallel composition via distributed scheduling. Theor. Comput. Sci. 365(1-2), 83–108 (2006)

10. Coste, N., Hermanns, H., Lantreibecq, E., Serwe, W.: Towards performance prediction of compositional models in industrial GALS designs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 204–218. Springer, Heidelberg (2009)
11. Georgievska, S., Andova, S.: Retaining the probabilities in probabilistic testing theory. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 79–93. Springer, Heidelberg (2010)
12. Giro, S.: On the automatic verification of distributed probabilistic automata with partial information. Ph.D. thesis, Universidad Nacional de Córdoba (2010)
13. Giro, S., D'Argenio, P.R.: Quantitative model checking revisited: Neither decidable nor approximable. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 179–194. Springer, Heidelberg (2007)
14. Giro, S., D'Argenio, P.R.: On the expressive power of schedulers in distributed probabilistic systems. Electron. Notes Theor. Comput. Sci. 253(3), 45–71 (2009)
15. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for web transactions. ACM Trans. Inf. Syst. Secur. 1(1), 66–92 (1998)
16. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)
17. Wang, Y.: Real-time behaviour of asynchronous agents. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 502–520. Springer, Heidelberg (1990)

Modular Reasoning about Differential Privacy in a Probabilistic Process Calculus*

Lili Xu[1,2,3]

[1] INRIA and LIX, École Polytechnique, France
[2] State Key Lab. of Comp. Sci., Institute of Software, Chinese Academy of Sciences
[3] Graduate University, Chinese Academy of Sciences
xulili@ios.ac.cn

Abstract. The verification of systems for protecting sensitive and confidential information is becoming an increasingly important issue. Differential privacy is a promising notion of privacy originated from the community of statistical databases, and now widely adopted in various models of computation. We consider a probabilistic process calculus as a specification formalism for concurrent systems, and we propose a framework for reasoning about the degree of differential privacy provided by such systems. In particular, we investigate the preservation of the degree of privacy under composition via the various operators. We illustrate our idea by proving an anonymity-preservation property for a variant of the Crowds protocol for which the standard analyses from the literature are inapplicable. Finally, we make some preliminary steps towards automatically computing the degree of privacy of a system in a compositional way.

1 Introduction

The most recent developments and usages of information technologies such as data profiling in databases, or user tracking in pervasive computing, pose serious threats to the confidential information of the users. For instance, the social networks Twitter and Flickr carefully protect their user's data by anonymization, and yet Narayanan and Smatikov [17] were able to conceive a de-anonymization algorithm which could re-identify 30% of the people who have accounts in both of them, with only a 12% error rate. The verification of systems for protecting sensitive and confidential information is becoming an increasingly important issue in the modern world.

Many protocols for protecting confidential information have been proposed in the literature. In order to obfuscate the link between the secret and the public information, several of them use randomized mechanisms. Typical examples are the DCNets [6], Crowds [19], Onion Routing [23] and Freenet [7]. Another common denominator is that various entities involved in the system to verify occur as concurrent processes and present typically a nondeterministic behavior. It is therefore natural and standard to apply process calculi, and the adaptation of the process-algebraic framework to specify

* This work has been jointly supported by the National Natural Science Foundation of China (Grant No.60833001), the project ANR-09-BLAN-0169-01 PANDA and the INRIA large scale initiative CAPPRIS.

C. Palamidessi and M.D. Ryan (Eds.): TGC 2012, LNCS 8191, pp. 198–212, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and reason about security protocols is an active line of research. See e.g. the CCS approach [16,9,4]. In this paper, we consider a probabilistic extension of CCS, which we call CCS_p. In addition to the standard parallel composition and nondeterministic choice of CCS, CCS_p provides also a primitive for the probabilistic choice.

Several formalizations of the notion of protection have been proposed in the literature. Among those based on probability theory, there are *strong anonymity* and *conditional anonymity* [6,13,3] and *probable innocence* [19]. Different from the previous notions providing only true-or-false properties, the concepts from Information Theory [8] based on *entropy*, notably *mutual information* and *capacity*, express the degree of protection in a quantitative way.

Differential privacy [10,12,11] is a promising definition of confidentiality that has emerged recently from the field of statistical databases. It provides strong privacy guarantees, and requires fewer assumptions than the information-theoretical approach. We say that a system is ϵ-differentially private if for every pair of *adjacent* datasets (i.e. datasets which differ in the data of an individual only), the probabilities of obtaining a certain answer differ at most by a factor e^{ϵ}. Differential privacy captures the intuitive requirement that the (public) answer to a query should not be affected too much by the (private) data of each singular individual. In this paper we consider a version of differential privacy using a generic notion of adjacency, which provides the basis for formalizing also other security concepts, like anonymity.

The main contribution of this work is to investigate differential privacy for concurrent systems in the context of a probabilistic process calculus (CCS_p). We present a modular approach for reasoning about differential privacy, where the modularity is with respect to the constructs of CCS_p. More specifically, we show that the restriction operator, the probabilistic choice, the nondeterministic choice and a restricted form of parallel composition are safe under composition, in the sense that they do not decrease the privacy of a system. Compositionality plays an important role in the construction and analysis of security systems: Rather than analyzing a complex system as a whole, the safe constructs allow us to split the system in parts, analyze the degree of privacy of each part separately, and combine the results to obtain the global degree of privacy.

We illustrate our compositionality results by proving an anonymity-preservation property for an extension of the *Crowds* protocol [19]. Crowds is an anonymity protocol which allows Internet users to perform web transactions without revealing their private identity. This is achieved by using a chain of forwarders, chosen randomly, rather than sending the message directly to the final recipient. In the standard Crowds all members have the same probability of being used as forwarders, which gives the protocol a symmetric structure (cf. equations (13-14) in [5]). In practice, however, the protocol can be asymmetric, because a sender may trust some agents (as forwarders) more than others, or may prefer those which are geographically closer, or more stable, in order to achieve a better performance. In this paper, our extension of Crowds consists in allowing each member to have a set of *trusted* users to which they can forward the message. This breaks the symmetry properties of the original protocol, thus making the standard analyses of [19] inapplicable. In contrast, our compositional method gives a simple proof.

Finally, we make some preliminary steps towards automatically computing the degree of privacy of a CCS_p process in a compositional way. In this paper, we only

consider the case of finite processes in which the secret choices are all at the top level. We leave the general case for future work.

