Pre-proceedings of
1st Interaction and Concurrency Experience (ICE’08)

Synchronous and Asynchronous Interactions
in
Concurrent Distributed Systems

Filippo Bonchi, Davide Grohmann, Paola Spoletini, Angelo Troina, and Emilio Tuosto
Preface

This document contains the pre-proceedings of ICE’08, the 1st Interaction and Concurrency Experience held in Reykjavik (Iceland) on July 6 2008 and co-located with ICALP 2008.

ICE’08 aims to provide an international forum where researchers can present and discuss emergent ideas for modelling and reasoning on theoretical and applied aspects of concurrent and distributed computing. The 1st ICE meeting focuses on Synchronous and Asynchronous Interactions in Concurrent Distributed Systems.

The Program Committee has selected the papers included in this pre-proceedings after a careful and innovative revision process supported by an on-line forum fostering discussions among reviewers and authors. All the 12 papers submitted have been reviewed by at least 3 reviewers and each of the 8 papers collected here has been selected not only according to the reviewers comments but also considering the discussions held on the forum.

This volumes also contains the abstracts of the invited contributions by Catuscia Palamidessi (École Polytechnique) and Joseph Sifakis (VERIMAG).

ICE’08 was made possible by the contribution of many people and institutions. We express our gratitude to all authors and PC members for having contributed papers and scientific discussions on the ICE forum. We have been delighted to see how the reviewing process worked nicely (the merit of which is entirely of authors and PC members), improved the already high quality of the submitted papers, and hopefully gave authors of non accepted papers insights on how to improve their work.

Finally, we acknowledge our sponsors the ESF project AutoMathA, the University of Pisa, and the ICALP organisers, moreover Emilio Tuosto has been supported by the Nuffield Foundation (grant for Newly Appointed Lecturers 2006, Ref: NAL/32612, ”History Dependent Automata for Service Oriented Computing (HiDeA4SOC)”).

Filippo Bonchi
Davide Grohmann
Paola Spoletini
Angelo Troina
Emilio Tuosto
A randomized implementation of synchronous communication in presence of mixed choice

Catuscia Palamidessi

INRIA and LIX, École Polytechnique, Rue de Saclay, 91128 Palaiseau, France

We consider the problem of encoding the π-calculus with mixed choice into the asynchronous π-calculus via a uniform translation while preserving a reasonable semantics. Although it has been shown that this is not possible with an exact encoding, we suggest a randomized approach using a probabilistic extension of the asynchronous π-calculus, and we show that our solution is correct with probability 1 under any proper adversary wrt a notion of testing semantics. This result establishes the basis for a distributed and symmetric implementation of mixed choice which, differently from previous proposals in literature, does not rely on assumptions on the relative speed of processes and it is robust to attacks of proper adversaries.
Component-based Construction of Heterogeneous Real-time Systems in BIP

Joseph Sifakis

VERIMAG

We present a framework for the component-based construction of real-time systems. The framework is based on the BIP (Behavior, Interaction, Priority) semantic model, characterized by a layered representation of components. Compound components are obtained as the composition of atomic components specified by their behavior and interface, by using connectors and dynamic priorities. Connectors describe structured interactions between atomic components, in terms of two basic protocols: rendezvous and broadcast. Dynamic priorities are used to select amongst possible interactions - in particular, to express scheduling policies.

BIP supports a methodology for incremental construction within a three-dimensional space: Behavior × Interaction × Priority. The separation between behavior and architectural constraints expressed by interactions and priorities, eases compositional verification of systems through a separate analysis of their atomic components and their architectural constraints.

The BIP framework has been implemented in a language and a toolset. The BIP language offers primitives and constructs for modelling and composing atomic components described as state machines, extended with data and functions in C. The BIP toolset includes an editor and a compiler for generating from BIP programs, C++ code executable on a dedicated platform. It allows simulation and verification of BIP programs by using model checking or compositional techniques for some properties such as deadlock-freedom.

We provide several examples illustrating the use of BIP for modeling heterogeneous systems. Further information is available at: http://www-verimag.imag.fr/~async/bip.html

This paper is electronically published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
Symmetric and Asymmetric Asynchronous Interaction

Rob van Glabbeek\textsuperscript{1}  
NICTA, Sydney, Australia  
University of New South Wales, Sydney, Australia

Ursula Goltz\textsuperscript{2}  Jens-Wolfgang Schicke\textsuperscript{3}  
Institute for Programming and Reactive Systems  
TU Braunschweig  
Braunschweig, Germany

Abstract

We investigate classes of systems based on different interaction patterns with the aim of achieving distributability. As our system model we use Petri nets. In Petri nets, an inherent concept of simultaneity is built in, since when a transition has more than one preplace, it can be crucial that tokens are removed instantaneously. When modelling a system which is intended to be implemented in a distributed way by a Petri net, this built-in concept of synchronous interaction may be problematic. To investigate the problem we assume that removing tokens from places can no longer be considered as instantaneous. We model this by inserting silent (unobservable) transitions between transitions and their preplaces. We investigate three different patterns for modelling this type of asynchronous interaction. Full asynchrony assumes that every removal of a token from a place is time consuming. For symmetric asynchrony, tokens are only removed slowly in case of backward branched transitions, hence where the concept of simultaneous removal actually occurs. Finally we consider a more intricate pattern by allowing to remove tokens from preplaces of backward branched transitions asynchronously in sequence (asymmetric asynchrony).

We investigate the effect of these different transformations of instantaneous interaction into asynchronous interaction patterns by comparing the behaviours of nets before and after insertion of the silent transitions. We exhibit for which classes of Petri nets we obtain equivalent behaviour with respect to failures equivalence. It turns out that the resulting hierarchy of Petri net classes can be described by semi-structural properties. In case of full asynchrony and symmetric asynchrony, we obtain precise characterisations; for asymmetric asynchrony we obtain lower and upper bounds.

We briefly comment on possible applications of our results to Message Sequence Charts.

Keywords: reactive systems, Petri nets, distributed systems, asynchronous interaction, equivalence notions

1 Introduction

In this paper, we investigate classes of systems based on different asynchronous interaction patterns with the aim of achieving distributability, i.e. the possibility to execute a system on spatially distributed locations, which do not share a common
van Glabbeek, Goltz and Schicke

Fig. 1. Transformation to the symmetrically asynchronous implementation

clock. As our system model we use Petri nets. The main reason for this choice is the detailed way in which a Petri net represents a concurrent system, including the interaction between the components it may consist of. In an interleaving based model of concurrency such as labelled transition systems modulo bisimulation semantics, a system representation as such cannot be said to display synchronous or asynchronous interaction; at best these are properties of composition operators, or communication primitives, defined in terms of such a model. A Petri net on the other hand displays enough detail of a concurrent system to make the presence of synchronous communication discernible. This makes it possible to study asynchronous communication without digressing to the realm of composition operators. In a Petri net, a transition interacts with its preplaces by consuming tokens. An inherent concept of simultaneity is built in, since when a transition has more than one preplace, it can be crucial that tokens are removed instantaneously, depending on the surrounding structure or—more elaborately—the behaviour of the net.

When modelling a distributed system by a Petri net, this built-in concept of synchronous interaction may become problematic. Assume a transition $t$ on a location $l$ models an activity involving another location $l'$, for example by receiving a message. This can be modelled by a preplace $s$ of $t$ such that $s$ and $t$ are situated in different locations. We assume that taking a token can in this situation not be considered as instantaneous; rather the interaction between $s$ and $t$ takes time. We model this effect by inserting silent (unobservable) transitions between transitions and their preplaces. We call the effect of such a transformation of a net $N$ an asynchronous implementation of $N$.

An example of such an implementation is shown in Figure 1. Note that $a$ can be disabled in the implementation before any visible behaviour has taken place. This difference will cause non-equivalence between the original and the implementation under branching time equivalences.

Our asynchronous implementation allows a token to start its journey from a place to a transition even when not all preplaces of the transition contain a token. This design decision is motivated by the observation that it is fundamentally impossible to check in an asynchronous way whether all preplaces of a transition are marked—it could be that a token moves back and forth between two such places.

We investigate different interaction patterns for the asynchronous implementation of nets. The simplest pattern (full asynchrony) assumes that every removal of a token from a place is time consuming. For the next pattern (symmetric asynchrony), tokens are only removed slowly when they are consumed by a backward branched transition, hence where the concept of simultaneous removal actually occurs. Fi-
nally we consider a more intricate pattern by allowing to remove tokens from preplaces of backward branched transitions asynchronously in sequence (asymmetric asynchrony).

Given a choice of interaction pattern, we call a net $N$ asynchronous when there is no essential behavioural difference between $N$ and its asynchronous implementation $I(N)$. In order to formally define this concept, we wish to compare the behaviours of $N$ and $I(N)$ using a semantic equivalence that fully preserves branching time, causality and their interplay, whilst of course abstracting from silent transitions. By choosing the most discriminating equivalence possible, we obtain the smallest possible class of asynchronous nets, thus excluding nets that might be classified as asynchronous merely because a less discriminating equivalence would fail to see the differences between such a net and its asynchronous implementation. To simplify the exposition, here we merely compare the behaviours of $N$ and $I(N)$ up to failures equivalence [6]. This interleaving equivalence abstracts from causality and respects branching time only to some degree. However, we conjecture that our results are in fact largely independent of this choice and that more discriminating equivalences, such as the history preserving ST-bisimulation of [21], would yield the same classes of asynchronous nets. Using a linear time equivalence would give rise to larger classes; this possibility is investigated in [19].

Thus we investigate the effect of our three transformations of instantaneous interaction into asynchronous interaction patterns by comparing the behaviours of nets before and after insertion of the silent transitions up to failures equivalence. We show that in the case of full asynchrony, we obtain equivalent behaviour exactly for conflict-free Petri nets. Further we establish that symmetric asynchrony is a valid concept for $N$-free Petri nets and asymmetric asynchrony for $M$-free Petri nets, where $N$ and $M$ stand for certain structural properties; the reachability of such structures is crucial. For symmetric asynchrony we obtain a precise characterisation of the class of nets which is asynchronously implementable. For asymmetric asynchrony we obtain lower and upper bounds.

