Event structures: maps, monads and spans: invited talk

Citation for published version (APA):

Document status and date:
Published: 01/01/2005

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Preface

The EXPRESS workshops aim at bringing together researchers interested in the relations between various formal systems in computer science, in particular in the field of concurrency. More specifically, they focus on the comparison between programming concepts (such as concurrent, functional, imperative, logic and object-oriented programming) and between mathematical models of computation (such as process algebras, Petri nets, event structures, modal logics, rewrite systems etc.) on the basis of their relative expressive power.

The EXPRESS workshops were originally held as meetings of the HCM project EXPRESS, which was active with the same focus from January 1994 until December 1997. The first three workshops were held in Amsterdam (1994, chaired by Frits Vaandrager), Tarquinia (1995, chaired by Rocco De Nicola) and Dagstuhl (1996, co-chaired by Ursula Goltz and Rocco De Nicola). The workshop in 1997, which took place in Santa Margherita Ligure and was co-chaired by Catuscia Palamidessi and Joachim Parrow, was organized as a conference with a call for papers and a significant attendance from outside the project. As of 1998 (so, also this year), the workshops are held as satellite workshops of the CONCUR conferences. In 1998, this was in Nice, co-chaired by Ilaria Castellani and Catuscia Palamidessi, in 1999 in Eindhoven, co-chaired by Ilaria Castellani and Björn Victor, in 2000 at Pennsylvania State University, co-chaired by Luca Aceto and Björn Victor, in 2001 at BRICS, Aalborg University, co-chaired by Luca Aceto and Prakash Panangaden, in 2002 in Brno, co-chaired by Uwe Nestmann and Prakash Panangaden, in 2003 in Marseille, co-chaired by Flavio Corradini and Uwe Nestmann, and finally in 2004 in London, co-chaired by Jos Baeten and Flavio Corradini.

This year, in response to the call for papers, we received 15 submissions. The program committee selected 9 of these for presentation at the workshop. One of these was a short paper, describing work in progress, that does not appear in the proceedings. In addition, the workshop contains two invited presentations, by Tom Henzinger and Glynn Winskel, of which abstracts appear in the proceedings. Thus, in total, this proceedings contains 2 abstracts and 8 papers. We would like to thank the authors of the submitted short and full papers, the invited speakers, the members of the program committee and their subreferees for their contribution to both the meeting and this volume. We thank the Technische Universiteit Eindhoven for printing the preliminary proceedings. We thank the CONCUR organizing committee for hosting EXPRESS’05, in particular the workshop organizer Leandro Dias Da Silva and CONCUR general chair Luca De Alfaro.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Michael Mislove helped with style files, Erik Luit with Cyberchair and Sonja Joosten took care of the webpage \url{http://www.win.tue.nl/Express05/}. The final proceedings will become available electronically at Elsevier Science Publisher’s website \url{http://www.elsevier.nl/locate/entcs}.

The editors

Jos Baeten (Technische Universiteit Eindhoven)
Iain Phillips (Imperial College, London)

EXPRESS 2005 Program Committee

Robert Amadio
Jos Baeten
Julian Bradfield
Michele Bugliesi
Mariangiola Dezani-Ciancaglini
Wan Fokkink
Thomas Hildebrandt

Kohei Honda
Richard Mayr
Catuscia Palamidessi
Iain Phillips
Julian Rathke
Eugene Stark

EXPRESS 2005 Subreferees

Paolo Baldan
Michele Boreale
Horatiu Cirstea
Lucia Cloth
Cas Cremers
Rocco De Nicola
Sibylle Fröschle
Fabio Gadducci
Francesco Gianfelicici
Jens Chr. Godskesen
Daniele Gorla
Michael Huth
Sławomir Lasota

Matteo Maffei
Massimo Merro
Marius Minea
Faron Moller
Marco Pistore
Paola Quaglia
Alwen Tiu
Erik de Vink
Marc Voorhoeve
Benjamin Wack
Glynn Winskel
Axelle Ziegler
Table of Contents

Event Structures --- Maps, Monads and Spans......................................................... 4
  Glynn Winskel

Causality Versus True-Concurrency................................................................. 6
  Sibylle Fröschle, Slawomir Lasota

Interaction Nets vs. the $\rho$-Calculus: Introducing Bigraphical Nets............... 18
  Maribel Fernández, Ian Mackie, François-Régis Sinot

Feasible Reactivity for Synchronous Cooperative Threads................................. 32
  Roberto Amadio, Frédéric Dabrowski

Static Equivalence is Harder than Knowledge............................................... 44
  Johannes Borgström

Processes as Games........................................................................................... 56
  Tom Henzinger

A Short Visit to the STS Hierarchy................................................................. 57
  Nathalie Bertrand, Phillippe Schnoebelen

A Petri Net Semantics of a Simple Process Algebra for Mobility..................... 69
  Raymond Devillers, Hanna Klaudel, Maciej Koutny

Separation of Synchronous and Asynchronous Communication via Testing.......... 81
  Diletta Cacciagrano, Flavio Corradini, Catuscia Palamidessi

Open Bisimulation, Revisited........................................................................ 93
  Sébastien Brias, Uwe Nestmann
Event Structures—maps, monads and spans
Invited talk

Glynn Winskel

University of Cambridge Computer Laboratory
University of Cambridge
Cambridge, England

Abstract

Event structures are a model of computational processes. They represent a process as a set of event occurrences with relations to express how events causally depend on others, or exclude other events from occurring. In one of their simpler forms they consist of a set of events on which there is a consistency relation expressing when events can occur together in a history and a partial order of causal dependency—writing $e' \leq e$ if the occurrence of $e$ depends on the previous occurrence of $e'$.

In detail, an event structure comprises $(E, \text{Con}, \leq)$, consisting of a set $E$, of events which are partially ordered by $\leq$, the causal dependency relation, and a predicate $\text{Con}$ consisting of finite subsets of $E$, the consistency relation, which satisfy

\[
\{e' \mid e' \leq e\} \text{ is finite for all } e \in E,
\{e\} \in \text{Con} \text{ for all } e \in E,
Y \subseteq X \in \text{Con} \Rightarrow Y \in \text{Con}, \text{ and}
X \in \text{Con} \& e \leq e' \in X \Rightarrow X \cup \{e\} \in \text{Con}.
\]

Our understanding of the consistency predicate and the enabling relation are expressed in the notion of configuration (or state or history) we adopt for event structures. The events are to be thought of as event occurrences; in any history an event is to appear at most once. A configuration is a set of events which have occurred by some stage in a process. According to our understanding of the consistency predicate and causal dependency relations a configuration should be consistent and such that if an event appears in a configuration then so do all the events on which it causally depends.

A configuration of an event structure $E$ is a subset $x \subseteq E$ which is

Consistent: $\forall X \subseteq x. \ X \in \text{Con}$, and

The configurations of an event structure are ordered by inclusion, where $x \subseteq x'$, i.e. $x$ is a sub-configuration of $x'$, means that $x$ is a sub-history of $x'$.

This is a preliminary version. The final version will be published in Électronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

4
Let $E$ and $E'$ be event structures. A map of event structures $f : E \to E'$ is a partial function on events $f : E \to E'$ such that for all configurations $x$ of $E$ its direct image $fx$ is a configuration of $E'$ for which

$$\text{if } e_1, e_2 \in x$$

The map expresses how the occurrence of an event $e$ in $E$ induces the coincident occurrence of the event $f(e)$ in $E'$ whenever it is defined. The partial function $f$ respects the instantaneous nature of events: two distinct event occurrences which are consistent with each other cannot both be coincident with the occurrence of a common event in the image. We say the map is total iff the function $f$ is total, and rigid iff it is total and preserves the order of causal dependency.

Recently, with Lucy Saunders-Evans, we have come to realize the primary nature of rigid maps in the sense that the other kinds of maps of event structures may be derived from them by a Kleisli construction. For example, total maps of event structures from $E$ to $E'$ can be obtained as rigid maps from $E$ to $T(E')$, for a suitable monad $T$ on the category of event structures with rigid maps. The monad is associated with changing the type of the events. A similar Kleisli construction for suitable monads yields many other kinds of maps too, though only through a slight generalisation of event structures. The generalisation is to extend event structures to allow “persistent” events, events which can have a significant duration. An event structure with persistence $(E, P)$ is an event structure $E$ together with a distinguished subset of persistent events $P$. Configurations are defined just as before. Maps $f : (E, P) \to (E', P')$ of event structures with persistence are partial functions on events $f : E \to E'$ such that $fP \subseteq P'$ and for all configurations $x$ of $E$ its direct image $fx$ is a configuration of $E'$ for which now

$$\text{if } e_1, e_2 \in x \text{ and } f(e_1) = f(e_2) \in (E' \setminus P'), \text{ then } e_1 = e_2.$$ 

A map on event structures with persistence is rigid iff it comprises a total function which preserves the order of causal dependency. This amounts to the same definition as before when no events are persistent.

In the talk I will motivate the generalisation to event structures with persistence, and discuss applications of maps, monads and spans of event structures, specifically to the semantics of nondeterministic dataflow and higher-order processes.

---

1 Email: Glynn.Winskel@cl.cam.ac.uk
Causality Versus True-Concurrency

Sibylle Fröschle\textsuperscript{1,2} and Sławomir Lasota\textsuperscript{1,3,4}

\textit{Institute of Informatics, Warsaw University}
\textit{02–097 Warszawa, Banacha 2, Poland}

Abstract

Category theory has been successfully employed to structure the confusing setup of models and equivalences for concurrency: Winskel and Nielsen have related the standard models via adjunctions and (co)reflections while Joyal et al. have defined an abstract notion of equivalence, known as open map bisimilarity. One model has not been integrated into this framework: the causal trees of Darondeau and Degano. Here we fill this gap. In particular, we show that there is an adjunction from causal trees to event structures, which we bring to light via a mediating model, that of event trees. Further, we achieve an open map characterization of history preserving bisimilarity: the latter is captured by the natural instantiation of the abstract bisimilarity for causal trees.

\textit{Key words:} event structures, causal trees, bisimulation

In [6] Winskel and Nielsen employ category theory to relate and unify the many models for concurrency. The basic idea is to represent models as categories: each model is equipped with a notion of morphism that shows how one model instance can be simulated by another. Category theoretical notions such as adjunctions and (co)reflections can then be applied to understand the relationships between the models. We give an example. \textit{Synchronization trees} are intuitively those transition systems with no cyclic behaviour. Formally, the two models are related by a coreflection: the inclusion functor embedding synchronization trees into transition systems is accompanied by a right adjoint that unfolds transition systems to synchronization trees.

The categorical approach has also been applied to bring uniformity to the confusing setup of behavioural equivalences. Joyal et al. define an abstract notion of bisimilarity in the following way [4]: given a category of models $M$ and a choice of path category $P$ within $M$, two model instances of $M$ are $P$-\textit{bisimilar} iff there is a span of $P$-open maps between them. $P$-open maps are morphisms that satisfy a special path-lifting property with respect to $P$. As one would expect, on transition

\textsuperscript{1} Supported by the EU Research Training Network GAMES.
\textsuperscript{2} Email: sib@mimuw.edu.pl
\textsuperscript{3} Partially supported by the Polish KBN grant No. 4 T11C 042 25.
\textsuperscript{4} Email: sl@mimuw.edu.pl

\textit{This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science}
\textit{URL: www.elsevier.nl/locate/entcs}
systems and synchronization trees the abstract bisimilarity gives rise to classical
bisimilarity [4]. Various well-known equivalences are motivated as instantiations
of $\mathcal{P}$-bisimilarity in a natural way [2].

Winskel and Nielsen’s framework has helped to clarify the connections between
truly-concurrent models such as event structures, asynchronous transition systems,
and Petri nets. These are all independence models: they have additional structure
which shows when two transitions are independent of each other. Common to
these models is that they come with a notion of event: given two runs $r_1$, $r_2$ and
two transitions $t_1$ on $r_1$, $t_2$ on $r_2$ it is possible to tell whether $t_1$ and $t_2$ represent
two occurrences of the same event and can thus be considered equivalent modulo
independent behaviour. The notion of event is primary in event structures; they can
be considered to be the independence model for unfolded behaviour.

On independence models $\mathcal{P}$-bisimilarity was shown to yield hereditary history
preserving bisimilarity ($hhp\text{-}b$) [4]. This left open whether it is at all possible
to capture history preserving bisimilarity ($hp\text{-}b$) via open maps, which was then
thought to be the truly-concurrent bisimilarity. In particular, it was found that the
characterization of $hhp\text{-}b$ is very robust with respect to the choice of path category.

Along a different strand of research, a new model emerged in the late 80’s: the
causal trees of Darondeau and Degano [3]. They are a variant of synchronization
trees with enriched action labels that supply information about which transitions
are causally dependent on each other. Thereby, they reflect one aspect of true-
concurrency, causality, while being different from the truly-concurrent models of
[6] in that they do not come with a notion of event. However, the precise relation-
ship between causal trees and the standard models has never been clarified.

Roughly one could say the strand of research along which causal trees have
emerged is that of syntax-enriched process calculi. A unifying framework for a
wide range of such calculi, including the $\pi$-calculus, has been provided by the
history-dependent automata of Pistore [5]. In this context a first, albeit indirect,
open map account of $hp\text{-}b$ has been achieved: in [5] history-dependent bisimilarity,
which induces $hp\text{-}b$ with respect to Petri nets, is captured via open maps. It has
remained open, though, whether $hp\text{-}b$ has a direct open map characterization: one
that is as natural as that of $hhp\text{-}b$ and illustrates the difference between the two
equivalences, one within a model related to event structures.

Our contribution is twofold. Firstly, we integrate the model of causal trees
into Winskel and Nielsen’s framework. We equip causal trees with a notion of
morphism, and thus define the category of causal trees, $\mathcal{C}$. We investigate how
$\mathcal{C}$ relates to the other model categories. In particular, we show that there is an
adjunction from causal trees to event structures. This is brought to light via a larger
model, called event trees: the adjunction arises as the composition of a coreflection
from causal trees to event trees and a reflection from event trees to event structures.

Secondly, we identify the natural instantiation of $\mathcal{P}$-bisimilarity for causal trees:
$CBran_\ell_1$-bisimilarity. It turns out that $CBran_\ell_1$-bisimilarity fills in a prominent
gap: it characterizes $hp\text{-}b$ in a direct fashion. Finally, we capture the difference
between $hp\text{-}b$ and $hhp\text{-}b$ by characterizing them within the category of event trees.
1 Relating Causal Trees to Other Models for Concurrency

We first define the category of transition systems, $\mathbf{T}$, and that of synchronization trees, $\mathbf{S}$.

A transition system is a tuple $(S, s^m, L, \text{Tran})$ where $S$ is a set of states, $s^m \in S$ is the initial state, $L$ is a set of labels, and $\text{Tran} \subseteq S \times L \times S$ is the transition relation. We write $s \xrightarrow{a} s'$ to denote that $(s, a, s') \in \text{Tran}$. We extend this notation to possibly empty strings of labels $v = a_1 \ldots a_n$, writing $s \xrightarrow{v} s'$ to indicate that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ for some $s_0, \ldots, s_n$ with $s = s_0$ and $s_n = s'$. Given $t = (s, a, s') \in \text{Tran}$, we use $\text{src}(t)$ for $s$, $\text{tgt}(t)$ for $s'$, and $l(t)$ for $a$.

A run of a transition system $T$ is a sequence of transitions $t_1 t_2 \ldots t_n$, $n \geq 0$, such that if $n > 0$ then $\text{src}(t_1) = s^m$ and for all $i \in [1, n - 1] \text{tgt}(t_i) = \text{src}(t_{i+1})$. We denote the set of runs of $T$ by $\text{Runs}(T)$.

Let $T_0 = (S_0, s_0^m, L_0, \text{Tran}_0)$ and $T_1 = (S_1, s_1^m, L_1, \text{Tran}_1)$ be transition systems. A morphism $f : T_0 \rightarrow T_1$ is a pair $f = (\sigma, \lambda)$ where $\sigma : S_0 \rightarrow S_1$ is a function and $\lambda : L_0 \rightarrow L_1$ is a partial function such that

(i) $\sigma(s_0^m) = s_1^m$,

(ii) $(s, a, s') \in \text{Tran}_0$ & $\lambda(a)$ defined $\implies (\sigma(s), \lambda(a), \sigma(s')) \in \text{Tran}_1$,

(iii) $(s, a, s') \in \text{Tran}_0$ & $\lambda(a)$ undefined $\implies \sigma(s) = \sigma(s')$.

Transition systems together with their morphisms form a category $\mathbf{T}$. Composition of morphisms is pairwise and identity for an object $T$ is $(1_S, 1_L)$ where $1_S$ is identity on the set of states $S$ of $T$ and $1_L$ is identity on the set of labels $L$ of $T$.

A synchronization tree is a transition system $(S, s^m, L, \text{Tran})$ such that

(i) every state is reachable: $\forall s \in S. \exists v. s^m \xrightarrow{v} s$,

(ii) the transition system is acyclic: $s \xrightarrow{v} s$ for some $v \in L^*$ $\implies v = \varepsilon$,

(iii) there is no backwards branching: $s' \xrightarrow{a} s \& s'' \xrightarrow{b} s \implies a = b \& s' = s''$.

Write $\mathbf{S}$ for the full subcategory of synchronization trees in $\mathbf{T}$.

We define causal trees explicitly as a generalization of synchronization trees. In particular, this means: we add causality information not via enriched labelling and backwards pointers as in [3] but by a causal dependency relation on transitions.

**Definition 1.1** A causal tree is a tuple $(S, s^m, L, \text{Tran}, <)$ where $(S, s^m, L, \text{Tran})$ is a synchronization tree and $< \subseteq \text{Tran} \times \text{Tran}$, the causal dependency relation, is a strict order, which satisfy:

(i) for all $t, t' \in \text{Tran}$, $t < t' \implies \text{tgt}(t) \xrightarrow{v} \text{src}(t')$ for some $v \in L^*$.

Axiom (i) expresses a natural property of causality: if $t$ is a cause of $t'$ then $t$ must have happened before $t'$. Causal trees inherit their notion of run from that of transition systems. We say two transitions $t, t' \in \text{Tran}$ are consistent, denoted by $t \text{ Con } t'$, iff they appear on the same branch: $t \text{ Con } t' \iff t = t' \lor \exists v \in L^*. \text{tgt}(t) \xrightarrow{v} \text{src}(t') \lor \text{tgt}(t') \xrightarrow{v} \text{src}(t)$.

The morphisms of the truly-concurrent models of [6] preserve concurrency. Let $t, t'$ be consistent transitions of a causal tree $C$; $t$ and $t'$ are concurrent iff they are
not identical and they are not related by \(<\). Note that in contrast to event-based models, here concurrency is only meaningful when interpreted with respect to a branch. Thus, we define causal tree morphisms as follows.

**Definition 1.2** Let \(C_0 = (S_0, s_0^0, L_0, \text{Tran}_0, \ll_0)\), \(C_1 = (S_1, s_1^0, L_1, \text{Tran}_1, \ll_1)\) be causal trees. A morphism \(f : C_0 \to C_1\) is a morphism of transition systems \((\sigma, \lambda) : (S_0, s_0^0, L_0, \text{Tran}_0) \to (S_1, s_1^0, L_1, \text{Tran}_1)\) such that \(\sigma\) preserves concurrency:

(i) for all \(t = (s, a, s')\), \(t' = (u, b, u') \in \text{Tran}_0\) such that \(t \ll_0 t'\), and \(\lambda(a)\), \(\lambda(b)\) are both defined, \((\sigma(s), \lambda(a), \sigma(s')) \ll_1 (\sigma(u), \lambda(b), \sigma(u')) \Rightarrow t \ll_0 t'\).

Causal trees and their morphisms give rise to the category of causal trees, \(\mathbf{C}\).

There is an obvious coreflection from \(S\) to \(C\): a synchronization tree can be regarded as a causal tree, one in which the causal dependency relation is given by the order of the transitions in the tree; the corresponding functor is accompanied by a right adjoint which forgets about the causality information. It is more difficult to understand the precise relationship between causal trees and event structures. We first give the definition of the category of event structures, \(E\).

A (labelled) event structure is a structure \((E, \ll, \text{Con}, L, l)\) consisting of a set \(E\) of events, which are strictly ordered\(^5\) by \(\ll\), the causal dependency relation, a consistency relation \(\text{Con}\) consisting of finite subsets of events, a set \(L\) of labels and a labelling function \(l : E \to L\), which satisfy

(i) \(e \downarrow = \{ e' \mid e' \ll e \}\) is finite,
(ii) \(\{ e \} \in \text{Con}\),
(iii) \(Y \subseteq X \in \text{Con} \Rightarrow Y \in \text{Con}\),
(iv) \(X \in \text{Con} \& e \ll e' \in X \Rightarrow X \cup \{ e \} \in \text{Con}\),

for all events \(e, e'\) and their subsets \(X, Y\). Axiom (i) ensures an event occurrence depends only on finitely many previous event occurrences. The consistency relation is thought to specify which finite subsets of events can occur together in a run. Axioms (ii)–(iv) express natural properties of this interpretation.

To define a run of an event structure \((E, \ll, \text{Con}, L, l)\), we need the notion of configuration, defined as any finite\(^6\) set \(X \subseteq E\) which is

(i) downwards-closed: \(e' \ll e \in X \Rightarrow e' \in X\), and
(ii) consistent: \(X \in \text{Con}\).

In particular, \(e \downarrow\) is always a configuration. For two configurations \(X, X'\) we write \(X \xrightarrow{e} X'\) when \(e \notin X\) and \(X' = X \cup \{ e \}\). A run is a possibly empty sequence \(e_1 \ldots e_n\) of events such that there is a sequence of transitions \(\emptyset \xrightarrow{e_1} X_1 \ldots \xrightarrow{e_n} X_n\) starting from the empty configuration, for some configurations \(X_1 \ldots X_n\). For runs, \(r \xrightarrow{e} r'\) means \(r' = re\). The set of all runs of an event structure \(E\) is denoted by \(\text{Runs}(E)\).

Let \(E_0 = (E_0, \ll_0, \text{Con}_0, L_0, l_0)\) and \(E_1 = (E_1, \ll_1, \text{Con}_1, L_1, l_1)\) be labelled

---

\(^5\) Defining causal dependency in terms of a strict rather than a partial order is more convenient here.

\(^6\) We deliberately restrict ourselves to finite configurations only.
event structures. A morphism $E_0 \to E_1$ is a pair $(\eta, \lambda)$ where $\eta : E_0 \to E_1$ and
\[ \lambda : L_0 \to L_1 \]
are partial functions such that
1. $\eta(e) \text{ defined } \Rightarrow \eta(e) \downarrow \subseteq \eta(e \downarrow)$,
2. $X \in \text{Con}_0 \Rightarrow \eta(X) \in \text{Con}_1$,
3. $\forall e, e' \in E_0, \{e, e'\} \in \text{Con}_0 \& \eta(e), \eta(e') \text{ both defined } \& \eta(e) = \eta(e') \Rightarrow e = e'$,
4. $\lambda \circ l_0 = l_1 \circ \eta$.

Event structures and their morphisms form the category of event structures, $E$.

The runs of an event structure give rise to a tree. Thus, any event structure can be transformed into a causal tree by abstracting away the notion of event; this operation has been defined in, e.g., [3]. On the other hand, there is no uniform way of reconstructing the notion of event so as to obtain a coreflection between $C$ and $E$. Indeed, there is one aspect in which event structures are less expressive than causal trees: their notion of run is induced abstractly by the consistency and causal dependency relation; in particular, this means the set of runs of any event structure is \textit{trace-closed}, that is closed under the shuffling of concurrent transitions. In the following, we expose an adjunction from $C$ to $E$ via a larger model, which we call event trees, that embeds $C$ as well as $E$. Event trees are like event structures in that causality and concurrency are event-based, global notions. They are like causal trees in that their possible runs are specified explicitly by a tree.

**Definition 1.3** A \textit{(labelled) event tree} is a tuple $(S, s^0, E, \text{Tran}, <, L, l)$ where $(S, s^0, E, \text{Tran})$ is a synchronization tree, $< \subseteq E \times E$ is a strict order on the set $E$ of events, $L$ is a set of labels, and $l : E \to L$ is a \textit{labelling function} such that
1. $e \in E \Rightarrow \exists s, s' \in S, s \xrightarrow{e} s'$,
2. $s \xrightarrow{e} s' \& s \xrightarrow{e''} s'' \Rightarrow s' = s''$,
3. $s \xrightarrow{e} s' \& u \xrightarrow{e} u' \Rightarrow \exists v \in E^*, s' \xrightarrow{v} u$,  
4. $e < e' \& s \xrightarrow{e} s' \Rightarrow \exists u \xrightarrow{e} u', v \in E^*, u' \xrightarrow{v} s$.

Axiom (i) says every event appears as a transition, and axiom (ii) that the occurrence of an event at a state leads to a unique state. (This is as for asynchronous transition systems.) Axiom (iii) expresses a natural property of acyclic models: every event appears at most once on a branch. Axiom (iv) ensures that if $e$ is a cause of $e'$ then $e$ must have happened before $e'$.

We say two events $e, e'$ are \textit{consistent} iff they appear on the same branch: $e \text{ Con } e' \iff e = e' \vee \exists s, s_1, s_2, s_3 \in S, v \in E^*, s \xrightarrow{e} s_1 \xrightarrow{v} s_2 \xrightarrow{e'} s_3 \vee s \xrightarrow{e'} s_1 \xrightarrow{v} s_2 \xrightarrow{e} s_3$. Event trees inherit a notion of \textit{run} from synchronization trees, where a run is a sequence of consecutive transitions. By axiom (ii) the sequence of events appearing along a run determines this run uniquely. Hence, we consider a run of an event tree to be a sequence of events rather than one of transitions.

A partial function $\eta : E_0 \to E_1$ induces a total function $\bar{\eta} : E^*_0 \to E^*_1$ defined inductively by: $\bar{\eta}(\varepsilon) = \varepsilon$, and $\bar{\eta}(re) = \bar{\eta}(r)\eta(e)$ if $\eta(e)$ defined, and $\bar{\eta}(r)$ otherwise.

**Definition 1.4** Assume two event trees $T_0 = (S_0, s_0^0, E_0, \text{Tran}_0, <_0, L_0, l_0), T_1 = \ldots$
(S_1, s_1^m, E_1, Tran_1, <_1, L_1, l_1). A morphism from T_0 to T_1 is a pair \( (\eta, \lambda) \) where \( \eta : E_0 \to E_1 \) and \( \lambda : L_0 \to L_1 \) are partial functions such that

(i) \( \eta(e) \) defined \( \Rightarrow \eta(e) \downarrow \subseteq \eta(e \downarrow) \),

(ii) \( r \in \text{Runs}(T_0) \Rightarrow \bar{\eta}(r) \in \text{Runs}(T_1) \),

(iii) \( \lambda \circ l_0 = l_1 \circ \eta \).

Clause (ii) implies that we also have: \( \forall e, e' \in E_0. e \text{ Con}_0 e' \& \eta(e), \eta(e') \) both defined \( \Rightarrow \eta(e) = \eta(e') \Rightarrow e = e' \). This is analogous to clause (iii) of event structure morphisms.

If \( (\eta, \lambda) : T_0 \to T_1 \) is a morphism of event trees then \( \bar{\eta} \) maps \( \text{Runs}(T_0) \) to \( \text{Runs}(T_1) \). Since each state of an event tree is reachable by a unique run, \( \bar{\eta} \) induces a total function, say \( \sigma_\eta \), from \( S_0 \) to \( S_1 \). It is routine to check:

**Proposition 1.5** If \( (\eta, \lambda) : T_0 \to T_1 \) is a morphism of event trees then \( (\sigma_\eta, \eta) \) is a morphism of transition systems \( (S_0, s_0^m, E_0, \text{Tran}_0) \to (S_1, s_1^m, E_1, \text{Tran}_1) \) such that \( \eta \) preserves concurrency: \( \forall e, e' \in E_0. e \text{ Con}_0 e' \& \eta(e), \eta(e') \) both defined \( \Rightarrow \eta(e) <_1 \eta(e') \Rightarrow e <_0 e' \).

Event trees and their morphisms give rise to the category of event trees, \( \text{ET} \).

Any event tree gives rise to a causal tree by forgetting about events. Considering axiom (i) of causal trees, we carry over the causal dependency relation from events to consistent transitions only. Extending this operation to a functor \( \text{et}2c : \text{ET} \to C \) we make use of Prop. 1.5 in our translation of morphisms.

**Definition 1.6** Let \( T = (S_T, s_T^m, E_T, \text{Tran}_T, <_T, L_T, l_T) \) be an event tree. Define \( \text{et}2c(T) = (S_T, s_T^m, L_T, \text{Tran}, <) \) where

- \( \text{Tran} = \{(s, l_T(e), s') \mid s \xrightarrow{e} s'\} \), and
- \( < = \{(s, l_T(e), s'), (u, l_T(e'), u') \mid s \xrightarrow{e} s', u \xrightarrow{e'} u', e <_T e' \& \exists v \in E_T^*, s' \xrightarrow{v} u\} \).

Let \( f = (\eta, \lambda) \) be a morphism of event trees. Define \( \text{et}2c(f) = (\sigma_\eta, \lambda) \).

On the other hand, every causal tree \( C \) determines an event tree: that induced by \( C \) when we assume each transition of \( C \) represents a separate event. We take as events the transitions of \( C \), and label each arc of \( C \) by the corresponding transition. This operation extends to a functor \( \text{c}2\text{et} : C \to \text{ET} \).

**Definition 1.7** Let \( C = (S_C, s_C^m, L_C, \text{Tran}_C, <_C) \) be a causal tree. Let \( \text{c}2\text{et}(C) = (S_C, s_C^m, \text{Tran}_C, <, L_C, l) \) where

- \( \text{Tran} = \{(s, (s, a, s'), s') \mid s \xrightarrow{a} C s'\} \), and
- \( l \) is given by \( l(s, a, s') = a \).

For \( f = (\sigma, \lambda) : C_0 \to C_1 \), define \( \text{c}2\text{et}(f) = (\eta, \lambda) \) where \( \eta : \text{Tran}_0 \to \text{Tran}_1 \) is given by: \( \eta(s, a, s') = \begin{cases} (\sigma(s), \lambda(a), \sigma(s')) & \text{if } \lambda(a) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases} \)

**Theorem 1.8** The functor \( \text{c}2\text{et} \) is left adjoint to \( \text{et}2c \). The adjunction is a coreflection, i.e., the unit is a (natural) isomorphism.
Proof. [Sketch] Let $C$ be a causal tree. Then $et2c(c2et(C)) = C$, and the unit of the adjunction at $C$, $\eta_C$, is the pair of identities $(1_S, 1_L)$.

The pair $(c2et(C), \eta_C)$ is free over $C$ wrt $et2c$, i.e. for any arrow $(\sigma, \lambda) : C \rightarrow et2c(T)$ in $C$, with $T$ an event tree, there is a unique arrow $f : c2et(C) \rightarrow T$ in $ET$ such that $et2c(f) \circ (1_S, 1_L) = (\sigma, \lambda)$: the label component of $f$ is necessarily $\lambda$, and the event component of $f$ is determined uniquely since events of $c2et(C)$ are transitions of $C$.

As a consequence, $C$ embeds fully and faithfully into $ET$ and is equivalent to the full subcategory of $ET$ consisting of those event trees $T$ that are isomorphic to $c2et(et2c(T))$. These event trees $T$ are exactly those in which each event occurs only once.

The runs of an event structure can be arranged into a tree. Hence, any event structure forms an event tree whose states are the runs of the event structure. This gives rise to a functor $e2et : ET \rightarrow E$.

Definition 1.9 Let $E = (E_E, <_E, Con_E, L_E, l_E)$ be an event structure. Define $e2et(E) = (\text{Runs}(E), \varepsilon, E_E, \rightarrow_E, <_E, L_E, l_E)$. On morphisms, $e2et(f) = f$.

On the other hand, any event tree determines an event structure: we define a set of events to be consistent iff they appear together on some branch, and, having extracted this information, we forget about the tree structure. Thereby we obtain a functor $et2e : ET \rightarrow E$.

Definition 1.10 Let $T = (S_T, s^0_T, E_T, Tran_T, <_T, L_T, l_T)$ be an event tree. Define $et2e(T) = (E_T, <_T, Con, L_T, l_T)$ where $Con$ exactly contains all sets $\{e_1, \ldots, e_n\}$ such that $s_1 \xrightarrow{e_1} s'_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{n-1}} s_n \xrightarrow{e_n} s'_n$ in $T$, for some states $s_1 \ldots s_n, s'_1 \ldots s'_n$ and sequences of events $v_1 \ldots v_{n-1}$. On morphisms, again $et2e(f) = f$.

Theorem 1.11 The functor $e2et$ is right adjoint to $et2e$. The adjunction is a reflection, i.e., the counit is a (natural) isomorphism.

Proof. [Sketch] Let $E$ be an event structure. Then $et2e(e2et(E)) = E$, essentially because the consistency relation derived from $e2et(E)$ recovers that of $E$. Hence, the counit $\varepsilon_E$ is the pair of identities $(1_E, 1_L)$.