Nowadays differential privacy is widely adopted in many models of computation, for instance, it has been investigated in the context of a SQL-like language [15], a linear type system [18], a relational Hoare logic [2], in MapReduce for cloud computing [20], and in probabilistic labeled transition systems for interactive systems [24]. To the best of our knowledge, this paper is the first to investigate differential privacy within the setting of a process algebra.

Summary of Contributions Our contributions are three-fold. We present a modular approach for reasoning about differential privacy for protocols expressed in a probabilistic process algebra (CCS$_p$). We apply our compositionality method to prove an anonymity-preservation property for an extended version of Crowds, i.e. with member-wise trusted forwarders. We show an algorithm for computing the privacy degree of a finite process.

Plan of the paper In Sections 2 - 4 we review the preliminary notions of CCS$_p$ and differential privacy. In Section 5 we investigate the compositionality of differential privacy with respect to CCS$_p$ constructs. In Section 6, we apply our compositionality result to Crowds with trust. In Section 7, we present the algorithm for computing the degree of privacy. Finally, Section 8 and 9 discuss further related work and conclude. Detailed proofs of the theorem in Section 5 can be found in the full version of this paper [25].

Acknowledgments The author gratefully acknowledges the contributions of Catuscia Palamidessi and Kostas Chatzikokolakis. Discussions with them helped the author to substantially improve the paper.

2 Preliminaries

2.1 Probabilistic Automata

We recall the formalism of *probabilistic automata* [22], to be used as the operational semantics for the probabilistic CCS.

We denote the set of all discrete probability distributions over a set X by $Disc(X)$. For $x \in X$, we denote by $\delta(x)$ (the *Dirac measure* on x) the probability distribution that assigns probability 1 to $\{x\}$.

A (*simple*) *probabilistic automaton* is a tuple $\mathcal{M} = (\mathcal{P}, P_{init}, Act, \mathcal{T})$ where $\mathcal{P}$ is a set of states, $P_{init} \in \mathcal{P}$ is the *initial state*, Act is a set of labels and $\mathcal{T} \subseteq \mathcal{P} \times Act \times Disc(\mathcal{P})$ is a *transition relation*. Informally, if $(P, a, \mu) \in \mathcal{T}$ then there is a transition from the state P performing a label a and then leading to a distribution μ over a set of states instead of a single state. Intuitively, the idea is that the transition in $\mathcal{T}$ is chosen nondeterministically, and the target state among the ones allowed by μ is chosen probabilistically. A *fully probabilistic* automaton is a probabilistic automaton without nondeterminism, at each state only one transition can be chosen.

An *execution* α of a probabilistic automaton is a (possibly infinite) sequence of alternating states and labels $P_{init}a_0P_1a_1P_2a_2P_3\cdots$, such that for each i, there is a transition $(P_i, a_i, \mu_i) \in \mathcal{T}$ with $\mu_i(P_{i+1}) > 0$. We will use $exec^*(\mathcal{M})$ to represent the set of all the finite executions of $\mathcal{M}$, $exec(\mathcal{M})$ to represent the set of all the executions of $\mathcal{M}$, and $lst(\alpha)$ to denote the last state of a finite execution $\alpha \in exec^*(\mathcal{M})$.

The *execution tree* of $\mathcal{M}$, denoted by $etree(\mathcal{M})$, is an automaton $\mathcal{M}' = (\mathcal{P}', P'_{init}, Act, \mathcal{T}')$ such that $\mathcal{P}' \subseteq exec(\mathcal{M})$, $P'_{init} = P_{init}$, and $(\alpha, a, \mu') \in \mathcal{T}'$ if and only if $(lst(\alpha), a, \mu) \in \mathcal{T}$ for some μ and $\mu'(\alpha a P) = \mu(P)$ for all P. Intuitively, $etree(\mathcal{M})$ is produced by unfolding all the executions of $\mathcal{M}$.

A *scheduler* of a probabilistic automation $\mathcal{M}$ is a function $\zeta : exec^*(\mathcal{M}) \to \mathcal{T}$ such that $\zeta(\alpha) = (P, a, \mu) \in \mathcal{T}$ implies that $P = lst(\alpha)$. (We omit $\mathcal{M}$ when it is clear from the context.) The idea is that a scheduler resolves the nondeterminism by selecting a transition among the ones available in $\mathcal{T}$, based on the history of the execution.

The *execution tree* of $\mathcal{M}$ *relative to a scheduler* ζ, denoted by $etree(\mathcal{M}, \zeta)$, is a fully probabilistic automaton $\mathcal{M}'' = (\mathcal{P}'', P''_{init}, Act, \mathcal{T}'')$, obtained from $\mathcal{M}'$ by removing all the transitions in $\mathcal{T}'$ that are not chosen by the scheduler, that is, $(\alpha, a, \mu'') \in \mathcal{T}''$ if and only if $\zeta(\alpha) = (lst(\alpha), a, \mu)$ for some μ and $\mu''(\alpha a P) = \mu(P)$ for all P. Intuitively, $etree(\mathcal{M}, \zeta)$ is produced from $etree(\mathcal{M})$ by resolving all nondeterministic choices using ζ. Note that $etree(M, \zeta)$ is a simple and fully probabilistic automaton.

3 Probabilistic CCS

In this section, we present a probabilistic version of Milner's CCS [16], that allows for both nondeterministic and probabilistic choice. Following [4] we make a distinction between *observable* and *secret* labels, for applications to security systems and protocols.

We consider a countable set Act of labels a, partitioned into a set Sec of *secrets* s, a set Obs of *observables* o, and the silent action τ. For each $o \in Obs$, we assume a complementary label $\overline{o} \in Obs$ with the convention that $\overline{\overline{o}} = o$. The syntax of CCS$_p$ is:

$P ::=$		*process term*
	$\bigoplus_i p_i P_i$	*probabilistic choice*
$\mid$	$\biguplus_i s_i.P_i$	*secret choice* ($s_i \in Sec$)
$\mid$	$\biguplus_i r_i.P_i$	*nondeterministic choice* ($r_i \in Obs \cup \{\tau\}$)
$\mid$	$P \mid P$	*parallel composition*
$\mid$	$(\nu a)P$	*restriction*
$\mid$	$!\,P$	*replication*

The term $\bigoplus_i p_i P_i$, in which the p_is are positive rationals that sum up to one, represents a *blind probabilistic choice*, in the sense that the choice of the branch is decided randomly (according to the corresponding probabilities) and there is no visible label associated to the decision. We use the notation $P_1 \oplus_p P_2$ to represent a binary sum with $p_1 = p$ and $p_2 = 1 - p$. Similarly, we use $a_1.P_1 \uplus a_2.P_2$ to represent a binary secret or nondeterministic choice. Finally the term $\mathbf{0}$, representing the terminated process, is syntactic sugar for an empty (secret or nondeterministic) choice.