In the concluding section, we discuss the use of our results for Message Sequence Charts, as an example how they may be useful for other models than Petri nets. When interpreting basic Message Sequence Chart as Petri nets, the resulting Petri nets lie within the class of conflict-free and hence $N$-free Petri nets. The more expressive classes give insights in the effect of choices in non-basic MSCs.

This is an extended abstract; for sake of brevity most proofs are omitted. They are contained in the full version of this paper [8], as well as in [19].

The paper is structured as follows. In Section 2 we establish the necessary basic notions. In Section 3 we introduce the fully asynchronous transformation and give a semi-structural characterisation of the resulting net class. In Section 4 we repeat those steps for the symmetrically asynchronous transformation. Furthermore we describe how the resulting net class relates to the classes of free-choice and extended free choice nets. In Section 5 we introduce the asymmetrically asynchronous transformation. We give semi-structural upper and lower bounds for the resulting net class and relate it to simple and extended simple nets. In the conclusion in Section 6 we compare our findings to similar results in the literature.
2 Basic Notions

We consider here 1-safe net systems, i.e. places never carry more than one token, but a transition can fire even if pre- and postset intersect. To represent unobservable behaviour, which we use to model asynchrony, the set of transitions is partitioned into observable and silent (unobservable) ones.

Definition 2.1

A net with silent transitions is a tuple $N = (S, O, U, F, M_0)$ where
- $S$ is a set (of places),
- $O$ is a set (of observable transitions),
- $U$ is a set (of silent transitions),
- $F \subseteq S \times T \cup T \times S$ (the flow relation) with $T := O \cup U$ (transitions) and
- $M_0 \subseteq S$ (the initial marking).

Petri nets are depicted by drawing the places as circles, the transitions as boxes, and the flow relation as arrows (arcs) between them. When a Petri net represents a concurrent system, a global state of such a system is given as a marking, a set of places, the initial state being $M_0$. A marking is depicted by placing a dot (token) in each of its places. The dynamic behaviour of the represented system is defined by describing the possible moves between markings. A marking $M$ may evolve into a marking $M'$ when a nonempty set of transitions $G$ fires. In that case, for each arc $(s, t) \in F$ leading to a transition $t$ in $G$, a token moves along that arc from $s$ to $t$. Naturally, this can happen only if all these tokens are available in $M$ in the first place. These tokens are consumed by the firing, but also new tokens are created, namely one for every outgoing arc of a transition in $G$. These end up in the places at the end of those arcs. A problem occurs when as a result of firing $G$ multiple tokens end up in the same place. In that case $M'$ would not be a marking as defined above. In this paper we restrict attention to nets in which this never happens. Such nets are called 1-safe. Unfortunately, in order to formally define this class of nets, we first need to correctly define the firing rule without assuming 1-safety. Below we do this by forbidding the firing of sets of transitions when this might put multiple tokens in the same place.

Definition 2.2

Let $N = (S, O, U, F, M_0)$ be a net. Let $M_1, M_2 \subseteq S$.
We denote the preset and postset of a net element $x$ by $x^\bullet := \{y \mid (y, x) \in F\}$ and $x^* := \{y \mid (x, y) \in F\}$ respectively. A nonempty set of transitions $G \subseteq (O \cup U), G \neq \emptyset$, is called a step from $M_1$ to $M_2$, notation $M_1 \mid G \rangle_N M_2$, iff
- all transitions contained in $G$ are enabled, that is
  \[ \forall t \in G. \, \bullet t \subseteq M_1 \land (M_1 \setminus \bullet t) \cap t^\bullet = \emptyset, \]
- all transitions of $G$ are independent, that is not conflicting:
  \[ \forall t, u \in G, t \neq u. \, \bullet t \cap \bullet u = \emptyset \land t^\bullet \cap u^\bullet = \emptyset, \]
- in $M_2$ all tokens have been removed from the preplaces of $G$ and new tokens
have been inserted at the postplaces of $G$:

$$M_2 = \left( M_1 \setminus \bigcup_{t \in G} \bullet t \right) \cup \bigcup_{t \in G} t^* .$$

To simplify statements about possible behaviours of nets, we use some abbreviations.

**Definition 2.3** Let $N = (S, O, U, F, M_0)$ be a net with silent transitions.

1. $\longrightarrow_{N} \subseteq \mathcal{P}(S) \times \mathcal{P}(O) \times \mathcal{P}(S)$ is defined by $M_1 \xrightarrow{G}_{N} M_2 \iff G \subseteq O \land M_1[G]_N M_2$
2. $\tau_{N} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ is defined by $M_1 \xrightarrow{\tau}_{N} M_2 \iff \exists t \in U. \ M_1 \{t\}_N M_2$
3. $\implies_{N} \subseteq \mathcal{P}(S) \times O^* \times \mathcal{P}(S)$ is defined by $M_1 \xrightarrow{\implies}_{N} M_2 \iff M_1 \overset{\text{trans.}}{\overset{t_1}{\overset{\cdot \cdot \cdot}{\overset{t_n}{\overset{\tau}{\longrightarrow}_{N}}}}} M_2$

where $\tau^*_{N}$ denotes the reflexive and transitive closure of $\tau_{N}$.

We write $M_1 \overset{G}{\longrightarrow}_{N} M_2$ for $\exists M_2$, $M_1 \overset{G}{\longrightarrow}_{N} M_2$, $M_1 \overset{G}{\longrightarrow}_{N}$ for $\forall M_2$. $M_1 \overset{\sigma}{\longrightarrow}_{N} M_2$ and similar for the other two relations.

A marking $M_1$ is said to be **reachable** if there is a $\sigma \in O^*$ such that $M_0 \overset{\sigma}{\implies} M_1$.

The set of all reachable markings is denoted by $[M_0]_N$.

We omit the subscript $N$ if clear from context.

As said before, here we only want to consider 1-safe nets. Formally, we restrict ourselves to **contact-free nets** where in every reachable marking $M_1 \in [M_0]$ for all $t \in O \cup U$ with $\bullet t \subseteq M_1$

$$(M_1 \setminus \bullet t) \cap t^* = \emptyset .$$

For such nets, in Definition 2.2 we can just as well consider a transition $t$ to be enabled in $M$ iff $\bullet t \subseteq M$, and two transitions to be independent when $\bullet t \cap \bullet u = \emptyset$.

In this paper we furthermore restrict attention to nets for which $\bullet t \neq \emptyset$, and $\bullet t$ and $t^*$ are finite for all $t \in O \cup U$. We also require the initial marking $M_0$ to be finite. A consequence of these restrictions is that all reachable markings are finite, and it can never happen that infinitely many independent transitions are enabled. Henceforth, we employ the name **$\tau$-nets** for nets with silent transitions obeying the above restrictions, and **plain nets** for $\tau$-nets without silent transitions, i.e. with $U = \emptyset$.

Our nets with silent transitions can be regarded as special **labelled nets**, defined as in Definition 2.1, but without the split of $T$ into $O$ and $U$, and instead equipped with a **labelling function** $\ell : T \rightarrow \text{Act} \cup \{\tau\}$, where $\text{Act}$ is a set of visible actions and $\tau \notin \text{Act}$ an invisible one. Nets with silent transitions correspond to labelled nets in which no two different transitions are labelled by the same visible actions, which can be formalised by taking $\ell(t) = t$ for $t \in O$ and $\ell(t) = \tau$ for $t \in U$.

To describe which nets are “asynchronous”, we will compare their behaviour to that of their asynchronous implementations using a suitable equivalence relation. As explained in the introduction, we consider here branching time semantics. Technically, we use failures equivalence, as defined below.

**Definition 2.4** Let $N = (S, O, U, F, M_0)$ be a $\tau$-net, $\sigma \in O^*$ and $X \subseteq O$.

$<\sigma, X>$ is a **failure pair** of $N$ iff

$$\exists M_1. \ M_0 \overset{\sigma}{\longrightarrow} M_1 \land M_1 \xrightarrow{\tau} \land \forall t \in X. \ M_1 \xrightarrow{\{t\}} .$$

5
\[ N: \quad \begin{array}{c}
\text{a} \quad \text{b}
\end{array} \]
\[ \text{FI}(N): \quad \begin{array}{c}
\tau \quad \tau
\end{array} \]

Fig. 2. A net which is not failures equivalent to its fully asynchronous implementation

We define \( \mathcal{F}(N) := \{ <\sigma, X> \mid <\sigma, X> \text{ is a failure pair of } N \} \).

Two \( \tau \)-nets \( N \) and \( N' \) are failures equivalent, \( N \approx_{\mathcal{F}} N' \), iff \( \mathcal{F}(N) = \mathcal{F}(N') \).

A \( \tau \)-net \( N = (S, O, U, F, M_0) \) is called divergence free iff there are no infinite chains of markings \( M_1 \xrightarrow{\tau} M_2 \xrightarrow{\tau} \cdots \) with \( M_1 \in [M_0] \).

### 3 Full Asynchrony

As explained in the introduction, we will examine in this paper different possible assumptions of how asynchronous interaction between transitions and their preplaces takes place. In this section, we start with the simple and intuitive assumption that the removal of any token by a transition takes time. This is implemented by inserting silent transitions between visible ones and their preplaces.

**Definition 3.1** Let \( N = (S, O, \emptyset, F, M_0) \) be a plain net.

The fully asynchronous implementation of \( N \) is defined as the net \( \text{FI}(N) := (S \cup S', O, U', F', M_0) \) with

\[
S' := \{s_t \mid t \in O, s \in \bullet t\}, \quad U' := \{t_s \mid t \in O, s \in \bullet t\} \text{ and } F' := (F \cap (O \times S)) \cup \{(s, t_s), (t_s, s_t), (s_t, t) \mid t \in O, s \in \bullet t\}.
\]

It is not hard to see that implementations of contact-free nets are contact-free and implementations are always divergence free; in fact an implementation of a plain net is always a divergence free \( \tau \)-net.

Whereas in a plain net \( N \) for any sequence of observable transitions \( \sigma \in O^* \) there is at most one marking \( M \) with \( M_0 \xrightarrow{\sigma} M \), in its fully asynchronous implementation \( \text{FI}(N) \) there can be several such markings. These markings \( M' \) differ from \( M \) in that some tokens may have wandered off into the added invisible transitions on the incoming arcs of visible ones. As a consequence, a visible transition \( t \) that is enabled in \( M \) need not be enabled in \( M' \)—we say that in \( \text{FI}(N) \) \( t \) can be refused after \( \sigma \).