The pair $(e2et(E), \varepsilon_E)$ is cofree over $E$ wrt $et2e$, i.e. for any arrow $(\eta, \lambda) : et2e(T) \rightarrow E$ in $E$, with $T$ an event tree, there is a unique arrow $f : T \rightarrow e2et(E)$ in $ET$ such that $(1_E, 1_L) \circ et2e(f) = (\eta, \lambda)$: it is $f = (\eta, \lambda)$, considering that $(\eta, \lambda)$ is a morphism from $T$ to $e2et(E)$ as well; $f$ is uniquely determined since $et2e$ is identity on morphisms.

As a consequence, $E$ embeds fully and faithfully into $ET$ and is equivalent to the full subcategory of $ET$ consisting of those event trees $T$ that are isomorphic to $e2et(et2e(T))$. Event trees that correspond to event structures are characterized as follows. We say that two distinct events $e_1, e_2$ of an event tree $T$ are concurrent, denoted by $e_1 c o_T e_2$, if they are consistent and neither $e_1 <_T e_2$ nor $e_2 <_T e_1$, similarly as it is done for event structures.
Proposition 1.12 An event tree $T$ is isomorphic to $e2et(e2et(T))$ iff $\text{Runs}(T)$ is trace-closed, i.e., satisfies the following condition: if $r e_1 e_2 r' \in \text{Runs}(T)$ and $e_1 \circ\!\!\!\circ e_2$ then $r e_2 e_1 r' \in \text{Runs}(T)$ as well.

The following diagram summarizes the four functors, which relate causal trees and event structures via event trees.

$$
\begin{array}{ccc}
C & \xleftarrow{et2c} & ET \\
\downarrow & & \downarrow \\
S & \xleftarrow{et2e} & E
\end{array}
$$

The hooks represent embeddings and the black arrows indicate the direction of left adjoints. Altogether, we have derived a composed adjunction between causal trees and event structures. It is not a coreflection, but induced by a coreflection and a reflection via a larger category. The object component of the right adjoint of this adjunction amounts to the transformation suggested in, e.g., [3]: it ‘linearizes’ an event structure into a causal tree by forgetting about events.

Integrating the coreflection from synchronization trees $S$ to $C$, and the well-known coreflection from $S$ to $E$ of [6] we obtain:

$$
\begin{array}{ccc}
C & \xleftarrow{et2c} & ET \\
\downarrow & & \downarrow \\
S & \xleftarrow{et2e} & E
\end{array}
$$

The diagram can be seen as a decomposition of the coreflection from $S$ to $E$ into three consecutive adjunctions. It is routine to check that the embeddings and left adjoints commute. The latter implies that right adjoints commute as well, and hence we obtain three different commuting squares:

$$
\begin{array}{ccc}
C & \xleftarrow{et2c} & ET \\
\downarrow & & \downarrow \\
S & \xleftarrow{et2e} & E
\end{array}
$$

2 Bisimulation from Open Maps

2.1 P-bisimilarity

Assume a category of models $M$ and a choice of path category $P \hookrightarrow M$, a subcategory of $M$. The choice for $P$ determines the notion of computation path that will be reflected by $P$-bisimilarity.

A morphism $f : X \rightarrow Y$ in $M$ is $P$-open iff it satisfies the following path-lifting condition. Whenever, for $m : P \rightarrow Q$ a morphism in $P$, a square (1) (c.f. diagrams below) in $M$ commutes, i.e. $q \circ m = f \circ p$, meaning the path $f \circ p$ in $Y$ can be extended via $m$ to a path $q$ in $Y$, then there is a morphism $p'$ such that in diagram (2) the two triangles commute, i.e. $p' \circ m = p$ and $f \circ p' = q$, meaning the path $p$ can be extended via $m$ to a path $p'$ in $X$ which matches $q$. 
Two objects \( X_1, X_2 \) of \( M \) are \( P \)-bisimilar iff there is a span of \( P \)-open morphisms \( f_1, f_2 \) as depicted in diagram (3). For the categories considered in this paper, \( P \)-bisimilarity is indeed an equivalence relation.

\[
\begin{array}{ccc}
P & \overset{P}{\xrightarrow{m}} & X \\
Q & \overset{q}{\xrightarrow{f}} & Y \\
\end{array}
\]

In the following, we work with respect to a fixed label set \( L \). Given a model category \( M \), whose objects have a label set, we restrict our attention to the fibre over \( L \) in \( M \) with respect to the obvious functor projecting the model objects to their label sets. This is exactly the subcategory of \( M \) with objects those models with label sets \( L \), and morphisms those having the identity on \( L \), as label component. We denote the fibre over \( L \) in \( M \) by \( M \_L \). Observe that all the adjunctions of Section 1 cut down to the fibres; in particular we have:

\[
C \_L \underset{\sim}{\rightarrow} E \_T \_L \underset{\sim}{\rightarrow} E \_L
\]

2.2 \( Hp\_b \) via Open Maps

To obtain a natural instantiation of \( P \)-bisimilarity for causal trees we single out a path category within \( C \_L \). Path objects are naturally taken to be causal branches, that is those causal trees which correspond to finite sequences of transitions.

**Definition 2.1** With respect to \( L \), define the category of causal branches \( C\text{Bran}_L \) to be the full subcategory of \( C \_L \) with objects those finite causal trees \( C \) satisfying:

(i) no forwards branching: \( s \xrightarrow{a} s' \& s \xrightarrow{b} s'' \Rightarrow a = b \& s' = s'' \).

A morphism \( m : P \rightarrow Q \) in \( C\text{Bran}_L \) shows how the causal branch \( Q \) can extend the causal branch \( P \): by additional transitions, and/or by increased concurrency. The \( C\text{Bran}_L \)-open morphisms are exactly those which are zig-zag (c.f. [4]) and additionally preserve causality; short we say they are causal zig-zag.

**Definition 2.2** Let \( f = (\sigma, 1_L) : C \rightarrow C' \) be a morphism in \( C \_L \). We say \( f \) is causal zig-zag iff it satisfies the following two conditions:

(i) zig-zag: for all \( s \in S_C \), if \( \sigma(s) \xrightarrow{a} s' \) in \( C' \) then \( s \xrightarrow{a} u \in C \) and \( \sigma(u) = s' \), for some \( u \in S_C \).

(ii) causality-preserving: for all \( t, t' \in \text{Tran}_C \), \( t <_C t' \Rightarrow f(t) <_{C'} f(t') \).

**Lemma 2.3** The \( C\text{Bran}_L \)-open morphisms of \( C \_L \) are exactly those which are causal zig-zag.

It turns out that \( C\text{Bran}_L \)-bisimilarity coincides with the well-known \( hp\_b \). Two systems are \( hp\_b \)-bisimilar iff their behaviour can be bisimulated while preserving the causal dependencies between their transitions. Technically, this can be realized by basing \( hp\_b \) on pairs of synchronous runs.
Let $C_1$, $C_2$ be causal trees with label sets $L$, $r_1 = t_1 \ldots t_n \in \text{Runs}(C_1)$, and $r_2 = t'_1 \ldots t'_m \in \text{Runs}(C_2)$. $r_1$ and $r_2$ are synchronous iff $n = m$, $\forall i \in [1,n]$, $l_i(t_i) = l_j(t'_j)$, and $\forall i, j \in [1,n]$, $t_i < t_j$ iff $t'_i < t'_j$. We denote the set of synchronous runs of $C_1$ and $C_2$ by $\text{SRuns}(C_1, C_2)$.

$\mathcal{H} \subseteq \text{SRuns}(C_1, C_2)$ is prefix-closed iff $(r_1t_1, r_2t_2) \in \mathcal{H}$ implies $(r_1, r_2) \in \mathcal{H}$. We assume hp-bisimulations to be prefix-closed; this restriction has no effect on the induced equivalence.

**Definition 2.4** Let $C_1$ and $C_2$ be causal trees with label sets $L$.

A history preserving (hp-) bisimulation relating $C_1$ and $C_2$ is a prefix-closed relation $\mathcal{H} \subseteq \text{SRuns}(C_1, C_2)$ that satisfies:

(i) $(\varepsilon, \varepsilon) \in \mathcal{H}$.

(ii) If $(r_1, r_2) \in \mathcal{H}$ and $r_1 t_1 \in \text{Runs}(C_1)$ for some $t_1 \in \text{Tran}_1$, then there is $t_2 \in \text{Tran}_2$ such that $(r_1 t_1, r_2 t_2) \in \mathcal{H}$.

(iii) Vice versa.

$C_1$ and $C_2$ are hp-bisimilar iff there exists a hp-bisimulation relating $C_1$ and $C_2$.

Given a morphism $f = (\sigma, 1_L) : C \rightarrow C'$ in $C_L$ we define the image of runs of $C$ in $C'$ inductively by: $f(\varepsilon) = \varepsilon$; $f(r(s, a, s')) = f(r)(\sigma(s), a, \sigma(s'))$. If $f$ is $\text{CBran}_L$-open and thus causality-preserving, it is easy to show that a run $r$ of $C$ and its image in $C'$ form a pair of synchronous runs.

**Proposition 2.5** Let $f : C \rightarrow C'$ be a $\text{CBran}_L$-open morphism in $C_L$. For any $r \in \text{Runs}(C)$ we have: $(r, f(r)) \in \text{SRuns}(C, C')$.

**Theorem 2.6** Two causal trees, with label sets $L$, are $\text{CBran}_L$-bisimilar iff they are hp-bisimilar.

**Proof.** ‘$\Rightarrow$’. Let $f : C \rightarrow C'$ be a $\text{CBran}_L$-open morphism in $C_L$. We show how from $f$ we obtain a hp-bisimulation relating $C$ and $C'$. By transitivity of hp-b this will clearly establish the ‘$\Rightarrow$’-direction. Define $\mathcal{H} = \{(r, f(r)) \mid r \in \text{Runs}(C)\}$. By Prop. 2.5 and prefix-closure of $\text{Runs}(C)$ it is clear that $\mathcal{H}$ is a prefix-closed subset of $\text{SRuns}(C, C')$. To prove that $\mathcal{H}$ is a hp-bisimulation for $C$ and $C'$ we further need to verify that conditions (i)-(iii) of Def. 2.4 are satisfied. (i) is obvious by $\varepsilon \in \text{Runs}(C)$. (ii) follows easily from $f$ being a morphism. (iii) can be obtained with the zig-zag condition, which $f$ satisfies by Lemma 2.3.

‘$\Leftarrow$’. Let $\mathcal{H}$ be a hp-bisimulation relating two causal trees $C_1$ and $C_2$, with label sets $L$. We observe that $\mathcal{H}$ can be regarded as a causal tree, $C_\mathcal{H}$, and that there are two morphisms $f_1 : C_\mathcal{H} \rightarrow C_1$ and $f_2 : C_\mathcal{H} \rightarrow C_2$ in $C_L$.

For $i \in \{1, 2\}$ we define a function $\pi_i : \text{SRuns}(C_1, C_2) \rightarrow S_i$ by: $\pi_i(\varepsilon, \varepsilon) = s_i^{\varepsilon\varepsilon}$, and $\pi_i(r_1 t_1, r_2 t_2) = tgt(t_i)$. Further, for $i \in \{1, 2\}$ we define the pair of maps $f_i = (\pi_i, 1_L)$. Given $(r, a, r') \in \text{SRuns}(C_1, C_2) \times L \times \text{SRuns}(C_1, C_2)$ we write $f_i(r, a, r')$ short for $(\pi_i(r), a, \pi_i(r'))$. 

15
Let \( C_H = (\mathcal{H}, (\varepsilon, \varepsilon), L, \text{Tran}_H, <_H) \) where 
\[ \text{Tran}_H = \{ ((r_1, r_2), a, (r'_1, r'_2)) \mid (r_1, r_2), (r'_1, r'_2) \in \mathcal{H}, r_1 \xrightarrow{a} r'_1 \& r_2 \xrightarrow{a} r'_2 \}, \]
\[ \forall u, u' \in \text{Tran}_H, u <_H u' \iff f_1(u) < f_1(u') \& f_2(u) < f_2(u'). \]

It is routine to check that \( C_H \) is indeed a causal tree, and that, with \( \pi_1, \pi_2 \) restricted to \( \mathcal{H} \), \( f_1 : C_H \to C_1 \) and \( f_2 : C_H \to C_2 \) are indeed morphisms in \( C_L \). Furthermore, \( f_1 \) and \( f_2 \) are causal zig-zag. But then by Lemma 2.3 there is a span of \( \text{CBran}_L \)-open morphisms as required.

\[ \square \]

2.3 Relating Hp-b and Hhp-b in ET

We capture the difference between hp-b and hhp-b by characterizing them within the category ET. We carry over hp-b to event structures and event trees. Two event structures \( E_1 \) and \( E_2 \) are hp-bisimilar iff \( e2et(e2et(E_1)) \) and \( e2et(e2et(E_2)) \) are hp-bisimilar; this is consistent with the standard definition. Analogously, it is natural to define: two event trees \( T_1 \) and \( T_2 \) are hp-bisimilar iff \( et2c(T_1) \) and \( et2c(T_2) \) are hp-bisimilar.

Consider the following instantiation of \( \text{P}-\)bisimilarity for event trees: as the path category within \( ET_L \) choose the image of \( \text{CBran}_L \) under the embedding functor \( e2et \); for simplicity, call it \( \text{CBran}_L \) as well. \( \text{CBran}_L \)-bisimilarity in \( ET_L \) characterizes hp-b:

**Proposition 2.7** Two event trees \( T_1 \) and \( T_2 \) are \( \text{CBran}_L \)-bisimilar iff they are hp-bisimilar.

Given a span of morphisms (as depicted in Section 2.1) in \( ET_L \), we say that the span is rooted in \( C_L \) if the root object \( X \) is \( e2et(C) \) for some causal tree \( C \), and that it is rooted in \( E_L \) if \( X \) is \( e2et(E) \) for some event structure \( E \). We have:

**Proposition 2.8** Two event trees \( T_1 \) and \( T_2 \) are \( \text{CBran}_L \)-bisimilar iff they are related by a \( \text{CBran}_L \)-open span rooted in \( C_L \).

By Prop. 2.7 and 2.8 it follows:

**Theorem 2.9** Two event structures \( E_1 \) and \( E_2 \) are hp-bisimilar iff \( e2et(E_1) \) and \( e2et(E_2) \) are related by a \( \text{CBran}_L \)-open span in \( ET_L \) rooted in \( C_L \).

Hhp-b is characterized in \( E_L \) as \( \text{Pom}_L \)-bisimilarity [4], where \( \text{Pom}_L \) is the full subcategory of finite \( \text{pomsets} \), i.e., of finite event structures without conflict (which means all finite subsets of events are consistent). We obtain:

**Lemma 2.10** Let \( f : E_1 \to E_2 \) be a morphism of event structures. Then \( f \) is \( \text{Pom}_L \)-open in \( E_L \) iff \( e2et(f) \) is \( \text{CBran}_L \)-open in \( ET_L \).

**Theorem 2.11** Two event structures \( E_1 \) and \( E_2 \) are hhp-bisimilar iff \( e2et(E_1) \) and \( e2et(E_2) \) are related by a \( \text{CBran}_L \)-open span in \( ET_L \) rooted in \( E_L \).

Theorems 2.9 and 2.11 indicate that \( C \) is the proper choice of model for hp-b while \( E \) is the natural choice for hhp-b.
3 Conclusions

Altogether we have advocated causality as a non-embedding but adjoining concept to true-concurrency. (We prefer the admittedly biased term ‘true-concurrency’ to ‘independence’ here since (in)dependence can be captured without a notion of event in the style of causal trees, just as well.) We summarize:

(i) Causality models are more basic than truly-concurrent models in that they capture causality without a notion of event. On the other hand, they are more expressive than the latter in that their possible runs can be freely specified in terms of a tree; in contrast, truly-concurrent models and their sets of runs adhere to certain axioms that express characteristics of independent events.

(ii) Hp-b turns out to have a straightforward open map characterization when we take causal trees to be the model category. Our results motivate that hp-b is the bisimilarity for causality while hhp-b remains the bisimilarity for true-concurrency.

Our work should be compared to [1], which relates causal trees to prioritized event structures. It would also be interesting to confirm our results with respect to models that keep the cyclic structure. A type of history-dependent automata, called causal automata, should be examined in this context.

Our investigation has led us to the new model of event trees. We are not keen on advertising yet another model for concurrency but event trees do arise in practice: given a truly-concurrent system, assume we restrict our attention to a subset of its runs that is not necessarily trace-closed. This is exactly what we do during a partial order reduction; indeed it is the intention here to lose trace-closure.

We are working on a characterization of those event structures E which correspond to causal trees in that $E = \text{et}^2e(\text{et}(C))$ for some causal tree C. We expect that such event structures are optimal for partial order reduction.

References


Interaction Nets vs. the $\rho$-calculus: Introducing Bigraphical Nets

Maribel Fernández, Ian Mackie

Department of Computer Science, King’s College London
Strand, London WC2R 2LS, UK

François-Régis Sinot

LIX, École Polytechnique, 91128 Palaiseau, France

Abstract
The $\rho$-calculus generalises both term rewriting and the $\lambda$-calculus in a uniform framework. Interaction nets are a form of graph rewriting which proved most successful in understanding the dynamics of the $\lambda$-calculus, the prime example being the implementation of optimal $\beta$-reduction. It is thus natural to study interaction net encodings of the $\rho$-calculus as a first step towards the definition of efficient reduction strategies. We give two interaction net encodings which bring a new understanding to the operational semantics of the $\rho$-calculus; however, these encodings have some drawbacks and to overcome them we introduce bigraphical nets—a new paradigm of computation inspired by Lafont’s interactions nets and Milner’s bigraphs.

Key words: Rewriting Calculus, Interaction Nets, Bigraphs.

1 Introduction

Pattern calculi [18,17,3,5,6,13] combine the expressiveness of pure functional calculi and algebraic term rewriting. The $\rho$-calculus [5,6] provides a simple framework generalising both term rewriting and the $\lambda$-calculus. It is an extension of the $\lambda$-calculus where we can abstract on patterns, not just on variables, hence providing a suitable foundational theory for modern programming languages with pattern-matching features.

Interaction nets [15] are graph rewrite systems which have been used for the implementation of efficient reduction strategies for the $\lambda$-calculus [12,16].

1 Projet Logical: PCRI, CNRS, École Polytechnique, INRIA, Université Paris-Sud
2 Email: maribel.ian@dcs.kcl.ac.uk
3 Email: frs@lix.polytechnique.fr

This is a preliminary version. The final version will be published in Électronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
One of the main features of interaction nets is that all the computation steps are explicit and expressed in the same formalism; there is no external machinery. Also, since reduction in interaction nets is local and strongly confluent, reductions can take place in any order, even in parallel (see [19]), which makes interaction nets well-suited for the implementation of programming languages and rewriting systems [9,8]. Since pattern calculi encompass term rewriting and the λ-calculus, clearly there are natural questions about the existence of efficient implementations using interaction nets. In this paper we will focus on the ρ-calculus, but our results can be applied to other pattern calculi.

We adapt interaction net implementations of the λ-calculus to deal with the specific features of the ρ-calculus. We concentrate on the problem of pattern-matching and take for granted that binders can be dealt with using one of the many available encodings of the λ-calculus mentioned previously. We give two alternative encodings of matching in interaction nets, both with certain advantages and drawbacks, and claim that a fully satisfactory solution cannot be obtained in the interaction net framework. This indeed comes as a surprise and highlights a deep difference between the λ-calculus and the ρ-calculus, namely that the ρ-calculus has more potential (implicit) parallelism. We then propose a third encoding using a particular class of bigraphs [14] which we call bigraphical nets. Bigraphs incorporate a notion of locality which is missing in interaction nets. We exploit this feature to overcome the drawbacks of the previous solutions, while still remaining close to a machine implementation.

Our work is modular with respect to the encoding of binders that is used, i.e. the same method can be applied to transform any interaction net implementation of the λ-calculus into an implementation of the ρ-calculus (even the bigraphical net encoding is modular, since interaction nets are also bigraphical nets). We can also easily export the solution to other calculi with patterns. Besides practical advantages (e.g. sharing of computations), graph representations are often useful to abstract away from syntactical details (e.g. α-conversion comes for free) and bring more understanding to the theory. In particular, the encodings presented in this paper highlight the differences between the various operational semantics defined for the ρ-calculus.

Section 2 recalls the ρ-calculus, interaction nets and bigraphs. Sections 3 and 4 give two interaction net encodings of the ρ-calculus, and their properties. Section 5 gives a third encoding using bigraphical nets, a new paradigm of graph rewriting incorporating locality. We conclude in Section 6.

2 Background

We start with a short presentation of the ρ-calculus; for more details see [5,6,2]. We write \( x, y, \ldots \) for variables and \( f, g, \ldots \) for constants. The set of ρ-terms (or just terms) \( \mathcal{T} \) is defined by: \( t, u ::=} x \mid f \mid p \rightarrow t \mid [p \ll u].t \mid (t u) \), where \( \mathcal{P} \) is an arbitrary subset of \( \mathcal{T} \) (\( p \in \mathcal{P} \) are called patterns); \( p \rightarrow t \) is a generalised abstraction (it can be seen either as a λ-abstraction on a pattern \( p \) instead of...
a single variable, or as a standard term rewriting rule); \([p \ll u].t\) is a delayed matching constraint denoting a matching problem \(p \ll u\) whose solutions (if any) will be applied to \(t\); \((t\ u)\) denotes an application (we omit brackets whenever possible, and associate to the left). Terms are always considered modulo \(\alpha\)-conversion (this is realised for free in interaction nets).

The \(\rho\)-calculus can be parametrised by the set \(\mathcal{P}\) of patterns. Here we use linear algebraic patterns: \(p := x \mid f \, p_1 \ldots p_n\) where each variable occurs at most once in \(p\). The reduction rules are the following:

\[
(p) \quad (p \to t) \; u \to [p \ll u].t
\]

\[
(\sigma) \quad [p \ll u].t \to \sigma_{p \ll u}(t)
\]

The rule \(\sigma\) asks for an external matching algorithm to find any solutions of the matching of \(p\) with \(u\), and applies the corresponding substitution to \(t\). Here we assume syntactic matching; under this assumption the calculus is confluent [6]. We focus on the implementation of the \(\sigma\) rule, isolating as much as possible the problem of matching from the problems of implementing binders. This methodology is justified by the fact that the terms \(p \to t\) and \(x \to [p \ll x].t\) are extensionally equivalent, so that we can safely precompile terms in order to abstract only on variables and have explicit matchings from the beginning. To give a full implementation of the calculus, including the matching algorithm, we first define a version of the \(\rho\)-calculus with explicit matching inspired by [4]. Substitution is still implicit since it will be realised for free in the graphical representation.

\[
(\rho) \quad (p \to t) \; u \to [p \ll u].t
\]

\[
(\sigma_v) \quad [x \ll u].t \to t\{x = u\}
\]

\[
(\sigma_{a_n}) \quad [f \, p_1 \ldots p_n \ll f \, u_1 \ldots u_n].t \to [p_1 \ll u_1] \ldots [p_n \ll u_n].t
\]

Note that \((\sigma_{a_n})\) represents an infinite family of rules; we thus replace it by a finite set of local rules, more suitable for an encoding into interaction nets:

\[
(a_c) \quad f \; t \to f \bullet t
\]

\[
(a_a) \quad (t \bullet u) \; v \to (t \bullet u) \bullet v
\]

\[
(\sigma_c) \quad [f \ll f].t \to t
\]

\[
(\sigma_a) \quad [(p \bullet r) \ll (u \bullet v)].t \to [p \ll u].[r \ll v].t
\]

A matching \([p \ll u]\) may have no solution; this is called a blocked matching. We can add a rule to detect failure for constants:

\[
(\perp) \quad [f \ll g].t \to \perp \quad \text{if } f \neq g
\]

Now if there is a \(\perp\) in a term, in which cases should the whole term evaluate to \(\perp\)? There are mainly two options:

\[
(\text{strict}) \quad C[\perp] \to \perp \quad \text{for any context } C[\cdot]
\]
The strict rule corresponds to an exception-like semantics of matching failure (as in ML) e.g. even if the argument of an application is not used by the function, the result is \( \bot \). In this context, a higher priority is given to this rule than to any other applicable rule. If we desire a non-strict semantics, this rule should be weakened to a particular class \( C \) of strict contexts:

\[
\text{(non-strict)} \quad C[\bot] \rightarrow \bot \quad \text{for any } C[\cdot] \in C
\]

We take \( C = \{([t], t \in T) \} \), although a larger class of contexts is acceptable. Rules of the form \([p \ll u].t \rightarrow \bot\) if \( p \) and \( u \) have different head symbols, are unsafe and lead to non-confluence [5]. Our encodings will not implement unsafe rules (automatically, due to the strong confluence of interaction nets).

A system of interaction nets is specified by a set \( \Sigma \) of symbols with fixed arities, and a set \( \mathcal{R} \) of interaction rules. An occurrence of a symbol \( \alpha \in \Sigma \) is called an agent. If the arity of \( \alpha \) is \( n \), then the agent has \( n + 1 \) ports: a principal port depicted by an arrow, and \( n \) auxiliary ports. Intuitively, a net \( N \) is a graph (not necessarily connected) with agents at the vertices and each edge connecting at most 2 ports. The ports that are not connected to another agent are free. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports. The interface of a net is its set of free ports. An interaction rule \((\alpha, \beta) =\rightarrow N) \in \mathcal{R}\) replaces a pair of agents \((\alpha, \beta) \in \Sigma \times \Sigma\) connected together on their principal ports (an active pair or redex) by a net \( N \) with the same interface. Reduction is local, and there can be at most one rule for each pair of agents. We use the notation \( =\rightarrow \) for the one-step reduction relation and \( =\rightarrow^* \) for its transitive and reflexive closure. If a net does not contain any active pairs then it is in normal form. One-step reduction satisfies the diamond property.

In [14] a notion of graph transformation system is defined, using nested (or hierarchical) graphs called bigraphs. Bigraphs represent two kinds of structure: locality (nodes may occur inside other nodes) and connectivity (nodes have ports that may be connected by links). We recall the basic terminology of bigraphs and refer the reader to [14] for details and examples.

Nodes are labelled by controls with fixed arities; the arity of a control corresponds to the number of ports of the node. Links are attached to nodes from the inside or the outside, so bigraphs have both an inner and an outer interface. A control is atomic if it cannot contain a nested graph, otherwise it is non-atomic. The reduction relation is defined by a set of reaction rules, which are pairs of bigraphs (called redex and reactum). The redex has a width, corresponding to the number of sites it occupies in the outer bigraph (see [14]).

A non-atomic control \( K \) can be specified as active, in which case reactions can occur inside, or passive, in which case reactions in the internal bigraph can only occur after the control \( K \) has been destroyed.

Interaction nets are a particular kind of bigraphs without nesting: all controls (called agents in interaction nets) are atomic, and have a distinguished port. Interaction rules can be seen as reactions in which both redex and
reactum have width 1, and redexes are restricted to just two controls connected by one link through the distinguished ports.

3 A Simple Interaction Net Encoding of the $\rho$-calculus

We can obtain an implementation of the $\rho$-calculus starting from any off-the-shelf interaction net encoding of the $\lambda$-calculus (that we will not describe) and adding a matching algorithm as specified in the explicit $\rho$-calculus (see Section 2). We first describe an encoding that aims at simplicity; however, we will see that this encoding is not able to represent a non-strict semantics of the $\rho$-calculus, which will motivate the following sections.

We give a translation $T(\cdot)$ of $\rho$-terms, and the interaction rules that will be used for solving matching constraints. A $\rho$-term $t$ with free variables $fv(t) = \{x_1, \ldots, x_n\}$ will be translated to a net $T(t)$ with the root edge at the top, and $n$ free edges corresponding to the free variables, as shown below (left)

If $t$ is a variable then $T(t)$ is just a wire. For each constant $f$ we introduce an agent as shown above (middle). A term of the form $[p \ll u].t$ is encoded as shown above (right) which can be interpreted as the substitution in $t$ of the (possible) solution of the matching (the left subnet corresponds to the matching problem $p \ll u$). We assume that terms have been precompiled to abstract only on variables, as described in the previous section; hence we can reuse the abstraction of the $\lambda$-calculus: We introduce an agent $\rightarrowtriangle$ with its principal port oriented upwards. Similarly for application and interaction between abstraction and application ($\beta/\rho$-reduction): we introduce an agent $@$ with its principal port oriented towards the left subterm, so that interaction with an abstraction is possible.

In this simple encoding, the matching algorithm is initiated by connecting the root of a pattern with the term to match. Thus, the rule $(\sigma_v)$ (matching against a variable) is realised for free, as in the $\lambda$-calculus. To simulate $(\sigma_a)$ and $(\perp)$, constants will interact. When two identical constants interact, they cancel each other to give the empty net, as indicated in Figure 1 (left). If the agents are not the same, then we introduce an agent $\text{fail}$, which represents a failure in the matching algorithm, as indicated in Figure 1 (right). Note that the right-hand side of the second rule in Figure 1 is a deadlocked net (because the matching is disconnected). Interaction is not possible between the agent $\text{fail}$ and the rest of the term; consequently, we interpret a net containing an agent $\text{fail}$ anywhere as an overall failure. Unfortunately, this implements the

---

4 A dashed edge represents a bunch of edges (a bus).
Fig. 1. Matching of constants (success and failure)

Fig. 2. Rules to transform patterns

Fig. 3. Matching applications

rule (**strict**) as a side effect.

In the $\rho$-calculus application is used in two very different ways: on one hand as the application of an abstraction to a term, and on the other hand as a term constructor in patterns, as shown in Section 2. We have to convert a usual application (@) into a pattern application (●) when it is part of an algebraic pattern (or term), using the rules in Figure 2. We also need a rule to match applications, which is given in Figure 3.

**Example 3.1** The term $(f \rightarrow h) \ g$ evaluates to a blocked matching (failure) $[f \ll g].h$. This reduction is represented in interaction nets by:

In this case there are two disconnected parts in the resulting net, one representing $h$ and the other one representing the matching failure.

**Properties of the encoding**
In most applications of interaction nets, nets which become disconnected are no longer of interest, i.e. they are considered garbage and thus ignored. This is especially done because, in general, garbage collection requires a full traversal and even full evaluation, which motivated some work on strategies in interaction net [10,20]. However, the previous example shows that disconnected
nets, even without any free ports, do matter in this interaction net encoding, which implies that we cannot stop reduction at the so-called interface-normal-forms [10]. On the other hand, this is very good news for a parallel implementation (which is also a motivation of interaction nets): disconnected components can easily be dispatched on different processors. Moreover, in our case, the only interesting information about these matching components is whether they finally evaluate to the empty net (successful matching) or not (matching failure, but this is undecidable).

Although good for parallelism, the nasty consequence of this observation is that we cannot implement a non-strict semantics. For instance, reduction of \((x \to h)((f \to i) g)\) gives (where the \(\epsilon\) agent is used for erasure):

We cannot distinguish the result of this computation (for which just \(h\) could be a desirable answer) from the previous failing one. This corresponds to adding the strict rule (cf. Section 2), treating pattern-matching failures as exceptions (cf. [6] for a corresponding big-step semantics, although we allow some terms to be reduced before matching). If we want pattern-matching failure to be treated as a matching failure in a lazy language rather than as an exception, then we cannot disconnect matching constraints. We present an alternative encoding which maintains connectivity in Section 4.

The simulation of the rules \((\rho)\) and \((\sigma_v)\) is under the responsibility of the encoding of the \(\lambda\)-calculus used. Rules corresponding to \((a_c), (a_a), (\sigma_c), (\sigma_a)\) and \((\bot)\) have been given. The (strict) rule is realised by the interpretation of agent \(\text{fail}\), as we explain below. Assuming correctness of the encoding of the \(\lambda\)-calculus we can show (where \(\to\) denotes reduction in the explicit \(\rho\)-calculus with the strict rule):

**Correctness:** If \(T(t) \Rightarrow^* T(u)\) connected, then \(t \to^* u\). If \(T(t) \Rightarrow^* T(u)\) disconnected but without failure, then there is a \(v\) such that \(T(u) = T(v)\) and \(t \to^* v\). If \(T(t) \Rightarrow^* N\) where \(N\) is a net with a failure, then \(t \to^* \bot\).

**Completeness:** If \(t\) is closed and \(t \to^* u \neq \bot\) in normal form, then \(T(t) \Rightarrow^* T(u)\). If \(t \to^* \bot\) then \(T(t) \Rightarrow^* N\) containing a fail agent.

The provisos of the correctness properties are justified by situations of the form \((f \to t) \Omega \to [f \ll \Omega].t\) where \(\Omega\) is a non-terminating term, thus the matching constraint cannot be eliminated, but it is represented by a disconnected interaction net (i.e. the readback is not unique). The restriction to
$t$ closed and $u$ in normal form for completeness are standard [16]. Since we disconnect the matchings from the rest of the net, we can only interpret a fail agent remaining at the end of the reduction as failure; hence we are modelling a strict calculus (i.e. the explicit $\rho$-calculus plus the $(\bot)$ and (strict) rules).