The operational semantics of a CCS$_p$ term P is a probabilistic automaton whose states $\mathcal{P}$ are the processes reachable from P, and whose transition relation is defined according to the rules in the Table 1, where we use $P \xrightarrow{a} \mu$ to represent the transition (P, a, μ). We denote by $\mu \mid P$ the measure μ' such that $\mu'(P' \mid P) = \mu(P')$ for all processes $P' \in \mathcal{P}$ and $\mu'(P'') = 0$ if P'' is not of the form $P' \mid P$. Similarly $(\nu a)\mu = \mu'$

Table 1. The semantics of CCS_p

$$\text{PROB}\ \frac{}{\bigoplus_i p_i\, P_i \xrightarrow{\tau} \sum_i p_i\, \delta(P_i)} \qquad \text{ACT}\ \frac{j \in I}{\biguplus_I a_i.P_i \xrightarrow{a_j} \delta(P_j)}$$

$$\text{PAR1}\ \frac{P_1 \xrightarrow{a} \mu}{P_1 \mid P_2 \xrightarrow{a} \mu \mid P_2} \qquad \text{PAR2}\ \frac{P_2 \xrightarrow{a} \mu}{P_1 \mid P_2 \xrightarrow{a} P_1 \mid \mu} \qquad \text{REP}\ \frac{P \mid !\,P \xrightarrow{a} \mu}{!\,P \xrightarrow{a} \mu \mid !\,P}$$

$$\text{COM}\ \frac{P_1 \xrightarrow{a} \delta(P_1') \quad P_2 \xrightarrow{\overline{a}} \delta(P_2')}{P_1 \mid P_2 \xrightarrow{\tau} \delta(P_1' \mid P_2')} \qquad \text{RES}\ \frac{P \xrightarrow{b} \mu \qquad b \neq a, \overline{a}}{(\nu a)P \xrightarrow{b} (\nu a)\mu}$$

such that $\mu'((\nu a)P) = \mu(P)$. A transition of the form $P \xrightarrow{a} \delta(P')$, having for target a Dirac measure, corresponds to a transition of a non-probabilistic automaton. From the rule PROB, we know that all transitions to non-Dirac measures are silent.

Following [4] we assume the secret labels to be the *inputs* of the system. Secrets are given in input to the scheduler and determine completely the secret choices. The scheduler then has to resolve the residual nondeterminism, which is originated by the nondeterministic choice and the parallel operator. From the observer's point of view, only the nondeterministic choices can be observed.

The definition of a scheduler of a CCS_p term is specified as follows. $X \rightharpoonup Y$ represents the partial functions from X to Y, and $\alpha_{|Sec}$ is the projection of α on Sec.

Definition 1. *Let P be a process in CCS_p and $\mathcal{M}$ be the probabilistic automaton generated by P. A scheduler is a function $\zeta : Sec^* \to exec^*(\mathcal{M}) \rightharpoonup \mathcal{T}$ such that if:*

(i) $\boldsymbol{s} = s_1 s_2 \ldots s_n$ and $\alpha_{|Sec} = s_1 s_2 \ldots s_m$ with $m < n$, and
(ii) there exists a transition $(lst(\alpha), a, \mu)$ such that $a \in Sec \Rightarrow a = s_{m+1}$

then $\zeta(\boldsymbol{s})(\alpha)$ is defined as one of such transitions. We will write $\zeta_{\boldsymbol{s}}(\alpha)$ for $\zeta(\boldsymbol{s})(\alpha)$.

We now define the *execution tree* of a CCS_p term, in a way similar to what is done in the probabilistic automata. The main difference is that in our case the execution tree depends not only on the scheduler, but also on the secret input.

Definition 2. *Let $\mathcal{M} = (\mathcal{P}, P_{init}, Act, \mathcal{T})$ be the probabilistic automaton generated by a CCS_p process P. Given an input $\boldsymbol{s}$ and a scheduler ζ, the* execution tree *of P, denoted by $etree(P, \boldsymbol{s}, \zeta)$, is a fully probabilistic automaton $\mathcal{M}' = (\mathcal{P}', P_{init}, Act, \mathcal{T}')$ such that:*

(i) $\mathcal{P}' \subseteq exec(\mathcal{M})$,
(ii) $(\alpha, a, \mu') \in \mathcal{T}'$ iff $\zeta_{\boldsymbol{s}}(\alpha) = (lst(\alpha), a, \mu)$ for some μ and $\mu'(\alpha a P) = \mu(P)$

Process Terms as Channels. We now show how CCS_p terms can be used to specify systems manipulating confidential information. A system can be seen as an information-theoretic channel [8]. Sequences of secret labels constitute the *secret information*

(or *secrets*), given in input to the channel, and sequences of observable labels constitute the *public information* (or *observables*), obtained as output from the channel. We denote secrets and observables by $\mathcal{S}$ and $\mathcal{O}$, and we assume that they have finite cardinality m and n, respectively. We also assume that each sequence in $\mathcal{S} \cup \mathcal{O}$ is finite. Thus, $\mathcal{S} \subseteq_{fin} Sec^*$ and $\mathcal{O} \subseteq_{fin} Obs^*$. This is usually enough to model the typical security systems, where each session is supposed to terminate in finite time.

Given an input $\boldsymbol{s} \in \mathcal{S}$, a run of the system will produce each $\boldsymbol{o} \in \mathcal{O}$ with a certain probability $p(\boldsymbol{o}|\boldsymbol{s})$ which depends on $\boldsymbol{s}$, on the randomized operations performed by the system, and also on the scheduler resolving the nondeterminism. The probabilities $p(\boldsymbol{o}|\boldsymbol{s})$, for a given scheduler ζ, constitute a $m \times n$ array M_ζ which is called the *matrix* of the channel, where the rows are indexed by the elements of $\mathcal{S}$ and the columns are indexed by the elements of $\mathcal{O}$. (See some examples of channel matrix in Section 7.2.)