This may occur for instance for the net \( N \) of Figure 2, namely with \( \sigma = \varepsilon \) (the empty sequence), \( M \) the initial marking of \( N \), \( M' \) the marking of \( \text{FI}(N) \) obtained by firing the rightmost invisible transition, and \( t = a \).

When this happens, we have \( <\sigma, \{t\} > \in \mathcal{F}(\text{FI}(N)) \setminus \mathcal{F}(N) \), so the nets \( N \) and \( \text{FI}(N) \) are not failures equivalent. If, on the other hand, the wandering off of
tokens into $\tau$-transitions never disables a transition that would be enabled otherwise, then there is no essential behavioural difference between $N$ and $FI(N)$, and they are equivalent in any reasonable behavioural equivalence that abstracts from silent transition firings. In that case, $N$ could be called fully asynchronous.

**Definition 3.2**

The class of *fully asynchronous nets respecting branching time equivalence* is defined as $FA(B) := \{ N \mid FI(N) \approx_{B} N \}$.

As for any plain net $N$ we have $F(N) \subseteq F(FI(N))$ [8], the class of nets $FA(B)$ can equivalently be defined as $FA(B) := \{ N \mid \mathcal{F}(FI(N)) \subseteq \mathcal{F}(N) \}$.

It turns out that there exists a quite structural characterisation of those nets which are failures equivalent to their fully asynchronous implementation.

**Definition 3.3**

A plain net $N = (S, O, \emptyset, F, M_{0})$ has a partially reachable conflict iff $\exists t, u \in O. \ t \neq u \land \bullet t \cap \bullet u \neq \emptyset$ and $\exists M \in [M_{0}]. \ \bullet t \subseteq M \land \bullet u \subseteq M$.

The nets $N$ of Figures 2 and 3, for instance, have a partially reachable conflict.

**Theorem 3.4** A plain net $N$ is in $FA(B)$ iff $N$ has no partially reachable conflict.

**Proof.** See [19] or [8].

## 4 Symmetric Asynchrony

For investigating the next interaction pattern, we change our notion of asynchronous implementation of a net. We only insert silent transitions wherever a transition has multiple preplaces. These are the situations where the synchronous removal of tokens is really essential.

**Definition 4.1** Let $N = (S, O, \emptyset, F, M_{0})$ be a net. Let $O^{b} = \{ t \mid t \in O, |\bullet t| > 1 \}$.

The *symmetrically asynchronous implementation* of $N$ is defined as the net $SI(N) := (S \cup S^{\tau}, O, U', F', M_{0})$ with

\[
S^{\tau} := \{ s_{t} \mid t \in O^{b}, s \in \bullet t \},
U' := \{ t_{s} \mid t \in O^{b}, s \in \bullet t \} \text{ and}
F' := F \cap \left( (O \times S) \cup (S \times (O \setminus O^{b})) \right)
\cup \{ (s, t_{s}, t_{s}, s_{t}, s_{t}) \mid t \in O^{b}, s \in \bullet t \}.
\]

An example is shown in Figure 3.

As for the fully asynchronous case, an implementation of a plain net is always a divergence-free $\tau$-net.

Again, the only difference in behaviour between the original net and its implementation is that observable transitions can potentially be refused in the implementation, as in Figure 3. This yields a concept of a *symmetrically asynchronous* net.
Definition 4.2
The class of symmetrically asynchronous nets respecting branching time equivalence is defined as $\mathcal{SA}(B) := \{ N \mid SI(N) \approx_{\mathcal{F}} N \}$.

Again we have $\mathcal{F}(N) \subseteq \mathcal{F}(SI(N))$ for any plain net $N$ \cite{8}. We now show that plain nets can be implemented symmetrically asynchronously with respect to failure equivalence exactly when they do not contain reachable structures of the form shown in Figure 3.

Definition 4.3
A plain net $N = (S, O, \emptyset, F, M_0)$ has a partially reachable $N$ iff $\exists t, u \in O. t \neq u \land \bullet t \cap \bullet u \neq \emptyset \land |\bullet t| > 1 \land \exists M \in [M_0]_N. \bullet t \subseteq M \lor \bullet u \subseteq M$.

Theorem 4.4 A plain net $N$ is in $\mathcal{SA}(B)$ iff $N$ has no partially reachable $N$.

Proof. See \cite{19} or \cite{8}.

The following proposition shows that the current class of nets strictly extends the one from the previous section.

Proposition 4.5 $FA(B) \subset \mathcal{SA}(B)$.

Proof. A net without partially reachable conflict surely has no partially reachable $N$. The inequality follows from the example in Figure 2.

It turns out that our class of nets $\mathcal{SA}(B)$ is strongly related to the following established net classes \cite{2,3}.

Definition 4.6 Let $N = (S, O, \emptyset, F, M_0)$ be a plain net.

(i) $N$ is free choice, $N \in FC$, iff $\forall p, q \in S. p \neq q \land p^* \cap q^* \neq \emptyset \Rightarrow |p^*| = |q^*| = 1$.

(ii) $N$ is extended free choice, $N \in EFC$, iff $\forall p, q \in S. p^* \cap q^* \neq \emptyset \Rightarrow p^* = q^*$.

(iii) $N$ is behaviourally free choice, $N \in BFC$, iff $\forall u, v \in O. \bullet u \cap \bullet v \neq \emptyset \Rightarrow (\forall M_1 \in [M_0]. \bullet u \subseteq M_1 \Leftrightarrow \bullet v \subseteq M_1)$.

The above definition of a free choice net is in terms of places, but the notion can equivalently be defined in terms of transitions:

$N \in FC$ iff $\forall t, u \in T. t \neq u \land \bullet t \cap \bullet u \neq \emptyset \Rightarrow |\bullet t| = |\bullet u| = 1$.

Both conditions are equivalent to the requirement that $N$ must be $N$-free, where $N$ is defined as in Definition 4.3 but without the reachability clause. Also the notion of an extended free choice net can equivalently be defined in terms of transitions:

$N \in EFC$ iff $\forall t, u \in T. \bullet t \cap \bullet u \neq \emptyset \Rightarrow \bullet t = \bullet u$.
This condition says that $N$ may not contain what we call a pure $N$: places $p, q$ and transitions $t, u$ such that $p \in \cdot t \cap \cdot u$, $q \in \cdot u$ and $q \not\in \cdot t$.

In [3] it has been established that $FC \subsetneq EFC \subsetneq BFC$. In fact, the inclusions follow directly from the definitions, and Figure 4 displays counterexamples to strictness.

The class of free choice nets is strictly smaller than the class of symmetrically asynchronous nets respecting branching time equivalence, which in turn is strictly smaller than the class of behavioural free choice nets. The class of extended free choice nets and the class of symmetrically asynchronous nets respecting branching time equivalence are incomparable.

**Proposition 4.7** $FC \subsetneq SA(B) \subsetneq BFC$, $EFC \subsetneq SA(B)$ and $SA(B) \nsubseteq EFC$.

**Proof.** The first inclusion follows because a partially reachable $N$ is surely an $N$, and also the second inclusion follows directly from the definitions. The four inequalities follow from the examples in Figure 4. The first net is unmarked and thus trivially in $SA(B)$. The second ones symmetrically asynchronous implementation has the additional failure $<\varepsilon, \{a, b\}>$ and hence this net is not in $SA(B)$. $\square$

In Figure 5 the relations between our semantically defined net class $SA(B)$, the structurally defined classes $FC$, $EFC$, and the more behaviourally defined class $BFC$ are summarised. These relations may be interpreted as follows.

Starting at the top of the diagram, free choice nets are characterised structurally, enforcing that for every place, a token therein can choose freely (i.e. without inquiring about the existence of tokens in any other places) which outgoing arc to take. This property makes it possible to implement the system asynchronously. In particular, the component which holds the information represented by a token can choose arbitrarily when and into which of multiple asynchronous output channels to forward said information, without further knowledge about the rest of the system. As this decision is solely in the discretion of the sending component and not based upon any knowledge of the rest of the system, no synchronisation with other components is necessary.

The difference between $SA(B)$ and $FC$ is that in $SA(B)$ the quantification over the places is dropped, making the requirement more straightforward: Every token can choose freely which outgoing arc to follow. Thus, $SA(B)$ allows for non-free-choice structures as long as these never receive any tokens.

This also explains why $BFC$ includes $SA(B)$. Since $SA(B)$ guarantees that all tran-
itions of a problematic structure are never enabled, transitions in such structures are never enabled while others are disabled.

The incomparability between the left and the right side of the diagram stems from the conceptual allowance of slight transformations of the net before evaluating whether it is free choice or not. Extended free choice nets and behavioural free choice nets were proposed as nets that are easily seen to be behaviourally equivalent to free choice nets, and hence share some of their desirable properties: in [2,3] constructions can be found to turn any extended free choice net into an equivalent free choice net, and any behavioural free choice net into an extended free choice net.\footnote{In [2,3] the nature of the equivalence between the original and transformed net is not precisely specified. However, it can be argued that whereas the transformation from \(EFC\)-nets to \(FC\)-nets preserves branching time as well as causality, the transformation from \(BFC\)-nets to \(EFC\)-nets preserves branching time only: the third net of Figure 4 is interleaving bisimulation equivalent with its \(EFC\)-counterpart in Figure 6, but whereas the original net can perform the transitions \(a\) and \(c\) concurrently (in one step), the transformed net cannot.}

Applied on the last two nets in Figure 4 these constructions yield:

\[\]

For the second net of Figure 4, a \(\tau\)-transition is introduced, which collects both tokens and then marks a single postplace from which the two original transitions are enabled. Hence the choice between the two transitions is centralised in the newly introduced place and thus free again. In the definition of our symmetrically asynchronous implementation \(SI\), we do not allow any insertion of such “helping” \(\tau\)-transitions, as it seems unclear to us how much computing power should be allowed in possibly larger networks of such transitions. This becomes especially problematic if these networks somehow track part of the global status of the net inside themselves and thus make quite informed decisions about what outgoing transition to enable.
5 Asymmetric Asynchrony

As seen in the previous section, the class of symmetrically asynchronous nets is quite small. It precludes the implementation of many real-world behaviours, like waiting for one of multiple inputs to become readable, a Petri net representation of which will always include non free-choice structures.