4 Introducing a Matching Agent

Disconnecting matching constraints increases the parallelism of the implementation but does not allow us to implement a non-strict semantics. The obvious solution to this problem is to maintain connectivity by using an explicit agent for matching. Then we can keep track of the point where a matching failure may have occurred, thus staying closer to the syntax and behaviour of standard $\rho$-calculus. We make the minimum amount of changes to the previous encoding. The counterpart of the $\rho$-rule now creates an explicit matching agent $\ll$ linked to the rest of the net by an agent $[]$ (the right-hand side of the rule is thus the new representation of a matching constraint):

We have introduced two new agents: $\ll$ will take care of the matching and $[]$ will simply attach the matching to the right place, exactly mimicking the structure of terms: $[p \ll u].t$. Note that the agent $[]$’s principal port is looking towards the agent $\ll$, i.e. it will wait until the matching is done and look at the result, which will be either success or failure. If the matching cannot be solved in either way, this agent will stay there forever.

Success will be represented by an agent $\top$, failure by an agent $\bot$, both 0-ary, with the expected interactions with agent $[]$. The ideas are settled, now the development of the matching algorithm is straightforward.

It is clear that each rewrite step in the explicit $\rho$-calculus corresponds to a sequence of interaction rules in this encoding, hence properties of simulation are stronger than with the previous encoding. Here $\rightarrow$ denotes the explicit non-strict reduction, and $t$, $u$ belong to $\mathcal{T} \cup \{\bot\}$.

Proposition 4.1 (Correctness and completeness) If $\mathcal{T}(t) \Longrightarrow^* \mathcal{T}(u)$, then $t \rightarrow^* u$. If $t$ is closed and $t \rightarrow^* u$ in normal form, then $\mathcal{T}(t) \Longrightarrow^* \mathcal{T}(u)$.

The problem with this encoding is that the agent $[]$ blocks any further computation between the root and what was the body of the abstraction. For instance, in the term $(a \rightarrow b) ((p \rightarrow c) \ t)$ where $a, b, c$ are constants and $p, t$ are terms, we will have to complete the matching of $t$ against $p$ (which may be costly) before noticing that $a$ and $c$ do not match. This was not the case in the first encoding. The moral interpretation of this result is that
the drawback identified in the first encoding is not about connectivity but about locality. Hence we naturally turn our attention towards an extended framework inspired by bigraphs: bigraphical nets.

5 Using Bigraphical Nets

Bigraphs [14] introduce a notion of locality (using nesting to indicate that a graph is local to a certain node) which is missing in interaction nets, and which is a key to solving the problems of the previous encodings: if we can specify that a pattern (and later a matching constraint) is local to a certain abstraction, we can keep track of occurrences of failure and implement a non-strict $\rho$-calculus without introducing an explicit matching agent. Note that bigraphs permit links between nested nets and external subgraphs (unlike hierarchical graphs [7]) and rewriting can take place across control boundaries, both features which will be of use in our encoding of the $\rho$-calculus. To encode the $\rho$-calculus we will use a subclass of bigraphs that contains interaction nets, and that we call bigraphical nets.

**Definition 5.1** Bigraphical nets are bigraphs in which each control has a distinguished principal port (the remaining ports are called auxiliary), and links connect at most 2 ports. Reaction rules define interactions between two controls connected by their principal ports (and their sites), or interactions of a control with its local sites, preserving the interfaces.

For examples of a bigraphical net and a reaction rule, see Figure 4. Note that, in contrast with interaction nets, a left-hand side can specify the location in which the reacting controls are, or the locations contained in these controls, and reactions can take place across boundaries. A full study of bigraphical nets as a computational framework is beyond the scope of this paper, however, we remark that all the examples of bigraphs for the $\pi$-calculus and ambient calculus given in [14] (part I) can be redefined as bigraphical nets by adding principal ports and copy/erase controls to preserve the interface of the reactions. Comparing with the properties of interaction nets, we remark that confluence does not hold in general for bigraphical nets, because of the possibility of interactions across boundaries. However reduction is still local. The latter point is crucial for implementation.

Bigraphical nets can be seen as a particular class of higher-order nets [11]. Following the usual terminology of interaction nets, controls in bigraphical nets are also called agents, and reaction rules are called interaction rules.

We will use the following agents for the encoding:
- $\rightarrow$ of arity 2, which is a non-atomic agent representing abstraction;
- $@$ of arity 2, which is an atomic agent representing application;
- $f, g, \ldots$ of arity 0, for constants;
- a non-atomic agent $M$ of arity 1 to represent matching problems;
• a family of non-atomic agents $\alpha_M$, where $\alpha$ is any of the agents above except $M$;
• $\bot$ of arity 0 (atomic), to represent matching failure.

All non-atomic agents permit interactions to take place inside nested nets, and also across the agent boundary (i.e. they are active). The translation of an application, a variable, or a constant, are the same as in the first interaction net encoding. We give the translation of an abstraction $p \rightarrow u$ in Figure 4(a), where we omit the encoding of the box, as before. The reaction implementing the $\rho$-rule is given in Figure 4(b). The rules to implement the matching algorithm are given in Figure 5. The first rule allows agents from the body of the abstraction (except for copy and erase agents) to interact at the root, keeping track of the matching constraint. We assume that the non-atomic agents $\alpha_M$ have the same behaviour as $\alpha$ (i.e. same interaction rules), and permit the same interactions as $M$ between the net inside and the outside (whence the name $\alpha_M$). We omit the rule for $M$ and $\epsilon$ which should erase $M$ and its content, sending $\epsilon$ agents along the interface. If $\alpha$ is itself an $\alpha_M$ then the interaction produces again $\alpha_M$ with an additional matching constraint. The next rule eliminates the $M$ agents after the matching constraint has been solved. The last three rules decompose matching problems with application, and detect success or failure when constants interact (we assume $\alpha \neq f$). Similar rules for $\alpha_M$ are omitted (with an empty net it reduces to $\alpha$).

![Figure 4. Bigraphical encodings of abstraction and rule ($\rho$)](image)

**Simulation of $\rho$-calculus reductions**

The interaction rules above simulate the rules $\rho$ and $\sigma$ of the explicit $\rho$-calculus (the rule $\sigma_v$ is obtained for free). This encoding allows us to implement a non-stric semantics (and also a strict one, see below), similarly to the encoding that uses an explicit matching agent. Unlike the latter, the matching agent does not block interactions between the net representing the body of the abstraction and the context of the term. We have the following results, which are similar to those stated in Section 4.
Proposition 5.2 (Correctness and completeness) If $T(t) \Rightarrow^* T(u)$, then $t \rightarrow^* u$. If $t$ is closed and $t \rightarrow^* u$ in normal form, then $T(t) \Rightarrow^* T(u)$.

Example 5.3 We now reconsider the examples of Section 4 with this encoding. Evaluation of the term $(f \rightarrow h) g$ produces a matching failure (both under the strict and non-strict semantics) as shown in Figure 6. On the other hand, the term $(x \rightarrow h)((f \rightarrow i) g)$ reduces to $h$, since the $\epsilon$ agent erases the failing matching (therefore implementing the non-strict semantics), as shown in Figure 7. To implement a strict semantics it is sufficient to take out the rule between $\epsilon$ and a fail agent, and interpret any net containing $\bot$ as failure.
Fig. 7. Evaluation of \((x \rightarrow h)((f \rightarrow i) \ g)\) with bigraphical nets

6 Conclusion

Although there are several good encodings of the \(\lambda\)-calculus in interaction nets, the problem of designing a satisfactory encoding of the \(\rho\)-calculus is non-trivial, because the problem of matching introduces more potential parallelism than in the \(\lambda\)-calculus, which is difficult to handle satisfactorily with interaction nets because they lack a notion of locality. We finally proposed a framework which is expressive enough for this.

For the sake of clarity, we omitted some details (like free variables in patterns, and non-linear patterns) in order to isolate the real problem, that is the problem of matching. We nevertheless assure the reader that the issue would stay unchanged in a more realistic implementation.

The original motivations for this work were to provide grounds for implementing in a distributed setting the \(\rho\)-calculus, which can be seen as a foundational model for functional languages featuring pattern-matching, or for rewriting. This actually led us far beyond, and we have also proposed bigraphical nets as a model of distributed computation with local synchronisations which is suitable for our particular problem (and for a larger class of problems). This will be the subject of future work.
References


Feasible Reactivity for Synchronous Cooperative Threads

Roberto M. Amadio 1

Université de Paris 7

Frédéric Dabrowski 2

INRIA Sophia-Antipolis

Abstract
We are concerned with programs composed of cooperative threads whose execution proceeds in synchronous rounds called instants. We develop static analysis methods to guarantee that each instant terminates in time polynomial in the size of the parameters of the program at the beginning of the computation.

Key words: Synchronous and cooperative programming, Resource bounds, Quasi-interpretations, Termination, Polynomial time.

1 Introduction

In [8], Boussinot and De Simone introduced the Synchronous Language (SL). A program in SL is a set of cooperative threads interacting through shared signals whose execution proceeds in synchronous rounds called instants. A fundamental hypothesis of the model is that the reaction to the absence of a signal within an instant can only happen in the following instant. Reactivity is the essential property that one should guarantee of an SL program. This means that at each instant the program fed with an input will ‘react’ producing an output.

The SL language has gradually evolved into a general purpose programming language for concurrent applications and has been implemented in various programming environments such as C, JAVA, SCHEME, and CAML. Typical applications effectively developed in these languages include event-driven controllers, data flow architectures, graphical user interfaces, simulations, web services, and multi-player games (see, e.g., [13,5]).

1 Email: amadio@cmi.univ-mrs.fr
2 Email: frederic.dabrowski@inria.fr

This is a preliminary version. The final version will be published in Électronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
All the extensions of the SL language mentioned above introduce data types such as integers, lists, trees. What does it mean to ensure reactivity in this context? One may consider three increasingly ambitious goals. The first one, is to ensure that every instant terminates. The second one, is to guarantee that the computation of an instant terminates within feasible bounds which depend on the size of the parameters at the beginning of the instant. The third one, is to guarantee that the parameters of the program stay within certain bounds, and thus the resources needed for the execution of the program are controlled for arbitrarily many instants.

In this note we introduce a basic version of the SL model enriched with data types and develop static analysis methods to guarantee that each instant terminates in time polynomial in the size of the parameters of the program at the beginning of the computation. Following previous work by one of the authors [1], the method is based on a combination of standard termination techniques for term rewriting systems and an analysis of the size of the computed values based on the notion of quasi-interpretation. With respect to [1], the main novelties are: (1) A more general and abstract formalisation of the model. (2) A method to generate inequalities whose satisfaction in suitable structures entails a polynomial bound on the size of the parameters of the program for arbitrarily many instants (theorem 4.2). (3) A new method to ensure polynomial time termination which can be regarded as a specialisation of size change termination (theorem 5.1).

2 Model

A program is a multi-set of threads described by a list of mutually recursive type, function, and behaviour definitions. Threads interact through shared signals which may carry general values (including signals). The language should be regarded as an intermediate code where complex control structures have been compiled into a simple tail-recursive form.

Types and Constructors. We assume a list of types \( t, t', \ldots \) and a list of constructors \( c, c', \ldots \). For constructors of particular ‘signal’ types we may use the notation \( r, r', \ldots \) and refer to them as reference values. A value \( v \) is a first order term built out of constructors. The size \(|v|\) of a value \( v \) is defined by \(|c| = 0\) and \(|c(v_1, \ldots, v_n)| = 1 + |v_1| + \cdots + |v_n|\).

We will use the notation \( \mathbf{a} \) to denote a vector \( a_1, \ldots, a_n \) of elements and denote with \( \sigma, \sigma', \ldots \) a substitution \([v/x]\) mapping variables to values. Types and constructors are declared by a system of equations having one of the following shapes:

\[
\begin{align*}
(1) \quad t &= \cdots | c \text{ of } t_1, \ldots, t_n | \cdots \\
(2) \quad t &= \text{Sig}(t') \text{ with } \cdots | r := v | \cdots
\end{align*}
\]

In equation (1), we declare a type \( t \) and a constructor \( c \) with functional type \( (t_1, \ldots, t_n) \to t \). In equation (2), we declare a type of evaluated signals.
including a signal $r$ whose value at the beginning of each instant is $v$. It is intended that the value $v$ has type $t'$ (see below) and for the sake of simplicity we assume $|v| = 0$. Signals can be read and written and their values are always defined.

The system of equations is subject to the usual convention that types and constructors occurring in it are declared exactly once. This means that we can assign to every constructor a unique type $(t_1, \ldots, t_n) \rightarrow t$ where $n \geq 0$. With respect to this assignment, values are typed according to the rule: the value $c(v_1, \ldots, v_n)$ has type $t$ if $c$ is assigned the type $(t_1, \ldots, t_n) \rightarrow t$ and the values $v_i$ have type $t_i$ for $i = 1, \ldots, n$. Finally, we have a special behaviour type beh: elements of this type do not return a value but produce side effects. In the following we will manipulate various syntactic concepts: variables, values, patterns, expressions, expression bodies, substitutions, behaviours, programs, . . . and we will always assume that they are well typed. The typing rules are standard and are omitted.

**Expressions.** Let $x, y, \ldots$ denote variables ranging over values. A pattern $p$ is a well-typed term built out of constructors and variables. In particular, a linear pattern $p$ is a pattern whose variables are all distinct. In the following all the patterns are supposed to be linear.

An expression $e$ has the shape $h(e_1, \ldots, e_n)$ where $n \geq 0$ and $h$ can be either a variable $x$, or a constructor $c$, or a function symbol $f$. A function symbol $f$ of type $(t_1, \ldots, t_n) \rightarrow t$ is specified by an equation $f(x) = eb$. Here $eb$ is an expression body defined by the grammar:

$$eb ::= e | \text{match } x \text{ with } \cdots \Rightarrow eb \cdots .$$

To simplify the presentation, we assume that a (well-typed) value matches exactly one pattern.

A closed expression body $eb$ evaluates to a value $v$, written $eb \Downarrow v$, according to the following standard rules where $\sigma$ denotes a matching substitution and $e \Downarrow v$ stands for $e_1 \Downarrow v_1, \ldots, e_n \Downarrow v_n$.

$$
\frac{e \Downarrow v}{c(e) \Downarrow c(v)} \quad \frac{e \Downarrow v, f(x) = eb, [v/x]eb \Downarrow v}{f(e) \Downarrow v} \quad \frac{\sigma p = v' \quad \sigma eb \Downarrow v}{\text{match } v' \text{ with } \cdots \Rightarrow eb \cdots \Downarrow v}.$$

**Thread behaviours.** In the following we let $\varrho, \varrho', \ldots$ range over both variables and reference values. We denote with $b, b', \ldots$ behaviours defined as follows:

$$b ::= \text{stop} \mid f(e) \mid \text{yield}.b \mid \text{next}.f(e) \mid \varrho := e.b$$

$$\text{match } x \text{ with } \cdots \Rightarrow b \cdots \mid \text{read } \varrho \text{ with } \cdots \Rightarrow b \cdots [x] \Rightarrow f(e)$$

where $f$ is a functional symbol of type $(t_1, \ldots, t_n) \rightarrow \text{beh}$ and defined by an equation $f(x) = b$. In the examples, we may omit the branches $p \Rightarrow b$ or the
branch \([x] \Rightarrow b\) of the \textit{read} instruction. Behaviours produce side effects and their execution is relative to a \textit{store} whose elements we denote with \(s, s', \ldots\). A store is a finite partial function mapping reference values to values which is type compatible. We will denote with \(s_o\) the default store with which the computation is initialised at the beginning of each instant.

Behaviour reduction is described by the eight rules below. A reduction \((b, s) \Downarrow (b', s')\) means that the behaviour \(b\) with store \(s\) runs an atomic sequence of actions till \(b'\), producing a store \(s'\) and returning the control to the scheduler.

\[
\begin{align*}
(b_1) & \quad (\text{stop}, s) \Downarrow (\text{stop}, s) \\
(b_2) & \quad (\text{yield} \cdot b, s) \Downarrow (b, s) \\
(b_3) & \quad (\text{next} \cdot f(e), s) \Downarrow (\text{next} \cdot f(e), s) \\
(b_4) & \quad \sigma p = v, (\sigma b, s) \Downarrow (b', s') \\
(b_5) & \quad e \Downarrow v \quad (b, s[v/r]) \Downarrow (b', s') \\
(b_6) & \quad (\text{match} v \Rightarrow b \Rightarrow \cdots, s) \Downarrow (b', s') \\
(b_7) & \quad s(r) = \sigma p, (\sigma b, s) \Downarrow (b', s') \\
(b_8) & \quad (\text{read} r \Rightarrow \cdots p \Rightarrow b \Rightarrow \cdots, s) \Downarrow (b', s')
\end{align*}
\]

The effect of the various instructions is informally described as follows: \textit{stop}, terminates the executing thread for ever; \textit{yield} \cdot b, halts the execution and hands over the control to the scheduler — the control should return to the thread later in the same instant and execution resumes with \(b\); \(f(e)\) and \textit{next} \cdot f(e)\) switch to another behaviour immediately or at the beginning of the following instant; \(r := e.b\), evaluates the expression \(e\), assigns its value to \(r\) and proceeds with the evaluation of \(b\); \textit{read} \(r \Rightarrow \cdots p \Rightarrow b \Rightarrow \cdots, x \Rightarrow f(e)\), waits until the value of \(r\) matches one of the patterns \(p\) (there could be no delay) and yields the control otherwise; if at the end of the instant the thread is always stuck waiting for a matching value then it starts the behaviour \(f(e)\) in the following instant where \(x\) is the value of \(r\) at the end of the instant; \textit{match} \(v \Rightarrow \cdots p \Rightarrow b\) filters the value \(v\) according to the pattern \(p\); the execution never blocks since we have assumed there is always a matching pattern. We say that a behaviour \(b\) with the store \(s\) is suspended, written \((b, s)^\dagger\), if \((b, s) \Downarrow (b, s)\) by means of a proof of height 1 (rules \((b_1), (b_3)\) or \((b_5)\)). Thus a suspended behaviour \(b\) has necessarily the shape \textit{stop} or \textit{next} \cdot f(\ldots) or \textit{read} \ldots.

\textbf{Programs.} We represent abstractly a program \(P\) as a non-empty multi-set of behaviours. The evaluation of a program \(P\) with store \(s\) is defined as follows:

\[
\begin{align*}
(p_1) & \quad \forall b \in P \quad (b, s)^\dagger \quad (P, s) \Downarrow ([P]_s, s) \\
(p_2) & \quad \exists b \in P \quad (\neg(b, s)^\dagger \quad (b, s) \Downarrow (b'', s'') \quad (P \setminus \{b\} \cup \{b''\}, s'') \Downarrow (P', s')) \quad (P, s) \Downarrow (P', s')
\end{align*}
\]
where the program \([P]_s\) representing the result of the execution of the threads at the end of the instant is defined as follows:

\[
[P]_s = \{[[b]_s \mid b \in P]\} \quad [\text{stop}]_s = \text{stop}
\]

\[
\langle \text{next} \rangle f(e)_s = f(e) \quad \langle \text{read} \tau \ldots [x] \Rightarrow f(e) \rangle_s = [s(r)/x]f(e)
\]

**Remark 2.1** [fairness] The execution of a program within an instant consists of a serialisation of the execution of the behaviours that compose it until all behaviours are suspended. Rule \((p_2)\) allows a completely non-deterministic scheduling of the behaviours. We say that an execution (within an instant) is *unfair* if there is a behaviour which is run (at least) twice and between the two runs there is a distinct behaviour which is continuously enabled but never run. Programs including the *yield* instruction may rely on the hypothesis that all executions are fair (cf. example 3.2).

### 3 Constraints generation

We introduce a suitable control flow analysis associating with a program a set of inequalities over first order terms.

**Read once condition.** We require that threads perform any given read instruction at most once in an instant. This can be checked by a simple control flow analysis rejecting programs that may traverse several times within an instant the same read instruction. Following this check, we assign to every read instruction in a program a distinct fresh label, \(y\), and we collect all these labels in an ordered sequence, \(y_1, \ldots, y_m\). In the following, we will sometimes use the notation \(\text{read}_{(y)} @ \text{with} \ldots\) in the code of a behaviour to make visible the label of a *read* instruction. Then with every function symbol \(f\) defining a behaviour we associate a list \(y_f\) composed of the labels of the read instructions we may execute within an instant starting from \(f\). The important point is that the computation of \(f\) within an instant can be regarded as a function of the parameters and the values read within the instant.

**Control points.** A *control point* is a triple \((f(p), bc, i)\) where, intuitively, \(f\) is the currently called function, \(p\) represents the patterns crossed so far in the function definition plus possibly the labels of the read instructions that still have to be executed, \(bc\) is the continuation, and \(i\) is an integer flag in \(\{0, 1, 2\}\) that will be used to associate with the control point various kinds of conditions. If the function \(f\) returns a value and is defined by the equation \(f(x) = eb\), then we associate with \(f\) the set \(\mathcal{C}(f, x, eb)\) defined as follows:

\[
\mathcal{C}(f, p, eb) = \text{case } eb \text{ of } \\
\quad e : \{(f(p), e, 0)\}
\]

\[
\text{match } x \text{ with } \ldots p \Rightarrow eb' \ldots \cdot \cdot \cdot \cup \mathcal{C}(f, [p/x]p, eb') \cup \cdot \cdot \cdot
\]
On the other hand, suppose the function $f$ defines a behaviour by the equation $f(x) = b$. Then we generate a fresh function symbol $f^+$ whose arity is that of $f$ plus the number of variables in $y_f$ (the ordered sequence of labels corresponding to read instructions that may be performed by $f$ within an instant). When unfolding the definition of $C$ the parameters $x$ and the labels $y_f$ of the function $f^+$ may be replaced by patterns.

When going from one instant to the following, we need to control the size of the parameters. The basic idea is that if $f$ may call $g$ in the current instant or in the following then the parameters of $f$ should control the size of the parameters of $g$. This idea has to be implemented with some care because some values may depend on read instructions and some parameters may be discarded before the following instant begins. Therefore, we identify first the function symbols that may start a behaviour. They include all function symbols that follow a next instruction or a $[y] \Rightarrow \ldots$ branch. We call these function symbols initial. Next we need some notation. Let $0$ be a fresh constant. If $h$ is a function of arity $n$ and $I \subseteq \{1, \ldots, n\}$ then $h(e_1, \ldots, e_n)_I$ is defined as $h(e'_1, \ldots, e'_n)$ where $e'_i = e_i$ if $i \in I$ and $e'_i = 0$ otherwise. Intuitively, in $h(e_1, \ldots, e_n)_I$ we set to $0$ all arguments that are not in $I$. For each function symbol $f$ defining a behaviour of arity $n$ with a related function symbol $f^+$ of arity $n + m$ we define a set $I_f \subseteq \{1, \ldots, n\}$ with the condition that $I_f = \{1, \ldots, n\}$ if $f$ is initial. In particular this means that we neglect all arguments that correspond to the read instructions.

With this convention, the set of control points associated with $f^+$ is the set $C(f^+, (x, y_f), b)$ defined as follows:

\[
C(f^+, p, b) = \text{case } b \text{ of } \\
(C_1) \quad \text{stop} \quad : \emptyset \\
(C_2) \quad g(e) \quad : \{(f^+(p), g^+(e, y_g), 0), (f^+(p), g^+(e, y_g)_{I_p}, 2)\} \\
(C_3) \quad \text{yield } b' \quad : C(f^+, p, b') \\
(C_4) \quad \text{next } g(e) \quad : \{(f^+(p)_{I_f}, g^+(e, y_g)_{I_p}, 2)\} \\
(C_5) \quad \varrho := e, b' \quad : \{(f^+(p), e, 1)\} \cup C(f^+, p, b') \\
(C_6) \quad \text{match } x \text{ with } \ldots p \Rightarrow b' \ldots \ldots \cup C(f^+, (p/x)p, b') \cup \ldots \\
(C_7) \quad \left( \begin{array}{l}
\text{read } (y) \varrho \text{ with } \ldots \\
p \Rightarrow b' \ldots [y] \Rightarrow g(e)
\end{array} \right) \quad : \{(f^+(p)_{I_f}, g^+(e, y_g)_{I_p}, 2)\} \cup \\
\ldots C(f^+, [p/y]p, b') \ldots 
\]

Note that in the clause $(C_2)$, the read once condition guarantees that the labels $y_g$ occur in the patterns $p$. An instance of a control point $(f(p), be, i)$ is an expression body or a behaviour $be' = \sigma(be)$, where $\sigma$ is a substitution mapping the free variables in $be$ to values. In order to carry on the proofs, it is convenient to reformulate expression body evaluation and behaviour evaluation on instances of control points. A hint on how to do this is given in the
proof of theorem 4.2 and a complete treatment is available in [1].

We associate with a control point \((f(p), b, i)\) a constraint \(f^+(p) \succeq_i b\) for \(i = 0, 1, 2\), and say that the constraint has index \(i\). Intuitively, we rely on the constraints of index 0 to enforce termination of the instant, on those of index 0, 1 to enforce a bound on the size of the computed values within an instant, and on those of index 0, 1, 2 to guarantee a bound on the size of the computed values for arbitrarily many instants. Note that the constraints are on pure first order terms, a property that allows us to reuse techniques developed in the standard term rewriting framework.

**Example 3.1** As a running example, we consider the case of a server \(f(s, x)\) that at every instant, yields the control, reads a list of requests on the signal \(s\), and serves the requests:

\[
\begin{align*}
  f(s, x) &= \text{yield.read } s \text{ with } l \Rightarrow f'(s, x, l) \\
  f'(s, x, l) &= \text{match } l \text{ with } \text{nil} \Rightarrow \text{next.f}(s, x) | \\
  &\quad \text{cons(req}(r, y), l') \Rightarrow r := h_1(y, x).f'(s, h_2(y, x), l')
\end{align*}
\]

The server maintains a state \(x\). A request contains a data \(y\) and a return signal \(r\). We leave the functions \(h_1\) and \(h_2\) unspecified; the first is used to reply to the request and the second to compute the following state of the server. A client \(g(s, r, y)\) that wishes to use the server could be defined as follows:

\[
g(s, r, y) = \text{read } s \text{ with } l \Rightarrow s := \text{cons(req}(r, y), l). \\
\text{yield.read } r \text{ with } z \ldots
\]

Notice that the operation of inserting a message in a list requires a read operation and therefore the read once condition forbids to iterate this kind of operation within an instant. However, arbitrarily many behaviours may perform the operation within an instant. We compute the constraints of index 0, 1, 2 assuming that \(f\) is initial, \(f'\) is not initial, and \(I_f = \{1, 2\} = I_{f'}\).

\[
\begin{align*}
  f'^+(s, x, \text{cons(req}(r, y), l')) &\succeq_0 f'^+(s, h_2(y, x), l) & f'^+(s, x, l) &\succeq_0 f'^+(s, x, l) \\
  f'^+(s, x, \text{cons(req}(r, y), l')) &\succeq_1 h_1(y, x) & f^+(s, x, 0) &\succeq_2 f'^+(s, x, 0) \\
  f'^+(s, x, 0) &\succeq_2 f^+(s, x, 0) & f'^+(s, x, 0) &\succeq_2 f'^+(s, h_2(y, x), 0)
\end{align*}
\]

**Example 3.2** [registers] In our framework, registers can be regarded as signals that preserve their values from one instant to the following. We can simulate a register \(r\) with a signal (with the same name) and a thread whose behaviour \(f(r)\) is described by:

\[
f(x) = \text{read } x \text{ with } [y] \Rightarrow g(x, y), \ g(x, y) = x := y.f(x)
\]

The behaviour \(f(r)\) waits the end of the instant to read the value \(y\) of \(r\) and in the following instant it writes \(y\) again in \(r\). Since in the simulation \(r\) is a signal, at the beginning of the instant its value is reset. Then we have to make sure that the behaviour \(f(r)\) runs before any other behaviour tries to read \(r\).
For this purpose, we rely on the fairness hypothesis (cf. remark 2.1), and transform all other behaviour definitions so that they start with a yield. We can extract from this simulation conditions to control the size of the values in the registers. In particular, we note that the constraint $f^+(x,0) \succeq_2 g^+(x,y)$ can only be satisfied if the value $y$ contained in the register has bounded size. This is an important restriction we have to impose on the programming language.

4 Size bounds

In order to bound the size of the values computed by a program we rely on the notion of quasi-interpretation (see [6,2,3,7]). In a nutshell, a quasi-interpretation $q_f$ of a function symbol $f$ is a numerical function ensuring that there is a constant $k$ such that the size of the largest value computed by $f$ when called with arguments $v_1, \ldots, v_n$ is bounded by $q_f(k|v_1|, \ldots, k|v_n|)$.

The synthesis of quasi-interpretations can be mechanised to some extent. The existence of a quasi-interpretation does not entail termination but it does allow to control the complexity of the computed function following a well-known result of S. Cook [9] who showed that a polynomially bounded auxiliary push down automaton can be simulated by a Turing Machine in exponential time using a ‘table’ to store intermediate results. We refer to [3] for an extended discussion of these issues.

Assignments and quasi-interpretations. Suppose given a program. An assignment $q$ associates with each constructor and function symbol $h$, a function $q_h$ over the natural numbers $\mathbb{N}$ such that:

1. If $c$ is a constructor with arity $n$ then $q_c = 0$ if $n = 0$ and $q_c(x_1, \ldots, x_n) = d + \sum_{i=1}^n x_i$ if $n > 0$ where $d \in \mathbb{N}$ and $d \geq 1$ (this guarantees that the quasi-interpretation of a value is proportional to its size). In particular, the quasi-interpretation of the special constant $0$ introduced in the constraints is the natural number $0$.

2. If $f$ is a function symbol with arity $n$ then $q_f : (\mathbb{N})^n \to \mathbb{N}$ is a monotonic function over the natural numbers.

We say that a function $U : \mathbb{N} \to \mathbb{N}$ bounds the assignment $q$ if $\forall x \in \mathbb{N} \ q_h(x, \ldots, x) \leq U(x)$. In particular, we say that $q$ is polynomially bounded if the function $U$ can be a polynomial. We associate with an expression $e$ without variables a natural number $q_e$ as follows:

$$q_h(e_1, \ldots, e_n) = q_h(q_{e_1}, \ldots, q_{e_n})$$

We write $q \models e_1 \succeq e_2$ if the assignment $q$ satisfies a constraint $e_1 \succeq e_2$ where $e_1, e_2$ are expressions (possibly with variables). This is defined as:

$$q \models e_1 \succeq e_2 \text{ if } \forall \sigma \ q_{\alpha e_1} \succeq q_{\alpha e_2}$$
where $\sigma$ is a substitution associating values with variables. We also write: $q \models e_1 \succ e_2$ if $\forall \sigma \ q_{\sigma e_1} > q_{\sigma e_2}$.

An assignment $q$ is a quasi-interpretation if it satisfies the constraints of index 0, 1, 2 generated by the program and moreover if it satisfies $f(x_1, \ldots, x_n) \succeq x_i$ for all $i = 1, \ldots, n$ and all function symbols $f$. This last condition allows to control the size of the values computed by a function and not just the size of its result. Thus if $f(v_1, \ldots, v_n) \Downarrow v$ then we know that $q_{f(v_1, \ldots, v_n)} \succeq q_v$ and moreover that for any value $u$ computed by the function $q_{f(v_1, \ldots, v_n)} \succeq q_u$.

**Example 4.1** We define a quasi-interpretation for the running example 3.1. We suppose the functions $h_1$ and $h_2$ operate over values of bounded size. Then we can just take $q_{h_1} = q_{h_2} = \lambda(x, y).k$ for some suitable constant $k \in \mathbb{N}$. We can also set $q_{\text{cons}}(x, l) = x + l + 1$ and $q_{\text{eq}}(r, y) = r + y + 1$. Then we can satisfy all the constraints by assuming $q_{f^+}(s, x, l) = q_{f^+}(s, x, l) = \max(l, k)$.

**Theorem 4.2** If a program $P$ has a polynomially bounded quasi-interpretation $q$ then the size of the largest value computed by $P$ is polynomial in the size of the parameters of the program at the beginning of the computation.

In order to prove theorem 4.2, we define a small step reduction of behaviours on instances of control points. The reduction makes abstraction of the memory and the scheduler while depending on an assignment $\delta$ associating values with the labels of the read instructions. The assignment $\delta$ is a kind of oracle that provides the thread with the values it may read within the current instant. A fresh assignment is generated whenever we move from one instant to the following one (rules $b'_3$ and $b'_6$).

\[
\begin{align*}
(b'_2) \quad & (f^+(p), \text{yield}, b, \sigma, \delta) \quad \rightarrow (f^+(p), b, \sigma, \delta) \\
(b'_3) \quad & (f^+(p), \text{next}, g(e), \sigma, \delta) \quad \rightarrow (f^+(p), g(e), \sigma, \delta') \\
(b'_4) \quad & (f^+(p), \text{match}, x \text{ with } \ldots p \Rightarrow b \ldots, \sigma, \delta) \quad \rightarrow (f^+([p/x]p), b, \sigma_1 \circ \sigma, \delta) \text{ if (1)} \\
(b'_5) \quad & (f^+(p), g := e, b, \sigma, \delta) \quad \rightarrow (f^+(p), b, \sigma, \delta) \text{ if } \sigma e \Downarrow v \\
(b'_6) \quad & (f^+(p), \text{read}, y, \ldots y, \sigma, \delta) \quad \rightarrow (f^+(p), g(e), [\delta(y)/y] \circ \sigma, \delta') \text{ if (2)} \\
(b'_7) \quad & (f^+(p), \text{read}, y, \ldots p \Rightarrow b \ldots, \sigma, \delta) \quad \rightarrow (f^+(p), b, \sigma_1 \circ \sigma, \delta) \text{ if (3)} \\
(b'_8) \quad & (f^+(p), g(e), \sigma, \delta) \quad \rightarrow (g^+(x, y), b, \sigma', \delta) \text{ if (4)}
\end{align*}
\]

where: (1) $\equiv \sigma_1 p = \sigma x$, (2) $\equiv$ (no pattern matches $\delta(y)$), (3) $\equiv \sigma_1(p) = \delta(y)$, and (4) $\equiv (\sigma e \Downarrow v, g(x) = b, \sigma' = [v/x])$. Relying on the small step semantics we then prove the following lemma from which theorem 4.2 follows directly.