Definition 3 ([4]). *Given a term P and a scheduler $\zeta : \mathcal{S} \to exec^* \rightharpoonup \mathcal{T}$, the matrix $M_\zeta(P)$ associated to P under ζ is defined as the matrix such that, for each row $\boldsymbol{s} \in \mathcal{S}$ and column $\boldsymbol{o} \in \mathcal{O}$, the element at their intersection, $p_\zeta(\boldsymbol{o}|\boldsymbol{s})$, is the probability of the set of the maximal executions in $etree(P, \boldsymbol{s}, \zeta)$ whose projection in Obs is $\boldsymbol{o}$.*

4 Differential Privacy

Differential Privacy [10] captures the idea that a query on a dataset does not provide too much information about a particular individual, regardless of whether the individual's record is in the dataset or not. In order to achieve this goal, typically some probabilistic noise is added to the answer. The formal definition is the following (where κ denotes the randomized answer, $\Pr$ the probability measure, and ϵ a finite non-negative number):

Definition 4 (Differential Privacy, [10]). *A mechanism κ provides ϵ-*differential privacy *iff for all datasets D_1 and D_2 differing in only one record, and for all $S \subseteq Range(\kappa)$,*

$$\Pr[\kappa(D_1) \in S] \leq e^\epsilon \Pr[\kappa(D_2) \in S]$$

Clearly, the smaller the value ϵ is, the higher is the protection.

We now adapt the notion of differential privacy to our framework. We consider a symmetric adjacency relation $\sim$ between secrets, which extends the dataset-based notion of "differing for only one record" to more general settings. The confidential data can be complex information like sequences of keys or tuples of individual data (see Example 2).

Example 1 (Anonymity). In the case of anonymity, the confidential data $\mathcal{S}$ are the agents' identities. Since the identities are just names without any particular structure, it is natural to assume that each name is adjacent to any other. Hence $(\mathcal{S}, \sim)$ is a clique, i.e. for all $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathcal{S}$ we have $\boldsymbol{s}_1 \sim \boldsymbol{s}_2$.

Example 2 (Geolocation). In the case of geolocation, the confidential data are the coordinates $(latitude, longitude)$ of a point on the earth's surface. If the purpose is to protect the exact location, a good definition of adjacency is: two points are adjacent if their Manhattan distance is 1, i.e. $(x_1, y_1) \sim (x_2, y_2)$ iff $|x_2 - x_1| = 1$ or $|y_1 - y_2| = 1$.

It can be proved that if the set of answers is discrete (which is our case) Definition 4 can be equivalently stated in terms of singleton S's. This leads to the following:

Definition 5. *A process P provides* ϵ-differential privacy *iff for all schedulers ζ, for all secret inputs $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathcal{S}$ such that $\boldsymbol{s}_1 \sim \boldsymbol{s}_2$, and for all observable $\boldsymbol{o} \in \mathcal{O}$,*

$$p_\zeta(\boldsymbol{o}|\boldsymbol{s}_1) \leq e^\epsilon \, p_\zeta(\boldsymbol{o}|\boldsymbol{s}_2)$$

We use $dp_\zeta[\![P]\!]$ to denote the smallest ϵ such that the process term P, under the scheduler ζ, provides ϵ-differential privacy. Furthermore we define

$$dp[\![P]\!] = \sup_\zeta dp_\zeta[\![P]\!]$$

Note that if there are both zero and non-zero probabilities occurring in the same column of the channel matrix, when the respective secrets are connected by $\sim$, then the process does not provide differential privacy for any ϵ.

Relation between Differential Privacy and Strong Anonymity. Strong anonymity for purely probabilistic systems was formalized by Chaum [6] as the property that the observation $\boldsymbol{o}$ does not change the probabilistic knowledge of the culprit's identity $\boldsymbol{s}$, i.e. $p(\boldsymbol{s}|\boldsymbol{o}) = p(\boldsymbol{s})$. This notion was extended in [3] to probabilistic and nondeterministic systems, essentially by requiring that the equation holds under any scheduler. Next proposition is an immediate consequence of the characterization given in [3].

Proposition 1. *An anonymity system P is strongly anonymous iff $dp[\![P]\!] = 0$.*

Relation between Differential Privacy and Probable Innocence. Probable innocence was defined in [19] as the property that, to the eyes of an observer, each user is more likely to be innocent rather than culpable (of having initiated the message). In [5] it was shown that this is equivalent to requiring $(m-1)p(o|x_i) \geq p(o|x_j)$ for all o, i and j, where m is the number of anonymous users, and $p(o|x_i)$ denotes the probability of detecting o given that the initiator is i. It is straightforward to see the following:

Proposition 2. *An anonymity system P provides probable innocence iff $dp[\![P]\!] \leq \ln(m-1)$.*

5 Modular Reasoning

In this section we investigate the compositional properties of CCS_p constructs with respect to differential privacy and state our first main result. We start by introducing the notions of *safe component* and *sequential replication*. The latter can be used to represent a sequence of sessions re-executing the same protocol. .

Definition 6. *Consider a process P, and observables $o_1, o_2, \ldots, o_k$, we say that $(\nu o_1, o_2, \ldots, o_k)P$ is* a safe component *if*

(i) P does not contain any secret label, and
(ii) all the observable labels of P are included in $o_1, o_2, \ldots, o_k$.

Definition 7. *Given a process term P assumed to terminate by performing a specific action $\overline{done}$, the* sequential replication *of P n* times *is defined as*

$$\circlearrowleft^n P = (\nu done)(P \mid !^n done.P)$$

where $!^0 P = \mathbf{0}$ and $!^{n+1} P = P \mid !^n P$.

We now show that the nondeterministic choice, the probabilistic choice, the restriction operator and a restricted form of parallel composition are *safe*, in the sense that combining components does not compromise the privacy of the system, while the sequential replication degrades the privacy in proportion to the number of replication times.

Theorem 1. *Consider a set of processes $\{P_i\}_i$, for $i = 1, 2, \ldots$, and assume that for each i, $dp(\llbracket P_i \rrbracket) = \epsilon_i$. Then:*

(1) $dp(\llbracket \biguplus_i o_i.P_i \rrbracket) \leq \max_i\{\epsilon_i\}$;

(2) $dp(\llbracket \bigoplus_i p_i P_i \rrbracket) \leq \max_i\{\epsilon_i\}$;

(3) $dp(\llbracket (\nu o) P_i \rrbracket) \leq \epsilon_i$;

(4) Assume that $(\nu o_1, o_2, \ldots, o_k)P_i$ is a safe component, that P_i and P_j can communicate with each other only via the labels of the set $\{o_h, \ldots, o_k\}$, with $1 \leq h \leq k$, and that $dp(\llbracket (\nu o_1, \ldots, o_{h-1})P_j) \rrbracket) = \epsilon_j$. Then $dp(\llbracket (\nu o_1, o_2, \ldots, o_k)\,(P_i \mid P_j) \rrbracket) \leq \epsilon_j$.