Therefore we propose a less strict definition of asynchrony such that actions may depend synchronously on a single predetermined condition. In a hardware implementation the places which earlier could always forward a token into some silent transitions must now wait until they receive an explicit token removal signal from their posttransitions.

To this end we introduce a static priority over the preplaces of each transition. Every transition first removes the token from the most prioritised preplace and then continues along decreasing priority. To formalise this behaviour in a Petri net we insert a silent transition for each incoming arc of every transition. These silent transitions are forced to execute in sequence by newly introduced buffer places between them. In the final position of this chain, the original visible transition is executed. An example of this transformation is given in Figure 7.

**Definition 5.1** Let \( N = (S, O, \varnothing, F, M_0) \) be a plain net.

Let \( g \subseteq (S \times O) \times (S \times O) \) be a relation on \( F \cap (S \times O) \) such that for each \( t \in O \)
\( g \cap (\bullet t \times \{t\}) \) is a total order \( \leq^{t}_g \) over \( \bullet t \times \{t\} \).

We write \( \text{min}^t_g \) for the \( \leq^t_g \)-minimal element of \( \bullet t \) and \( (s - 1)^t_g \) for the next place in \( \bullet t \) that is \( \leq^t_g \)-smaller than \( s \).

We define a set of silent transitions as \( X := \{t_s \mid t \in O, s \in \bullet t\} \).

Let \( h : X \to X \cup O \) be the function

\[
h(t_s) = \begin{cases} 
  t & \text{iff } s = \text{min}^t_g \\
  t_s & \text{otherwise}
\end{cases}
\]
N ∈ AA(B)

As before, we are interested in the relationship between nets and their possible implementations. The definition of asymmetric asynchrony however allows different implementations for the same net. We define a net to be asymmetrically asynchronous if any of the possible implementations simulates the net sufficiently.

**Definition 5.2**

The class of asymmetrically asynchronous nets respecting branching time equivalence is defined as $AA(B) := \{ N \mid \exists g. AL_g(N) \approx_F N \}$.

As before, we have $\mathcal{F}(N) \subseteq \mathcal{F}(AL_g(N))$ for any plain net $N$ and any priority relation $g$ [8]. Additionally we would like to obtain a semi-structural characterisation of $AA(B)$ in the spirit of Theorems 3.4 and 4.4. Unfortunately we didn’t succeed in this, but we obtained structural upper and lower bounds for this net class.

**Definition 5.3**

A net $N = (S, O, \emptyset, F, M_0)$ has a left and right reachable $M$ iff $\exists t, u, v \in O \exists p \in t u \searrow u \searrow v: t \neq u \wedge u \neq v \wedge p \neq q \wedge \exists M_1, M_2 \in [M_0]. t u \cup u \subseteq M_1 \wedge v \cup u \subseteq M_2$.

A net $N = (S, O, \emptyset, F, M_0)$ has a left and right border reachable $M$ iff $\exists t, u, v \in O \exists q \in \searrow u \wedge v: t \neq u \wedge v = p \neq q \wedge \exists M_1, M_2 \in [M_0]. t \subseteq M_1 \wedge v \subseteq M_2$.

**Theorem 5.4**

A plain net $N$ in $AA(B)$ has no left and right reachable $M$.

A plain net $N$ which has no left and right border reachable $M$ is in $AA(B)$.

**Proof.** See [19] or [8].
Figure 8 shows two nets, each with a left and right border reachable M but no left and right reachable M, that thus fall in the grey area between our structural upper and lower bounds for the class $AA(B)$. In this case the first net falls outside $AA(B)$, whereas the second net falls inside. The crucial difference between these two examples is the information available to u about the execution of y.

There exists an implementation for the right net, namely by u taking the tokens from r, q and s in that order. The first token (from r) conveys the information that y was executed, and thus t is not enabled. Collecting the last token (from s) could fail, due to v removing it earlier. Even so, removing the tokens from r and q did not disable any transition that could fire in the original net. In the left net such an implementation will not work.

The following proposition says that our class of symmetrically asynchronous nets strictly extends the corresponding class of asymmetrically asynchronous nets.

**Proposition 5.5** $SA(B) \subset AA(B)$.

**Proof.** A net which has no partially reachable N also has no left or right border reachable M. The inequality follows from the example in Figure 3. \qed

As before, our class $AA(B)$ is related to some known net classes [3].

**Definition 5.6** Let $N = (S, O, \emptyset, F, M_0)$ be a plain net.

(i) $N$ is simple, $N \in SPL$, iff $\forall p, q \in S. p \neq q \land p^* \cap q^* \neq \emptyset \Rightarrow |p^*| = 1 \lor |q^*| = 1$.

(ii) $N$ is extended simple, $N \in ESPL$, iff $\forall p, q \in S. p^* \cap q^* \neq \emptyset \Rightarrow p^* \subseteq q^* \lor q^* \subseteq p^*$.

Extended simple nets appear in [2] under the name asymmetric choice systems. Note that simple is equivalent to $M$-free, where $M$ is as in Definition 5.3 but without the reachability clauses. Clearly, we have $FC \subset SPL \subset ESPL$ and $EFC \subset ESPL$, whereas $EFC \not\subset SPL$ and $SPL \not\subset EFC$: the inclusions follow immediately from the definitions, and the first two nets of Figure 4 provide counterexamples to the inequalities.

The class of asymmetrically asynchronous nets respecting branching time equivalence strictly extends the class of simple nets, whereas it is incomparable with the class of extended simple nets.

**Proposition 5.7** $SPL \subset AA(B)$, $AA(B) \not\subset ESPL$ and $ESPL \not\subset AA(B)$.

**Proof.** The inclusion is straightforward, and the inequalities follow from the counterexamples in Figure 4 (the second one) and Figure 9. The missing tokens in the latter example are intended. As no action is possible there will not be any additional implementation failures. \qed

Fig. 9. $N \in AA(B)$, $N \notin ESPL$
Fig. 10. Overview of asymmetric-choice-like net classes

The relations between the classes $SPL$, $ESPL$ and $AA(B)$ are summarised in Figure 10. Similarly to what we did in Section 4, we now try to translate Figure 10 into an intuitive description.

The basic intuition behind $SPL$ is that for every transition there is only one preplace where conflict can possibly occur. Whereas in $SPL$ that possibility is determined by the static net structure, in $AA(B)$ reachability is also considered.

Similar to the difference between $FC$ and $EFC$ there exists a difference between $ESPL$ and $SPL$ which originates from the fact that $ESPL$ allows small transformations to a net before testing whether it lies in $SPL$. Again our class $AA(B)$ does not allow such “helping” transformations.

6 Conclusion and Related Work

We have investigated the effect of different types of asynchronous interaction, using Petri nets as our system model. We propose three different interaction patterns: fully asynchronous, symmetrically asynchronous and asymmetrically asynchronous. An asynchronous implementation of a net is then obtained by inserting silent (unobservable) transitions according to the respective pattern. The pattern for asymmetric asynchrony is parametric in the sense that the actual asynchronous implementation of a net depends on a chosen priority function on the input places of a transition. For each of these cases, we investigated for which types of nets the asynchronous implementation of a net changes its behaviour with respect to failures equivalence (in the case of asymmetric asynchrony, the ‘best’ priority function may be used). It turns out that we obtain a hierarchy of Petri net classes, where each class contains those nets which do not change their behaviour when transformed into the asynchronous version according to one of the interaction patterns. This is not surprising because later constructions allow a more fine-grained control over the interactions than earlier ones.

We did not consider connections from transitions to their postplaces as relevant to determine asynchrony and distributability. This is because we only discussed contact-free nets, where no synchronisation by postplaces is necessary. In the spirit of Definition 3.1 we could insert $\tau$-transitions on any or all arcs from transitions to their postplaces, and the resulting net would always be equivalent to the original.

Although we compare the behaviour of a net and its asynchronous implementations in terms of failures equivalence, we believe that the very same classes of nets are obtained when using any other reasonable behavioural equivalence that respects branching time to some degree and abstracts from silent transitions—no matter if
this is an interleaving equivalence, or one that respects causality. We would get larger classes of nets, for example for the case of full asynchrony including the net of Figure 2, if we merely required a net $N$ and its implementation to be equivalent under a suitably chosen linear time equivalence. This option is investigated in [19].

The central results of the paper give semi-structural characterisations of our semantically defined classes of nets. Moreover, we relate these classes to well-known and well-understood structurally defined classes of nets, like free choice nets, extended free choice nets and simple nets.

To illustrate the potential interpretation of our results in other models of distributed systems, we give an example.

Message sequence charts (MSCs), also contained in UML 2.0 under the name sequence diagrams, are a model for specifying interactions between components (instances) of a system. A simple kind are basic message sequence charts (BMSCs) as defined in [13], where choices are not allowed. A Petri net semantics of BMSCs with asynchronous communication and a unique sending and receiving event for each message will yield Petri nets with unbranched places (see for instance [10]). Hence in this case the resulting Petri nets are conflict-free and therefore fully asynchronously implementable according to Theorem 3.4.

However in extended versions of MSCs, e.g. in UML 2.0 or in live sequence charts (LSCs, see [11]), inline expressions allow to describe choices between possible behaviours in MSCs. Consider for example the MSC given in Figure 11 and a naive Petri net representation. The instances i1 and i2 can either communicate or execute their local actions. Obviously, this requires some mechanism in order to make sure that the choice is performed in a coherent way (see e.g. [7] for a discussion of this type of problem). In the Petri net representation, we find a reachable $N$, hence with Theorem 4.4 the net does not belong to the class $SA(B)$ of symmetrically asynchronously implementable nets. However, the net is $M$-free, and thus does belong to the class $AA(B)$ of asymmetrically asynchronously implementable nets. By giving priority to the collection of the message token (choosing the appropriate function $g$ in our notion of implementation), it can be assured that instance i2 does not make the wrong choice and gets stuck (however it is still not clear whether the message will actually be consumed).

The obvious question is whether the naive Petri net interpretation we have given is conform with the intended semantics of the *alt*-construct (according to the informal
UML semantics the alternatives always have to be executed completely; in LSCs it is specified explicitly whether messages are assured to arrive. However, on basis of a maybe more elaborate Petri nets semantics, it could be discussed what types of MSCs can be used to describe physically distributed systems, in particular which type of construct for choices is reasonable in this case.