**Lemma 4.3** Suppose $q$ is a polynomially bounded quasi-interpretation for a program $P$ with $n$ distinct read instructions and $n$ threads. Let $i$ denote a thread of the program and $P(i)$ the associated behaviour.

1. Suppose that at the beginning of the computation $P(i) = f(v)$ and that $P(i) = g(e)$ at the beginning of a following instant. Then $q_{f(v)} \succeq q_{g(e)}$. 

40
Suppose that at the beginning of an instant \( P(i) = f(v) \). Then the size of the values computed by the thread \( i \) during that instant is bounded by 
\[
q_f + (v, u)
\]
where \( u \) are the values contained in the signals at the time they are read by the thread (or their default value, otherwise).

Suppose \( c \) is a bound on the quasi-interpretation of the parameters of the behaviours at the beginning of an instant and that \( U \) is a polynomial bound on the quasi-interpretation. Then the size of the values computed by the program \( P \) during an instant is (polynomially) bounded by 
\[
U^{m+1}(c)
\]

5 Polynomial time reactivity

Marion et al. [11,6,7] have shown how to ensure termination in polynomial space or time by combining the existence of a polynomially bounded quasi-interpretation with termination by suitable restrictions of the recursive path ordering (see, e.g., [4]). In this section we propose a more flexible approach where we compare the arguments of the functions using the quasi-interpretation rather than the restricted recursive path order.

The constraints of index 0 have one of the following shapes: (A) \( f(p) \geq_0 e \) or (B) \( f^+(p) \geq_0 g^+(e) \). We start by building the least pre-order (reflexive and transitive) \( \geq_F \) on the function symbols such that \( f \geq_F g \) if \( f \) appears on the left hand side and \( g \) on the right hand side of a constraint of index 0. We write \( f =_F g \) if \( f \geq_F g \) and \( g \geq_F f \). We note that a function symbol that returns a value can never call a function symbol that generates a behaviour, so we have that the latter is always larger than the former.

As in recursive path orderings, we associate a status with each function symbol which determines how to compare the arguments of the function. In our case, we consider either a lexicographic or a multi-set status. We assume that function symbols that are equivalent with respect to the pre-order \( \geq_F \) have the same arity and the same status.

Following [7], we say that a constraint is linear if there is at most one function symbol on the right hand side which is equivalent to the function symbol on the left hand side. Henceforth, we assume that all the constraints of index 0 are linear (note that the constraints of shape (B) are always linear).

Suppose given a polynomially bounded quasi-interpretation \( q \) for the program. We rely on the quasi-interpretation to compare the arguments of equivalent function symbols. Therefore, we depart from the standard RPO method and rely on a size change analysis as in [10].

For lexicographic comparison, we write: \( q \models (p_1, \ldots, p_n) >_{lex} (e_1, \ldots, e_n) \) if there is an \( i < n \) such that for all \( \sigma \), \( q_{\sigma p_1} \geq q_{\sigma e_1}, \ldots, q_{\sigma p_{i-1}} \geq q_{\sigma e_{i-1}}, \) and \( q_{p_i} > q_{e_i} \). For multi-set comparison, we write: \( q \models (p_1, \ldots, p_n) >_{mset} (e_1, \ldots, e_n) \) if for all \( \sigma \), \( \{q_{\sigma p_1}, \ldots, q_{\sigma p_n}\} >_{mset} \{q_{\sigma e_1}, \ldots, q_{\sigma e_n}\} \), where \( \{\ldots\} \) is our notation for multi-sets and \( >_{mset} \) is the multi-set order over the natural numbers. We say that the quasi-interpretation is compatible with the order if in all con-
constraints of index 0 of the shape:

\[ f(p_1, \ldots, p_n) \succeq_0 C[g(e_1, \ldots, e_n)] \]

where \( C \) is a one hole context and \( f =_F g \) with status \( st \in \{ \text{lex}, \text{mset} \} \) we have that \( q \models (p_1, \ldots, p_n) >_st (e_1, \ldots, e_n) \).

**Theorem 5.1** Suppose the program has a compatible and polynomially bounded quasi-interpretation. Then the computation of an instant terminates in time polynomial in the size of the parameters at the beginning of the instant.

**Proof hint.** In our language all function definitions are first-order. Then the state of each thread can be represented by a stack of frames. A frame is a triple \((f, pc, v_1 \cdots v_n)\) where \( f \) is the name of a function, \( pc \) points to the instruction of the function to be executed, and \( v_1 \cdots v_n \) is a stack of values. The maximum number of values that can be on the stack can be statically determined. Thus the size of a frame is determined by the size of the values that can be found on the stack and the quasi-interpretation provides a polynomial bound for that. Our task is to bound the number of frames that a thread can allocate within an instant. The rank of a function symbol \( f \) is the length of the longest chain of functions such that \( f >_F f_1 >_F \cdots >_F f_n \) with respect to the preorder on function symbols. We estimate the number \( ncall(o) \) of frames that a function \( f \) of rank \( o \) can generate when called with arguments of size at most \( B \) using the linearity of the constraints of index 0 and the hypothesis that the quasi-interpretation is over the natural numbers.

**Example 5.2** With reference to the running example 3.1, 4.1, we assume \( f^+ >_F f'^+ \) and a lexicographic status (from right to left) for the function \( f'^+ \).

**Remark 5.3** The method in [7] is based on a recursive path ordering that coincides with the homeomorphic embedding \( \succ_{emb} \) on constructors. We observe that if \( p \succ_{emb} p' \) and \( q \) is an assignment then \( q \models p > p' \). It follows that the method presented here succeeds whenever the one presented in [7] does (and therefore all PTIME functions can be represented). The converse fails since we can never have \( p \succ_{emb} e \) when \( e \) contains a function symbol.

6 Conclusion

We have presented a static analysis that guarantees that a program composed of synchronous cooperative threads reacts every instant in time polynomial in the size of the parameters at the beginning of the computation. The conditions we have imposed refine and extend those proposed in [1] to control the resources within an instant. As it could be expected, the possibility of controlling the resources for arbitrarily many instants comes at a price. First, the parameters of each thread at the beginning of each instant have to be essentially non-size increasing. To satisfy this requirement it is often necessary to rely on the fair yield hypothesis and to reprogram the application.
so that an instant is sufficiently large; typically, an instant corresponds to a
protocol transaction or to a logical simulation step. Second, the registers (or
persistent signals) have to carry values of bounded size while unbounded data
structures can still be allocated on signals (which are reset at the beginning
of each instant).

Acknowledgement The authors were partially supported by ACI CRISS.
This work was started while the first author was at the Université de Provence.

References

in Theoretical Computer Science.


ACM PPDL, 2005.


November 2004. Available from the authors.


at the USENIX Technical Conference, 1996.

www-sop.inria.fr/mimosa/rp.
Static Equivalence is Harder than Knowledge

Johannes Borgström

School of Computer and Communication Sciences, EPFL, Switzerland

Abstract
There are two main formulations of secrecy of cryptographic protocols. The first version checks if the adversary can have knowledge of the value of a secret parameter. In the second version, one checks if the adversary can notice any difference between two protocol runs with different values of the secret parameter.

In this paper we give a new proof that when considering more complex equational theories than partially inversible functions, these two kinds of secrecy are not equally difficult to verify. More precisely, we identify a message language equipped with a convergent rewrite system such that after a completed protocol run, the first problem mentioned above is decidable but the second problem (static equivalence) is not. The proof is by reduction of the ambiguity problem for context-free grammars.

Key words: Security protocol analysis, Term rewriting, Decidability.

1 Introduction
There are two main ways of specifying secrecy for a cryptographic protocol.

(1) One common approach is to take the set of messages intercepted by the attacker after some interaction with honest protocol participants, and check if he can deduce the value of a secret parameter of the protocol. This disclosure-based approach is intuitive and straightforward to define, and is taken in, e.g., [15,17,13].

(2) The other approach is to check whether the attacker can notice any difference between protocol runs with different values of the secret parameter. This indistinguishability-based approach fits naturally into the standard process calculus framework [5,8], is a standard notion of secrecy of cryptographic primitives [12], and is thus often used for protocol analysis in the probabilistic polynomial-time tradition [16]. This approach can also be used for other properties than secrecy, by comparing an implementation of the protocol with an executable specification.

1 Email: Johannes.Borgstroem@EPFL.ch, home page: http://lamp.epfl.ch/~jobo/.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
Independently of the particular security properties to be verified, the formal cryptography tradition [11] is moving towards a more complete treatment of algebraic properties of cryptographic primitives [4] as well as a more fine-grained treatment of “compound primitives” such as block encryption algorithms used in electronic code book or cipher block chaining mode, or message authentication codes [14]. However, algorithms treating such more complex message algebras are often defined ad-hoc [9] and/or without termination guarantees (e.g., naive additions to ProVerif [6]). Recent work [1,3] aims at finding a sufficiently large class of message algebras, where the relevant properties still are decidable.

In this paper, we prove that there exist message algebras in which after a protocol run, disclosure is decidable but indistinguishability is not. The proof is by reducing the ambiguity problem for context-free grammars to an indistinguishability problem. Previously, a proof sketch for this separation result, based on another undecidable problem relating two pairs of Turing machines, appeared in [1,2]. However, the present paper is, to the knowledge of the author, the first published instance of a full proof.

2 Formal Cryptography

The basic idea behind formal cryptography is to abstract from the actual encryption algorithms used, and instead work with some suitable message algebra. The reason for this is that cryptographic primitives are often in themselves fairly complex algorithms and the guarantees that they provide are usually based on probabilities and computation time, which together makes for a complicated model for the verification. Formal cryptography, on the other hand, works with algebraic relationships between cryptographic primitives. Implicit in this approach is that the only operations on messages are the ones permitted by the algebra. Thus, formal cryptography is the study of protocols under assumptions of perfect cryptography.

2.1 Message Algebras

Definition 2.1 We assume countably infinite sets of names $n \in N$, variables $x \in V$ and function symbols $f \in F$, and a finite signature $\Sigma : F \to N$ taking function symbols to their arity (which may be 0). The set of terms $T_{\Sigma}$ is then defined by $t, u ::= n | x | f(t_1, \ldots, t_n)$ where $\Sigma(f) = n$. Let $|t|_u$ be the number of occurrences of $u$ in $t$. We let $n(t)$ be the names and $v(t)$ be the variables of a term $t$. The concrete terms $T_{\Sigma}^c$ are those that do not contain any variables.

In algebras for cryptography, message equality is typically induced by some rewrite system. In the case of symmetric cryptography, this may be as simple as the single rule $\text{dec}(\text{enc}(x, k), k) \rightarrow x$, stating that a message $x$ encrypted (\text{enc}) under the key $k$ can be decrypted (\text{dec}) using the same key.
In order to more accurately model the behavior of particular implementations of cryptographic primitives, one can add to and modify this rule [10]. One drawback with such refinements is that the rewrite system might no longer be convergent, so the decidability of equality must be proven for each variation. Since names are often used to model many different types of cryptographic data, such as public and private keys, nonces, and primitive messages, we also permit rewrite rules that apply only to names of a certain type. This gives the adversary increased distinguishing power.

**Definition 2.2** A rewrite rule is of the form “$t_1 \rightarrow t_2$ if $\phi$”, where $t_1, t_2 \in T_\Sigma$ and $\phi$ is a conjunction of membership predicates $x_i \in S_i$ for certain $S_i \subseteq N$. We require $v(t_2) \cup v(\phi) \subseteq v(t_1)$. An equational theory $E$ is defined by a finite set of rewrite rules. A term $t$ matches a rewrite rule of the form above if there is a substitution $\sigma : v(t_1) \rightarrow T_\Sigma$ such that $t = t_1 \sigma$ and $\phi \sigma$ is true. If $E$ is an equational theory defined by a set containing this rewrite rule, $t$ can be head rewritten to $t_2 \sigma$, which we write $t \rightarrow^h_E t_2 \sigma$. We let $\rightarrow_E$ be the closure of $\rightarrow_E$ under contexts, and $\equiv_E$ be the transitive, reflexive and symmetric closure of $\rightarrow_E$. When $E$ is clear from the context, we often omit it.

As an example, if we assume a set of DES keys $K_{DES} \subset N$, the rewrite rule “IsDESKey($x$) $\rightarrow$ true if $x \in K_{DES}$” permits checking if a message $x$ is a name that can be used as key for the symmetric encryption algorithm DES.

Note that since theories are defined by a finite set of rewrite rules, the set of names has a finite partitioning into equivalence classes with respect to these rules, so exhaustive enumerations can work modulo this equivalence without any impact on decidability properties.

In what follows, we will assume that $\equiv_E$ is decidable; this is notably the case if the rewrite system $\rightarrow_E$ is confluent and terminating. For these (convergent) rewrite systems, we write $t \downarrow$ for the unique term such that $t \rightarrow^* E t \downarrow \not\rightarrow_E$.

### 2.2 Frames and Operations

The most important dynamic characteristic of a Dolev-Yao adversary is the set of messages that it has learned by communicating with the legitimate participants of the protocol. This message set is the only information needed to verify if the adversary knows a particular (confidential) datum. For the indistinguishability-based approach we want to compare results of corresponding operations on the knowledge of two adversaries, so we need some way of relating corresponding messages. One way of modelling this, used in [8] for the spi calculus, is to represent the attacker knowledge as a substitution.

As usual, the adversary can apply any combination of cryptographic functions to the messages he possesses. He can also freshly generate names (nonces, keys, ...), that are different from all other names in the system. In order to preserve this distinction, we augment the substitution representing attacker knowledge with a tuple of names that cannot be freshly generated. This augmented knowledge is called a frame, following [1].
**Definition 2.3** A frame is a pair \((\nu N)\sigma\), where \(N \subset \mathcal{N}\) is finite and \(\sigma : \mathcal{V} \rightarrow \mathcal{T}_\Sigma^\mathcal{E}\) is partial with finite domain. We let \(\text{bn}((\nu N)\sigma) := N\).

The frame \((\nu N)\sigma\) can **primitively generate** the message (term) \(t\), written \((\nu N)\sigma \vdash^p t\), if there is \(t'\) such that \(\nu(t') \cap N = \emptyset\), \(\nu(t') \subseteq \text{dom}(\sigma)\) and \(t'\sigma = t\). Given an equational theory \(E\), \((\nu N)\sigma\) **generates** \(t\) in \(E\), written \((\nu N)\sigma \vdash_E t\), if there is \(t'\) such that \((\nu N)\sigma \vdash^p t'\) and \(t' \equiv_E t\).

The disclosure-based definition of secrecy corresponds to asking whether, after a completed run of the protocol, the frame representing the adversary knowledge can generate the value of the secret parameter. For the indistinguishability-based definition we ask whether one can notice any difference, using only \(\equiv_E\), when studying pairs of messages generated simultaneously.

**Definition 2.4** Two frames \((\nu N_1)\sigma_1\) and \((\nu N_2)\sigma_2\) where \(\text{dom}(\sigma_1) = \text{dom}(\sigma_2)\) are **indistinguishable**, written \((\nu N_1)\sigma_1 \approx_s (\nu N_2)\sigma_2\), if for all \(t, u\) such that \((\nu(t) \cup \nu(u)) \cap (N_1 \cup N_2) = \emptyset\) and \((\nu(t) \cup \nu(u)) \subseteq \text{dom}(\sigma_1)\), we have \(t\sigma_1 \equiv_E u\sigma_1\) iff \(t\sigma_2 \equiv_E u\sigma_2\).

In regard to automated verification, since \(\mathcal{T}_\Sigma\) is enumerable we immediately get that the message construction problem is semidecidable and the indistinguishability problem is co-semidecidable (assuming that \(\equiv_E\) is decidable). An important question for automated verification is for which message algebras these problems are decidable. In [1], the authors proved that in message algebras with the encryption rule mentioned above, decidability of \(\approx_s\) implies decidability of \(\vdash\). Moreover, they gave an example of a convergent rewrite system with \(\vdash\) decidable but \(\approx_s\) undecidable. In this paper, we exhibit another rewrite system with the same properties but in a simpler setting (context-free grammars versus Turing machines), and develop a full proof.

## 3 Reduction of Ambiguity to Static Equivalence

Our example message algebra, where deduction is decidable but static equivalence is not, is based on leftmost derivations of context-free grammars in Chomsky normal form. We first recall some definitions for such grammars.

### 3.1 Context-free grammars

A context-free grammar \(G = (A_G, X_G, s_G, T_G \cup N_G)\) in Chomsky normal form (CNF) consists of terminal symbols \(A_G\), non-terminal symbols \(X_G\) (with \(A_G \cap X_G = \emptyset\)), an initial symbol \(s_G \in X_G\), and two kinds of derivation rules: **terminal** and **non-terminal** rules. Terminal rules \((n \rightarrow t) \in T_G\) take a non-terminal symbol \(n\) to a terminal symbol \(t\), whereas non-terminal rules \((n \rightarrow n_1n_2) \in N_G\) take a non-terminal symbol to two non-terminal symbols.

A leftmost derivation of \(\tilde{w} \in A_G^*X_G^*\) is a word \(r_1 \cdots r_k \in (T_G \cup N_G)^*\) where there exist words \(\tilde{a}^0, \tilde{a}^1, \ldots, \tilde{a}^k \in A_G^*\) and \(\tilde{x}^0, \tilde{x}^1, \ldots, \tilde{x}^k \in X_G^*\) such that \(\tilde{a}^0 \tilde{x}^0 = s_G, \tilde{a}^k \tilde{x}^k = \tilde{w}\) and for all \(i = 1, \ldots, k\) we have that either
$r_i = (n \rightarrow t) \in T_G$, $\tilde{a}^i = \tilde{a}^{i-1}t$ and $n\tilde{x}^i = \tilde{x}^{i-1}$, or $r_i = (n \rightarrow n_1n_2) \in N_G$, $\tilde{a}^i = \tilde{a}^{i-1}$ and $\tilde{x}^i = n\tilde{y}$ and $\tilde{x}^{i-1} = n_1n_2\tilde{y}$ for some $\tilde{y}$. It is easy to show that $k$ above (the length of the derivation) is equal to $|\tilde{w}| + |\tilde{a}^n| - 1$. Such a derivation is called partial if $\tilde{w} \notin A^*_G$. The language of a grammar $L(G)$ is the set of words over $A_G$ that have a leftmost derivation. Additionally, a grammar in CNF has no useless non-terminals, in the following sense.

$$\forall x \in X_G. (L(A_G, X_G, x, T_G \cup N_G) \neq \emptyset \land \exists \tilde{w}_1, \tilde{w}_2, \tilde{r}. \tilde{r} \text{ is a leftmost derivation of } \tilde{w}_1x\tilde{w}_2)$$

A grammar $G$ is ambiguous if there exists a word $\tilde{w} \in L(G)$ that has two different leftmost derivations. A standard result in formal language theory is that it is undecidable whether a given context-free grammar (in CNF) is ambiguous. In what follows, we define a rewrite system such that this problem is equivalent to the indistinguishability problem for a particular frame pair.

### 3.2 Message algebra

We begin by introducing a message algebra intended to model leftmost derivation according to the rules of a context-free grammar in Chomsky normal form. Let $\Sigma$ be the following signature.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Intuitive meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Nil}$</td>
<td>0</td>
<td>Nil</td>
</tr>
<tr>
<td>$\text{id}$</td>
<td>1</td>
<td>Identifier</td>
</tr>
<tr>
<td>$(\cdot \cdot)$</td>
<td>2</td>
<td>Pair</td>
</tr>
<tr>
<td>$\text{OK}$</td>
<td>2</td>
<td>Name type check</td>
</tr>
<tr>
<td>$T$</td>
<td>2</td>
<td>Terminal grammar rule</td>
</tr>
<tr>
<td>$N$</td>
<td>3</td>
<td>Non-terminal grammar rule</td>
</tr>
<tr>
<td>$\text{dc}$</td>
<td>5</td>
<td>Derivation context</td>
</tr>
</tbody>
</table>

The five arguments of the derivation context ($\text{dc}$) have the following meanings:

1. The symbol with which a derivation started.
2. (Ensures that rewriting does not reduce the size of terms.)
3. A list of terminals forming a prefix of the word that is derived.
4. A list of the non-terminals that remain to be rewritten.
5. A list of the derivation rules that have not yet been applied.
Let $E$ be the equational theory on $\Sigma$ induced by the following rewrite rules:


dc(\text{Nil, Nil, Nil, Nil}, (T(y,t) . u)) \rightarrow
dc(y, (\text{OK} (\text{Nil, Nil}) . \text{Nil}), (t . \text{Nil}), \text{Nil}, u) \quad (1)

dc(\text{Nil, Nil, Nil, Nil}, (N(y,t_1,t_2) . u)) \rightarrow
dc(y, (\text{OK} (\text{Nil, Nil}) . \text{Nil}), (t_1 . (t_2 . \text{Nil})), u) \quad (2)

dc(v, w, x, (y . z), (T(y,t) . u)) \rightarrow dc(v, (\text{OK} (y, y) . w), ((t . x), z), u) \quad (3)

dc(v, w, x, (y . z), (N(y,t_1,t_2) . u)) \rightarrow
dc(v, (\text{OK} (y, y) . w), x, (t_1 . (t_2 . z)), u) \quad (4)

\text{OK}(m,n) \rightarrow \text{OK}(\text{Nil, Nil}) \text{ when } m, n \in \mathcal{N} \quad (5)

Note that these rules are terminating and confluent when oriented left to right, so the equality problem is clearly decidable. Intuitively, the rules denote the following operations related to leftmost derivations:

1. Initial derivation step, using a terminal rule.
2. Initial derivation step, using a nonterminal rule.
3. Subsequent derivation step, using a terminal rule.
4. Subsequent derivation step, using a nonterminal rule.
5. Hiding of the non-terminal that is discharged (iff it is a name).

**Theorem 3.1** The deduction problem for $E$ is decidable.

**Proof.** By inspection, the rewrite rules have the property that $T \rightarrow T'$ implies that $|T| \leq |T'|$, so no term is of greater syntactic size than its normal form. Thus, all equivalence classes are finite modulo injective renaming. To check deducibility, we check if any of a finite (modulo injective renaming as above) number of terms can be primitively generated, which clearly is decidable. □

### 3.3 Translation

Given the rewrite system above and a context-free grammar, we look for a pair of frames that are indistinguishable if and only if the grammar is unambiguous.

**Definition 3.2** If $G := (A_G, X_G, s_G, T_G \cup N_G)$ is in CNF where $A_G \cup X_G \subset \mathcal{N}$, and $f_x : \mathcal{N} \times \mathcal{N} \rightarrow x$ and $g_x : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \rightarrow x$ are families of injective functions with range($f_x$) \cap range($g_x$) = $\emptyset$ for $x = V, \mathcal{N}$, then we let

$$T_1(G) := (\nu A_G \cup X_G) \{ [T(a,b)]_{f_x(a,b)} \mid (a \rightarrow b) \in T_G \}$$

$$\quad \cup \{ [T(a,b,c)]_{g_x(a,b,c)} \mid (a \rightarrow bc) \in N_G \},$$

$$T_2(G) := (\nu \text{ n} (\text{ range}(T_2(G)))) \{ [1^{id} (f_x(a,b))]_{f_x(a,b)} \mid (a \rightarrow b) \in T_G \}$$

$$\quad \cup \{ [1^{id} (g_x(a,b,c))]_{g_x(a,b,c)} \mid (a \rightarrow bc) \in N_G \}$$
At the corresponding point in the proof of [2] (Proposition 5, page 17) the authors conclude: “Then we can verify that [an undecidable property holds] if and only if [the two frames are statically equivalent].” However, they say nothing of how to verify that. To clarify this for ourselves and others, we devote the remainder of this paper to a proof of this proposition in our setting.

### 3.4 Derivations

In what follows, we assume a fixed context-free grammar \( G \) in CNF where \( G := (A_G, X_G, s_G, T_G \cup N_G) \). The following lemma shows that partial derivations of \( G \) can be simulated by the rewrite system. In order to state the lemma, we first need some auxiliary definitions.

**Definition 3.3** We define the following shorthand notations for terms.

- **list** Let \([ \epsilon ] := \text{Nil} \) and \([ \hat{w}v ] := (v \cdot [\hat{w}])\).
- **derivation length** Let \( dl(0) := \text{Nil} \) and \( dl(n + 1) := (\text{OK}(\text{Nil}, \text{Nil}) \cdot dl(n))\).
- **rule** Let \( \text{rule}(k \rightarrow lm) := N(k, l, m) \) and \( \text{rule}(n \rightarrow a) := T(n, a) \).
- **derivation** Let \( \text{der}_x(\epsilon) := x \) and \( \text{der}_x(r_1 \tilde{r}') := (\text{rule}(r_1) \cdot \text{der}_x(\tilde{r}')) \).

We can then state the lemma.

**Lemma 3.4** Let \( \text{tail}^k(\hat{w}) := w_{k+1} \ldots w_{|w|} \). Then \( s_G \xrightarrow{k} A \tilde{n} \) using the partial leftmost derivation \( \tilde{\tau} := r_1 \tilde{r}_2 \ldots r_k \), where \( A \in A^*_G \) and \( \tilde{n} \in X^*_G \) iff for any \( x \),

\[
\text{dc}(\text{Nil, Nil, Nil, Nil, \text{der}_x(\tilde{\tau}))} \rightarrow^{2k-1} \text{dc}(s_G, dl(k), [\hat{a}], [\hat{n}], \text{der}_x(\text{tail}^k(\hat{w})))}
\]

**Proof.** By induction on \( k \). \( \square \)

**Example 3.5** As an example, let us consider a context-free grammar for a parenthesis language. Let \( G := (\{1, r, a\}, \{S, S', L, R\}, S, T_G \cup N_G) \) where \( T_G := \{S \rightarrow a, L \rightarrow l, R \rightarrow r\} \) and \( N_G := \{S \rightarrow SS, S \rightarrow LS', S' \rightarrow SR\} \). It is straightforward to verify that \( G \) is in CNF.

Numbering the rules from 1 to 6 according to the order of appearance above, a leftmost derivation of the word \textbf{lara} is given by \( \tilde{\tau} := 4, 5, 2, 6, 1, 3, 1 \) (i.e., \( S \rightarrow SS \rightarrow LS'S \rightarrow 1S'S \rightarrow 1SRS \rightarrow 1aRS \rightarrow 1aS \rightarrow \text{lara} \)). Moreover,

\[
\begin{align*}
\text{dc}(\text{Nil, Nil, Nil, Nil, \text{der}_x(\tilde{\tau}))} &= \text{dc}(\text{Nil, Nil, Nil, Nil, } (N(S, S, S) \cdot \text{der}_x(\text{tail}^1(\tilde{\tau})))) \\
&\rightarrow \text{dc}(S, dl(1), \text{Nil, } (S \cdot (S \cdot \text{Nil})), (N(S, L, S') \cdot \text{der}_x(\text{tail}^2(\tilde{\tau})))) \\
&\rightarrow \text{dc}(S, (\text{OK}(S, S) \cdot dl(1)), \text{Nil, } (L \cdot (S' \cdot (S \cdot \text{Nil}))), \text{der}_x(\text{tail}^2(\tilde{\tau}))) \\
&\rightarrow \text{dc}(S, dl(2), \text{Nil, } (L \cdot (S' \cdot (S \cdot \text{Nil}))), (T(L, 1) \cdot \text{der}_x(\text{tail}^3(\tilde{\tau})))) \\
&\rightarrow \text{dc}(S, (\text{OK}(L, L) \cdot dl(2)), (1 \cdot \text{Nil, } (S' \cdot (S \cdot \text{Nil})), \text{der}_x(\text{tail}^3(\tilde{\tau}))) \\
&\rightarrow \cdots \rightarrow \text{dc}(S, dl(7), (a \cdot (r \cdot (a \cdot (1 \cdot \text{Nil}))))), \text{Nil, Nil})
\end{align*}
\]
This lemma can be generalized to show that $T_1(G) \vdash_E$ accurately models leftmost derivations of the grammar $G$.

**Proposition 3.6** If $w \in A_G^*$ then $w \in L(G)$ iff $T_1(G) \vdash_E \text{dc}(s_G, dl(1 + 2|w|), [w], \text{Nil}, \text{Nil})$.

**Proof.**

$\Rightarrow$ Assume that $w \in L(G)$. Then there exists a leftmost derivation $s_G \rightarrow^* w$ described by the tuple $\hat{r} := r_1r_2\ldots r_{|w|−1}$. By Lemma 3.4 we have

$$\text{dc}(\text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, \text{der}_{\text{Nil}}(\hat{r})) \rightarrow^{4|w|−3} \text{dc}(s_G, dl(1 + 2|w|), [w], \text{Nil}, \text{Nil}).$$

Clearly $T_1(G) \vdash_E \text{dc}(\text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, \text{der}_{\text{Nil}}(\hat{r}))$.

$\Leftarrow$ Assume that $T_1(G) \vdash_E U := \text{dc}(s_G, dl(1 + 2|w|), [w], \text{Nil}, \text{Nil})$. Then there exists $U' \equiv E U$ such that $T_1(G) \vdash_E U'$. Note that no rule creates a dc function symbol at the top level if there was not already one. Thus, since the frame does not contain any dc symbols, at the top level of $U'$ there must be a dc function application.

By inspection of the grammar rules, and since all letters of $w$ are restricted in the frame, no subterm of $[w]$ except for Nil is deducible. Thus, by inspection of the rewrite rules, the subterm $[w]$ of $U$ must have been generated by repeated application of rule (1) or (3).

Note that all terms in the frame $T_1(G)$ are in normal form. Since no rewrite rule introduces a T function symbol, and all terminal and nonterminal symbols of the grammar are restricted in the frame, $T(x, t)$ terms where $t \in A_G$ can only be deduced using a single application of the frame lookup rule, and thus $x \in X_G$.

Thus, whenever the third argument to the top-level dc function symbol grows (rules (1) and (3)), it is by using a terminal rule of $G$. Since the fourth argument only shrinks by application of rule (3), we can conclude that it always is a list of non-terminal symbols of the grammar.

By a similar argument, whenever the fourth argument to the top-level dc function symbol grows (rules (2) and (4)), it is by using a non-terminal rule of $G$. From this follows that there must exist $\hat{r}$ such that the last argument of the top-level dc function symbol of $U'$ is equal to $\text{der}_{\text{Nil}}(\hat{r})$.

By the restriction on the frame, the subterm $s_G$ of $U$ is not deducible. By inspection of the rules, it must have been generated using rule (1) or (2). Thus, $U' = \text{dc}(\text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, \text{der}_{\text{Nil}}(\hat{r}))$, so by Lemma 3.4 $s_G \rightarrow^* w$.

Our main technical lemma is a full characterization of the set of terms that can be derived by $T_1(G)$, in the case where $G$ is unambiguous. In this case, when starting from a primitively generated term that was in normal form before
applying the substitution, rewrite rules can only be applied as intended, i.e.,
extending partial derivations of the grammar $G$. To show this, we need to
find a deterministic rewrite strategy and prove it to be injective for this class
of initial terms ($\mathcal{L}_0$ below).

**Lemma 3.7** Let $G$ be fixed as above, and assume that $G$ is unambiguous. Let
$\mathcal{L}_0'$ be the set of (possibly open) terms in normal form that do not contain any
name in $A_G \cup X_G$. Let $\mathcal{D}_0(x) := \{ dc(\text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, x) \}$ and for $k > 0$
\[
\mathcal{D}_k(x) := \{ dc(n, dl(k), [\tilde{a}], [\tilde{n}], x) \mid \tilde{a} \in A_G \land \tilde{n} \in X_G^* \land \n \rightarrow^k_G \tilde{a} \tilde{n} \text{ using a leftmost partial derivation} \}
\]
Let the sets $\mathcal{L}_k'$ for $k > 0$ be the smallest sets satisfying this inference rule.
\[
\frac{U \in \mathcal{L}_k' \land V \in \mathcal{L}_0' \land W \in \mathcal{D}_l(V)}{\begin{cases}
U \in \mathcal{L}_k' & \text{if } V \in \mathcal{L}_0' \\
W \in \mathcal{D}_l(V) & \text{if } k \geq 0, l > 0
\end{cases}}
\]
Let $\mathcal{L}_k := \{ UT_1(G) \mid U \in \mathcal{L}_k' \land v(U) \subseteq \text{dom}(T_1(G)) \}$ and $\mathcal{L} := \cup_{k \in \mathbb{N}} \mathcal{L}_k$.
We then have:
(i) If $T_1(G) \vdash U$ with $U$ in normal form, then $U \in \mathcal{L}$.
(ii) If $U, U' \in \mathcal{L}_0$ and $U \equiv_E U'$, then $U = U'$.