(5) $dp(\llbracket \circlearrowleft^n P_i \rrbracket) \leq n\,\epsilon_i$.

Properties (1) and (2) point out that the degree of privacy of a system, consisting of some subsystems in a nondeterministic or probabilistic choice, is determined by the subsystem with the lowest degree of privacy. Properties (3) and (4) intuitively say that, turning an observable label to be unobservable, and paralleling with a safe component, maintain the level of privacy. Property (5) means that the degree of privacy of a process degrades through multiple runs, since more information may be exposed.

Unfortunately the secret choice and the unrestricted form of parallel composition do not preserve the privacy, essentially due to the presence of nondeterminism. This is illustrated by the following counterexamples taken from [4]. (In Examples 3 - 5, we use the original definition of the adjacency relation, that is, the difference in only one label.)

Example 3 (For the secret choice). Let $Sec = \{s_1, s_2\}$ and assume that $\mathcal{S}$ does not contain the empty sequence. Consider the process $P = o_1.\mathbf{0} \uplus o_2.\mathbf{0}$. Clearly, P provides 0-differential privacy, because for every sequence $\boldsymbol{s} \in \mathcal{S}$ we have $p(o_1|\boldsymbol{s}) = p(o_2|\boldsymbol{s})$. Consider now a new process $P' = s_1.P \uplus s_2.P$, and the scheduler ζ for P' which selects o_1 if the secret is s_1, and o_2 if the secret is s_2. The resulting matrix under ζ does not preserve differential privacy, since $p(o_1|s_1\boldsymbol{s}) = p(o_2|s_2\boldsymbol{s}) = 1$ while $p(o_1|s_2\boldsymbol{s}) = p(o_2|s_1\boldsymbol{s}) = 0$.

Example 4 (For the need of condition (i) in Def. 6). Let Sec and $\mathcal{S}$ be as in Example 3. Define $P_1 = s_1.\mathbf{0} \uplus s_2.\mathbf{0}$ and $P_2 = o_1.\mathbf{0} \uplus o_2.\mathbf{0}$. Clearly, P_2 provides 0-differential privacy. Consider now the parallel term $P_1 \mid P_2$ and define a scheduler that first executes a secret label s in P_1 and then, if s is s_1, it selects o_1, while if s is s_2, it selects o_2. The rest proceeds like in Example 3.

Example 5 (For the need of condition (ii) in Def. 6). Let Sec and $\mathcal{S}$ be as in Example 3. Define $P_1 = o.\mathbf{0}$ and $P_2 = s_1.(o_1.\mathbf{0} \oplus_{.5} o_2.\mathbf{0}) \boxplus s_2.(o_1.\mathbf{0} \oplus_{.5} o_2.\mathbf{0})$. It is easy to see that P_2 provides 0-differential privacy. Consider the term $P_1 \mid P_2$ and define a scheduler that first executes a secret label s in P_2 and then, if s is s_1, it selects first P_1 and then the continuation of P_2, while if s is s_2, it selects first the continuation of P_2 and then P_1. Hence, under this scheduler, for every sequence $\boldsymbol{s} \in \mathcal{S}$, $p(oo_1|s_1\boldsymbol{s}) = p(oo_2|s_1\boldsymbol{s}) = 0.5$ and also $p(o_1o|s_2\boldsymbol{s}) = p(o_2o|s_2\boldsymbol{s}) = 0.5$ while $p(oo_1|s_2\boldsymbol{s}) = p(oo_2|s_2\boldsymbol{s}) = 0$ and $p(o_1o|s_1\boldsymbol{s}) = p(o_2o|s_1\boldsymbol{s}) = 0$. Therefore s_1 and s_2 are disclosed.

Intuitively, the existence of free observables (i.e. o of P_1 in this example) may create different interleavings, which can be used by the scheduler to mark different secrets.

6 A Case Study: The Crowds Protocol

In this section, we apply the result in Theorem 1 to prove the anonymity-preservation property of an extended Crowds protocol with member-wise trusted forwarders. The novelty of our proof is that it is simple. Moreover, it does not require the symmetry property that is usually made in the literature in order to simplify the analysis.

Crowds is an anonymity protocol which allows Internet users to send messages without revealing their identity. More specifically, a crowd is a group of n participants constituted by m *honest members* and $c\ (= n - m)$ *corrupted members* (the *attackers*). The destination of messages is named the *server*. The protocol works as follows:

- When a member, called the *initiator*, wants to send a message to the server, instead of sending it directly to the server, she randomly selects a member (possibly herself) in the crowd and she forwards the message to this member.
- Every member who receives the message, either
  - with probability $1 - p_f$, delivers the message to the server, or
  - with probability p_f, randomly selects another member (possibly herself) in the crowd as the new *forwarder* and relays the message to this new forwarder to repeat the same procedure again.

In this way, even if the message is caught by an attacker, the attacker cannot be sure whether the previous forwarder is the initiator or just a forwarder on behalf of somebody else. Members (including attackers) are assumed to have only access to messages routed through them, so that they only know the identities of their immediate predecessors and successors in the path, and of the destination server.

6.1 The Crowds with Member-Wise Trusted Forwarders

The above standard Crowds protocol implicitly requires the *symmetry conditions* (see [5]), in the sense that a new forwarder is chosen among all the members in the crowd with uniform probability. In this context, it has been proved in [19] that Crowds can satisfy probable innocence under certain conditions.