Another model of reactive systems where we can transfer our results to are process algebras. When giving Petri net semantics to process algebras, it is an interesting question to investigate which classes of nets in our classification are obtained for certain types of operators or restricted languages, and to compare the results with results on language hierarchies (as summarised below).

We now give an overview on related work. A more extensive discussion is contained in [19]. We start by commenting on related work in Petri net theory.

The structural net classes we compare our constructions to were all taken from [3], where Eike Best and Mike Shields introduce various transformations between free choice nets, simple nets and extended variants thereof. They use “essential equivalence” to compare the behaviour of different nets, which they only give informally. This equivalence is insensitive to divergence, which is also relied upon in their transformations. As observed in Footnote 1, it also does not preserve concurrency. They continue to show conditions under which liveness can be guaranteed for some of the classes.

In [1], Wil van der Aalst, Ekkart Kindler and Jörg Desel introduce two extensions to extended simple nets, by allowing self-loops to ignore the discipline imposed by the ESPL-requirement. This however assumes a kind of “atomicity” of self-loops, which we did not allow in this paper. In particular we do not implicitly assume that a transition will not change the state of a place it is connected to by a self-loop, since in case of deadlock, the temporary removal of a token from such a place might not be temporary indeed.

In [18] Wolfgang Reisig introduces a class of systems which communicate using buffers and where the relative speeds of different components are guaranteed to be irrelevant. The resulting nets are simple nets. He then proceeds introducing a decision procedure for the problem whether a marking exists which makes the complete system live.

The most similar work to our approach we have found is [12], where Richard Hopkins introduces the concept of distributable Petri Nets. These are defined in terms of locality functions, which assign to every transition \( t \) a set of possible machines or locations \( L(t) \) on which \( t \) may be executed, subject to the restriction that a set of transitions with a common preplace must share a common machine. A plain net \( N \) is distributable iff for every locality function \( L \) that can be imposed on it, it has a “distributed implementation”, a \( \tau \)-net \( N' \) with the same set of visible transitions, in which each transition is assigned a specific location, subject to three restrictions:

- the location of a visible transition \( t \) is chosen from \( L(t) \),
- transitions with a common preplace must have the same location
- and there exists a weak bisimulation between \( N \) and \( N' \), such that all \( \tau \)-transitions involved in simulating a transition \( t \) from \( N \) reside on one of the locations \( L(t) \).
The last clause enforces both a behavioural correspondence between $N$ and $N'$ and a structural one (through the requirement on locations). Thus, as in our work, the implementation is a $\tau$-net that is required to be behaviourally equivalent to the original net. However, whereas we enforce particular implementations of an original net, Hopkins allows implementations which are quite elaborate and make informed decisions based upon global knowledge of the net. Consequently, his class of distributable nets is larger than our asynchronous net classes. As Hopkins notes, due to his use of interleaving semantics, his distributed implementations do not always display the same concurrent behaviour as the original nets, namely they add concurrency in some cases. This does not happen in our asynchronous implementations.

Another branch of related work is in the context of distributed algorithms. In [5] Luc Bougé considers the problem of implementing symmetric leader election in the sublanguages of CSP obtained by either allowing all guards, only input guards or no communication guards at all in guarded choice. He finds that the possibility of implementing it depends heavily on the structure of the communication graphs, while truly symmetric schemes are only possible in CSP with input and output guards.

Quite a number of papers consider the question of synchronous versus asynchronous interaction in the realm of process algebras and the $\pi$-calculus. In [4] Frank de Boer and Catuscia Palamidessi consider various dialects of CSP with differing degrees of asynchrony. In particular, they consider CSP without output guards and CSP without any communication based guards. They also consider explicitly asynchronous variants of CSP where output actions cannot block, i.e. asynchronous sending is assumed. Similar work is done for the $\pi$-calculus in [17] by Catuscia Palamidessi, in [16] by Uwe Nestmann and in [9] by Dianele Gorla. A rich hierarchy of asynchronous $\pi$-calculi has been mapped out in these papers. Again mixed-choice, i.e. the ability to combine input and output guards in a single choice, plays a central role in the implementation of truly synchronous behaviour. It would be interesting to explore the possible connections between these languages and our net classes.

In [20], Peter Selinger considers labelled transition systems whose visible actions are partitioned into input and output actions. He defines asynchronous implementations of such a system by composing it with in- and output queues, and then characterises the systems that are behaviourally equivalent to their asynchronous implementations. The main difference with our approach is that we focus on asynchrony within a system, whereas Selinger focusses on the asynchronous nature of the communications of a system with the outside world.

Finally, there are approaches on hardware design where asynchronous interaction is an intriguing feature due to performance issues. For this, see the papers [14] and [15] by Leslie Lamport. In [15] he considers arbitration in hardware and outlines various arbitration-free “wait/signal” registers. He notes that nondeterminism is thought to require arbitration, but no proof is known. He concludes that only marked graphs can be implemented using these registers. Lamport then introduces “Or-Waiting”, i.e. waiting for any of two signals, but has no model available to characterise the resulting processes. The used communication primitives bear a striking similarity to our symmetrically asynchronous nets.
References


Counting the cost in the picalculus (Extended Abstract)

Matthew Hennessy $^{1,2}$

Department of Computer Science
Trinity College Dublin
Dublin, Ireland

Manish Gaur$^3$

Department of Informatics
University of Sussex
Brighton, UK

Abstract

We design a new variation on the picalculus, $\pi_{\text{cost}}$, in which the use of channels or resources must be paid for. Processes operate relative to a cost environment, and communications can only happen if principals have provided sufficient funds for the channels associated with the communications.

We define a bisimulation-based behavioural preorder in which two processes are related if, intuitively, they exhibit the same behaviour but one may be more efficient than the other. We justify our choice of preorder by proving that it is characterised by three intuitive properties which behavioural preorders should satisfy in a framework in which the use of resources must be funded.

Keywords: picalculus, resources, cost bisimulations

1 Introduction

The picalculus [20] is a basic abstract formal language for describing communicating processes and has a very developed behavioural theory [28], expressed as an equivalence relation between process descriptions; $P \approx Q$ signifies that, although $P$ and $Q$ may be intentionally very different they offer essentially the same behaviour to users.

The basic language and its related theory has been extended in myriad ways in order to incorporate many different aspects of concurrent behaviour [1,26,8]. In one family of

---

1 This work is a part of the doctoral studies of the second author and is supported by Commonwealth Scholarship Commission UK (Ref: INCS-2005-145). The support of SFI, Ireland is also acknowledged.
2 Email: Matthew.Hennessy@cs.tcd.ie
3 Email: M.H.C.Gaur@sussex.ac.uk
extensions the judgements of the behavioural theory take the form

\[ \Gamma \models P \approx Q \]  

(1)

where \( \Gamma \) represents some aspect of the infrastructure in which the processes \( P, Q \) operate. The primary example, initiated in [25], is when \( \Gamma \) is a type environment describing the type of the communicating channels used by \( P, Q \). But in [9,23] it represents the state of the underlying network, recording for example the current connectivity between the sites at which processes execute, or the failures which have occurred.

In this short paper we show how this framework, in particular the version from [13,11], can also be adapted to develop a theory in which there is a cost associated with the resources used in a computation. Here \( \Gamma \) will represent a cost environment, which could record for example the cost of using particular channels or resources, the current funds available to the various principals involved, and could also keep a tally of the total funds which have been expended so far. Indeed if the latter is included in the notion of a cost environment then (1) could be adapted to judgements of the form

\[ \Gamma \models P \preccurlyeq Q \]  

(2)

meaning informally that, relative to the cost environment \( \Gamma \), processes \( P \) and \( Q \) offer essentially the same behaviour to users, but that \( Q \) is as least as efficient as \( P \), and possibly more efficient.

We envisage two immediate applications for these ideas. The first is web services, [2]. In [17,6] a basic theory of contracts for web services is introduced, based on a variation of CCS, [19]. Our use of cost environments could immediately be applied here, and indeed we intend to pursue this line of work in future publications. The second is in the development of a more realistic theory of networked processes. Communication across a network is not instantaneous; by introducing some representation of routers into the process description language, we can associate as the cost of a communication the number of routers through which the message has to travel. This is pursued in [10].

The current paper seeks to lay the foundations for a theory of costed process behaviour. In Section 2 we describe a very simple variation on the (asynchronous) \( \pi \)-calculus, which we call \( \pi_{\text{cost}} \), in which channels are viewed as resources, as in [6], but which can only be used if sufficient funds are available. The reduction semantics is relative to a cost environment, so that the judgements are of the form

\[ \Gamma_1 \triangleright P_1 \rightarrow \Gamma_2 \triangleright P_2 \]

We refer to the pairs \((\Gamma_i \triangleright P_i)\) as systems. The rules governing the judgements are minor variations on those used in the standard reduction semantics for the (asynchronous) \( \pi \)-calculus; it turns out that the rules only depend on three high-level operations on cost environments. However we also give a concrete instantiation of cost environment which supports these operations.

In Section 4 we define a labelled transition system for \( \pi_{\text{cost}} \), and use the resulting actions to define the relation referred to above, (2), using a (minor) variation on the standard definition of (asynchronous) bisimulation equivalence, [14,4,28]. We claim that this does indeed form the basis for an adequate theory of costed process behaviour. To support this claim
we offer one theorem, Theorem 4.4, which says that this relation is completely determined by three natural properties of behavioural relations between systems. These properties are outlined in Section 3, and the main ingredient is the manner in which processes are observed, in particular who pays the cost of performing observations. The paper ends with some remarks on related and future work.

2 The language $\pi_{\text{cost}}$

We assume a set of channel or resource names $\text{Chan}$, ranged over by $a, b, c, \ldots$ whose use requires some cost. As already stated we have two examples in mind. The first is where these names actually represent web services, as in [6], and the second is where they represent the transmission of data through routers in a distributed network. We also assume a set of principals or owners $\text{Own}$, ranged over by $o$, who register for these resources and pay for their use. The syntax of $\pi_{\text{cost}}$ is then given in Figure 1, and is essentially a very minor extension to the picalculus; the meta-variables $u, v$ range over identifiers, which are either resource names $a \in \text{Chan}$, or variables $x$ from a distinct set $\text{Var}$. We employ the standard abbreviations associated with the picalculus, and associated terminology.