**Proof.** Assume a well-ordering on contexts compatible with the partial well-
ordering induced by the depth of the hole, and let $\sim$ be rewriting where the
redex with the greatest context is always chosen. Note that this strategy is
deterministic and complete.

To compare terms in different stages of $\sim$-rewriting, we let $\equiv_C$ for a context
$C$ relate terms (or contexts) that coincide down to (exclusive) the depth of
the “hole” in $C$ and on the contents (or position) of the “hole”.

Note that the $\mathcal{L}_k$ are disjoint for different $k$.

Since equivalence is based on a convergent rewrite system and preserved
by arbitrary substitution of terms for variables, we have that $T_1(G) \vdash U'$ iff
there is $U \in \mathcal{L}_0$ such that $U' \equiv_E U$.

Let $P(i)$ be the following predicate:

(1) If $U_0 \in \mathcal{L}_0$ and $U_0 \sim^* U_i \in \mathcal{L}_i$ where $U_i \sim$ then one of (a) to (d) holds.
(a) $U_i \sim^{(1)} U_{i+1} \in \mathcal{L}_{i+1}$ by some $\mathcal{D}_0((T(y, t) \cdot u)) \ni U \rightarrow^h_{(1)} \in \mathcal{D}_1(u)$
where $T(y, t) \in \text{range}(T_1(G))$; or
(b) $U_i \sim^{(2)} U_{i+1} \in \mathcal{L}_{i+1}$ by some $\mathcal{D}_0(N(y, t_1, t_2)) \ni U \rightarrow^h_{(2)} \in \mathcal{D}_1(u)$
where $N(y, t_1, t_2) \in \text{range}(T_1(G))$; or
(c) $U_i \sim^{(3)} U_{i, 5} \sim^{(5)} U_{i+1} \in \mathcal{L}_{i+1}$ by some $\mathcal{D}_j((T(y, t) \cdot u)) \ni U \rightarrow^h_{(3)} \in \mathcal{D}_{j+1}(u)$
where $T(y, t) \in \text{range}(T_1(G))$; or
(d) $U_i \sim^{(4)} U_{i, 5} \sim^{(5)} U_{i+1} \in \mathcal{L}_{i+1}$ by some $\mathcal{D}_j((N(y, t_1, t_2) \cdot u)) \ni U \rightarrow^h_{(4)} \in \mathcal{D}_{j+1}(u)$
where $N(y, t_1, t_2) \in \text{range}(T_1(G))$; or
\[ \overline{U} \overset{h_{(4)}}{\rightarrow} \in D_{j+1}(u) \text{ where } \mathbb{N}(y, t_1, t_2) \in \text{range}(T_1(G)). \]

(II) Moreover, for each \( U'_i \in \mathcal{L}_0 \) such that \( U'_0 \approx^* U'_i \in \mathcal{L}_i \) and \( U_i \approx^* U_{i+1} \in \mathcal{L}_{i+1} \) as above, we have that \( U_{i+1} = U_{i+1} \) implies \( U'_0 = U_0 \).

The proof that \( \forall i \in \mathbb{N}. P(i) \) is omitted from this version of the paper due to space constraints. Given this, the statement of the lemma follows quickly.

(i) Assume that \( T_1(G) \vdash U \) with \( U \) in normal form. Since equality is based on a convergent rewrite system and preserved by arbitrary substitution of terms for variables, we have that \( T_1(G) \vdash U \) iff there is \( U' \in \mathcal{L}_0 \) such that \( U \equiv_E U' \). By \( \forall i \in \mathbb{N}. P(i) \), \( U' \downarrow \in \mathcal{L} \), so \( U \in \mathcal{L} \) by confluence.

(ii) Assume that \( U_1, U_2 \in \mathcal{L}_0 \) and \( U_1 \equiv_E U_2 \). By definition there is \( V \) such that \( V \not\approx \), and \( U_1 \approx^* V \) and \( U_2 \approx^* V \). By \( \forall i \in \mathbb{N}. P(i) \) there is \( k \) such that \( V \in \mathcal{L}_k \), and \( U_1 \approx^* V \) as by \( P \). Since the \( \mathcal{L}_k \) are disjoint for different \( k \), we also have \( U_2 \approx^* V \) as by \( P \). \( P(k - 1) \) then yields \( U_1 = U_2 \).

\( \square \)

Note that the statement of this lemma does not hold if \( G \) is ambiguous since in that case, two different elements in \( \mathcal{L}_0 \) can rewrite to the same term. For this reason, a similar characterization is hard to find in the general case. As an aside, in the setting of [2] we also have that two different terms (in the counterpart to our \( \mathcal{L}_0 \) can rewrite to the same term, making it more difficult to fully characterize the set of terms that can be generated.

3.5 Reduction

We now know in sufficient detail how the grammar \( G \) relates to \( T_1(G) \), and can proceed to the main result of this paper:

**Theorem 3.8** A grammar \( G \) in CNF is unambiguous iff \( T_1(G) \approx_{E} T_2(G) \).

**Proof.** As above, we write \( G := (A_G, X_G, s_G, T_G \cup N_G) \).

\( \Leftarrow \) We prove the contrapositive of the implication from right to left. Assume that \( G \) is ambiguous. Then there exists \( w \in A_G^* \) with two different leftmost derivations \( \overrightarrow{r}_1 \) and \( \overrightarrow{r}_2 \). Let \( \text{varOf}(k \rightarrow lm) := g_{0}(k, l, m) \), \( \text{varOf}(n \rightarrow a) := f_{0}(n, a) \) and \( t_{i} := \text{dc}(\text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, [\text{varOf}(\overrightarrow{r}_i)]) \) for \( i = 1, 2 \). By Lemma 3.4, we have that

\[ t_{1}T_{1}(G) \rightarrow^{*} \text{dc}(s_G, dl(1 + 2|w|), |w|, \text{Nil}, \text{Nil}) \]
\[ t_{2}T_{1}(G) \rightarrow^{*} \text{dc}(s_G, dl(1 + 2|w|), |w|, \text{Nil}, \text{Nil}) \]

so \( t_{1}T_{1}(G) = t_{2}T_{1}(G) \). By inspection, \( t_{1}T_{2}(G) \not\rightarrow^{*} \) and \( t_{2}T_{2}(G) \not\rightarrow^{*} \), so \( t_{1}T_{2}(G) \neq t_{2}T_{2}(G) \). Thus \( T_{1}(G) \) and \( T_{2}(G) \) are not statically equivalent.

\( \Rightarrow \) Assume that \( G \) is unambiguous. Let \( M \) and \( N \) be terms in normal form such that \( (n(M) \cup n(N)) \cap (bn(T_{1}(G)) \cup bn(T_{2}(G))) = \emptyset \) and \( (v(M) \cup v(N)) \subseteq \text{dom}(T_{2}(G)) \). Let \( M_{1} := MT_{1}(G), M_{2} := MT_{2}(G), N_{1} := NT_{1}(G), \) and \( N_{2} := NT_{2}(G) \).
• Since $T_2(G)$ is injective, $\text{range}(T_2(G))$ is in normal form, $\mathcal{N} \cap \text{range}(T_2(G)) = \emptyset$, $n(T_2(G)) \setminus \text{bn}(T_2(G)) = \emptyset$, and $\text{range}(T_2(G))$ does not contain any function symbols that appear in rewrite rules, we have that $M_2$ and $N_2$ are in normal form. Then, by the injectivity of $T_2(G)$, $M_2 \equiv_E N_2$ implies that $M = N$, so $M_1 \equiv_E N_1$.

• Assume instead that $M_2 \not\equiv_E N_2$. Then $M \not= N$, so by the injectivity of $T_1(G)$, we do not have $M_1 = N_1$. By Lemma 3.7, $M_1 \not\equiv_E N_1$.

\[\square\]

**Corollary 3.9** Since the ambiguity problem for context-free grammars is undecidable, $\approx_s$ is undecidable for $\equiv_E$.

### 4 Conclusions

In conclusion, we have showed that there exists a message language where the construction problem is decidable but the indistinguishability problem is not. Since $\vdash_E$ can be reduced to $\approx_s$ in the presence of encryption [1], this means that there is a price to pay for the more sophisticated indistinguishability-based definition of secrecy: Static equivalence is harder than knowledge!

Since the adversary can apply any combination of cryptographic operations in the course of a man-in-the-middle attack, the state-space of cryptographic protocols is infinitely branching on protocol input. Bounding the number of operations reduces the branching factor to finite but often intractable levels. The standard solution to this problem is to switch to symbolic semantics, where each input only gives raise to one (constrained) variable. Finding suitable classes of rewrite systems that yield decidable static equivalence and knowledge problems in this setting is an interesting possible topic for further work; the STA tool [7] already implements a decision procedure for adversary knowledge under any image-finite message algebra.

### Acknowledgments

Many thanks to Martín Abadi, who introduced me to this subject and encouraged me to complete a detailed proof of this result. Thanks also to Véronique Cortier, who commented on an early draft of this paper, and Uwe Nestmann, who pinpointed some problematic parts.

### References


We view processes as games between an input player and an output player. This view has important ramifications on the design of a process algebra. First, it naturally supports a definition of process composition that preserves deadlock-freedom. Second, it leads to a definition of process refinement that is based on alternating simulation.

More precisely, a process is well-formed if the input player has a strategy to avoid deadlocks. A well-formed process encodes a design-time assumption about the inputs that the environment will provide to the process, namely, the assumption that the process will be deployed only in contexts that avoid deadlocks (well-formedness guarantees that at least one such context exists). Two processes, then, are compatible if their product is well-formed; that is, if they can be put together in an environment that ensures that they satisfy each other’s input assumptions. Furthermore, the composition of two compatible processes must represent the weakest assumptions that are necessary about such an environment.

In alternating simulation, inputs of the specification process must be matchable by the implementation process, and outputs of the implementation must be matchable by the specification. Using such an alternating notion of refinement, we obtain the substitutivity of implementations: if two processes A and B are compatible, and A is implemented by A’, then A’ and B are still compatible, and their composition implements the composition of A and B. This property allows the refinement of a process independent of its context.

This talk is based on joint work with Luca de Alfaro.
A short visit to the STS hierarchy

Nathalie Bertrand\textsuperscript{1} and Philippe Schnoebelen\textsuperscript{2}

Lab. Spécification & Vérification, CNRS & ENS de Cachan,
61, av. Pdt. Wilson, 94235 Cachan Cedex France

Abstract
The hierarchy of Symbolic Transition Systems, introduced by Henzinger, Majumdar and Raskin, is an elegant classification tool for some families of infinite-state operational models that support some variants of a symbolic “backward closure” verification algorithm. It was first used and illustrated with families of hybrid systems.

In this paper we investigate whether the STS hierarchy can account for classical families of infinite-state systems outside of timed or hybrid systems.

Key words: Symbolic transition systems, well-structured transition systems, STS hierarchy.

1 Introduction
Verification of infinite-state systems is a very active field of research where one studies how the algorithmic techniques that underly the successful technology of model checking for finite-state systems can be extended to more expressive computational models [BCMS01]. Many different models have been studied, ranging from infinite-data models (like channel systems) to infinite-control models (like process algebras), including timed automata and hybrid systems. General undecidability results are worked around by discovering special restricted subclasses where decidability can be recovered for specific verification problems, and our understanding of the compromises between expressivity and tractability improves regularly.

There have been some attempts at bringing some order inside the existing plethora of scattered results. One way to do this is to discover conditions that (1) support some generic verification algorithms, and (2) can account

\textsuperscript{1} Email: bertrand@lsv.ens-cachan.fr
\textsuperscript{2} Email: phs@lsv.ens-cachan.fr
\textsuperscript{3} This research was supported by Persée, a project funded by the ACI Sécurité Informatique of the French Ministry for Scientific Research.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
for a rich enough variety of models. The well-structured transition systems (WSTS) of [ACJT00,FS01] are one such attempt, where the key notion is the existence of a well-quasi-order between configurations that is compatible with transitions. The WSTS idea applies widely, and instances exist in many classes of models [FS01].

The symbolic transition systems (STS) of [HMR05] are another attempt. Actually [HMR05] defines a hierarchy of five different levels: STS1 to STS5. All levels are defined in the same way: a system is STS$k$ iff its set of configurations yields a finite quotient modulo $\approx_k$, an equivalence relation that relates states with similar “behavior”. The equivalences from $\approx_1$ to $\approx_5$ are coarser and coarser, and systems in the STS$k$ class are also in STS$(k + 1)$. Additionally, five variants of a generic symbolic closure algorithm are given, one for each class, allowing verification of properties ranging from $\mu$-calculus model checking (for the class STS1) to reachability properties (for the class STS5).

While the STS idea is illuminating, its weak point is that it is not widely applicable. In [HMR05], all the given examples of classes STS1 to STS5 are some restricted families of hybrid systems. And no instance of STS4 systems is given. As a consequence it is not clear whether the classification has any impact beyond hybrid and timed systems.

**Our contribution.** We look at well-known families of models for which verification results exist, and that are not related to hybrid systems: Petri nets, pushdown systems, and channel systems. In particular, we consider several variants of lossy channel systems [AJ96,CFP96]. For these families, a natural question is whether they give rise to systems sitting inside some level of the STS hierarchy.

Here we are only considering semantical issues: we ask whether a given system model with a given set of observable properties gives rise to STS$k$ transition systems. We are not concerned with algorithmic issues and symbolic verification, even though the STS hierarchy meets its purpose when systems can be equipped with a working region algebra [HMR05].

A general outcome of our investigation is that only systems that are well-structured in the sense of [FS01] can fit in the STS hierarchy, at level STS5 (or, sometimes, STS4). Indeed, [HMR05] uses the name “well-structured systems” for its STS5. We argue that the close links between the two notions do not provide a perfect fit.

## 2 The STS hierarchy

Henzinger, Majumdar and Raskin introduced symbolic transition systems (STS) in [HMR05]. These are labeled transition systems equipped with a region algebra. However, since we do not consider algorithmic issues or symbolic verification in this paper, we will work with a simplified definition.
Definition 2.1 A labeled transition system (LTS) is a tuple $S = (S, \rightarrow, P)$ where $S$ is a (possibly infinite) set of states, $\rightarrow \subseteq S \times S$ is a transition relation, and $P \subseteq 2^S$ is a finite set of observable properties (or observables) that covers the state space: $S = \bigcup_{p \in P} p$.

An observation is a set of observables. The observation $P(\sigma)$ of a state $\sigma$ is \{p $\in P \mid \sigma \in p$\}.

Classically we write $\sigma \rightarrow \sigma'$ rather than $(\sigma, \sigma') \in \rightarrow$, and say that $\sigma'$ is one of the successors of $\sigma$. $\sigma$ is a deadlock state if it has no successors. A (finite) path in $S$ is a sequence of states $\sigma_1, \ldots, \sigma_n$ such that for all $i$, $\sigma_i \rightarrow \sigma_{i+1}$.

The STS hierarchy is based on well-known notions of simulations and traces (see, e.g., [Gla01]).

We recall the definitions for simulations. Let $S = (S, \rightarrow, P)$ be a LTS. A binary relation $R \subseteq S \times S$ is a simulation on $S$ if $R$ entails:

(i) $\forall p \in P$, $\sigma \in p \Leftrightarrow \tau \in p$,

(ii) $\forall \sigma \rightarrow \sigma'$, $\exists \tau \rightarrow \tau'$ s.t. $\sigma'R \tau'$.

$\sigma$ and $\tau$ are bisimilar – denoted by $\sigma \equiv^S \tau$ – if there is a symmetric simulation $R$ such that $\sigma R \tau$.

They are simulation-equivalent – denoted by $\sigma \equiv^S_2 \tau$ – if there are two simulations $R_1$ and $R_2$ such that $\sigma R_1 \tau$ and $\tau R_2 \sigma$.

It is well-known that bisimilarity and simulation-equivalence are equivalence relations.

We now recall the definitions for traces. Let $S = (S, \rightarrow, P)$ be a LTS. A trace from state $\sigma$ is the observation of a path originating from $\sigma$. Formally it is a sequence $P_1 \cdots P_n$ of observations such that there exists a path $\sigma_1 \cdots \sigma_n$ with $\sigma_1 = \sigma$ and $P_i = P(\sigma_i)$ for $i = 1, \ldots, n$. Any $p$ in $P_n$, the last observation along the trace, is called a target of the trace and we write $\sigma \rightarrow^n p$ when such a trace exists.

Two states $\sigma$ and $\tau$ are trace-equivalent – denoted by $\sigma \equiv^T_3 \tau$ – if every trace from $\sigma$ is a trace from $\tau$, and vice-versa.

They are distance-equivalent, – denoted by $\sigma \equiv^T_4 \tau$ – if for every trace from $\sigma$ with length $n$ and target $p$ there is a trace from $\tau$ of length $n$ and target $p$, and vice versa.

They are bounded-reach equivalent – denoted by $\sigma \equiv^S_5 \tau$ – if for every trace from $\sigma$ with length $n$ and target $p$ there is a trace from $\tau$ with length at most $n$ and target $p$, and vice versa.

Clearly, trace equivalence, distance equivalence, and bounded-reach equivalence are equivalence relations.

Definition 2.2 (The STS hierarchy) [HMR05].

A labeled transition system $S = (S, \rightarrow, P)$ belongs to the class $\text{STS}_k$ (for $1 \leq k \leq 5$) iff the relation $\equiv^T_k$ has finite index (i.e., induces a finite number
of equivalence classes in $S$).

Some immediate properties of STS classes are:

**Hierarchy:** If $S$ is in $\text{STS}_k$, it is in $\text{STS}(k+1)$.

**Finite systems:** If $S = \langle S, \rightarrow, P \rangle$ has finite $S$, then $S$ is in $\text{STS}_1$.

**Trivial observables:** If $S = \langle S, \rightarrow, P \rangle$ has $P = \{S\}$, then $S$ is in $\text{STS}_5$. If no state is $S$ a deadlock state, then $S$ is even in $\text{STS}_1$.

**Monotonicity w.r.t. observables:** If $S = \langle S, \rightarrow, P \rangle$ and $S' = \langle S, \rightarrow, P' \rangle$ only differ by $P' \subseteq P$ (i.e., $S$ has more observable properties than $S'$), and $S$ is in $\text{STS}_k$, then $S'$ too is in $\text{STS}_k$.

3 Well-structured transition systems and the STS hierarchy

In [HMR05] the class $\text{STS}_5$ is said to coincide with well-structured transition systems, a class of infinite-state transition systems supporting generic verification algorithms [Fin87, ACJT00, FS01]. This claim is supported by an alternative characterization of $\text{STS}_5$ systems, using well-quasi-orderings [HMR05, Theorem 5A]. However, the link with WSTS is not made more explicit.

In this section we show that WSTS are in $\text{STS}_5$ and consider the converse question: can any $\text{STS}_5$ transition system be turned into a WSTS by equipping it with a “compatible” well-quasi-ordering?

We recall that a *well-quasi-ordering* (wqo) is a reflexive and transitive relation $\leq$ (over some set $S$) such that for any infinite sequence $x_0, x_1, \cdots$ in $S$, there exists indexes $i < j$ with $x_i \leq x_j$. As a consequence, a wqo is well-founded and only admits finitely many minimal elements. (In the sequel we often write, as we just did, that a set has finitely many minimal elements when we really mean “finitely many distinct minimal elements up to the equivalence induced by the wqo”. This nuance is not required when the wqo is a partial ordering, i.e., is antisymmetric.)

**Definition 3.1 (Well-Structured Transition Systems) [FS01].**

A Well-Structured Transition System is a transition system $S = \langle S, \rightarrow, \leq \rangle$ equipped with a relation $\leq \subseteq S \times S$ which is a well-quasi-ordering (upward-) compatible with $\rightarrow$, i.e., for all $\sigma_1 \leq \tau_1$ and $\sigma_1 \rightarrow \sigma_2$ there exists $\tau_1 \rightarrow \tau_2$ with $\sigma_2 \leq \tau_2$.

This notion of compatibility is called *strong compatibility* in [FS01]. We say $S = \langle S, \rightarrow, \leq \rangle$ has reflexive compatibility if for all $\sigma_1 \leq \tau_1$ and $\sigma_1 \rightarrow \sigma_2$, there exists $\tau_2 \geq \sigma_2$ with either $\tau_2 = \tau_1$ or $\tau_1 \rightarrow \tau_2$ (which is denoted $\tau_1 \xrightarrow{0/1} \tau_2$ in the sequel). It is immediate that a given WSTS with strong compatibility has also reflexive compatibility.

Petri nets with $k$ places equipped with the partial order on $\mathbb{N}^k$ are an example of well-structured transition systems (with strong compatibility), see
section 4.2. Another example is the class of lossy channel systems using the subword ordering on channel contents, see section 4.3.

Definition 3.1 does not coincide with the definition used in Theorem 5A of [HMR05]. There, a well-structured system is a LTS that can be equipped with a wqo \( \leq \) on the states such that for all observable properties \( p \) and \( d \in \mathbb{N} \), the set of states that can reach \( p \) in less than \( d \) steps is upward-closed (a set \( S' \subseteq S \) is upward-closed if \( \sigma \in S' \) and \( \sigma \leq \tau \) entail \( \tau \in S' \)). This is shown to coincide with STS5 systems.

Let us consider a WSTS \( S = \langle S, \rightarrow, \leq \rangle \) and ask whether there is a set \( P \) of observables that turn \( S \) into an STS5 system. Of course, setting \( P = \{ S \} \) works, but this does not exploit the fact that \( S \) is well-structured. It turns out that any set \( P \) of upward-closed observables will work, and this holds even if \( S \) has reflexive compatibility.

Theorem 3.2 Let \( S = \langle S, \rightarrow, \leq \rangle \) be a WSTS with reflexive compatibility, and \( P \) be a finite set of upward-closed observables that covers \( S \). Then \( \langle S, \rightarrow, P \rangle \), denoted \( S_P \), is in STS5.

Since [HMR05] uses a different definition, our proof of Theorem 3.2 is not a copy of the proof of [HMR05, Theorem 5A]. Moreover it is also a more direct proof since we do not deal with algorithmic aspects of predecessors computation.

Proof. For an observable \( p \in P \), we let \( \text{Orig}(p) \) denote the set of pairs \((\sigma, n) \in S \times \mathbb{N} \) such that \( \sigma \) can reach \( p \) within \( n \) steps: \( \text{Orig}(p) \overset{\text{def}}{=} \{ (\sigma, n) \mid \sigma \overset{n}{\rightarrow} p \} \).

The canonical product wqo on \( S \times \mathbb{N} \) is defined by

\[(\sigma, n) \leq (\tau, m) \overset{\text{def}}{\iff} (\sigma \leq \tau \text{ and } n \leq m).\]

Let \( \text{MinOrig}(p) \) be the set of minimal elements in \( \text{Orig}(p) \): \( \text{MinOrig}(p) \) is finite since \( \subseteq \) is a wqo. We define \( \approx \subseteq S \times S \) with:

\[\sigma \approx \sigma' \overset{\text{def}}{\iff} \forall p \in P, \forall (\tau, m) \in \text{MinOrig}(p), \tau \leq \sigma \Leftrightarrow \tau \leq \sigma' \quad (1)\]

and claim it is a bounded-reach equivalence of finite index. That \( \approx \) has finite index comes from the finiteness of \( \text{MinOrig}(p) \). To see that it is a bounded-reach equivalence, assume \( \sigma \approx \sigma' \) and \( \sigma \overset{n}{\rightarrow} p \) for some \( n \in \mathbb{N} \) and \( p \in P \). Then \((\sigma, n) \in \text{Orig}(p) \) and there is some \((\tau, m) \in \text{MinOrig}(p) \) with \( \tau \leq \sigma \) and \( m \leq n \). From (1), we deduce \( \tau \leq \sigma' \).

Now pick a path \( \tau \rightarrow \tau_1 \rightarrow \cdots \rightarrow \tau_m \) with \( \tau_m \in p \). By induction on \( m \), and using the reflexive compatibility of \( S \), we show that there exist states \( \sigma'_1 \cdots \sigma'_m \) such that \( \sigma' \overset{0/1}{\rightarrow} \sigma'_1 \overset{0/1}{\rightarrow} \cdots \overset{0/1}{\rightarrow} \sigma'_m \) and \( \tau_i \leq \sigma'_i \) for \( i = 1, \ldots, m \). (Fig. 1 illustrates the proof.) Since \( p \) is upward-closed, \( \tau_m \in p \) implies \( \sigma'_m \in p \). Thus we have found a path witnessing \( \sigma' \overset{m'}{\rightarrow} p \) for some \( m' \leq m \leq n \). Hence \( \approx \) is a bounded-reach equivalence and \( S_P \) is in STS5. \( \Box \)
Note that, since reflexive compatibility is more general than strong compatibility, Theorem 3.2 shows a more general connection between STSₜₕ systems and WSTS’s.

A converse problem is to consider a LTS $S = \langle S, \rightarrow, P \rangle$ in STSₜₕ, and try to find a well-quasi-ordering $\leq$ on $S$ such that $\langle S, \rightarrow, \leq \rangle$ is a WSTS. Since Finkel and Schnoebelen showed that any (finitely branching) transition system could be equipped with a well-quasi-ordering $\leq$ to get a well-structured transition system [FS01], this can always be done.

However we would appreciate if the wqo that turns $S$ into a WSTS were “compatible” with $P$. For example it would be nice if the observables in $P$ become upward-closed sets w.r.t. the wqo since this is how $P$ is defined in the proof of Theorem 3.2. We do not know if such a wqo can be defined for all $S$ in STSₜₕ and must leave this question open for the moment.

**Remark 3.3** Given $S = \langle S, \rightarrow, P \rangle$ in STSₜₕ, [HMR05] proves that there exists a wqo on $S$ such that, for all $p \in P$ and $d \in \mathbb{N}$, the set $\{ \sigma \mid \sigma \leq^d p \}$ is upward-closed. Hence in particular every $p$ is upward-closed (pick $d = 0$). However, the wqo they define is in general not compatible with transitions and hence does not transform $S$ into a WSTS in the sense of [FS01].

## 4 Looking at classical infinite-state models

All the examples of STS systems in [HMR05] are hybrid systems: timed automata, two-dimensional rectangular automata, networks of timed automata, etc. Here we study classical infinite-state systems such as pushdown automata, Petri nets and lossy channel systems and consider whether they give rise to systems in one of the STSk classes.

### 4.1 Pushdown automata

Pushdown automata are systems with finite control and a pushdown stack.

Formally, a pushdown automaton $PD = \langle Q, \Gamma, \Delta \rangle$ is composed of a finite set of locations $Q$, a stack alphabet $\Gamma$ and a finite set of transition rules $\Delta$. The rules in $\Delta$ are of the form $l \xrightarrow{\text{pop } a} l'$ or $l \xrightarrow{\text{push } a} l'$ for $l, l'$ locations and $a \in \Gamma$. The operational semantics of $PD$ is given as a transition system $S_{PD}$ where
a state (or configuration) has the form $\sigma = \langle l, w \rangle$ with $l \in Q$ a location and $w \in \Gamma^*$ a stack contents. We omit the obvious definition for the transitions $\sigma \rightarrow \sigma'$ (see for example [BEF+00]).

Pushdown automata are a family of infinite-state systems for which verification is relatively easy in the sense that the iterated successor relation $\rightarrow^*$ is recognizable and can be described by a finite transducer effectively derivable from $PD$ [Cau92]. Of course there exist questions, e.g., trace equivalence, that are undecidable for these systems.

One obtains LTS’s from pushdown automata by equipping the transition systems they induce with some sets of observables.

Assume $PD = (Q, \Gamma, \Delta)$ is a pushdown automaton. The simplest and most natural observable properties are based on the locations: for each location $l \in Q$, let $p_l \overset{\text{def}}{=} \{ \langle l, w \rangle \mid w \in \Gamma^* \}$ and $P \overset{\text{def}}{=} \{ p_l \mid l \in Q \}$. We write $PD_l$ for the class of LTS’s obtained from pushdown automata with locations for observable properties.

Another option is to look at the stack and distinguish the states depending on the emptiness (or non-emptiness) of the stack. In this case there are two observable properties: $p_{\text{empty}} \overset{\text{def}}{=} \{ \langle l, \epsilon \rangle \mid l \in Q \}$ and $p_{\text{empty}}^R \overset{\text{def}}{=} S \setminus p_{\text{empty}}$. This gives rise to a class of LTS’s we denote $PD^s$. Finally, we write $PD_l^{s\,s}$ for the class of LTS’s one obtains by considering both types of observables.

**Theorem 4.1** The classes $PD_l$, $PD^s$ and $PD_l^{s\,s}$ give rise to LTS’s that are not in STS$_5$ in general.

**Proof.** We only prove the result for $PD_l$ since similar arguments work for $PD^s$ (and $PD_l^{s\,s}$ is dealt with using monotonicity of observables).

Consider $PD_0$, the pushdown automaton depicted in Fig. 2. Here from location $l_1$, one must pop all $a$’s before a move to location $l_2$ is allowed. Hence two states $\langle l_1, a^n b \rangle$ and $\langle l_1, a^m b \rangle$ are not bounded-reach equivalent unless $n = m$ (since from $\langle l_1, a^n b \rangle$ one can only reach target $l_2$ in $n + 1$ steps). Therefore bounded-reach equivalence does not have finite index, and the STS associated with $PD_0$ in $PD_l$ is not in STS$_5$. □

**4.2 Petri nets**

We do not recall here the definition of Petri nets (see [Esp98]). Let $PN$ be a Petri net with $k$ places. Its operational semantics is given by a transition
system where the states (or markings) are tuples from $\mathbb{N}^k$. Markings are partially ordered by the product ordering $(\mathbb{N}, \leq)^k$, or, formally

$$\langle x_1, \ldots, x_k \rangle \leq \langle y_1, \ldots, y_k \rangle \iff x_1 \leq y_1 \land \cdots \land x_k \leq y_k.$$ 

That $\leq$ is a wqo on $\mathbb{N}^k$ is known as Dickson’s Lemma [Dic13]. For observables we consider the set of all upward closures $\uparrow m \overset{\text{def}}{=} \{ m' \mid m \leq m' \}$ where $m$ is a marking in $\{0,1\}^k$. Hence an observation sees whether a place is marked or not, but does not see how many tokens are in a given place. Note that $P$ covers $S$ since $S = \uparrow \langle 0, \ldots, 0 \rangle$. We denote by $PN$ the class of LTS’s obtained from Petri nets with the observable properties defined above.

**Theorem 4.2** The class $PN$ gives rise to LTS’s that are in $STS_5$ but not in $STS_4$ in general.

**Proof.** Petri nets with $\leq$ are WSTS with strong compatibility (see [FS01] for example). A direct consequence of Theorem 3.2 is that they are STS5.

To see that they are not in STS4 in general, consider the Petri net with a single place and a single transition described in Fig. 3. Starting with $n$ tokens,

Fig. 3. A simple Petri net

the longest trace has exactly length $n$. Hence two different markings cannot be distance-equivalent and the distance equivalence does not have finite index on this system. $\square$

4.3 Lossy channel systems

Several different definitions for Lossy Channel Systems (LCS) can be found in the literature: see, e.g., [Fin94,CFP96,AJ96]. In this paper we will follow the approach of Abdulla and Jonsson [AJ96] which works smoothly and is more commonly cited. For this model we introduce two variants (allowing idling or not) and consider different cases for the observables.

**Definition 4.3** (LCS’s).

A lossy channel system $L = (Q, C, M, \Delta)$ is composed of a finite set of locations $Q$, a finite set of channels $C$, a finite alphabet $M$ and a finite set of transition rules $\Delta$. The rules have the form $q \xrightarrow{op} q'$ where $q$ and $q'$ are locations, and $op$ is an operation of the form:

- **send:** $clm$ writing message $m$ to channel $c$;
- **receive:** $c?m$ reading message $m$ from channel $c$;
- **internal action:** $\sqrt{\text{(no input/output operation)}}$. 

64
Operational semantics. The operational semantics of \( L = (Q, C, M, \Delta) \) is given by a transition system where a state (or a configuration) is a pair \( \langle q, w \rangle \) composed of a location \( q \) and a mapping \( w : C \to M^* \) describing the channels contents.

The effect of an operation \( op \) on a channel contents \( w \), denoted \( op(w) \), is the channel contents \( w' \) such that:

\[
\begin{align*}
\text{op} = c!m: & \text{ then } w'(c) = w(c).m \text{ and } w'(c') = w(c') \text{ for } c' \neq c; \\
\text{op} = c?m: & \text{ then } m.w'(c) = w(c) \text{ and } w'(c') = w(c') \text{ for } c' \neq c; \\
\text{op} = \sqrt{ } : & \text{ then } w'(c) = w(c) \text{ for all } c \in C.
\end{align*}
\]

We observe that \( op(w) \) is not defined when \( op = c?m \) and \( w(c) \) does not start with \( m \).