In this paper, we consider a *member-wise trusted forwarders* scenario, in which the member currently holding the message selects a forwarder only among the members which she thinks are trustable. We also consider the case of *reputed initiators*. The idea

Table 2. The CCS_p code for standard Crowds (i.e. with the symmetry conditions)

$$
\begin{aligned}
\mathit{Initiator} &= \textstyle\bigoplus^{\mathcal{U}}_{x_i \in \mathcal{H}_r} p_i.\mathit{act}_i.(\bigoplus^{\mathcal{U}}_{x_j \in \mathcal{C}} p_j.\bar{x}_j\langle i\rangle)\\
\mathit{honest}_i &= x_i.((\textstyle\bigoplus^{\mathcal{U}}_{x_j \in \mathcal{C}} p_j.\bar{x}_j\langle i\rangle.\overline{\mathit{done}}) \oplus_{p_f} \overline{\mathit{ser}}.\overline{\mathit{done}})\\
\mathit{Honest}_i &= \circlearrowleft \mathit{honest}_i\\
\mathit{Attacker}_i &= x_i(j).\overline{\mathit{detect}}\langle j\rangle\\
\mathit{Server} &= \mathit{ser}.\overline{S}\\
\mathit{crowd}_n &= \mathit{Server} \mid \mathit{Initiator} \mid \textstyle\prod_{x_i \in \mathcal{H}} \mathit{Honest}_i \mid \prod_{x_j \in \mathcal{A}} \mathit{Attacker}_j\\
\mathit{Crowd}_n &= (\nu \mathit{ser})(\nu x_1, x_2, \ldots, x_n)\mathit{crowd}_n
\end{aligned}
$$

Table 3. The new definitions for Crowds with member-wise trusted forwarders

$$
\begin{aligned}
\mathit{Initiator} &= \textstyle\bigoplus^{\mathcal{U}}_{x_i \in \mathcal{H}_r} p_i.\mathit{act}_i.(\bigoplus^{\mathcal{U}}_{x_j \in \mathcal{T}_i} p_j.\bar{x}_j\langle i\rangle)\\
\mathit{honest}_i &= x_i.((\textstyle\bigoplus^{\mathcal{U}}_{x_j \in \mathcal{T}_i} p_j.\bar{x}_j\langle i\rangle.\overline{\mathit{done}}) \oplus_{p_f} \overline{\mathit{ser}}.\overline{\mathit{done}})
\end{aligned}
$$

is that a new member has the right to initiate requests only when she has acquired a certain level of *reputation*. This is motivated by some kinds of social networks in which, when a new agent wants to join a web conversation, her behavior needs to be examined for a while until she becomes a totally legal member.

The standard Crowds protocol expressed in CCS_p is stated in the Table 2. For simplicity, we introduce a notation for value-passing in CCS_p, following standard lines.

$$
\begin{aligned}
\mathit{Input} \quad & x(i).P = \underline{+}_j\, x_j.P[j/i]\\
\mathit{Output} \quad & \bar{x}\langle i\rangle = \bar{x}_i
\end{aligned}
$$

We use $\mathcal{H}$, $\mathcal{H}_r$ and $\mathcal{A}$ to denote the set of honest members, of reputed honest members and of attackers, respectively. $\mathcal{C} = \mathcal{H} \cup \mathcal{A}$ representing the set of all participants in Crowds. We use $x_1, x_2, \ldots, x_n$ to range over $\mathcal{C}$, and $\bigoplus^{\mathcal{U}}$ to represent an uniform distribution. For simplicity we assume that once an attacker receives a message from an honest member, it will terminate after reporting the detection through the observable $\overline{\mathit{detect}}$. The reason is that by forwarding the message after the first detection, attackers can not gain more useful information. Hence at most one honest member can be detected. Precisely, the set of secret labels is $\{\, \mathit{act}_i \mid x_i \in \mathcal{H}_r \}$ and the set of observable labels is $\{\, \overline{\mathit{detect}}\langle i\rangle \mid x_i \in \mathcal{H} \,\} \cup \{\, \overline{S} \,\}$. We denote by $\mathcal{T}_i$ the subset of Crowds members which the ith honest member trusts. Then the process terms for Crowds with member-wise trusted forwarders are similar to the terms showed in Table 2, except that the process $\mathit{Initiator}$ and the process honest_i are modified as shown in Table 3.

An Anonymity-Preservation Property. Consider the scenario in which there exists a Crowds with n members (shown in Table 2). Assume that a new honest agent is allowed to join the crowd but she does not enjoy the reputation to be an initiator right away.

Table 4. Specification of the addition of the $n+1$th agent

$$crowd_{n+1} = crowd_n \mid Honest_{n+1}$$

$$Crowd_{n+1} = (\nu ser)(\nu x_1, x_2, \ldots, x_{n+1})crowd_{n+1}$$

Old members in Crowds can decide by themselves to trust the agent or not, which means that there is no restriction of how the new agent is added, thus resulting in a Crowds with member-wise trusted forwarders. Applying the compositionality theory in Section 5, we show that the privacy level of Crowds with $n+1$ members can be estimated by the value of privacy of a simplified Crowds, obtained by considering the non-reputed agent as an attacker and therefore ignoring her following trust links to successors. The fact is supported by the following theorem.

Theorem 2. $dp[\![Crowd_{n+1}]\!] \leq dp[\![Crowd_n]\!]$.

Proof. The addition of a new honest agent to a Crowds with n participants is presented in Table 4. Basically, it is a parallel composition of the process term $crowd_n$ of old crowd, and the process term $Honest_{n+1}$ of the new agent. In $crowd_n$, although there is no entity of $Honest_{n+1}$, we assume that the identity x_{n+1} is already available in the set $\mathcal{C}$ as a free label, to be selected as a forwarder.

Consider the term $Crowd_{n+1}$. Remove the term $Honest_{n+1}$ and the corresponding restriction operators. Note that the free labels through which old Crowds communicates with the new agent are $\{ser\} \cup \{x_j | x_j \in \mathcal{T}_{n+1}\} \cup \{\bar{x}_{n+1}\langle i\rangle | x_i \in \mathcal{S}_{n+1}\}$, where $\mathcal{S}_{n+1}$ is the subset of Crowds members who trust the new agent. The label $\bar{x}_{n+1}\langle i\rangle$ behaves like an attacker, because it can reveal the identity of member (ex. x_i) who is sending the request. Since this new agent is known not to be an initiator (because she is not reputed). Her presence will not induce an addition of the secret label act_{n+1}. In the sense that the process $Honest_{n+1}$ does not contain any secret labels. Clearly,

$$(\nu ser)(\nu x_1, x_2, \ldots, x_{n+1})Honest_{n+1}$$

is a safe component. By Theorem 1(4), $Crowd_{n+1}$ provides at least as much privacy as $Crowd_n$.