Since resource usage incurs a cost, the execution of processes is now relative to a cost environment $\Gamma$; this records which owners are registered for which resources, and both the costs required to use resources, and the effect of actually using them. Thus judgements of the reduction semantics take the form

$$\Gamma_1 \triangleright P_1 \rightarrow \Gamma_2 \triangleright P_2$$

where $P_i$ are processes, that is closed terms from $\pi_{\text{cost}}$, and $\Gamma_i$ represent cost environments.

There are many possibilities for cost environments, and we will provide a particular instance shortly. But no matter how they are defined, we need to be able to define at least three operations on them:

- **resource charging**: $\Gamma_1 \xrightarrow{a} \Gamma_2$ means that relative to $\Gamma_1$ sufficient funds are available for
the use of resource $a$, and if it is used, the consumption of appropriate funds is recorded in the transformation from $\Gamma_1$ to $\Gamma_2$.

- **resource subscription**: $\Gamma_1 \xrightarrow{\text{sub}(o,a,c)} \Gamma_2$ records the effect of allowing owner $o$ to subscribe, with the funds $c$, to the resource $a$.

- **resource registration**: $\Gamma_1, a:R \xrightarrow{\text{new } b:R} \Gamma_2$ records the result of extending $\Gamma$ with a new resource named $a$, with the information contained in the type $R$. In this paper these types will take the form $\langle R_a, R_b \rangle$, where $R_a$ is a usage cost, and and $R_b$ records the amount of funds which owners have allocated to the resource.

Relative to these operations, the reduction semantics for $\pi_{\text{const}}$ is then defined as the least relation which satisfies the rules in Figure 2. This uses the standard structural equivalence between process terms of the picalculus which is recalled in Figure 3.

The idea behind this semantics is that $a!(v).P$ is a request to use the service $a$ with parameter $v$; so with rule (r-out) it spawns an atom $\text{del}(a, v)$, which is implicitly delivered through the network to the site of the resource $a$. In (r-comm) this request is satisfied, at least if the cost environment allows it, that is $\Gamma_1 \xrightarrow{v} \Gamma_2$. We have not yet actually specified this relation, but one would also expect it to record the cost of this request. Most of the remaining rules are standard from the picalculus, but the novel (r-subscribe) allows an owner to subscribe to a channel, that is allocate funds for the use of the channel.

But the final rule (r-new) is non-standard. In $\Gamma_1 \xrightarrow{\text{new } b:R} P$, the process $P$ may evolve by using the internal resource $b$. In general this requires the expenditure of funds, and therefore will effect funds available for any subsequent use of $b$. This is reflected in the
The required operations are defined as follows:

\[(\text{s-scope} - \text{extrusion}) \quad (\text{new}\ a:R)(P \mid Q) \equiv P \mid (\text{new}\ a:R)\ Q \quad \text{if}\ a \notin \text{fn}(P)\]

\[(\text{s-monoind} - \text{com}) \quad P \mid Q \equiv Q \mid P\]

\[(\text{s-monoind} - \text{assoc}) \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)\]

\[(\text{s-monoind} - \text{id}) \quad P \mid \text{stop} \equiv P\]

\[(\text{s-new} - \text{flip}) \quad (\text{new}\ a:R)(\text{new}\ b:S)\ P \equiv \text{new}\ b:S\ \text{new}\ a:R\ P \quad \text{if}\ a \neq b\]

\[(\text{s-new} - \text{co}) \quad (\text{new}\ a:R)\ P \equiv P \quad \text{if}\ a \notin \text{fn}(P)\]

\[(\text{s-rec}) \quad *P \equiv P \mid *P\]

Fig. 3. Structural equivalence of \(n\)-Cost

change of the type, from \(R\) to \(R'\); the possible values for \(R'\) are deduced by examining the possible evolution of \(P\) relative to the extended cost environment \(\Gamma_1, b:R\).

For the remainder of the paper we take a cost environment \(\Gamma\) to consist of a 4-tuple \(\langle \Gamma^c, \Gamma^o, \Gamma^s, \Gamma^r \rangle\) where

- \(\Gamma^c: \text{Chan} \rightarrow N^{\infty}\)
  \(\Gamma^c(a)\) records the cost of using the resource \(a\); since \(\Gamma^c\) is a partial function it also implicitly records the valid resources known to \(\Gamma\), namely \(\text{dom}(\Gamma^c)\).

- \(\Gamma^o: \text{Own} \rightarrow N^{\infty}\)
  For an owner \(o \in \text{Own}\), \(\Gamma^o(o)\) records the (unsubscribed) funds which \(o\) has in the system. These are available for \(o\) to allocate to particular resources, via the \(\text{subscribe}(o, a, c)\) command.

- \(\Gamma^s: \text{Chan} \rightarrow (\text{Own} \rightarrow N^{\infty})\)
  \(\Gamma^s(a)\) records the subscriptions that owners have on resource \(a\); since \(\Gamma^s(a)\) is a partial function it also implicitly records the owners registered to use \(a\), namely \(\text{dom}(\Gamma^s(a))\). We also use \(\Sigma(\Gamma^s(a))\) to denote \(\sum\{\Gamma^s(a)(o) \mid o \in \text{dom}(\Gamma^s(a))\}\), the entire funds available for the use of the resource \(a\).

- \(\Gamma^r: N^{\infty}\)
  This is a record of the cost which has already been expended by the system.

The required operations are defined as follows:

- **resource charging**: informally \(\Gamma_1 \xrightarrow{\text{cost}} \Gamma_2\) if there are sufficient funds subscribed to \(a\) in \(\Gamma_1\) to cover the costs of using it, and \(\Gamma_2\) records their consumption. Formally it holds when
  \[
  \cdot |\Gamma^s_2(a)| = |\Gamma^s_1(a)| - \Gamma^c_1(a)
  \cdot \Gamma^s_2 = \Gamma^s_1 + \Gamma^s_1(a)
  \cdot \Gamma^r_2 = \Gamma^r_1, \ \Gamma^o_2 = \Gamma^o_1, \text{ and } \Gamma^s_2(b) = \Gamma^s_1(b) \text{ whenever } b \neq a.
  \]
  Note that here no record is kept of which owners actually contributed to this particular use of the resource \(a\).

- **resource subscription**: Intuitively \(\Gamma_1 \xrightarrow{\text{subscribe}(o, a, c)} \Gamma_2\) if \(\Gamma_2\) can be constructed from \(\Gamma_1\), by decreasing \(\Gamma^o(o)\) by \(c\), and increasing \(\Gamma^s_1(a)(o)\) by the same amount. Formally it holds when
  \[
  \cdot o \in \text{dom}(\Gamma^s_1(a)); \text{ that is } o \text{ is actually registered to use resource } a
  \cdot \Gamma^o_2(o) = \Gamma^o_1(o) - c
  \]
\[
\begin{align*}
\cdot \quad \Gamma'_2(a)(o) &= \Gamma'_1(a)(o) + c \\
\cdot \quad \Gamma'_2 = \Gamma'_1, \quad \Gamma'_2 = \Gamma'_1, \quad \text{and} \quad \Gamma'_2(o') &= \Gamma'_1(o') \quad \text{whenever} \quad o' \neq o, \quad \text{and} \quad \Gamma'_2(b) &= \Gamma'_1(b) \quad \text{for every other} \quad b \quad \text{different from} \quad a.
\end{align*}
\]

- **resource registration**: The cost environment \( \Gamma, a : R \), is only defined if \( a \) is fresh to \( \Gamma \), that is, if \( a \) is not in \( \text{dom}(\Gamma) \). In this case it gives the new cost environment \( \Phi \) defined by

\[
\begin{align*}
\cdot \quad \Phi'(b) &= \begin{cases} 
\Gamma'(b) & \text{if} \quad b \in \text{dom}(\Gamma) \\
R_c & \text{if} \quad a = b \\
\end{cases} \\
\cdot \quad \Phi'(b) &= \begin{cases} 
\Gamma'(b) & \text{if} \quad b \in \text{dom}(\Gamma) \\
R_c & \text{if} \quad a = b \\
\end{cases}
\]

So we require \( R_c \) to be a partial function in \( (\text{Own} \rightarrow N^\omega) \). Note that this also implicitly defines the set of owners registered to use the new channel \( a \), namely \( \text{dom}(R_c) \).

- \( \Phi' \) and \( \Phi'' \) are taken to be \( \Gamma' \) and \( \Gamma'' \) respectively.

The pair \( (\Gamma \triangleright P) \) is called a **system** if \( \text{fn}(P) \subseteq \text{dom}(\Gamma') \), that is every free resource name in \( P \) is known to the **cost environment** \( \Gamma \). We use \( S \) to denote the set of all systems.

**Proposition 2.1**  If \((\Gamma_1 \triangleright P_1) \) is a system and \((\Gamma_1 \triangleright P_1) \rightarrow (\Gamma_2 \triangleright P_2) \) then \((\Gamma_2 \triangleright P_2) \) is also a system. \( \square \)

Reductions in a system affects it’s cost environment, and as a sanity check we can describe precisely the kinds of changes which are possible:

**Proposition 2.2**  Suppose \((\Gamma_1 \triangleright P_1) \rightarrow (\Gamma_2 \triangleright P_2) \). Then

- \( \Gamma_1 = \Gamma_2 \)
- or \( \Gamma_1 \overset{a}{\longrightarrow} \Gamma_2 \), for some resource \( a \)
- or \( \Gamma_1 \overset{\text{subjcost}}{\rightarrow} \Gamma_2 \), for some resource \( a \), owner \( o \) and cost \( c \). \( \square \)

### 3 Observing systems

Here we adapt the standard theory of reduction barbed congruence, \([14, 28, 13, 11]\), to \( \pi_{\text{cost}} \).

The theory enables one to say that relative to an environment \( \Gamma \) the processes \( P_1 \) and \( P_2 \) are observationally equivalent. We modify this in two ways. In the first we will actually relate systems, \( \Gamma_1 \triangleright P_1 \) and \( \Gamma_2 \triangleright P_2 \), thereby enabling us to compare, for example, the same process running under different cost environments. Secondly, because our cost environments accumulate expenditure we will be able to define what it means for one system to be more efficient than another, while offering similar observational behaviour to observers:

\((\Gamma_1 \triangleright P_1) \preceq_{\text{cbp}} (\Gamma_2 \triangleright P_2) \).