The perfect steps between configurations are all pairs \( \langle q, w \rangle \xrightarrow{\text{perf}} \langle q', w' \rangle \) such that there is a rule \( q \xrightarrow{op} q' \) in \( \Delta \) with \( w' = op(w) \).

Given two channels contents \( w \) and \( w' \), we write \( w \sqsubseteq w' \) if \( w \) can be obtained from \( w' \) by deleting messages (whatever their place in \( w' \)). This is extended to states as follows:

\[ \langle q, w \rangle \sqsubseteq \langle q', w' \rangle \stackrel{\text{def}}{\iff} q = q' \text{ and } w \sqsubseteq w'. \]

This is a wqo between states (by Higman’s Lemma [Hig52]).

What we are really interested in are the lossy steps, obtained from perfect steps by preceding and following them by arbitrary message losses (possibly none). Formally:

\[ \sigma \xrightarrow{\text{loss}} \tau \stackrel{\text{def}}{\iff} \exists \sigma', \exists \tau'. \sigma \sqsupseteq \sigma' \land \sigma' \xrightarrow{\text{perf}} \tau' \land \tau' \sqsupseteq \tau. \]

Idling. Starting with this definition, a natural variant is to enable idling in all configurations [BS03]. This assumption, which amounts to adding all pairs \( \sigma \xrightarrow{\sigma} \sigma \) on top of lossy steps, is a way of getting rid of deadlock states.

Observables. Natural observable properties for LCS’s are associated with the locations (exactly as with pushdown automata) and we let \( S_L \) ("L" for “locations”) denote the LTS associated in such a way with LCS \( L \).

One may prefer to observe the contents of the channels but this requires some care in order to obtain upward-closed observables. A simple solution is to only consider upward-closed and location-independent properties, i.e., properties \( p \) such that for all \( q, q' \in Q \) and all \( w \sqsubseteq w' \), \( \langle q, w \rangle \in p \) implies \( \langle q', w' \rangle \in p \). For every \( c \in C \), one such property is \( p_c \stackrel{\text{def}}{=} \{ \langle q, w \rangle \mid w(c) \neq \varepsilon \} \), that allows to observe (non-)emptiness of \( c \). One obtains a set of observables that covers \( S \) by letting \( P = \{ p_c \mid c \in C \} \cup \{ S \} \) and we write \( S_L^c \) ("c" for “channels”) for the resulting LTS. One can also mix the two approaches and observe both locations and channels, giving rise to LTS’s denoted \( S_L^{L,c} \).
Finally, we write $S^{i,l}_L$ (“i” for “idling”) and, respectively, $S^{c,i}_L$, or $S^{l,c,i}_L$, for the variant STS’s obtained by considering idling steps in the transition relation. For a nonempty $\alpha \subseteq \{i,c,l\}$, we write $LCS^\alpha$ for the class of all $S^\alpha_L$.

Observe that all variants of Lossy Channel Systems are WSTS’s with strong compatibility when equipped with $\subseteq$ as a wqo between states. Therefore they are in STS5 by Theorem 3.2.

In the next theorem we give tight results for all variants of lossy channel systems. When idling is allowed, LCS’s are in STS4, otherwise they are in STS5, whatever the observable properties.

**Theorem 4.4** • The class $LCS^l$ gives rise to LTS’s that are in STS5 but not in STS4 in general.

• The class $LCS^{i,l}$ gives rise to LTS’s that are in STS4 but not in STS3 in general.

**Proof.**

$LCS^l$: Let us give a counter-example to show that LCS’s with locations as observable properties are not in STS4 in general. Consider the simple LCS $L_1$ in the left of Fig. 4 with only one rule $l \xrightarrow{a} l$ (the name of the single channel is irrelevant). Starting from a configuration with $n$ a’s in the channel, a trace of length $n$ is possible but no longer trace is. As a consequence, trace equivalence does not have finite index and $S^l_L$ is not in STS4.

$LCS^{i,l}$: We first show that LCS’s with idling are in STS4. To see this we consider the $\approx$ relation defined in the proof of Theorem 3.2: in the case of LCS$^{i,l}$, the proof that $\approx$ is a bounded-reach equivalence can be continued and, using idling steps, one shows that it is a distance equivalence.

For showing that in general LCS$^{i,l}$ does not give rise to systems in STS3, we consider the LCS $L_2$ in the right of Fig. 4. Starting from $\langle l, a^n \rangle$ there is a trace $p_1, p_v, p_1, p_v, \cdots$ of length $n$ but no such trace longer than $n$ (that is, longer traces must use idling steps and cannot alternate between $p_1$ and $p_v$). Hence trace equivalence does not have finite index and $S^{i,l}_L$ is not in STS3.

□

**Theorem 4.5** • The class $LCS^c$ gives rise to LTS’s that are in STS5 but not in STS4 in general.

• The class $LCS^{i,c}$ gives rise to LTS’s that are in STS4 but not in STS3 in general.
• *The class* \( \text{LCS}^{l_c} \) *gives rise to* LTS’s *that are in* STS5 *but not in* STS4 *in general.*

• *The class* \( \text{LCS}^{l,l_c} \) *gives rise to* LTS’s *that are in* STS4 *but not in* STS3 *in general.*

The proofs for these assertions (both positive parts and counter-examples) are very similar to the proof of Theorem 4.4 and are left to the reader.

5 Concluding remarks

We considered the STS hierarchy as a potential classification tool for various families of infinite-state models of systems. Given a class SC of systems (with its operational semantics), it is natural to ask the question of where the systems in SC fit in the STS hierarchy. This is a semantical question that can be answered independently of whether some region algebra and the associated algorithmics are available for class SC.

All previously known examples for levels STS1 to STS5 were some classes of hybrid or timed systems [HMR05]. We considered classical families of systems outside the world of timed/hybrid systems (Petri nets, pushdown systems, lossy channel systems) that support verification techniques. It turns out that only well-structured systems can fit in the STS hierarchy, and at the weakest levels (i.e., STS4 and STS5). As a side effect, we clarified the links between level STS5 and the well-structured systems of [FS01].

We are left with the conclusion that, at the moment, the STS hierarchy does not appear very enlightening outside the world of timed/hybrid systems or well-structured systems.

Acknowledgments

We thanks the anonymous referees for their many useful remarks and suggestions.

References


A Petri net semantics of a simple process algebra for mobility

Raymond Devillers

Département d’Informatique, Université Libre de Bruxelles
CP212, B-1050 Bruxelles, Belgium

Hanna Klaudel

LaMI UMR 8042 CNRS, Université Evry-Val d’Essonne
523, Place des Terrasses 91000 Evry, France

Maciej Koutny

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract

In this paper, we propose a structural translation of terms from a simple variant of the Klaim process algebra into behaviourally equivalent finite high level Petri nets. This yields a formal semantics for mobility allowing one to deal directly with concurrency and causality. Moreover, the equivalence results provide a characterisation of the expressivity of the class of Petri nets we use.

Key words: Petri nets, process algebras, mobility, recursion.

1 Introduction

In this paper, we propose a structural and compositional translation of terms from ToyKlaim, which is a simple variant of the Klaim process algebra (designed, in particular, to represent mobility) into behaviourally equivalent finite high level Petri nets. This yields a formal semantics for mobility allowing one to deal directly with concurrency and causality. The new representation may also be used for automatically verifying liveness or reachability properties using model-checking.

This is a preliminary version. The final version will be published in Électronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
The paper is structured in the following way. We first introduce the syntax and semantics of ToyKlaim. After that we outline the net algebra used in our translation from ToyKlaim to Petri nets. We do not detail the basic concepts of high-level Petri nets; however, we provide intuition in order to keep the paper accessible to readers not very familiar with that field.

2 A simple process algebra

We start by defining ToyKlaim which is basically cKlaim [3] extended with a few features taken from OpenKlaim [3] and StockKlaim [5].

We assume that \( \mathcal{L} \) is a set of localities ranged over by \( l, l', l_1, \ldots \), and \( \mathcal{U} \) is a disjoint set of locality variables ranged over by \( u, v, w, u', v', w', u_1, v_1, w_1, \ldots \). Their union, together with the distinguished name \( \text{self} \), forms the set of names ranged over by \( n, n', n_0, n_1, \ldots \). In addition, \( \mathcal{A} = \{ A_1, \ldots, A_K \} \) is a finite set of process identifiers, each identifier \( A \in \mathcal{A} \) having an arity \( m_A \geq 0 \).

The syntax of ToyKlaim is given below:

\[
\begin{align*}
\text{networks} & \quad N := l :: P | l :: \{ l \} | N | N \\
\text{actions} & \quad a ::= \text{out}(n)@n \mid \text{newloc}(u) \mid \text{in}(T)@n \mid \text{eval}(A(n_1, \ldots, n_{m_A}))@n \\
\text{processes} & \quad P ::= \text{nil} | A(n_1, \ldots, n_{m_A}) | a \cdot P | P + P | P|P \\
\text{templates} & \quad T ::= n | !u
\end{align*}
\]

For each \( A \in \mathcal{A} \), there is exactly one definition of the form \( A(u_1, \ldots, u_{m_A}) \overset{df}{=} P_A \), where \( u_i \neq u_j \) for \( i \neq j \). It is assumed that such a definition is global. A structural equivalence on networks is the smallest congruence such that the following hold (below \( \{n_1/u_1, \ldots, n_{m_A}/u_{m_A}\} \) denotes substitution):

\[
\begin{align*}
\text{(COM)} \quad N_1 || N_2 & \equiv N_1 || N_2 \\
\text{(ASSOC)} \quad (N_1 || N_2) || N_3 & \equiv N_1 || (N_2 || N_3) \\
\text{(ABS)} \quad l :: P & \equiv l :: (P|\text{nil}) \\
\text{(CLONE)} \quad l :: (P_1|P_2) & \equiv l :: P_1 || l :: P_2 \\
\text{(PRINV)} \quad l :: A(n_1, \ldots, n_{m_A}) & \equiv l :: \{n_1/u_1, \ldots, n_{m_A}/u_{m_A}\}P_A
\end{align*}
\]

Networks. These are finite (flat, due to the rules COM and ASSOC) collections of computational nodes, where data and processes can be located. Each node consists of a locality \( l \) identifying it and a process or a datum (itself a locality in this simple presentation). There can be several nodes with the same locality part. Effectively, one may think of a network as a collection of uniquely named nodes, each node comprising its own data space and a possibly concurrent process which runs there (see the CLONE rule).

Actions. These are the basic (atomic) operations which can be executed by processes: (i) \text{out}(n')@n deposits a fresh copy of the locality denoted by \( n' \) inside the locality addressed by \( n \); (ii) \text{in}(T)@n retrieves an item matching the template \( T \) from the locality addressed by \( n \); (iii) \text{eval}(A(n_1, \ldots, n_{m_A}))@n
launches a new instance of the process identified by \( A \) in the locality \( n \); and
(iv) \texttt{newloc}(u) \ creates a fresh network node whose address is passed to the system using the locality variable \( u \). Note that the special meaning of the distinguished name \texttt{self} is that it refers to the node address at which an action is executed, and that instantiating a process in an arbitrary locality allows one to model mobility.

**Processes.** These are the only computational units, acting upon the data stored at various nodes, creating new locations and spawning new processes. The algebra of processes is built upon the (hanging) process \texttt{nil} and three composition operators: prefixing by an action \((a \cdot P)\); choice \((P_1 \mid P_2)\); and parallel composition \((P_1 || P_2)\).

**Well-formedness.** The action prefixes in (sub-)processes \texttt{newloc}(u) \( . \) \( P \) and \texttt{in}(!u)@n \( . \) \( P \) (i.e., location creation and input) \texttt{bind} the locality variable \( u \) within \( P \), and we then denote by \( fv(P) \) the free variables of \( P \) (and similarly for networks). For the process definition as above, we assume that \( fv(P_A) \subseteq \{ u_1, \ldots, u_{m_A} \} \), so that the free variables of \( P_A \) are in fact parameter bound.

As usual, processes are defined up to the alpha-conversion, meaning that bound variables may be coherently renamed avoiding potential clashes. Moreover, \( \{ u / u', \ldots \} P \) is obtained from \( P \) by replacing all free occurrences of \( u' \) by \( u \), etc, possibly after alpha-converting \( P \) in order to avoid clashes; e.g.,
\[
\{ v' / u / w' \} \texttt{in}(!u)@v'. \texttt{out}(u)@w. A(u') = \texttt{in}((u')@v. \texttt{out}(u')@w. A(u)).
\]

Given a network \( N \), one can apply alpha-conversion so as to obtain a well-formed system definition. By this we mean that no variable across the network and process definitions is both free and bound, and no variable ever generates more than one binding. Moreover, we assume that there are no free variables in the network (a variable \( u \) may only yield evolutions if it occurs under the scope of an \texttt{in}(!u) or \texttt{newloc}(u), or as a formal parameter\(^4\)).

**Semantics.** The operational semantics of networks and processes is detailed in tables 1 and 2. It is based on the structural equivalence defined above (see the \texttt{STRUCT} rule) and labelled transition rules augmented with an explicit information about known localities (the sets \( L, L' \subseteq \mathcal{L} \)):

\[
L \vdash N \xrightarrow{act} L' \vdash N'
\]

where \( act \) is the record of an execution of a prefix and \( loc(N) \subseteq L \subseteq L' \supseteq loc(N') \), and where \( loc(N) \) (resp. \( loc(N') \)) denote the set of localities present in \( N \) (resp. in \( N' \)). The \( act \) can be \texttt{o}(l,l'',l') or \texttt{i}(l,l'',l') or \texttt{c}(l,A(l_1, \ldots, l_{m_A}),l') or \texttt{n}(l,l'), where the initial symbol identifies the type of action, \( l \) the node where the action is executed, \( l' \) identifies the node where the action takes effect, and \( l'' \) is an optional parameter (argument of the action).

**Example.** We will use the \textit{Simple Mobile Robot} example (SMR). Its intended behavior can be understood as the following indefinite loop:

\[
\text{input locality } u; \text{ output the current locality; move to } u
\]

\(^4\) E.g., \( l :: \texttt{in}(!u)@u. \texttt{out}(u)@l'. A() \) is not well-formed (as \( u \) is both bound and free).
Devillers, Klaudel and Koutny

\[
\begin{align*}
\text{(OUT)} & \quad \text{if } n_0 = \text{self} \text{ then } l' = l \text{ else } l' = n_0, \\
\text{if } n_1 = \text{self} \text{ then } l'' = l \text{ else } l'' = n_1
\end{align*}
\]

\[
L \vdash l :: \text{out}(n_1)@n_0. P \xrightarrow{\alpha(l,l'',\bar{l})} L \vdash l :: P \parallel l' :: \langle l'' \rangle
\]

\[
\begin{align*}
\text{(PAR)} & \quad L \vdash N \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash N \parallel N'' \xrightarrow{\text{act}} L' \vdash N' \parallel N''
\end{align*}
\]

\[
\begin{align*}
\text{(SUM1)} & \quad L \vdash l :: P \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash l :: P + P' \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash l :: P + P \parallel l' :: \langle l'' \rangle \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash l :: P + P' \parallel l' :: \langle l'' \rangle \xrightarrow{\text{act}} L' \vdash N'
\end{align*}
\]

\[
\begin{align*}
\text{(SUM2)} & \quad L \vdash l :: P \parallel l' :: \langle l'' \rangle \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash l :: P + P' \parallel l' :: \langle l'' \rangle \xrightarrow{\text{act}} L' \vdash N' \\
L \vdash l :: P + P \parallel l' :: \langle l'' \rangle \xrightarrow{\text{act}} L' \vdash N'
\end{align*}
\]

\[
\begin{align*}
\text{(STRUCT)} & \quad N \equiv N_1 \\
\text{if } n_0 = \text{self} \text{ then } l' = l \text{ else } l' = n_0, \\
\text{if } n_1 = \text{self} \text{ then } l'' = l \text{ else } l'' = n_1
\end{align*}
\]

\[
L \vdash l :: \text{out}(n_1)@n_0. P \xrightarrow{\alpha(l,l'',\bar{l})} L \vdash l :: P \parallel l' :: \langle l'' \rangle
\]

Table 1

Operational semantics rules I.

Using TOYKLAIM’s syntax and 0, 1, 2 as localities, we model SMR as:

\[
0 :: \text{eval}(\text{SMR}(0))@1. \text{nil} \parallel 0 :: \langle 1 \rangle \parallel 1 :: \langle 2 \rangle \parallel 2 :: \langle 0 \rangle
\]

with the recursive process definition given by:

\[
\text{SMR}(u_1) \overset{\text{df}}{=} \text{in}(!u)@\text{self}. \text{out}(u_1)@\text{self}. \text{eval}(\text{SMR}(\text{self}))@u. \text{nil}.
\]

Then, the beginning of its execution sequence is:

\[
[1: 2: 0] \xrightarrow{\epsilon(0,\text{SMR}(0),1)} [1: 2: 0] \xrightarrow{i(1,2,1)} [1: 0: 0] \xrightarrow{\epsilon(1,0,1)} [1: 0: 0] \xrightarrow{\epsilon(1,\text{SMR}(1),2)} [1: 0: 0] \xrightarrow{i(2,0,2)} [1: 0: ] \xrightarrow{o(2,1,2)} [1: 0: 1] \xrightarrow{\epsilon(2,\text{SMR}(2),0)} [1: 0: 1],
\]

where the notation for the intermediate states means

\[
[\text{data in location } 0: \text{data in location } 1: \text{data in location } 2]
\]

and the process part of a state as well as the (unchanging) set of known localities are omitted.
if $n_0 = \text{self}$ then $l' = l$ else $l' = n_0$.
\[ (\text{Eval}) \]
\[ L \vdash l :: \text{eval}(A(\ldots n_i \ldots))@n_0 \cdot P \]
\[ \xrightarrow{c(l,A(\ldots l''_{i} \ldots),l')} L \vdash l :: P \parallel l' :: \{ \ldots l''_{i} / u_i \ldots \} P_A \]

if $n_i = \text{self}$ then $l''_i = l$ else $l''_i = n_i$ ($i = 1, \ldots, m_A$)

\[ (\text{InVar}) \]
\[ L \vdash l :: \text{in}(!u)@n_0 \cdot P \parallel l' :: (l'') \]
\[ \xrightarrow{u(l'\nu,l')} L \vdash l :: \{ l'' / u \} P \parallel l' :: \text{nil} \]

if $n_0 = \text{self}$ then $l' = l$ else $l' = n_0$.
\[ (\text{InLoc}) \]
\[ L \vdash l :: \text{in}(n_1)@n_0 \cdot P \parallel l' :: (l'') \]
\[ \xrightarrow{u(l'\nu,l')} L \vdash l :: P \parallel l' :: \text{nil} \]

if $n_1 = \text{self}$ then $l'' = l$ else $l'' = n_1$

\[ (\text{New}) \]
\[ L \vdash l :: \text{newloc}(u) \cdot P \]
\[ \xrightarrow{n(l', l')} L \cup \{ l' \} \vdash l :: \{ l' / u \} P \parallel l' :: \text{nil} \]

Table 2
Operational semantics rules II.

3 An algebra of nets

The development of our Petri net model, called \textit{klaim-nets}, has been inspired by the box algebra [1,2,7] and by the rp-net algebra used in [6] to translate \( \pi \)-calculus terms. In particular, we use coloured tokens and read-arcs (allowing any number of transitions to simultaneously check for the presence of a resource stored in a place [4]). Transitions in klaim-nets have the following labels: (i) \textbf{o} to denote outputting of data to data spaces; (ii) \textbf{i} to denote retrieving of data from data spaces; (iii) \textbf{n} to denote creating of new nodes; (iv) \textbf{c}_A to denote a creation of an instance of the process \( A \); and (v) \textbf{\tau} to denote silent transitions.

A key idea behind our translation is to view the system as consisting of a main program together with a number of procedure declarations. We represent the control structure of the main program and the procedures using disjoint unmarked nets, one for the main program and each of the procedure declarations. The program is executed once, while each procedure can be invoked several times (even concurrently), each such invocation being uniquely identified by a structured token which corresponds to the sequence of recursive
calls along the execution path leading to that invocation. With this in mind, we use the notion of a trail $\sigma$ to denote a finite (possibly empty) sequence of hierarchical transitions (i.e., those labelled by $c_A$ and $\tau$) of a klaim-net. The places of the nets which are responsible for control flow carry tokens which are trails. (The empty trail $\epsilon$, used for the main program, will be treated as the usual ‘black’ token.) Procedure invocation is then possible if each of the input places of a transition $t$ labelled with $c_A$ contains the same trail token $\sigma$. These tokens are then removed and a new token $\sigma t$ is inserted in each initial (entry) place of the net representing the process $P_A$ defining $A(\ldots)$, together with other tokens representing the actual parameters.

Places in klaim-nets are labelled in ways reflecting their intended role:

- **Control flow places**: These model control flow and are labelled by their status symbols (internal places by $i$, and interface places by $e$ and $x$, for entry and exit, respectively). The tokens they carry are simply the trails $\sigma$.

- **Locality places (or loc-places)**: These carry structured tokens representing localities known and used by the main program and procedure invocations. Each such token, called a trailed locality, is of the form $\sigma.l$ where $\sigma$ is a trail and $l$ is a locality in $L$. Intuitively, $\sigma$ identifies the invocation in which the token is available, while $l$ provides its value. In the diagrams, locality places have thick borders and are labelled by the localities, locality variables and distinguished names. For example, if a loc-place $u$ contains a trailed locality $\sigma.l$ then this means that some procedure invocation corresponding to trail $\sigma$ has the value of its locality variable $u$ set to $l$. Loc-places labelled by $\texttt{self}$ indicate where processes are executed (and when they are executed).

- **Data-place**: This is a distinguished place, labelled by $\texttt{DS}$, used to represent data stored at various locations. It will store structured tokens of the form $l :: \langle l' \rangle$, each such token corresponding to $l :: \langle l' \rangle$ in the process algebra.

- **Name-place**: This again is a distinguished place, labelled by $\texttt{FN}$ and containing all fresh (i.e., unused) localities which can then be picked up by the $n$-labelled transitions in order to incarnate new network nodes.

Directed arcs and read arcs are labelled by (one or many) annotations of the kind $\omega, \omega.z, \omega.t, \omega.t.x, x, x :: \langle y \rangle$, where $\omega, x, y, z$ are (high-level net) variables and $t$ is a (Petri net) transition name.

**Dynamics and bindings.** A transition may be executed under a binding mapping $b$ assigning values to the variables occurring around it. The transition then has to “check for the presence of” or “absorb” or “produce tokens” of the kind $b(\omega), b(\omega).b(z), b(\omega).t, b(\omega).t.b(x), b(x), b(x) :: \langle b(y) \rangle$, respectively.

For instance, in figure 2, transition $t_1$ may be executed under the binding $\omega \mapsto \epsilon, x \mapsto 1, x_1 \mapsto 0, z \mapsto 0$. Its firing consumes the $\epsilon$-token in the entry place, checks the presence of tokens $\epsilon.0, \epsilon.1$ and $\epsilon.0$ in the loc-places $\texttt{self}_1$,
1 and 0, respectively, and produces: $\epsilon$-token in the leftmost internal place, $t_1$-token in the upper internal place, $t_1.1$-token in the place $\text{self}_{SMR}$, and $t_1.0$-token in the place $u_1$.

4 Translating networks into klaim-nets

We assume that the following well-formed network is to be translated:

$$L \vdash (|_{i=1}^k l_i :: P_i) \mid (|_{j=1}^k l_j' :: (l_j''))$$

together with the necessary process identifier definitions; we also assume that $l_i \neq l_j$, for $i \neq j$. The translation for the basic actions of TOYKLAIM is given in figure 1:

**Input actions.** We have two different translations, depending on the form of the template used.

The first one, $K(\text{in}(n'@n))$, has to find in the data-place $\mathcal{DS}$ a matching pair corresponding to the current values of $n$ and $n'$, for the considered trail, and only then the action can be executed (the effect is that the matched pair disappears from the data-place). We do not assume that $n'$, $n$ and $\text{self}$ are distinct, and if some of them are the same, we collapse the corresponding loc-places, and gather together the annotations of read arcs. In the extreme case, for $K(\text{in}(\text{self}@\text{self}))$, the three loc-places are collapsed into a single one, labelled $\text{self}$, and the read-arc linking it with the only transition is annotated by $\omega.x$, $\omega.y$ and $\omega.z$.

The second translation, $K(\text{in}(u@n))$, where $u$ is a variable, works similarly, except that (one of) the value(s) corresponding in the data-place $\mathcal{DS}$ to the locality observed in $n$ for the considered trail is deposited in the loc-place $u$. We do not assume that $n$ and $\text{self}$ are distinct, and if they are the same, we collapse the corresponding loc-places, and gather together the annotations of read arcs.

When executed under a binding $\flat$, in both cases, the translation generates the visible label $i(\flat(z), \flat(y), \flat(x))$.

**Output actions.** In $K(\text{out}(n'@n))$, we do not assume that $n'$, $n$ and $\text{self}$ are distinct, and if some of them are the same, we collapse the corresponding loc-places, and gather together the annotations of read arcs. When executed under a binding $\flat$, the net generates the visible label $o(\flat(z), \flat(y), \flat(x))$.

**New location.** In $K(\text{newloc}(u))$, executing the transition under a binding $\flat$ generates the visible label $n(\flat(z), \flat(x))$.

**Process creation.** The translation $K(\text{eval}(A(n_1, \ldots, n_m)@n))$ assumes that the defining equation for $A$ is $A(u_1, \ldots, u_m) \equiv P_A$. We do not assume that $\text{self}$, $n$ and $n_1, \ldots, n_m$ are different, and if some of them are the same, we collapse the corresponding loc-places, and gather together the annotations of read arcs. On the other hand, $\text{self}, u_1, \ldots, u_m$, and $e_A$ are all distinct. In the extreme case, for $K(\text{eval}(A(\text{self}, \ldots, \text{self})@\text{self}))$, the $m_A+2$ loc-places
on the left are collapsed into a single one, labelled \texttt{self}, and the read-arc linking it with the only transition is annotated by \( \omega.z, \omega.x, \omega.x_1, \ldots, \omega.x_{m_A} \).

The idea of process creation is to spawn a new thread of activity within the sub-net corresponding to the process identifier definition. In particular, the parameters \( u_1, \ldots, u_{m_A} \) are assigned values corresponding to the current locality values of \( l_1, \ldots, l_{m_A} \) (notice that the newness of the thread is captured by extending the trails of tokens residing in the places on the left of \( t \) to make them distinct from any other tokens). Moreover, the auxiliary place labelled by \texttt{eA} will later be used to start-off the flow of control in the sub-net corresponding to the process identifier definition. When executed under a binding \( b \), the translation generates the visible label \( \texttt{c}(b(z), A(b(x_1), \ldots, b(x_{m_A})), b(x)) \).

\textbf{Nil process.} The translation, \( K(\texttt{nil}) \), without any transitions, has three places common to all the elementary translations.

\textbf{Composition operators.} For syntax driven translation we need operators corresponding to those in \textsc{ToyKlaim}, allowing one to construct klaim-nets
compositionally. These operators are *prefixing* \((R \cdot R')\), *choice* \((R + R')\) and *parallel composition* \((R|R')\). All three operators merge the non-control places (i.e., loc-places, DS and FN) with the same label; this corresponds to the asynchronous links used in [7]. For two operand nets, their transitions and control flow places are made disjoint before applying a composition operator in order to allow to properly handle the cases when, e.g., \(R = R'\).

In the choice composition, similar to the choice operation in the box algebra [1], the entry and exit places of \(R\) and \(R'\) are combined together through a cartesian product. This has the following effect: if we start from a situation where each entry place contains a copy of a common trail token \(\sigma\), then either \(R\) or \(R'\) can be executed.

The prefixing operator combines the exit place of the prefix \(R\) with the entry places of \(R'\) into internal places, and the effect is that the execution of \(R\) after reaching the terminal marking, where the only exit place is marked, is followed by that of \(R'\).

The parallel composition of \(R\) and \(R'\) puts them side by side allowing to execute both parts in parallel.

**Main translation.** It proceeds in the following three phases:

*Phase I.* For each \(i \leq h\), hence for each \(l_i :: P_i\) component, we first translate \(P_i\) compositionally (i.e., homomorphically) into \(\mathbb{K}(P_i)\). Notice that, while \(e\) denotes control places, \(e_A\) does not and is therefore subject to merging in the various net constructions. After that, we change the (unique) loc-place label \(\text{self}\) into \(\text{self}_i\). The result is denoted \(\mathbb{K}(l_i :: P_i)\).

*Phase II.* For each process definition \(A(u_1, \ldots, u_{m_A}) df = P_A\), we first translate compositionally \(P_A\) into \(\mathbb{K}(P_A)\). After that, we add loc-places labelled \(u_i\) for each \(i \leq m_A\), unless such a place is already present. Finally, we re-label the only \(\text{self}\) loc-place to \(\text{self}_A\), and if there is another loc-place \(\text{self}_A\) present (due to direct recursions), we merge them. The result is denoted \(\mathbb{K}(A)\).

*Phase III.* We take the parallel composition of the \(\mathbb{K}(A)\)'s and \(\mathbb{K}(l_i :: P_i)\)'s; then, if \(\mathbb{K}(A)\) has \(r\) entry places, we create \(r\) copies of the (unique) \(e_A\)-labelled place and identify them with the entry places of \(\mathbb{K}(A)\) (with the same input arcs as the \(e_A\)-labelled place and the same output arcs as the entry places of \(\mathbb{K}(A)\)). We then turn the entry and exit places of \(\mathbb{K}(A)\) into internal places.

After that we set the initial marking as follows: for \(i \leq h\), we insert \(\epsilon.l_i\) into the \(\text{self}_i\)-labelled place (\(\epsilon\) denoting the empty trail); each place labelled by a locality \(l \in L\) (we assume that \(\text{self}_i \notin L \cup U\) for each \(i\)) receives a single \(\epsilon.l\)-token; each entry place receives a single \(\epsilon\)-token; for each \(l \in L \setminus L\) we insert a single \(l\)-token into the FN-labelled place (if present); and for each \(j \leq k\), we insert a single \((l'j :: (l''j))\)-token into the DS-labelled place.

**Example.** In the case of the Simple Mobile Robot system, we get the net represented in figure 2, and it may easily be checked that the evolutions mentioned above for this system are also valid for the corresponding net.

For specifications without direct procedure calls (i.e., no usage of processes of the kind \(A(n_1, \ldots, n_{m_A})\); only actions \(\text{eval}(A(n_1, \ldots, n_{m_A})) @ n\) are allowed),

77
our translation yields a (sequential) Petri net semantics which coincides with the original ToyKlaim operational semantics.

**Proposition 4.1** For specifications without direct procedure calls, the labelled transition system generated by the above translation corresponds in an operationally strong way to the one generated by the rules in tables 1 and 2 for the original network.

The precise formulation and proof of this (and subsequent) results can be found in [8]. Note also that the above translation allows for direct concurrency and causality analyses.

## 5 Direct procedure calls

In order to translate direct procedure calls, we have a choice between at least two different approaches which, though up to a certain extent equivalent, still exhibit subtle differences.

The first one is to use translation similar to that for procedure invocations in figure 1, as shown in figure 3. The $t$ transition is a silent procedure call activating the body of the process and transmitting the parameters. It may happen, however, that $P_A$ cannot perform anything and the call occurs in a choice $\alpha + A(\ldots)$; hence it is necessary to roll back the call. This is realised by the ‘inverse’ transition, $t^{-1}$, acting as a silent uncall$^6$. Note that in ToyKlaim’s syntax directs calls are processes which, like $\text{nil}$, never terminate. As

---

$^6$ If a direct call is guarded (preceded by a prefix), then the uncall transition is not necessary and the net may be simplified.
a consequence, the call and uncall transitions do not need to be connected to
the exit place.

Figure 3. General translation \( \mathbb{K}(A(n_1, \ldots, n_{mA})) \) of a direct call (simplified by
using a single annotation for pairs of arcs attached to the same place).

**Proposition 5.1** The labelled transition system generated by the above trans-
lation corresponds in an operationally weak way to the one generated by the
rules in tables 1 and 2 for the original network.

The second approach consists in defining an equivalence on markings gen-
erated by the \( \tau \)-labelled transitions, which effectively are no longer needed.
This amounts to considering the equivalence generated by the following re-
lations: for any marking \( M \), for any process identifier \( A \), for any call-like
transition \( t \) of \( A \), and for any trail \( \sigma \),

\[
M \oplus \sigma.M^t_R \oplus M^t_\sigma \equiv M \oplus \sigma.M^t_R \oplus M^t_\sigma \oplus \sigma.t.M^t_O
\]

where \( \oplus \) denotes the (multiset) sum of markings and:

- \( M^t_R \) is any marking of the loc-places checked by \( t \),
- \( \sigma.M \) is the marking obtained by prefixing each token of \( M \) by “\( \sigma \).”,
- \( M^t_\sigma \) is the marking with a token \( \sigma \) in each input place of \( t \),
- \( M^A_\sigma \) is the marking with a token \( \sigma.t \) in each entry place of \( \mathbb{K}(A) \), and
- \( M^t_O \) is the marking of the output loc-places of \( t \) corresponding to \( M^t_R \).