7 Computing the Degree of Privacy

In this section, we study the possibility of computing the degree of privacy in CCS_p in a fine-grained way, precisely, in a bottom-up fashion. We make a first step by considering finite processes in which the secret choices are only at the top-level, and give some toy examples to illustrate the basic ideas of the approach.

7.1 The Algorithm

Consider a process term $T = \biguplus_i s_i.T_i$ starting with a secret choice. For simplicity, in this paper, we assume that there is no secret choice inside any of T_i, and no occurrence of ! , (while considering ! is a real difficult problem, the restriction about the

secret choice can be lifted by using more complicated definitions of how to combine the schedulers of the components. We however leave them for future work). We also assume that every occurrence of the parallel operator is removed by using the *expansion law* [16]. We construct the set of *all possible schedulers* for a process P, denoted by $\Delta(P)$. We denote its size by $|\Delta(P)|$. Thus the residual constructs possibly occurring in T_i are the following cases:

- Case $P = \bigoplus_i p_i P_i$: Intuitively, P's scheduler selects for each P_i a scheduler from $\Delta(P_i)$. Hence, $|\Delta(P)| = O(\prod_i |\Delta(P_i)|)$.
- Case $P = \biguplus_i r_i.P_i$: P's scheduler first resolves the top nondeterministic choice and then proceeds the continuation of the process corresponding to the selected label. Thus, $|\Delta(P)| = O(\sum_i |\Delta(P_i)|)$.
- Case $P' = (\nu o)P$: A scheduler of P' is obtained from a scheduler in $\Delta(P)$ by removing the assignments of the executions containing o. Hence, $|\Delta(P')| = O(|\Delta(P)|)$.
- Case $P' = \circlearrowleft^n P$: Intuitively, a scheduler of P' selects for each run of P a scheduler from $\Delta(P)$, and use them in a sequential way. Hence, $|\Delta(P')| = O(|\Delta(P)|^n)$.

For every term T_i in T, we are able to obtain the scheduler set $\Delta(T_i)$. The corresponding matrix under each scheduler in $\Delta(T_i)$ is computed in a bottom-up way (see [25]). We now turn to the construction of the matrix of T. For every secret label s_i, T's scheduler ζ chooses a scheduler ζ_i from the set $\Delta(T_i)$. Let $p_\zeta(\boldsymbol{o}|s_i\boldsymbol{s})$ (resp. $p^i_{\zeta_i}(\boldsymbol{o}|\boldsymbol{s})$) be the probability in the matrix $M(T)_\zeta$ (resp. $M_{\zeta_i}(T_i)$). Hence $p_\zeta(\boldsymbol{o}|s_i\boldsymbol{s}) = p^i_{\zeta_i}(\boldsymbol{o}|\boldsymbol{s})$. By the definition of differential privacy we get:

$$dp_\zeta[\![T]\!] = \min\{\epsilon \mid \boldsymbol{s} \sim \boldsymbol{s}' \Rightarrow \forall \boldsymbol{o} \in \mathcal{O}.\ p_\zeta(\boldsymbol{o}|\boldsymbol{s}) \leq e^\epsilon p_\zeta(\boldsymbol{o}|\boldsymbol{s}')\}$$

and

$$dp[\![T]\!] = \max_{\zeta \in \Delta(T)} dp_\zeta[\![T]\!].$$

Complexity Analysis. The time complexity of the above algorithm is determined by the size of the set of all possible schedulers, that is, the time complexity is $O(|\Delta(T)|)$. However we can make it more efficient in the case in which differential privacy does not hold: Whenever we find a scheduler ζ_i in $\Delta(T_i)$ producing an observable $\boldsymbol{o}$ which is not included in the set of observables generated by a previous scheduler ζ_j in $\Delta(T_j)$ (with $j < i$ and $s_i \sim s_j$), then we can halt the algorithm and claim that T does not provide differential privacy for any ϵ. In fact, assigning the scheduler ζ_i (resp. ζ_j) to the secret s_i (resp. s_j) differentiates the two secrets by producing a non-null (resp. null) probability in the column of $\boldsymbol{o}$.

7.2 Some Toy Examples

Example 6. Let $Sec = \{s_1, s_2\}$, $Obs = \{o_1, o_2\}$, and consider the following processes: $P_1 = o_1.\mathbf{0} \oplus_{0.3} o_2.\mathbf{0}$, $P_2 = o_1.\mathbf{0} \oplus_{0.5} o_2.\mathbf{0}$, $P_3 = o_1.\mathbf{0} \oplus_{0.8} o_2.\mathbf{0}$, $P = P_1 \biguplus P_2 \biguplus P_3$ and $P' = s_1.P \biguplus s_2.P$.

For P_1, P_2 and P_3, we have $\Delta(P_1) = \Delta(P_2) = \Delta(P_3) = \{\varnothing\}$, $M(P_1) =$ | o_1 | o_2 |
|---|---|
| 0.3 | 0.7 |

, $M(P_2) =$ | o_1 | o_2 |
|---|---|
| 0.5 | 0.5 |

, and $M(P_3) =$ | o_1 | o_2 |
|---|---|
| 0.8 | 0.2 |

.

For the term P, we have $\Delta(P) = \{\zeta_1, \zeta_2, \zeta_3\}$ with ζ_i representing the choice of P_i. The corresponding matrices are: $M_{\zeta_1}(P) = M(P_1)$, $M_{\zeta_2}(P) = M(P_2)$ and $M_{\zeta_3}(P) = M(P_3)$.

For the term P' we can define the scheduler ζ' which selects ζ_1 and ζ_3 for the secret s_1 and s_2, respectively. The corresponding matrix is $M_{\zeta'}(P') =$ | | o_1 | o_2 |
|---|---|---|
| s_1 | 0.3 | 0.7 |
| s_2 | 0.8 | 0.2 |

, which gives $(\ln 3.5)$-differential privacy.

Example 7. Let $Sec = \{s_1, s_2\}$, $Obs = \{o, o_1, o_2\}$, and consider the processes $P_1 = o_1.\mathbf{0} \oplus_{0.3} o_2.\mathbf{0}$, $P_2 = o.\mathbf{0}$, $P = P_1 \mid P_2$, and $P' = s_1.P \boxplus s_2.P$.

First we use the expansion law to rewrite P into $(o_1.o.\mathbf{0} \oplus_{0.3} o_2.o.\mathbf{0}) \boxplus (o.o_1.\mathbf{0} \oplus_{0.3} o.o_2.\mathbf{0})$. Through steps similar to the above example, we can find a scheduler producing a matrix breaking the differential privacy, for example | | o_1o | o_2o | oo_1 | oo_2 |
|---|---|---|---|---|
| s_1 | 0.3 | 0.7 | 0 | 0 |
| s_2 | 0 | 0 | 0.3 | 0.7 |

.