**Observations:**

There is lots of scope for defining what it means to observe processes in scenarios where communication, and therefore observation, must be paid for. In this preliminary paper we take a simple approach, in which the observations of a system \((\Gamma \triangleright P)\) are paid for by the funds available within the cost environment \( \Gamma \); in other words observers are allowed access to the funds available in \( \Gamma \).

Because \( \pi_{\text{cost}} \) is based on the asynchronous picalculus it turns out that only one kind of observable is required. Intuitively \((\Gamma \triangleright P) \downarrow^c \text{del}(a) \) means that it will cost the system at
most $c$ for an observer to be assured that some value can be delivered to the resource $a$.

First let us define strong observations. We write $(\Gamma \triangleright P) \Downarrow^c \del(a)$ whenever

- $P \equiv (\new \tilde{b} : (\del(a, v) \mid Q)$, where $a$ does not occur in $\tilde{b}$
- $\Gamma \not\xrightarrow{\cdot} \Gamma'$ for some $\Gamma'$
- $\Gamma^c(a) \leq c$

So this means that an observer can immediately obtain some value on resource $a$, and the cost of obtaining it is at most $c$. Then weak observations are defined by letting

$$(\Gamma \triangleright P) \Downarrow \del(a)$$

whenever $(\Gamma \triangleright P) \xrightarrow{\cdot}^* (\Phi \triangleright Q)$ where $(\Phi \triangleright Q) \Downarrow^d$, for some $d$ such that $d + (\Gamma' - \Gamma') \leq c$.

Here the total cost to the system is still at most $c$, taking into account the cost required to get to the state where the actual (strong) observation can be made.

We say that a relation $R \subseteq S \times S$ is observation improving if, whenever $S_1 \triangleright R \triangleright S_2$, $S_1 \Downarrow^c \del(a)$ implies $S_2 \Downarrow^c \del(a)$.

Intuitively this means that any observation made on the system $S_1$ can be made on $S_2$ for a possibly smaller cost.

**Contextual:**

A relation $R \subseteq S \times S$ is called contextual if

(i) $(\Gamma_1 \triangleright P_1) \not\xRightarrow{R} (\Gamma_2 \triangleright P_2)$ implies $(\Gamma_1 \triangleright P_1 \mid O) \not\xRightarrow{R} (\Gamma_2 \triangleright P_2 \mid O)$, whenever $(\Gamma_1 \triangleright P_1 \mid O)$ and $(\Gamma_2 \triangleright P_2 \mid O)$ are both systems

(ii) $(\Gamma_1 \triangleright P_1) \not\xRightarrow{R} (\Gamma_2 \triangleright P_2)$ implies $(\Gamma_1, a : R \triangleright P_1) \not\xRightarrow{R} (\Gamma_2, a : R \triangleright P_2)$, whenever $a$ is fresh to $\Gamma_i$.

**Reduction cost improving:**

A relation $R \subseteq S \times S$ is called reduction cost improving if, whenever $(\Gamma_1 \triangleright P_1) \not\xRightarrow{R} (\Gamma_2 \triangleright P_2)$

(i) $(\Gamma_1 \triangleright P_1) \rightarrow^* (\Gamma'_1 \triangleright P'_1)$ implies $(\Gamma_2 \triangleright P_2) \rightarrow^* (\Gamma'_2 \triangleright P'_2)$ for some system $(\Gamma'_2 \triangleright P'_2)$ such that $(\Gamma'_2 - \Gamma'_2) \leq (\Gamma'_2 - \Gamma'_2)$ and $(\Gamma'_1 \triangleright P'_1) \not\xRightarrow{R} (\Gamma'_2 \triangleright P'_2)$.

(ii) Conversely $(\Gamma_2 \triangleright P_2) \rightarrow^* (\Gamma'_2 \triangleright P'_2)$ implies $(\Gamma_1 \triangleright P_1) \rightarrow^* (\Gamma'_1 \triangleright P'_1)$ for some system $(\Gamma'_1 \triangleright P'_1)$ such that $(\Gamma'_1 - \Gamma'_1) \leq (\Gamma'_1 - \Gamma'_1)$ and $(\Gamma'_1 \triangleright P'_1) \not\xRightarrow{R} (\Gamma'_2 \triangleright P'_2)$.

Here $(\Gamma'_2 - \Gamma'_2)$ represents the cost of doing the reduction $(\Gamma_1 \triangleright P_1) \rightarrow (\Gamma'_1 \triangleright P'_1)$. So $(\Gamma_1 \triangleright P_1) \not\xRightarrow{R} (\Gamma_2 \triangleright P_2)$ means that the systems can mimic each other’s reductions, but the reductions from $(\Gamma_2 \triangleright P_2)$ are no more expensive, and possibly cheaper, than those from $(\Gamma_1 \triangleright P_1)$.

**Definition 3.1 Cost barbed precongruence:**

Let $\ll_{cbp} \subseteq S \times S$ be the largest relation which is

(i) observation improving

(ii) contextual

(iii) reduction cost improving.
The main result of the paper is a non-contextual purely coinductive characterisation of this observational preorder between systems.

4 Bisimulation equivalence for $\pi_{\text{cost}}$

In Figure 4 we give a set of rules for deriving judgements of the form $(\Gamma_1 \triangleright P_1) \xrightarrow{\mu} (\Gamma_2 \triangleright P_2)$, where $\mu$ can take one of the forms

(i) internal action, $\tau$:

(ii) input, $a?b$, $(b:R)a?b$: input by resource $a$ of a known or fresh name, respectively

(iii) output: $\text{del}(a,b)$, $(b:R)\text{del}(a,b)$: delivery of known or fresh name, respectively, to resource $a$

(iv) external subscription, $\text{sub}(o,a,c)$: subscription by owner $a$ to resource $a$

(v) external consumption, $\tau_o$: use by some external entity of resource $a$.

We use $\alpha$ to range over the free actions $a?b$ or $\text{del}(a,b)$, and in the rules we employ the standard complementary notation for them, $\overline{\alpha}$ denoting the complement of $\alpha$. For convenience, we sometimes use $(b:R)\alpha$ to denote an arbitrary action; $\alpha$ is considered to be a degenerate instance of $(b:R)\alpha$, where the sequence $(b:R)$ is empty. We will also assume, as usual, that all bound names are fresh in the context in which they are used.

Many of the rules are a very simple modification of those used in the standard action semantics for the asynchronous $\pi$-calculus, to take into account the presence of cost environments. Resource charging $\Gamma \xrightarrow{\mu} \Gamma'$ is required for both input (l-in) and delivery (l-del). Resource registration is required in (l-open), as is usual for the $\pi$-calculus, but also in (l-cntx) because of the effect that internal moves may have on resource types. The rule (l-asy) is required because our language is asynchronous. Intuitively it represents an attempt by a user to observe a process $P$ performing the input action $a?v$, by sending it the package $\text{del}(a,v)$. This is ignored by $P$, and the resulting system is $P|\text{del}(a,v)$. Note it does not require any intervention of the cost environment; intuitively a request has been made to the resource $a$, but is has not yet been serviced. The use of (l-asy) has been discussed at length in [11], and was originally suggested in [14].

There are two novel actions which take into account the indirect effect that observers may have on the cost environment of a system. The first, (l-ext.subscribe), models some observer adding some funds to the resource $a$, while (l-ext.comm) is required to take into account the use of a resource $a$ by some external party.

We can perform a number of sanity checks on these rules. For example one can show that if $(\Gamma_1 \triangleright P_1) \xrightarrow{\mu \text{reg}} (\Gamma_2 \triangleright P_2)$ Then $\Gamma_2 = \Phi, b:R$ for some $\Gamma_2$ such that $\Gamma_1 \xrightarrow{\mu} \Phi$, where $a$ is the channel used in $\alpha$. In fact the cost to the system of performing this action is precisely the cost of using this channel: $\Phi' - \Gamma_1' = \Gamma_1'(a)$.

The actions also preserve systems:

**Proposition 4.1** If $(\Gamma_1 \triangleright P_1)$ is a system and $(\Gamma_1 \triangleright P_1) \xrightarrow{\mu} (\Gamma_2 \triangleright P_2)$ then $(\Gamma_2 \triangleright P_2)$ is also a system. \hfill $\square$

As another sanity check we can relate the internal actions of the action semantics of Figure 4 with the reduction semantics of Figure 2. We lift the structural equivalence from processes to systems by writing $\Gamma_1 \triangleright P_1 \equiv \Gamma_2 \triangleright P_2$ to mean $P_1 \equiv P_2$ and $\Gamma_1 = \Gamma_2$. 

8
Lemma 4.2

(i) $S_1 \rightarrow S_2$ implies $S_1 \xrightarrow{i} S'_2$ for some $S'_2 \equiv S_2$

(ii) $S_1 \xrightarrow{i} S_2$ implies either $S_2 \equiv S_1$ or $S_1 \rightarrow S_2$.  

The bisimulation equivalence is defined using a slight abstraction from these judgements. As usual we ignore the type information on the freshly exported resource names [13], but more importantly we explicitly record the cost of actions:
In this short paper we have shown how the well-established theory of typed bisimulation equivalence for the picalculus can be easily adapted to provide an adequate theory of costed process behaviour, in which actions can only be performed if there are sufficient funds available to pay for them. Moreover the theory is relatively independent of the precise details of the cost environment relative to which computations take place.

We intend to pursue this line of work in two directions. In the first we wish to apply it to the various calculi being developed for web services, such as those in [17,6], and to see to what extent practical examples can be treated. In the second, more theoretical, we intend to revisit the idea of observing costed processes, as discussed in Section 3. There we assumed
that observers of a system had access to the funds of the system; a more realistic point of view would be that observers were required to provide themselves the funds necessary to perform observations. This change should have some implications for the required labelled transition system, but at the moment their extent is unclear.

There is already a considerable literature on topics related to this line of research. For example in [16] an efficiency preorder is defined between CCS processes; here the cost, or speed of a (weak) action simply depends on the number of internal moves it contains. Interesting properties of this preorder were further studied in [21]. In [15] a cost is associated with a subset of actions (which can not be synchronised) and a theory of amortised bisimulations is developed in this framework. Here amortised refers to the fact that the cost of each individual action is not compared; instead it is the overall cost which counts, where the high cost of one action may be compensated for by another at a low cost. It should be possible to develop amortised bisimulations for πcost; but an interesting theoretical question is how the resulting equivalence can be justified in terms of observations. Faster than preorders between processes have also been developed in work on timed process algebras; see [18] for an attempt at unifying different approaches.