**Proposition 5.2** The labelled transition system generated by the above trans-
lation (up to the \( \equiv \)-equivalence classes of markings) corresponds in an opera-
tionally strong way to the one generated by the rules in tables 1 and 2 for the
original network.

The above translation also allows for causality and concurrency analyses,
but it uses a non-standard Petri Net transition system.
6 Conclusions

We succeeded in translating ToyKLAIM specifications into the class of klaimnets, thus showing the expressive power of this class of Petri nets. Other KLAIM features, like tuple spaces, could easily be modelled as well.

This new semantics expresses in a natural way the concurrency and the causality features and allows to apply model checking techniques to verify their behavioural properties.

7 Acknowledgements

We would like to thank the anonymous referees for their helpful and constructive comments. This research was supported by the EC IST grant 511599 (RODIN).

References


Separation of synchronous and asynchronous communication via testing

Diletta Cacciagrano¹, Flavio Corradini²

Dipartimento di Matematica e Informatica
Università degli Studi di Camerino, Italy

Catuscia Palamidessi³

INRIA Futurs Saclay and LIX
École Polytechnique, France

Abstract

One of the early results about the asynchronous π-calculus which significantly contributed to its popularity is the capability of encoding the output prefix of the (choiceless) π-calculus in a natural and elegant way. Encodings of this kind were proposed by Honda and Tokoro, and (independently) by Boudol. We investigate whether the above encodings preserve De Nicola and Hennessy’s testing semantics. In this sense, it turns out that those encodings are incorrect, and, more interestingly, under some fairly general conditions no encoding of output prefix is able to preserve the must testing. This negative result is due to (a) the non atomicity of the sequences of steps which are necessary in the asynchronous π-calculus to mimic synchronous communication, and (b) testing semantics’s sensitivity to divergence.

Key words: Pi-Calculus, Asynchronous Pi-Calculus, Synchronous Communication, Asynchronous Communication, Testing Semantics.

1 Introduction

In recent years, the asynchronous communication paradigm has become more and more popular in the process calculi community. Reasons include the facts that it is easy to implement in a distributed system and that it naturally represents the basic communication mechanism of most Internet and Web applications.

1 Email: diletta.cacciagrano@unicam.it
2 Email: flavio.corradini@unicam.it
3 Email: catuscia@lix.polytechnique.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
One of the most popular asynchronous calculi is probably the asynchronous π-calculus \[ 12,2 \]. This is a proper subset of the π-calculus \[ 14 \], the main differences being the absence of the output prefix and of the choice operator. It is in particular the (absence of) output prefix which is important. In fact, this construct allows us to express directly that when a process performs an output it suspends until the partner performs the complementary input.

Naturally, the relation between the expressive power of the two calculi has attracted the attention of many researchers. Since the π-calculus contains the asynchronous π-calculus, it is obviously at least as expressive. As for the other direction, the third author has shown a separation result, based on the fact that the choice operator, in combination with synchronous communication, allows to solve certain problems of distributed agreement that cannot be solved with the asynchronous π-calculus \[ 17 \].

If we consider the choiceless π-calculus, however, things are less clear. The result in \[ 17 \] does not say anything concerning the presence/absence of output prefix alone. On the other hand, Honda and Tokoro \[ 12 \] and, independently, Boudol \[ 2 \] have proposed (different) encodings of the output prefix in the asynchronous π-calculus, thus justifying the claim that synchronous communication can be “implemented” via asynchronous communication. In both cases the idea is to represent a synchronization via a sequence of asynchronous steps executing a “mutual inclusion” protocol, which involves an exchange of acknowledgement messages.

These encodings significantly contributed to the popularity of the the asynchronous π-calculus, but only some correctness result was provided for them. (Boudol proved, for his encoding, only an adequacy result w.r.t. the Morris preorder, which is a fairly weak relation \[ 2 \].)

In this paper we show that the situation is more complicated than expected. In fact, we prove that the encodings by Honda, Tokoro and Boudol do not preserve De Nicola and Hennessy’s testing semantics \[ 9,10,1,7 \]. They resort on a set of processes to be tested and a set of test environments (processes with the ability to perform a special action to report success). A process \( P \) is embedded into a test environment \( o \) via parallel composition. Then, we say that \( P \) may \( o \) if there exists a successful computation between \( P \) and \( o \), \( P \) must \( o \) if every computation between \( P \) and \( o \) is successful and \( P \) fair \( o \) (proposed in \[ 4,16 \]) if each state of every computation between \( P \) and \( o \) leads to success after finitely many interactions. Each criterion allows the natural definition of a preorder relation over processes: For any process \( P \)
and \( Q, P \sqsubseteq^O_{\text{sat}} Q \) if and only if for each test \( o \in \mathcal{O} \), \( P \text{ sat } o \) implies \( Q \text{ sat } o \), where \( \text{sat} \) stands for \( \text{may}, \text{must} \) or \( \text{fair} \).

The first two authors started to investigate the properties of Boudol’s encoding w.r.t. various testing theories in [5]. They were particularly interested in establishing conditions on \( \llbracket \cdot \rrbracket \) and on \( \mathcal{R} \) so that (1) would hold. They realized however that the only-if part of (1) cannot hold for testing theories for the reason that the processes which are mappings of some \( \pi \)-calculus process are a strict subset of the asynchronous \( \pi \)-calculus. Thus testing a process \( \llbracket P \rrbracket \) with a test which is not the coding of any process in the \( \pi \)-calculus means testing \( \llbracket P \rrbracket \) over a set of tests which is “more powerful” than that of \( P \). In fact, a test which is not the result of an encoding in general does not follow the “rules of the game” w.r.t. the communication protocol, and can interact with it in strange ways.

In [5] the first two authors proposed a refinement of the testing theories by considering only encoded tests on the right hand side, and proved that Boudol’s encoding \( \llbracket \cdot \rrbracket \) satisfies the following:

(i) \( P \sqsubseteq^C_{\text{may}} Q \) iff \( \llbracket P \rrbracket \sqsubseteq^\llbracket \mathcal{O} \rrbracket \text{ may } \llbracket Q \rrbracket \);

(ii) \( P \sqsubseteq^C_{\text{fair}} Q \) iff \( \llbracket P \rrbracket \sqsubseteq^\llbracket \mathcal{O} \rrbracket \text{ fair } \llbracket Q \rrbracket \).

The key idea for proving (i) and (ii) is given by the following statement:

\[
(2) \quad P \text{ sat } o \iff \llbracket P \rrbracket \text{ sat } \llbracket o \rrbracket
\]

where \( \text{sat} \) is either \( \text{may} \) or \( \text{fair} \). For the \( \text{must} \) preorder (2) is not valid; indeed, we gave examples showing that \( P \text{ must } o \) does not imply \( \llbracket P \rrbracket \text{ must } \llbracket o \rrbracket \).

In this paper we further investigate on the must preorder. We focus on the condition that would imply the if and only if version of Property (3), that is:

\[
(3) \quad P \text{ must } o \iff \llbracket P \rrbracket \text{ must } \llbracket o \rrbracket
\]

We call this condition preservation of must testing.

We consider general encodings \( \llbracket \cdot \rrbracket \) of the (choiceless) \( \pi \)-calculus into the asynchronous \( \pi \)-calculus. We prove that, under some fairly general conditions (which include the cases of Boudol’s encoding, but also Honda and Tokoro’s one) namely compositionality w.r.t. prefixes and existence of a diverging encoded term, \( \llbracket \cdot \rrbracket \) cannot preserve must testing.

The source of the problem is that an (atomic) synchronous communication between a sender and a receiver can be simulated in the asynchronous world but there is no way to guarantee that sender and a receiver will be resumed (after communication) at the same time. More precisely, it could be the case that when the sender is ready to proceed the receiver is still engaged in some parts of the protocol, or vice versa. Therefore, there are unfair computations in which one partner is never resumed, and a test based on the interaction, after the communication, with that partner, would not succeed. This is of course not a problem in the synchronous world where the communication partners resume simultaneously.
The fact that our result holds for a general class of encodings points out, to our opinion, an inherent shortcoming of asynchronous communication with respect to the synchronous one.

The rest of the paper is organized as follows. Section 2 presents the π-calculus and the asynchronous π-calculus. Section 3 formally defines the must testing. Section 4 recalls some basic definitions about encodings. Section 5 shows our main result and Section 6 investigates some consequences of it. Due to space limitation most of the proofs have been omitted; they are reported in the full version of the paper [6].

2 The pi-calculus and the asynchronous pi-calculus

In this section we briefly recall the basic notions about the (choiceless) π-calculus and the asynchronous π-calculus.

2.1 The pi-calculus

Let \( \mathcal{N} \) (ranged over by \( x, y, z, \ldots \)) be a set of names. The set \( \mathcal{P}_s \) (ranged over by \( P, Q, R, \ldots \)) of processes is generated by the following grammar:

\[
\begin{align*}
P &::= 0 \mid x(y).P \mid \tau.P \mid \bar{x}y.P \mid P \mid P \mid (\nu x)P \mid ! P
\end{align*}
\]

The input prefix \( y(x).P \), and the restriction \((\nu x)P\), act as name binders for the name \( x \) in \( P \). The free names \( fn(P) \) and the bound names \( bn(P) \) of \( P \) are defined as usual. The set of names of \( P \) is defined as \( n(P) = fn(P) \cup bn(P) \). Whenever \( fn(P) = \emptyset \), \( P \) is said closed.

The operational semantics of processes is given via a labelled transition system, whose states are the process themselves. The labels (ranged over by \( \mu, \gamma, \ldots \)) correspond to prefixes, input \( x(y) \), output \( \bar{x}y \) and tau \( \tau \), and to the bounded output \( \bar{x}(y) \) (which models scope extrusion). If \( \mu = x(y) \) or \( \mu = \bar{x}y \) or \( \mu = \bar{x}(y) \) we define \( sub(\mu) = x \) and \( obj(\mu) = y \). The functions \( fn, bn \) and \( n \) are extended to cope with labels as follows:

\[
\begin{align*}
bn(x(y)) &= \{ y \} & bn(\bar{x}(y)) &= \{ y \} & bn(\bar{x}y) &= \emptyset & bn(\tau) &= \emptyset \\
fn(x(y)) &= \{ x \} & fn(\bar{x}(y)) &= \{ x \} & fn(\bar{x}y) &= \{ x, y \} & fn(\tau) &= \emptyset
\end{align*}
\]

The transition relation is given in Table 2.1. The symbol \( \equiv \) used in Rule Cong stands for the structural congruence. This is the smallest congruence over the set \( \mathcal{P}_s \) induced by the axioms in Table 2.

**Definition 2.1** (Weak transitions) Let \( P \) and \( Q \) be \( \mathcal{P}_s \) processes. Then:

- \( P \overset{\epsilon}{\rightarrow} Q \) if and only if there exist \( P_0, P_1, \ldots, P_n \in \mathcal{P}_s \), \( n \geq 0 \), such that

\[
P = P_0 \overset{\tau}{\rightarrow} P_1 \overset{\tau}{\rightarrow} \ldots \overset{\tau}{\rightarrow} P_n = Q;
\]
Table 1

Early operational semantics for \( P_s \) terms.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x(y) )</td>
<td>( P \xrightarrow{x} P' ) where ( x \neq y )</td>
</tr>
<tr>
<td>( (\nu y)P \xrightarrow{y} P' )</td>
<td>( y \notin n(\mu) )</td>
</tr>
<tr>
<td>( P \xrightarrow{\mu} P' )</td>
<td>( \nu y )</td>
</tr>
<tr>
<td>( P \xrightarrow{\nu} P' )</td>
<td>( P \xrightarrow{\nu} (\nu y)P' )</td>
</tr>
<tr>
<td>( P \xrightarrow{\nu} P' )</td>
<td>( (\nu y)P \xrightarrow{\nu} (\nu y)P' )</td>
</tr>
<tr>
<td>( P \xrightarrow{\nu} P' )</td>
<td>( P \xrightarrow{\nu} (\nu y)(P' \mid Q) )</td>
</tr>
</tbody>
</table>

Table 2

The structural congruence.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>( P \equiv Q ) if and only if ( Q ) can be obtained from ( P ) by alpha-renaming</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( (P_s / \equiv, \mid, 0) ) is a commutative monoid</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>( ((\nu x)P \mid Q) \equiv (\nu x)(P \mid Q) ), if ( x \notin fn(Q) )</td>
</tr>
<tr>
<td>( a_4 )</td>
<td>( (\nu x)P \equiv P ), if ( x \notin fn(P) )</td>
</tr>
<tr>
<td>( a_5 )</td>
<td>( (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P )</td>
</tr>
</tbody>
</table>

- \( P \xrightarrow{\mu} Q \) if and only if there exist \( P_1, P_2 \in P_s \) such that

\[
P \xrightarrow{\varepsilon} P_1 \xrightarrow{\mu} P_2 \xrightarrow{\varepsilon} Q .
\]

**Notation 2.1** Sometimes we write \( P \xrightarrow{\mu} (P \xrightarrow{\mu} ) \) to mean that there exists \( P' \) such that \( P \xrightarrow{\mu} P' \xrightarrow{\mu} P' \) and we write \( P \xrightarrow{\varepsilon} \xrightarrow{\mu} \) to mean that there are \( P' \) and \( Q \) such that \( P \xrightarrow{\varepsilon} P' \xrightarrow{\mu} Q \). We say that \( P \) diverges, notation \( P \uparrow \), if there exists an infinite sequence of \( \tau \) transitions from \( P \), i.e. \( P = P_0 \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots P_i \xrightarrow{\tau} P_{i+1} \xrightarrow{\tau} \ldots \) for some \( P_0, P_1, \ldots P_i, P_{i+1}, \ldots \). In the opposite case, i.e. if \( P \downarrow \), we say that \( P \) converges, notation \( P \downarrow \).
2.2 The asynchronous pi-calculus

The set $P_a$ of processes of the asynchronous $\pi$-calculus is generated by the following grammar:

$$P ::= 0 \mid x(y).P \mid \tau.P \mid \bar{x}y \mid P | P \mid (\nu x)P \mid !P$$

The operational semantics of $P_a$ is given by the rules in Table 2.1, with the rule Output/Tau replaced by the rules Output and Tau in Table 3. The axioms defining the structural congruence are the same as the ones in Table 2.

<table>
<thead>
<tr>
<th>Output $\bar{x}y$</th>
<th>$\bar{x}y$</th>
<th>Tau $\tau.P$</th>
<th>$\tau.P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$0.$</td>
<td>$\tau.$</td>
<td>$\tau.$</td>
</tr>
</tbody>
</table>

Table 3
The rules for Output and Tau in $P_a$.

The definitions and notation given in the synchronous setting are assumed in the asynchronous one as well. Note that the asynchronous $\pi$-calculus is a sub-set of the $\pi$-calculus. Indeed, the output-action process $\bar{x}y$ can be thought as the special case of output prefix $\bar{x}y.0$.

3 Must preorder

In this section we briefly summarize the basic definitions behind the testing machinery for the $\pi$-calculi. In the following, $P$ will denote either $P_s$ or $P_a$.

**Definition 3.1 (Observers)**
- Let $\mathcal{N}' = \mathcal{N} \cup \{\omega\}$ be the set of names. By convention we let $fn(\omega) = \{\omega\}$, $bn(\omega) = \emptyset$ and $sub(\omega) = \omega$. The action $\omega$ is used to report success.
- The set $\mathcal{O}$ (ranged over by $o, o', o'', \ldots$) of observers is defined like $P$, where the grammar is extended with the production $P ::= \omega.P$.
- The operational semantics of $P$ is extended to $\mathcal{O}$ by adding $\omega.o \xrightarrow{\omega} o.$.

In the following we will use $\langle P \rangle$ to denote some restricted version of $P$, i.e. any process of the form $(\nu x_1)(\nu x_2)\ldots(\nu x_n)P$, for some $x_1, \ldots, x_n \in fn(P)$.

**Definition 3.2 (Maximal computations)** Given $P \in P$ and $o \in \mathcal{O}$, a maximal computation from $P | o$ is either an infinite sequence

$$P | o = \langle P_0 | o_0 \rangle \xrightarrow{\tau} \langle P_1 | o_1 \rangle \xrightarrow{\tau} \langle P_2 | o_2 \rangle \xrightarrow{\tau} \ldots$$

or a finite sequence

$$P | o = \langle P_0 | o_0 \rangle \xrightarrow{\tau} \langle P_1 | o_1 \rangle \xrightarrow{\tau} \ldots \xrightarrow{\tau} \langle P_n | o_n \rangle \xrightarrow{\tau} \ldots$$

We are now ready to present the definition of must testing preorder.
Definition 3.3 (Must relation) Given a process $P \in \mathcal{P}$ and an observer $o \in \mathcal{O}$, define $P \text{ must } o$ if and only if for every maximal computation $P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \ldots \langle P_n \mid o_n \rangle \xrightarrow{\tau} \ldots$ there exists $i \geq 0$ such that $\langle P_i \mid o_i \rangle \xrightarrow{\omega} \ldots$

Definition 3.4 (Must preorder) Given two processes $P, Q \in \mathcal{P}$ and a set of observers $\mathcal{O}$, define: $P \sqsubseteq_{\text{must}}^\mathcal{O} Q$ if and only if for every $o \in \mathcal{O}$, $P \text{ must } o$ implies $Q \text{ must } o$.

Note that, by Definition 3.3 and by the operational rule for a test $\omega . o$, for every $P \in \mathcal{P}$ and $o \in \mathcal{O}$ we have $P \text{ must } \omega . o$.

4 Encodings of the pi-calculus into the asynchronous pi-calculus

In this section we recall some notions about encodings. In general an encoding is simply a syntactic transformation between languages. We will focus on encodings of the $\pi$-calculus into the asynchronous $\pi$-calculus, and we will use the notation $[\cdot] : \mathcal{P}_s \rightarrow \mathcal{P}_a$ to represent one such transformation. In general a “good” encoding satisfies some additional properties, but there is no agreement on a general notion of “good” encoding. Perhaps indeed there should not be a unique notion, but several, depending on the purpose. Anyway, in this paper we focus on the most common requirements, which are the homomorphism or (weaker) the compositionality w.r.t. certain operators, and the correctness w.r.t. a given semantics.

Definition 4.1 (Homomorphism w.r.t. an operator) Let $[\cdot]$ be an encoding. Let $op$ be an $n$-ary operator of both $\mathcal{P}_s$ and $\mathcal{P}_a$. We say that $[\cdot]$ is homomorphic w.r.t. $op$ if and only if

$$[op(P_1, \ldots, P_n)] = op([P_1], \ldots, [P_n]).$$

Usually the homomorphism is required only for certain operators (typically, in distributed languages, it is required for the parallel construct) while for the others we simply require a compositional translation. To describe this notion we use contexts $C[\cdot]$, which are terms in $\mathcal{P}_a$ with one or more “holes” $[\cdot]$. Given $P_1, \ldots, P_n \in \mathcal{P}_a$ and a context $C[\cdot]$ with $n$ holes, $C[P_1, \ldots, P_n]$ denotes the term in $\mathcal{P}_a$ obtained by replacing the occurrences of $[\cdot]$ by $P_1, \ldots, P_n$ respectively.

Definition 4.2 (Compositionality w.r.t. an operator) Let $op$ be an $n$-ary operator of $\mathcal{P}_s$. We say that an encoding $[\cdot]$ is compositional w.r.t. $op$ if and only if there exists a context $C_{op}[\cdot]$ in $\mathcal{P}_a$ such that

$$[op(P_1, \ldots, P_n)] = C_{op}([[P_1], \ldots, [P_n]].$$

Concerning semantic correctness, we consider preservation of must testing:
Definition 4.3 (Soundness, completeness and must-preserving) Let \( \llbracket \cdot \rrbracket \) be an encoding from \( \mathcal{P}_s \) to \( \mathcal{P}_a \). We say that \( \llbracket \cdot \rrbracket \) is:
- **sound w.r.t. must** iff \( \forall P \in \mathcal{P}_s, \forall o \in O, \llbracket P \rrbracket \text{ must } \llbracket o \rrbracket \) implies \( P \text{ must } o \);
- **complete w.r.t. must** iff \( \forall P \in \mathcal{P}_s, \forall o \in O, P \text{ must } o \) implies \( \llbracket P \rrbracket \text{ must } \llbracket o \rrbracket \);
- **must-preserving** iff \( \llbracket \cdot \rrbracket \) is sound and complete w.r.t. must.

5 Non existence of a must-preserving encoding

This section is the core of the paper. We prove a general negative result for a large class of encodings of the \( \pi \)-calculus into the asynchronous \( \pi \)-calculus, which includes the ones of Boudol, but also of Honda and Tokoro. More specifically, our result, Theorem 5.3, states that any encoding \( \llbracket \cdot \rrbracket \) that is compositional w.r.t. input and output prefixes and for which there exists \( P \in \mathcal{P}_s \) such that \( \llbracket P \rrbracket \uparrow \), cannot be must-preserving. We start with two key (nice) properties of our encodings:

Lemma 5.1 Let \( \llbracket \cdot \rrbracket \) be an encoding must-preserving. If there exists \( P \in \mathcal{P}_s \) such that \( \llbracket P \rrbracket \uparrow \), then \( \llbracket \omega.0 \rrbracket \xrightarrow{\omega}. \)

Proof. Let \( P \in \mathcal{P}_s \) such that \( \llbracket P \rrbracket \uparrow \). Since \( P \text{ must } \omega.0 \) and the encoding \( \llbracket \cdot \rrbracket \) is must-preserving, then \( \llbracket P \rrbracket \text{ must } \llbracket \omega.0 \rrbracket \). Since \( \llbracket P \rrbracket \uparrow \), we have \( \llbracket \omega.0 \rrbracket \xrightarrow{\omega}. \)

Lemma 5.2 Let \( \llbracket \cdot \rrbracket \) be an encoding that satisfies:
1. compositionality w.r.t. input and output prefixes,
2. must-preservation,
3. \( \exists P \in \mathcal{P}_s \) such that \( \llbracket P \rrbracket \uparrow \).

Then each \( \llbracket \cdot \rrbracket \) in \( C_{x(y)}[\cdot] \) and \( C_{\bar{x}y}[\cdot] \) is guarded, i.e. it occurs after an input or \( \tau \) prefix.

We are now ready to state our main result. It is worth of noting that Theorem 5.3 does not apply to any hypothesis of homomorphism w.r.t. parallel operator.

Theorem 5.3 Let \( \llbracket \cdot \rrbracket \) be an encoding that satisfies:
1. compositionality w.r.t. input and output prefixes,
2. \( \exists P \in \mathcal{P}_s \) such that \( \llbracket P \rrbracket \uparrow \).

Then \( \llbracket \cdot \rrbracket \) is not must-preserving.

6 Other impossibility results

The existence of a divergent process in the target language of the encodings, which is one of the hypotheses of Theorem 5.3, can be guaranteed by suitable assumptions on the encoding itself and the preservation of the must
testing. This section investigates conditions as weak as possible on the encodings which, under the hypothesis of must-preservation, ensure the existence of such divergent terms and therefore, together with the compositionality w.r.t. the input and output prefixes, imply the non existence of a must-preserving encoding. The proofs can be found in the appendix.

The following theorem states that the existence of a divergent and a convergent term in the source language whose encodings do not interact with the context is a sufficient condition.

**Theorem 6.1** Let \([\cdot]\) be an encoding that satisfies:

1. compositionality w.r.t. input and output prefixes,
2. \(\exists Q \in P_s\) such that \(Q \uparrow\) and \(fn([Q]) = \emptyset\),
3. \(\exists R \in P_s\) such that \(R \downarrow\) and \(fn([R]) = \emptyset\).

Then \([\cdot]\) is not must-preserving.

The following theorem states that for the impossibility result it is also sufficient to have homomorphism w.r.t. \(\tau\)-prefixes. Note that we don’t require homomorphism w.r.t. the \(!\) operator. The homomorphism w.r.t. both \(\tau\)-prefixes and \(!\) would imply immediately the existence of a divergent process in the target language.

**Theorem 6.2** Let \([\cdot]\) be an encoding that satisfies:

1. compositionality w.r.t. input, output, and \(\tau\)-prefixes,
2. \(fn([\tau.\cdot]) = fn([0]) = \emptyset\).

Then \([\cdot]\) is not must-preserving.

The next result is, to our opinion, the most surprising. It states that a compositional encoding cannot be must-preserving if the encodings of \(\tau.\cdot\) and 0 do not interact with the environment.

**Theorem 6.3** Let \([\cdot]\) be an encoding that satisfies:

1. compositionality w.r.t. input, output, and \(\tau\) prefixes,
2. \(fn([\tau.\cdot]) = fn([0]) = \emptyset\).

Then \([\cdot]\) is not must-preserving.

### 7 Related and future work

In this section we provide a brief introduction of those papers that are close to our study and discuss possible future work.

An interesting paper that aims at studying the relationships between synchronous and asynchronous communication is certainly [18]. The authors consider the polyadic \(\pi\)-calculus and the asynchronous version of the monadic \(\pi\)-calculus as base languages, Boudol’s encoding from the former to the latter
language and barbed congruence as the semantics to be preserved by the mapping. Some of the ideas exploited in [5] are also present in [18]. By relying on a type system for processes, the authors restrict the asynchronous tests (contexts in their setting) to those which are mapping of synchronous tests. In such a way they get a full abstraction result for barbed congruence. However, we know that such a condition is necessary but not sufficient to get full abstraction for every semantic theory. In [5] it has been proven a similar result for may and fair testing but not for must. The proof technique in [5] still work when their barbed congruence is considered (and, actually, also when Morris’ testing preorder is taken into account).

Another very interesting paper is [17]. Also this paper is concerned with the attempt of solving or, at least, clarify how these two communication mechanisms (synchronous and asynchronous) can be implemented one into the other. The π-calculus and the asynchronous π-calculus are the considered languages together with their own transitional semantics. It has been shown that it is not possible to map the π-calculus in the asynchronous π-calculus for every possible “uniform” encoding (it is compositional w.r.t. parallel composition and “behaves well” w.r.t. renamings) and for every “reasonable” semantics (it distinguishes two processes \( P \) and \( Q \) whenever in some computation of \( P \) the actions on certain intended channels are different from those of any computation of \( Q \) which one wants to preserve. We cannot exploit Palamidessi’s result to justify our negative result with the must preorder because her proof technique strongly relies on the presence of mixed choices (input and output prefixes in alternative composition) in the π-calculus while we do not have explicit non deterministic compositions in our source language.

Another work with similar issues of ours is [11]. Honda concentrates on the ν-calculus, a subset of the asynchronous π-calculus, where only input guarded terms can be in the scope of the bang operator (notice that this is not a real restriction, since this kind of replicator is as expressive as the full bang operator [15]). He considers two operational semantics: the first one, called “synchronous”, relies on the standard input-prefix rule, while the latter one, called “asynchronous”, relies on a new input-prefix rule, which allows any process to perform an input action, also when not syntactically specified. Hence he proves soundness and completeness, w.r.t. a weak bisimulation, of a mapping within the same language, when equipped by the two different semantics (the synchronous and asynchronous ones). It is worth of noting that his synchronous semantics is our asynchronous semantics, a part from the different syntax, of course; his asynchronous semantics is a new one.

In [12] Honda and Tokoro concentrate on the π-calculus without sum and bounded output and provide terms of this algebra with two transitional semantics: one describes processes with a synchronous communication and the other as the asynchronous semantics in [11]. The former transitional semantics is standard. Then, various observational semantics based on trace, failure and bisimulation, are defined on the top of the considered transitional semantics.
The relationships between the synchronous bisimulation and its asynchronous counterpart are investigated. The main result of this study shows that the latter relation is strictly weaker than the former one. Similar results hold for trace and failure-based semantics. To obtain full abstraction results, they introduce the notion of I completion. This is a mapping from a term interpreted asynchronously into a term interpreted synchronously. Any target term is able to mimic all the asynchronous transitions via synchronous transitions. They do not mention, however, to full abstraction results for the opposite mapping; i.e., how to implement synchronous communication in terms of the asynchronous one which, instead, is the main purpose of the current work.

Another interesting work is [8]. In this paper the authors study the expressive power of variants of Klaim (a language with programming constructs for global computing) and for assessing the expressiveness of the various languages they exploit the idea of “mapped tests” in the target language.

As a future work, we plan to investigate the possibility of positive results under some “fair” scheduling assumption. The idea of trying the fairness assumption comes from the observation that the negative result for the must testing is essentially due to divergent components and unfair scheduling strategies. Of course, if we impose fairness on all parts of the computations, then we have to impose it on both the source and the target languages in order for the encoding to preserve the semantics. This in a sense weakens the intended result. To avoid this problem, we plan to impose fairness only on asynchronous computations and, more specifically, only on those actions which belong to simulations. We would like to check whether our results scale to versions of π-calculus with mixed choice by exploiting the results and some non-uniform encodings [Nes97].

It is also interesting to proceed in a similar way, comparing other communication mechanisms (for example, broadcasting vs peer to peer), and to import the ideas developed for testing in a bisimulation scenario (for this latter line of research it seems that [12] can provide a valid support to this investigation).

References


Open bisimulation, revisited*

Sébastien Briais** Uwe Nestmann

School of Computer and Communication Sciences, EPFL, Switzerland

Abstract

In the context of the π-calculus, open bisimulation is prominent and popular due to its congruence properties and its easy implementability. Motivated by the attempt to generalise it to the spi-calculus, we offer a new, more refined definition and show in how far it coincides with the original one.

1 Introduction

Open bisimulation, as introduced by Sangiorgi [San96] is an attractive candidate notion of bisimulation for the π-calculus for a number of different reasons. First, it constitutes a reasonably full congruence, i.e., it is preserved by all operators including input prefix. Second, it allows for simple axiomatizations (for finite terms). Third, it is rather straightforward to build tools that check open bisimilarity (see the MWB [Vic94] or the ABC [Bri03]).

The current paper arose from our attempt to “smoothly” generalise the definition of open bisimulation from the π-calculus to the spi-calculus, an extension of the former by cryptographic primitives to be used in the description of security protocols. It turns out that this is not easily doable, for reasons that we try to explain in the remainder of this Introduction. Driven by the quest for a meaningful definition of open-style bisimulation for the spi-calculus, we came up with a proposal that we then observed can also be meaningfully projected down to the case of the π-calculus. The resulting notion and its comparison to the original definition is the main contribution of this paper.

The flurry of notions of bisimulation for the π-calculus

1

1

, ranging from ground over early and late to open, results mainly from the different possible treatments of simulated symbolic input transitions, e.g., when

\[ P \xrightarrow{a(x)} P' \quad \text{by} \quad Q \xrightarrow{a(x)} Q'. \]

A long version is found at http://lamp.epfl.ch/~sbriaiss/.

**Supported by the Swiss National Science Foundation, grant No. 21-65180.1

1 Luckily, all of these notions collapse in certain sub-calculi, for example like the asynchronous π-calculus, that are still expressive enough for most practical purposes.

This is a preliminary version. The final version will be published in

Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
The problem is that after the execution of a symbolic input on channel $a$, the input variable $x$ becomes free in the resulting continuation processes $P'$ and $Q'$. Considering the possible instantiations of this input variable by received messages can be done either not at all (as in ground), or (as in early) before the simulating transition is chosen, or (as in late) right afterwards—or (as in open) considering all possible substitutions (not only affecting the just freed input variable) even before starting any bisimulation game. The latter case can also be seen as “very late” or “lazy” since all possible instantiations of the input variable will be checked the next time we try to continue with the bisimulation game with $P'$ and $Q'$.

For clarity of the following explanations, in an application $P\{M/x\}$ of a substitution, where $M$ replaces all (free) occurrences of $x$ in $P$, let us use the terms substitution subject for $x$ and substitution object for $M$.

What do we actually mean by all possible instantiations? By definition, only free names can ever be affected as substitution subjects. In a process, there are three kinds of free name. A free name may be free because:

(i) either it was already initially free,
(ii) or it has become free after having done an input (or been substituted),
(iii) or it has become free after having been created as a local name, and afterwards output to some observing process.

We argue$^3$ that names of the latter kind are constant, i.e., they should not be considered as substitution subjects, because they were created freshly and thus appropriately chosen. (We formally support this point of view in Lemma 3.6, and show that it gives rise to an equivalent freshness-aware notion of bisimulation.) In contrast, the first two kinds shall be considered. On the other hand, also not all substitution objects may be acceptable. More precisely: depending on the history of the ongoing bisimulation game, certain instantiations may sometimes be forbidden. There may be two different reasons for this.

The first reason concerns names names of kind (i) or (ii), say $a$, that were free in a process before another name, say $b$, got freshly created and extruded. Due to the freshness, any subsequent substitution for subject $a$ must not mention $b$ as substitution object, so not to retrospectively invalidate this freshness property. In open bisimulation, represented by an indexed family of binary relations, the indexing component is precisely a structure called distinction that keeps track of inequalities like $a \neq b$, as required above.

The second reason concerns only names of kind (ii) and resides on the intuition that substitution objects represent messages that may be sent from the observer to the observed process. In the $\pi$-calculus, there is no limitation beyond the above distinctions: the observer may send any name that it may have.