8 Related Work

- *Compositionality properties of probabilistic process calculi for security protocols.* In [9] Deng et al. used the notion of relative entropy to measure privacy. In [4] Braun et al. proved that the safety measured by Bayes risk is maintained under the composition of CCS$_p$ constructs with a restricted form of parallel composition and without the secret choice. The compositionality results in our paper are closely related to those of [4], although we use a different measure of protection (differential privacy).
- *Compositionality of Differential Privacy.* As stated in the introduction, there is a vast body of work on formal methods for differential privacy. Compositional methods, as one of the most important features, have been intensively investigated in the field of statistical databases [15] and programs [2,18]. These works investigate the so-called sequential and parallel compositions of queries (programs), which, in their context, mean a sequence of queries (programs) applied to the same dataset and to disjoint parts of the dataset, respectively. Under this setting, they have proved that the sequential composition decreases the privacy, and the parallel composition maintains the privacy. Our result about the sequential replication and the parallel composition in CCS$_p$ are reminiscent of the above results. But the context is different. In particular, the parallel composition concerned in this paper is different from the above one, in that the parallel operator here represents interactions of concurrent agents. Our restrictions on the parallel composition are incomparable with those of [15,2,18] (disjointness of databases).
- *Other extensions on the Crowds protocol.* Hamadou et al. [14] have taken into account attackers' additional information correlated to anonymous agents before attacking the

protocol. In [21] Sassone et al. extended Crowds by allowing members to turn corrupt, and considering a trust estimation over participants. This kind of trust estimation is the same from all members' points of view, and therefore can still be thought of as the symmetry condition. We consider a more realistic scenario in which users can choose to communicate only with the users they think are reliable, which is the most common case in web transactions of the real world.

9 Conclusion and Future Work

In this paper, we have defined the degree of differential privacy for concurrent systems expressed in a probabilistic process calculus, and investigated how the privacy is affected under composition of the CCS_p constructs. We have applied our approach to give a simple proof for the anonymity-preservation of an extended Crowds protocol with member-wise trusted forwarders. Finally, we have presented an algorithm for computing the degree of differential privacy for a finite process.

Fine-grained methods for computing the channel matrix have been studied in [1]. In future work, we plan to optimize our current algorithm, extend it to more general processes, more precisely, develop an approach that can deduce the global property of differential privacy from local information, w.r.t. the adjacency relation. Another interesting problem is the applicability of our approach to the problem of preserving privacy in geolocation-related applications. More specifically, we intend to use (a possibly extended version of) our probabilistic process calculus to express systems of concurrent agents moving in space and time, and interacting with each other in ways that depend on the adjacency relation. We believe that our compositional method will provide a way to synthesize differentially private mechanisms in a (semi-)automatic way.

References

1. Andrés, M.E., Palamidessi, C., van Rossum, P., Smith, G.: Computing the leakage of information-hiding systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 373–389. Springer, Heidelberg (2010)
2. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. In: Proc. of POPL. ACM (2012)
3. Bhargava, M., Palamidessi, C.: Probabilistic anonymity. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 171–185. Springer, Heidelberg (2005)
4. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Compositional methods for information-hiding. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 443–457. Springer, Heidelberg (2008)
5. Chatzikokolakis, K., Palamidessi, C.: Probable innocence revisited. Theor. Comp. Sci. 367(1-2), 123–138 (2006)
6. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
7. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A distributed anonymous information storage and retrieval system. In: Federrath, H. (ed.) Anonymity 2000. LNCS, vol. 2009, pp. 44–66. Springer, Heidelberg (2001)
8. Cover, T.M., Thomas, J.A.: Elements of Information Theory. J. Wiley & Sons, Inc. (1991)

9. Deng, Y., Pang, J., Wu, P.: Measuring anonymity with relative entropy. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2006. LNCS, vol. 4691, pp. 65–79. Springer, Heidelberg (2007)
10. Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006, Part II. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)
11. Dwork, C.: A firm foundation for private data analysis. Communications of the ACM 54(1), 86–96 (2011)
12. Dwork, C., Lei, J.: Differential privacy and robust statistics. In: Proc. of the 41st Annual ACM Symposium on Theory of Computing (STOC), pp. 371–380. ACM (2009)
13. Halpern, J.Y., O'Neill, K.R.: Anonymity and information hiding in multiagent systems. J. of Comp. Security 13(3), 483–512 (2005)
14. Hamadou, S., Palamidessi, C., Sassone, V., ElSalamouny, E.: Probable innocence in the presence of independent knowledge. In: Degano, P., Guttman, J.D. (eds.) FAST 2009. LNCS, vol. 5983, pp. 141–156. Springer, Heidelberg (2010)
15. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In: Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pp. 19–30. ACM (2009)
16. Milner, R.: Communication and Concurrency. Series in Comp. Sci. Prentice Hall (1989)
17. Narayanan, A., Shmatikov, V.: De-anonymizing social networks. In: Proc. of S&P, pp. 173–187. IEEE (2009)
18. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: a calculus for differential privacy. In: Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 157–168. ACM (2010)
19. Reiter, M.K., Rubin, A.D.: Crowds: anonymity for Web transactions. ACM Trans. on Information and System Security 1(1), 66–92 (1998)
20. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for MapReduce. In: Proc. of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 297–312. USENIX Association (2010)
21. Sassone, V., ElSalamouny, E., Hamadou, S.: Trust in crowds: Probabilistic behaviour in anonymity protocols. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 88–102. Springer, Heidelberg (2010)
22. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, Tech. Rep. MIT/LCS/TR-676 (1995)
23. Syverson, P.F., Goldschlag, D.M., Reed, M.G.: Anonymous connections and onion routing. In: Proc. of S&P, pp. 44–54 (1997)
24. Tschantz, M.C., Kaynar, D., Datta, A.: Formal verification of differential privacy for interactive systems (extended abstract). Electron. Notes Theor. Comput. Sci. 276, 61–79 (2011)
25. Xu, L.: Modular Reasoning about Differential Privacy in a Probabilistic Process Calculus (full version). Research report, INRIA (2012), `http://hal.inria.fr/hal-00691284`

Author Index