$^2$ Note that we do not introduce different syntactic categories for (constant) names and variables. It is only for convenience of the explanation that we call receiving names in bound input position “input variables”.

$^3$ And here we slightly differ from Sangiorgi’s definition of open bisimulation.
received earlier, or it may simply invent names on its own. However, it is pre-
cisely here that severe difficulties arise when moving to the spi-calculus. The
main reason there is the presence of complex messages $E_k(n \cdots E_k(M) \cdots)$,
which may dispose of some deeply nested structure that involves so-called en-
cryption keys $k_1 \ldots k_n$. Substitution objects are then all messages that the
observer (potentially a malicious attacker) could possibly have generated at
the moment the message was input. This generation is not arbitrary; it is con-
strained by the knowledge that the observer has acquired up to the moment
of interaction. For example, consider the spi-calculus process

$$P \triangleq (\nu k) (\nu m) \overline{a}(E_k(m)).a(x).\overline{a}(k).[x=m]\overline{a}(a).0$$

where $(\nu k)$ denotes the generation of a fresh name, $\overline{a}(k)$ the sending of name $k$
over channel name $a$, $a(x)$ the reception of a message over channel name $a$
with input variable $x$, $E_k(m)$ the previously mentioned encryption of datum $m$
with key $k$, and $[x=m]$ a test of equality of names. Intuitively, the output
$\overline{a}(a)$ is impossible, because it would require that $x$ could have been substituted
by $m$, which is itself impossible, because the private datum $m$ was passed on
to the observer only within message $E_k(m)$ encrypted with the private key $k$;
however, this key was unknown to the observer when it sent the message that
got received by $a(x)$ — it was published only afterwards.

Here, a simple distinction $k \neq m$ is not sufficient to characterise disallowed
substitutions because neither $m$, nor $E_k(m)$, nor $E_k(E_k(m))$, etc., are permitted
substitution objects. In contrast, the message $E_k(m)$ that the observer learnt
in the first exchange could have been sent back to the process.

The study of other notions of bisimulation for the spi-calculus (see an
overview in [BN02]) resulted in careful analyses of observer (attacker) knowl-
edge and various kinds of data structures for the representation of such knowl-
edge. Typically, all messages that were emitted by an observed process in the
course of a bisimulation game are stored. Likewise, in particular in the pro-
posal of symbolic bisimulation of [BBN04], some timing or ordering informa-
tion is stored that keeps track of which messages were known to the observer
at the moment of the reception of a message by a process.

Together with the above-mentioned freshness-awareness, we choose to rep-
resent the observer knowledge for our new notion of open bisimulation by
tuples of the form $(O, V, \prec)$, where $\prec \subseteq O \times V$. $O$ is the set of the emit-
ted messages, while $V$ is the set of the substitutable names. Note that the
freshly created and subsequently extruded names are $C = n(O) \setminus n(V)$ and
we add the condition that $O \cap V = \emptyset$. The relation $\prec$ indicates for each
substitutable variable $x \in V$, which part of $O$ was known when $x$ was input.
Thus, in bisimulation games, this kind of environment structure permits to
treat substitutable names of the kinds (i) and (ii) in the same way.

While the above motivated way to characterise permissible substitutions
was driven by an analysis of spi-calculus phenomena, it also makes sense to
apply it to the much simpler $\pi$-calculus, which is the goal of this paper. In
\[ P, Q ::= 0 \mid E(x).P \mid \mathcal{E}(F).P \mid \phi P \mid P \mid Q \mid P + Q \mid P \mid (\nu x)P \]

Table 1
Syntax of processes \( P \)

\[ M, N ::= a \] (messages \( M \))
\[ E, F ::= a \] (expressions \( E \))
\[ \phi, \psi ::= tt \mid \phi \land \psi \mid [E=F] \] (formulae \( F \))

Table 2
Syntax of messages, expressions and formulae for the \( \pi \)-calculus

\[ M, N ::= a \mid E_N(M) \] (messages \( M \))
\[ E, F ::= a \mid E_F(E) \mid D_F(E) \] (expressions \( E \))
\[ \phi, \psi ::= tt \mid \phi \land \psi \mid [E=F] \mid [E:N] \] (formulae \( F \))

Table 3
Syntax of messages, expressions and formulae for the spi-calculus

\[ n(M) \]
\[ n(E) \]
\[ n(\phi) \]

§2, we recall the original definition of open bisimulation in the \( \pi \)-calculus, for which we use a unified presentation of the \( \pi \)-calculus and the spi-calculus. In §3, we develop the details of our new proposal and prove its coincidence with the original notion. In §4, we comment on the advantages of our new notion.

2 Open bisimulation

2.1 Syntax of the \( \pi \)-calculus and the spi-calculus

A countably infinite set \( a, b, c, \ldots, k, l, m, n, \ldots, x, y, z, \ldots \) of names \( N \) is presupposed. In the following, we write \( \tilde{z} \) for a (possibly empty) finite sequence of names \( z_1, z_2, \ldots, z_n \). If \( \tilde{z} \) is such a sequence, then we write \( \{ \tilde{z} \} \) for the set of names appearing in the sequence \( \tilde{z} \). In order to unify the presentation of the \( \pi \)-calculus and the spi-calculus, we have parametrised the syntax of processes Table 1 by messages, expressions and formulae. Table 2 read in conjunction with Table 1 gives the syntax of the \( \pi \)-calculus, whereas for the spi-calculus, Table 3 and Table 1 should be considered.

The set of names appearing in a message \( M \) is written \( n(M) \). In the case of the \( \pi \)-calculus, it is simply the singleton set containing \( M \) (since \( M \) is a name). Similarly, the set of the names appearing in an expression \( E \) is written \( n(E) \) and the set of the names appearing in a formula \( \phi \) is written \( n(\phi) \). Finally, the set of free names \( fn(P) \) and bound names \( bn(P) \) of a process \( P \) are defined as usual taking into account that the name \( x \) is bound in \( P \) by the constructs \( E(x).P \) and \( (\nu x)P \). These notions are straightforwardly lifted to sets.
Definition of \([\cdot] : E \rightarrow M \cup \{\bot\}\):

\[
\begin{align*}
[a] & \overset{\text{def}}{=} a \\
[EF(E)] & \overset{\text{def}}{=} EN(M) \quad \text{if} \ [E] = M \in M \text{ and } [F] = N \in M \\
[DF(E)] & \overset{\text{def}}{=} M \quad \text{if} \ [E] = EN(M) \in M \text{ and } [F] = N \in M \\
[E] & \overset{\text{def}}{=} \bot \quad \text{in all other cases}
\end{align*}
\]

Definition of \([\cdot] : F \rightarrow \{\text{true}, \text{false}\}\):

\[
\begin{align*}
[tt] & \overset{\text{def}}{=} \text{true} \\
[\phi \land \psi] & \overset{\text{def}}{=} [\phi] \text{ and } [\psi] \\
[[E=F]] & \overset{\text{def}}{=} \text{true} \quad \text{if} \ [E] = [F] = M \in M \\
[[E:N]] & \overset{\text{def}}{=} \text{true} \quad \text{if} \ [E] = a \in N \\
[\phi] & \overset{\text{def}}{=} \text{false} \quad \text{in all other cases}
\end{align*}
\]

Definition of \(c(\cdot) : F \rightarrow 2^{M \cup \{\bot\}}\):

\[
\begin{align*}
c(tt) & \overset{\text{def}}{=} \emptyset \\
c(\phi \land \psi) & \overset{\text{def}}{=} c(\phi) \cup c(\psi) \\
c([E=F]) & \overset{\text{def}}{=} \emptyset \\
c([E:N]) & \overset{\text{def}}{=} \{[[E]]\}
\end{align*}
\]

Table 4
Evaluation of expressions and formulae

2.2 Labelled (late) semantics

Table 4 defines the straightforward evaluation of expressions and formulae, as well as some name constraints of a given formula. Table 5 defines a labelled transition \(P \xrightarrow{\mu} S P'\) where \(\mu\) is an action and \(S\) is a set of names. The set \(S\) collects the names that should be names in order for the transition to be enabled. In the \(\pi\)-calculus, where only names are considered, it can be simply ignored but it is useful for the case of spi-calculus. These names are those that are used as channels or that are assumed to be names by formulae.

Upon this transition system, the late semantics of the \(\pi\)-calculus and the spi-calculus is given by: \(P \xrightarrow{\Delta} P'\) if and only if there is \(S\) such that \(P \xrightarrow{\mu} S P'\).

The syntax of actions \(\mu\) is given by:

\[
\mu ::= \tau \mid a(x) \mid (\nu \hat{z}) \bar{a} M \quad \text{(actions)}
\]

The bound output actions \((\nu \hat{z}) \bar{a} M\) are such that \(\{\hat{z}\} \subseteq n(M)\). In the case of the \(\pi\)-calculus, since messages \(M\) are reduced to names, we have two cases:
The late semantics of the $\pi$-calculus

Table 5

The late semantics of the $\pi$-calculus

either $\tilde{z}$ is the empty sequence and $(\nu \tilde{z}) \overline{a} M$ is simply written $\overline{a} M$ or $\tilde{z} = M$ and the bound output action is simply $(\nu z) \overline{a} z$ where $z = M$.

The set of names $n(\mu)$ is defined by:

$$n(\tau) := \emptyset, \quad n(a(x)) := \{a, x\}, \quad n((\nu \tilde{z}) \overline{a} M) := \{a, \tilde{z}\} \cup n(M).$$

The set of bound names $bn(\mu)$ of $\mu$ is defined by:

$$bn(\tau) := \emptyset, \quad bn(a(x)) := \{x\}, \quad bn((\nu \tilde{z}) \overline{a} M) := \{\tilde{z}\}.$$

Moreover, if $\mu = a(x)$ or $\mu = (\nu \tilde{z}) \overline{a} M$, we define $ch(\mu) \stackrel{\text{def}}{=} a$.

2.3 Open bisimulation in the $\pi$-calculus

As mentioned in the Introduction, open bisimulation was introduced by Sangiorgi [San96]. It relies on the notion of distinction to keep track of inequalities of names in order to constrain the set of substitutions to be considered in the respective bisimulation game.

**Definition 2.1 (distinction)** A binary relation $D \subseteq N \times N$ on names is called **distinction** if it is finite, symmetric, and irreflexive.

By $n(D)$ we denote the set of names contained in $D$. 

<table>
<thead>
<tr>
<th>Rule</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>$[E] = a \in \mathcal{N}$</td>
<td>$[E] = a \in \mathcal{N}$</td>
</tr>
<tr>
<td></td>
<td>$E(x).P \xrightarrow{a(x)}_{(a)} P$</td>
<td>$E(F).P \xrightarrow{\pi M}_{(a)} P$</td>
</tr>
<tr>
<td><strong>Close-L</strong></td>
<td>$P \xrightarrow{a(x)}_{S} P'$</td>
<td>$(\nu z) \xrightarrow{\pi M}_{S'} Q'$</td>
</tr>
<tr>
<td></td>
<td>$P \mid Q \xrightarrow{S_{(\nu \tilde{z})}} (P'{M/\tilde{z}}) \mid Q'$</td>
<td>${z} \cap fn(P) = \emptyset$</td>
</tr>
<tr>
<td><strong>Open</strong></td>
<td>$P \xrightarrow{(\nu \tilde{z})\pi M}_{S} P'$</td>
<td>$(\nu z') \xrightarrow{(\nu \tilde{z})\pi M}_{S\setminus(z')} P'$</td>
</tr>
<tr>
<td></td>
<td>$z' \in n(M) \setminus {a, \tilde{z}}$</td>
<td>$a \in \mathcal{N}$</td>
</tr>
<tr>
<td><strong>Res</strong></td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$(\nu z) \overset{\mu}{\xrightarrow{S\setminus(z)}} P'$</td>
</tr>
<tr>
<td></td>
<td>$z \notin n(\mu)$</td>
<td>$z' \notin n(\mu)$</td>
</tr>
<tr>
<td><strong>Guard</strong></td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P \overset{\phi P}{\xrightarrow{S\cup(\phi)}} P'$</td>
</tr>
<tr>
<td></td>
<td>$[\phi] = \text{true}$</td>
<td>$\phi P$</td>
</tr>
<tr>
<td><strong>Par-L</strong></td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td></td>
<td>$P \mid Q \overset{\mu}{\xrightarrow{S}} P' \mid Q$</td>
<td>$P \mid Q \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td></td>
<td>$bn(\mu) \cap fn(Q) = \emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td><strong>Sum-L</strong></td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td></td>
<td>$P + Q \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P + Q \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td></td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td><strong>Rep</strong></td>
<td>$\not!P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$\not!P \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td></td>
<td>$\not!P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$\not!P \overset{\mu}{\xrightarrow{S}} P'$</td>
</tr>
<tr>
<td><strong>Alpha</strong></td>
<td>$P = a \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P = a \overset{\mu}{\xrightarrow{S}} P''$</td>
</tr>
<tr>
<td></td>
<td>$P \overset{\mu}{\xrightarrow{S}} P'$</td>
<td>$P \overset{\mu}{\xrightarrow{S}} P''$</td>
</tr>
<tr>
<td></td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
If \( A, B \) are two sets of names, we define the distinction \( A \otimes B \) to be \( \{(x, y) \in A \times B \cup B \times A \mid x \neq y\} \). \( A^\# \) abbreviates \( A \otimes A \).

**Definition 2.2 (substitution)** A substitution \( \sigma \) is a total function \( \mathcal{N} \to \mathcal{M} \) such that its support \( \text{supp}(\sigma) := \{x \mid x \sigma \neq x\} \) is a finite set.

The co-support of \( \sigma \) is \( \text{cosupp}(\sigma) := \{x \sigma \mid x \in \text{supp}(\sigma)\} \).

The set of names of \( \sigma \) is \( n(\sigma) := \text{supp}(\sigma) \cup n(\text{cosupp}(\sigma)) \).

As said previously, distinctions are to prevent substitutions to fuse two names that were assumed to be different at some point. Hence the definition of so-called respectful substitutions.

**Definition 2.3 (respectfulness)** Let \( D \) be a distinction, \( \sigma \) a substitution.

\( \sigma \) respects \( D \), written \( \sigma \triangleright D \), if and only if \( x \sigma \neq y \sigma \) for all \( (x, y) \in D \).

If \( \sigma \) respects \( D \), then \( D \sigma \) is defined as \( \{(x \sigma, y \sigma) \mid (x, y) \in D\} \).

Note that since \( \mathcal{M} = \mathcal{N} \) in the case of the \( \pi \)-calculus, \( D \sigma \) is itself a distinction.

An open bisimulation is a distinction-indexed family of symmetric relations between processes that satisfies some condition.

**Definition 2.4 (open bisimulation)** The family \( (\mathcal{R}_D)_{D \in \mathcal{D}} \) (where \( \mathcal{D} \) is a set of distinctions) of symmetric relations is an open bisimulation if for all \( D \in \mathcal{D} \), for all substitutions \( \sigma \) such that \( \sigma \triangleright D \), for all \( (P, Q) \in \mathcal{R}_D \), whenever \( P \sigma \xrightarrow{\mu} P' \) (with \( \text{bn}(\mu) \) fresh), there exists \( Q' \) such that \( Q \sigma \xrightarrow{\mu} Q' \) and

* if \( \mu = (\nu z) a z \) for some \( a \) and \( z \), \( D' \in \mathcal{D} \) and \( (P', Q') \in \mathcal{R}_{D'} \)
  where \( D' = D \sigma \cup \{z\} \otimes (\text{fn}((P + Q) \sigma) \cup n(D \sigma)) \)
* otherwise, \( D \sigma \in \mathcal{D} \) and \( (P', Q') \in \mathcal{R}_{D \sigma} \).

The induced equivalence is defined as usual, modulo the indexing component.

**Definition 2.5 (open bisimilarity)** Let \( P, Q \in \mathcal{P} \) and \( D \) a distinction. We say that \( P \) and \( Q \) are open \( D \)-bisimilar—written \( P \approx^O_D Q \)—if there exists an open bisimulation \( (\mathcal{R}_D)_{D \in \mathcal{D}} \) such that \( D \in \mathcal{D} \) and \( (P, Q) \in \mathcal{R}_D \).

Instead of families of binary relations between processes we may also use ternary relations, which is often done in the context of the spi-calculus. Thus, instead of \( (P, Q) \in \mathcal{R}_D \), we then write \( (D, P, Q) \in \mathcal{R} \), where \( D \) is usually called environment, and the ternary relation is called environment-sensitive. It is mainly for easier readability that we adopt the ternary style in the following, although a bit of care needs to be taken to lift the three equivalence properties to the ternary format. For example, a ternary environment-sensitive relation is called symmetric if and only if \( (e, P, Q) \in \mathcal{R} \Leftrightarrow (e, Q, P) \in \mathcal{R} \).

3 Open bisimulation, reloaded

Before proceeding to our new proposal to define open-style bisimulation, we provide a slightly different, but equivalent variant of the previously given standard notion. This variant will make it easier to relate to our new proposal.
3.1 A freshness-aware variant of open bisimulation

In this section, we define the notion of F-open bisimulation. The simple idea is, as we mentioned already in the Introduction, to prevent names that were previously (in the course of a bisimulation game) created freshly from being considered as permissible substitution subjects.

The knowledgeable reader may be reminded of the notion of quasi-open bisimulation, proposed by Sangiorgi and Walker [SW01b], and later on revisited by Fu [Fu05]. There, the use of distinctions as environments was adapted to the use of a simple set of names that were once freshly created and therefore deemed to remain constant. The resulting quasi-open bisimulation was recognised as being strictly weaker than open bisimulation. Sangiorgi and Walker intuitively summarised this difference as: “In open bisimilarity, when a name \( z \) is sent in a bound-output action, the distinction is enlarged to ensure that \( z \) is never identified with any name that is free in the processes that send it. In quasi-open bisimilarity, in contrast, at no point after the scope of \( z \) is extruded can a substitution be applied that identifies \( z \) with any other name.” [SW01b].

Like quasi-open bisimulation, the following definition also explicitly keeps track of previously freshly created names. However, it does not use this information to prevent the fusion of such fresh names like quasi-open bisimulation does. It only use this information to implement the idea that fresh names can be considered as constant names once chosen, such that they should afterwards never be used as substitution subjects. In fact, Lemmas 3.6 and 3.7 show that this change still faithfully retains the equational power of open bisimulation.

**Definition 3.1 (F-environment)** The pair \((D, C)\) where \(D\) is a distinction and \(C\) is a finite subset of names is a F-environment if \(C \neq \cap D\). The set of all F-environments is written \(F\).

The distinction \(D\) plays the same role as in open bisimulation, while the set \(C\) indicates which names can be considered as constant names. It is used to refine the notion of respectfulness, as follows.

**Definition 3.2 (respectful substitution)**
Let \((D, C)\) be a F-environment and \(\sigma\) a substitution. We say that \(\sigma\) respects \((D, C)\) – written \(\sigma \rhd (D, C)\) – if \(\sigma \rhd D\) and \(\text{supp}(\sigma) \cap C = \emptyset\).

**Definition 3.3 (F-relation)** A F-relation \(R\) is a subset of \(F \times P \times P\).

**Definition 3.4 (F-open bisimulation)** A symmetric F-relation \(R\) is a F-open bisimulation, if for all \(((D, C), P, Q) \in R\) and for all substitutions \(\sigma\) such that \(\sigma \rhd (D, C)\), whenever \(P\sigma \overset{\mu}{\rightarrow} P'\) (with \(\text{bn}(\mu)\) fresh), there exists \(Q'\) such that \(Q\sigma \overset{\mu}{\rightarrow} Q'\) and

- if \(\mu = (\nu z) a z\) for some \(a\) and \(z\), \(((D', C \cup \{z\}), P', Q') \in R\) where \(D' = D\sigma \cup \{z\} \otimes (\text{fn}((P+Q)\sigma) \cup n(D\sigma))\)
- otherwise, \(((D\sigma, C'), P', Q') \in R\)

The two only differences compared to open bisimulation is, first, that the
notion of respectfulness is slightly modified such that it takes into account the constant names of a F-environment and, second, that the extruded names are being accumulated in the pool of constant names of F-environments.

**Definition 3.5 (F-open bisimilarity)** Let \( P, Q \in \mathcal{P} \) and \((D, C) \in \mathcal{F} \).

\( P \) and \( Q \) are F-open \((D, C)\)-bisimilar, written \( P \approx_{\mathcal{F}}^{(D, C)} Q \), if there is a F-open bisimulation \( \mathcal{R} \) such that \(((D, C), P, Q) \in \mathcal{R} \).

The two notions of bisimilarity are equivalent in the following sense.

**Lemma 3.6** Let \( P, Q \in \mathcal{P} \) and \((D, C) \in \mathcal{F} \).

If \( P \approx_{\mathcal{F}}^{(D, C)} Q \), then \( P \approx_{\mathcal{O}} Q \).

**Proof.** The key of the proof is that it is possible, if \( \sigma \triangleright D \) and \( C^{\neq} \subseteq D \), to find a substitution \( \sigma' \) and a bijective substitution \( \theta \) such that \( \sigma = \sigma' \theta \) and \( \sigma' \triangleright (D, C) \).

**Lemma 3.7** Let \( P, Q \in \mathcal{P} \) and \( D \) a distinction.

If \( P \approx_{\mathcal{O}} Q \), then \( \forall C : C^{\neq} \subseteq D \Rightarrow P \approx_{\mathcal{F}}^{(D, C)} Q \).

**Proof.** This result is obvious because \( \sigma \triangleright (D, C) \) implies \( \sigma \triangleright D \).

### 3.2 A knowledge-aware variant of open bisimulation

As motivated in the Introduction, we propose a bisimulation that makes explicit the attacker who plays against the two players \( P \) and \( Q \) involved in the bisimulation game. The knowledge of the attacker is stored in K-environments of the form \((O, V, \prec)\). The set of names \( V \) represents all the substitutable free names (those that were initially free or become free after an input action). The set of messages \( O \) contains all the messages that were emitted by \( P \) and \( Q \), except the names of \( V \). Finally, the relation \( \prec \) indicates for each substitutable name \( x \) the available knowledge acquired by the attacker at the moment the name \( x \) was input. This relation characterises the admissible messages received from the attacker.

**Definition 3.8 (K-environment)*** A K-environment is a triple \((O, V, \prec)\) such that \( O \cup V \) is a finite subset of \( \mathcal{N} \), \( O \cap V = \emptyset \) and \( \prec \subseteq O \times V \). The set of all K-environments is \( \mathcal{K} \).

If \( E \) is a K-environment, and \( n \in \mathcal{N} \), it is possible to extend \( E \) with \( n \) in two ways. Either \( n \) is meant to be an emitted name and it is added to the constant part of \( E \), or \( n \) is meant to be a received name and it is added to the variable part of \( E \) and put in relation with all already emitted names. If \( n \) is already contained in \( E \), its addition to \( E \) has no effect.

**Definition 3.9 (Extension of a K-environment)*** Let \( E = (O, V, \prec) \) be a K-environment and \( n \in \mathcal{N} \). We define

(i) \( E \uplus_{O} n \overset{\text{def}}{=} (O', V, \prec) \) where \( O' \overset{\text{def}}{=} O \cup \{n\} \) if \( n \notin V \) and \( O' \overset{\text{def}}{=} O \) otherwise.

(ii) if \( n \notin O \cup V \), \( E \uplus_{V} n \overset{\text{def}}{=} (O, V \cup \{n\}, \prec') \) where \( \prec' \overset{\text{def}}{=} \prec \cup O \times \{n\} \).
Keeping in mind that a substitution represents the potential inputs the attacker could have generated, we define the set of respectful substitutions. A substitution $\sigma$ respects a K-environment $E = (O, V, \prec)$ if it affects only substitutable names (those in $V$) and if for each $x \in V$, it takes only values that were generatable at the moment when $x$ was input. This means that such a name $x$ can use any name in $V$ (this corresponds to fusing two substitutable names), or use any name in $O$ that was known by the attacker when $x$ was input (this is indicated by the relation $\prec$) or use any new fresh name not contained in $E$ (this corresponds to the creation of free names by the attacker).

In the $\pi$-calculus, since a substitution replaces a name by a name, this can be easily and concisely expressed by:

**Definition 3.10 (respectful substitution)**

A substitution $\sigma$ respects a K-environment $E = (O, V, \prec)$, written $\sigma \triangleright E$, if:

(i) $\text{supp}(\sigma) \subseteq V$

(ii) $\forall x \in V : x\sigma \in O \Rightarrow x\sigma \prec x$

Roughly speaking, in spi-calculus, $x\sigma$ is built using names from $V$, the messages from $O$ that are permitted by $\prec$ and some freshly generated names. In $\pi$-calculus, this is simplified to $x\sigma \prec x$ because $x\sigma \in N$.

Any K-environment $E = (O, V, \prec)$ may, under the impact of some a respectful substitution $\sigma$, be straightforwardly updated to $E^\sigma$. In general, the knowledge contained in $O$ should be updated to $O\sigma$. However, in the $\pi$-calculus, substitution deals only with names, and since $O \cap V = \emptyset$ we have $O\sigma = O$. The set $V$ of substitutable names should keep all the names that were not affected by $\sigma$, and in addition list all the new names that were created by the attacker, as visible in the substitution objects.\(^4\) Particular care must be taken when computing the new relation $\prec'$ because of the possibility that $\sigma$ fuses two names of $V$. Fusing two names $x$ and $y$ (by $x\sigma = y\sigma$) corresponds to a voluntary loss of power of the attacker: the only admissible values for the fused name are those that were admissible for both $x$ and $y$.

**Definition 3.11 (K-environment updating)**

Let $E = (O, V, \prec)$ be a K-environment and $\sigma$ a substitution such that $\sigma \triangleright E$. The updated environment is $E^\sigma \overset{\text{def}}{=} (O', V', \prec')$ of $E$ by $\sigma$ where

$$V' \overset{\text{def}}{=} (V \setminus \text{supp}(\sigma)) \cup \{ x\sigma \mid x \in \text{supp}(\sigma) \land x\sigma \notin O \}$$

$$\prec' \overset{\text{def}}{=} \{ (n, x') \mid \forall x \in V : x' \in n(x\sigma) \Rightarrow n \prec x \}$$

**Definition 3.12 (K-relation)** A K-relation $\mathcal{R}$ is a subset of $\mathcal{K} \times \mathcal{P} \times \mathcal{P}$ such that $\forall ((O, V, \prec), P, Q) \in \mathcal{R} : \text{fn}(P+Q) \subseteq O \cup V$.

The new variant of open bisimulation now simply keeps track of whether dynamically freed names are substitutable or not. If they are, then we explic-
itly state that previously created names may be used in future substitutions. Names that will be created later on—by the process—will not be permitted.

**Definition 3.13 (K-open bisimulation)** A symmetric K-relation \( R \) is a K-open bisimulation, if for all \((E, P, Q) \in R\) and for all substitutions \( \sigma \) such that \( \sigma \triangleright E \), whenever \( P \sigma \overset{\mu}{\rightarrow} P' \) (with \( \text{bn}(\mu) \) fresh), there exists \( Q' \) such that \( Q \sigma \overset{\mu}{\rightarrow} Q' \) and

- if \( \mu = \tau \), then \((E', P', Q') \in R\)
- if \( \mu = a(x) \) then \((E' \oplus V x, P', Q') \in R\)
- if \( \mu = (\nu z) \overline{a} z \) or \( \mu = \overline{a} z \) then \((E' \oplus O z, P', Q') \in R\)

We see in this definition that indeed \( O \) collects all the messages emitted by \( P \) and \( Q \) (but the addition \( E' \oplus O z \) has only effect when \( \mu = (\nu z) \overline{a} z \) because \( E \) contains all free names of \( P \) and \( Q \)) and \( V \) collects all substitutable names.

**Definition 3.14 (K-open bisimilarity)** Let \( P, Q \in \mathcal{P} \) and \( E \in K \). \( P \) and \( Q \) are K-open \( E \)-bisimilar, written \( P \approx_{E}^{K} Q \), if there is a K-open bisimulation \( R \) such that \( (E, P, Q) \in R \).

In the \( \pi \)-calculus, it is possible to represent any K-environment by some F-environment. The idea is that all names in \( O \) should be kept pairwise distinct (they were fresh names) and for all \((n, x) \in O \cup V \), if \( n \) cannot be used to generate \( x \) (i.e. \( n \not≺ x \)), then \( n \) and \( x \) should be distinct \((n \neq x)\).

**Definition 3.15 (F-environment of a K-environment)** Let \( E = (O, V, \prec) \) be a K-environment. We define \( f(E) = (D, O) \) where \( D = O \not\emptyset \cup \bigcup_{n \in O \land x \in V \land n \not≺ x} \{(n, x), (x, n)\} \). Clearly, \( f(E) \in F \).

The K-open bisimilarity is sound with respect to F-open bisimilarity.

**Lemma 3.16** Let \( P, Q \in \mathcal{P} \) and \((O, V, \prec) \in K \) such that \( \text{fn}(P+Q) \subseteq O \cup V \). Then we have:

\[
P \approx_{E}^{(O, V, \prec)} Q \Rightarrow P \approx_{F}^{f((O, V, \prec))} Q
\]

Under the condition that the F-environment \((D, C)\) is representable by a K-environment \( E \), F-open \((D, C)\)-bisimilarity is sound with respect to K-open \( E \)-bisimilarity.

**Lemma 3.17** Let \( P, Q \in \mathcal{P} \) and \((D, C) \in F \). Then we have

\[
P \approx_{F}^{(D, C)} Q \Rightarrow \begin{cases} C \cap V = \emptyset \\ \forall V, \prec : & \text{fn}(P+Q) \subseteq C \cup V \\
\land (D, C) = f((C, V, \prec))
\end{cases} \Rightarrow P \approx_{E}^{(C, V, \prec)} Q
\]

The proof of this lemma also shows that F-environments that are not representable by any corresponding K-environment are negligible.

It is known that open \( D \)-bisimilarity is a \( D \)-congruence, i.e., it is preserved by all contexts in which the occurrence of the hole is not underneath an input prefix binding a name in \( D \) (cf. [SW01a]). We conjecture that, based on our new notion of K-open-bisimilarity and with respect to \((D, C) = f((C, V, \prec))\),
we can define a bigger classes of contexts that preserve open bisimilarity. The idea is (1) to admit contexts with the same above condition w.r.t. names \( C \) as \( D \)-congruence imposes w.r.t. \( D \), and furthermore (2) to admit contexts where the hole occurs underneath an input prefix that binds a name \( x \) of \( V \), but only if, in addition, every name of \( \{ n \in C \mid \neg n \prec x \} \) appears underneath a respective restriction on the “path” from the hole-binding input prefix for \( x \) to the hole. We leave a formal treatment of this issue for future work, and just explain the conjecture by means of a simple example.

**Example 3.18** Let \( P = \pi y \) and \( Q = \pi . y + y. \pi \).

It is known and easily verifiable that \( P \approx^D Q \) with \( D = \{ (x, y), (y, x) \} \).

Let \( C = \{ y \} \) and \( V = \{ x \} \), and note that \( (D, C) = f((C, V, \emptyset)) \).

Observe that \( P \approx^{(C, V, \prec)}_K Q \).

Now, let us regard the context \( X[\cdot] = a(x)(\nu y)[\cdot] \).

Then \( X[P] \approx^{0}_K X[Q] \), although \( X[\cdot] \) is not considered by \( D \)-congruence.

However, \( X[\cdot] \) follows our above informal rule of admissible contexts.

Finally, just note that also \( X[P] \approx^{(\emptyset, \{a\}, \emptyset)}_K X[Q] \).

In summary, we can conclude from the previous results our new notion of open-style bisimilarity semantically coincides with the original style.

**Theorem 3.19** \( P \approx^0 Q \Leftrightarrow P \approx^F Q \Leftrightarrow P \approx^{(\emptyset, \emptyset)}_K Q \)

### 4 Conclusion and future work

The main contribution of this paper is the definition of a new notion of open-style bisimulation in the \( \pi \)-calculus guided by knowledge-sensitive notions of bisimulation that arose in the context of the spi-calculus. We have proved that the new notion corresponds to the original open bisimilarity in a precise and informative way that indicates improved congruence properties.

The new definition of open-style bisimulation can now indeed be smoothly extended in the spi-calculus (a first proposal is given in appendix but we can mention close work such as [Bri02] or [BBN04]). Our proposal in spi-calculus uses the same environment shape as our proposal in \( \pi \)-calculus. But it is necessary, as noticed by Abadi and Gordon in [AG98], to introduce also a notion of indistinguishability. Some type constraints should also be ensured: a free name used as a channel should never be substituted by anything else than a name. Hence, the environment we propose for spi-calculus are quadruple \((h, v, \prec, \gamma)\) where \( h \) stores all the emitted messages and moreover implements this notion of indistinguishability, \( v \) contains all the substitutable names, \( \prec \) governs which messages can be used to generate inputs for names in \( v \) and \( \gamma \) stores which names should keep the type of names.

Next, we plan to study congruence properties of our K-open bisimilarity. We will do the same for our extension to the spi-calculus and also study its relation to symbolic bisimilarity as defined in [BBN04].
References