In [3] there is a slightly different notion of the cost of a computation. The setting is mobile ambients [5] and the cost of computation is in terms of space consumption; essentially mobile agents can only migrate if the target location has sufficient capacity to accommodate it. Finally in [7] (and related publications such as [27]) a quite general theory of resource-based computation is being developed. The setting is SCCS [22], but the operational semantics is with respect to a resource. The generality is obtained by only requiring certain operations on the resource; in effect their use of resource is very similar to our use of cost environments, although the required operations are quite different. However they also have resource based modal logic for expressing properties of processes. The interesting point about the logic, a variation on Hennessy-Milner logic [12], is that satisfiability is resource dependent, being based on the bunched logic of [24]. It would be interesting to see if a similar logic could be developed for πcost.

References


Secrecy for rewriting in weakly adhesive categories

Tobias Heindel
Abteilung für Informatik und angewandte Kognitionswissenschaft
Universität Duisburg-Essen
Duisburg, Germany

Abstract
Inspired by the scope extrusion phenomenon of name passing calculi that allow to reason about knowledge of (secret) names, we propose an abstract formulation of the concept of secret in any weakly adhesive category. The guiding idea is to mark part of a system state as visible or publicly accessible; further, in principle, something that has become public knowledge will stay accessible indefinitely. The main technical contribution consists in providing a proof which shows that a recently proposed categorical construction, which produces a category having monomorphisms as objects and pullback squares as morphisms, preserves weak adhesivity. Finally we sketch how it is possible to verify certain secrecy properties using unfolding based verification approaches that lately have been generalized to rewriting systems in weakly adhesive categories.

Keywords: adhesive categories, interaction models, verification

1 Introduction

In every day communication one usually does not disclose private information unless one trusts the communication partner. Moreover, special care is advisable when making private information publicly available. And more often than not, one would rather prefer that a certain piece of private information will never be become known to the public. Here we are not only talking about issues of embarrassment or reputation, but also about secret data like personal identification numbers for (on-line) banking accounts.

Nevertheless, as the example of on-line banking illustrates, there often occur situations in which secret data need to be transmitted via protected channels between trustworthy communication partners, and moreover the critical data must not become disclosed to a third party. The running example of this paper will be concerned with access keys to a private network, e.g. the intranet of some banking institute. Obviously, in this scenario, it is important that such access keys do not become publicly available.
One of the earlier approaches to formally reason about the security of key exchange protocols using cryptographic methods, is the spi-calculus [1], which extends the π-calculus [15] by cryptographic primitives. Based on this name passing calculus, there has been carried out a large amount of work concerning the verification of concrete protocols. The actual protocol verification tools however do sometimes use techniques from other fields of computer science (see e.g. [3]). Alternatively, protocols might also be specified and verified using graph transformation systems [4,13]; the latter have the advantage that they are often easier understandable by laypersons.

Now the aim of this paper is not another concrete proposal of a modelling technique for protocols. Instead we strive for a better understanding of the fundamental distinction between private and public knowledge, which corresponds to the open/bound names dichotomy of name passing calculi. Moreover the scope extrusion phenomenon of the latter captures the possibility to exchange secret information and, in the extreme, to make secret information publicly available.

Taking a more abstract point of view, given an arbitrary state of a system, then part of of this state is open to public access (while at the same time other parts are still secret). In the process calculus world, the open part corresponds to the free names of a process. In graph transformations systems using the borrowed context approach [5], the open part is singled out by a sub-graph of the graph which models the whole system state.

The main question is now, when the private part and the public part (in the model) of a given system state should be considered “sufficiently” distinct such that all secret information is protected from public access. Though this question usually has an intuitive answer in concrete example cases, the question seems more difficult in the abstract setting of this paper, as we consider system states as objects of an arbitrary (weakly) adhesive category.

To help answer this question, we proceed as follows. First we introduce the protected links calculus as an example of a simple name passing calculus, which nevertheless illustrates the private/public dichotomy and allows to give a precise characterization of secrecy violations. Then we give the graphical representation of this calculus in section 3. With these concrete examples at hand, in section 4, we set out to lift the notion of secrecy violation to the abstract setting of adhesive categories in such a way that the results of [2] apply.

2 The protected links calculus

The running example of this paper will be the protected links calculus (plc), which couples the ideas of (the implementation of) the explicit fusion calculus [17] with a basic access control mechanism. Recall that the explicit fusion calculus was developed with the goal of providing an implementation of Milner’s π-calculus [15]. The “machine model” was the fusion machine described in [17]; a simplified version of the latter has been proposed in [9], where also a “low-level” encoding of the π-calculus was presented.

Now the main characteristic of the protected links calculus that it shares with
the explicit fusion calculus and the fusion machine, is that it does not use any name substitution at the meta-level but instead uses a “low-level” approach similar to the one of the fusion machine. In the latter, substitution of names is implemented by a forwarding mechanisms that was explored in more detail in [9].

Indeed, the major part of the primitives of the protected links calculus are taken from the calculus of explicit fusions and its fusion machine, namely (asynchronous) input and output, parallel composition, and forwarders, which we here more often call links; the latter however are equipped with an access control mechanism based on the notion of access right, which is the new entity kind of the PLC.

**Definition 2.1 (Syntax of the protected links calculus)** Let \( N \) be a collection of names, which is the disjoint union of public names \( \cdot N \) and private names \( \cdot \cdot N \), which means \( N = \cdot N \uplus \cdot \cdot N \). Then the set of (raw) terms of the \( \text{plc} \)-calculus is given by the following specification.

\[
P ::= u(x) \quad u, x \in N \quad \text{(output action)}
\]

\[
| u(y) \quad u, y \in N \quad \text{(input action)}
\]

\[
| u\cdot w \quad u, w \in N \quad \text{(protected link)}
\]

\[
| x\cdot u \quad x \in N, u \in \cdot \cdot N \quad \text{(access right)}
\]

\[
| 0 \quad \text{(inaction)}
\]

\[
| P \parallel P \quad \text{(parallel composition)}
\]

Let \( P \) be a raw term; then the set of free names of \( P \), written \( \text{fn} P \), contains all names that occur in \( P \) but are not private, i.e. \( \text{fn} P = \{ x \in \cdot N \mid x \text{ occurs in } P \} \).

A process of the \( \text{plc} \)-calculus is a raw term up to structural congruence, written \( \equiv \), which is the smallest equivalence relation on raw terms satisfying the following axioms where \( P, Q, R \) range over \( \text{plc} \)-terms.

\[
P \parallel 0 \equiv P \quad P \parallel Q \equiv Q \parallel P \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R
\]

If a name \( v \) is public, i.e. \( v \in \cdot N \), we sometimes write \( \cdot v \) instead of \( v \) to emphasize this fact; conversely, if \( \cdot v \) is a name, then we implicitly assume that \( \cdot v \in \cdot N \); by a similar convention, if we write \( \cdot v \), then we silently presuppose that \( \cdot v \in \cdot N \).

As mentioned above, the main difference w.r.t the calculus of explicit fusions consists in the new entity, called access right. Following a common interpretation of process calculi, names are often referred to as *channels* through which input and output actions may synchronise and communicate. Further we will talk about *scopes*, which are those parts of terms that share a private name \( \cdot u \), or names that are related to \( \cdot u \) via a chain of bi-directional links, i.e. \( \cdot u \) and \( v \) are in the same scope, if there is a chain \( \cdot u\cdot w_1 \cdot w_1\cdot \cdot u \cdot \cdot v \cdot w_n \cdot w_n\cdot v \). Relying on this word usage, the ideas of the protected links calculus can be described as follows.

The protection mechanism of links ensures that an output action can enter into a scope only if it has the access right for the scope in question. This is captured by the conditional forwarding mechanism of protected links, which checks access rights before output actions are relocated. In contrast, modelling the possibility of attacks
and careless users, access rights may always spread between linked channels. These phenomena are made precise in the formal definition of the reaction relation over PLC terms, and is discussed in more detail afterwards.

**Definition 2.2 (Reaction in the PLC)** The reaction relation over PLC terms, written \( \rightarrowtriangle \), is the smallest relation satisfying the axioms and rules of Figure 1.

\[
\begin{align*}
\text{PROT} & \quad \hat{v} \in \mathcal{N} \\
& \quad (u(x) \mid x \triangleright \hat{v} \mid u \triangleright v) \rightarrow (\hat{v}(x) \mid x \triangleright v \mid u \triangleright v) \\
\text{PUBL} & \quad \hat{v} \in \mathcal{N} \\
& \quad (u(x) \mid u \triangleright v) \rightarrow (\hat{v}(x) \mid u \triangleright v) \\
\text{EXCH} & \quad (x \triangleright \hat{v} \mid x \cdot y) \rightarrow (y \cdot \hat{v} \mid x \cdot y) \\
\text{COMM} & \quad (u(x) \mid u(y)) \rightarrow (x \cdot y \mid y \cdot x) \\
\text{STRUCT} & \quad P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q \\
& \quad P \rightarrow Q
\end{align*}
\]

Fig. 1. Reaction rules of the protected links calculus

These axioms and rules can be explained as follows. The protection mechanism of links is captured by the PROT-axiom: an output action \( u(x) \) is forwarded along a link \( u \triangleright v \) only if the data \( x \) come equipped with the access right \( x \triangleright v \) to the channel \( v \), to which the action could be transported via the link. To simplify the modelling process we think of the data \( x \) as a representation of the actual user that is trying to send \( x \).

However, the protection mechanism does not restrict transmissions to public channels, i.e. everyone can send on public channels. This is formalized by the PUB-rule, which says that given a link \( u \triangleright v \), which models a direct network link connecting \( u \) to \( v \), an output action \( u(x) \) at the origin \( u \) can always travel to the target \( v \), provided that the target channel \( v \) is a public.

Next we come to the formal counterpart of the phenomenon that, as known from practical experience, keys are often stolen or exchanged with untrustworthy partners. Hence, assuming the worst case, the distribution of access keys is unconditional, i.e. whenever there is a direct means of communication a key may be exchanged. Precisely the EXCH-axiom says that whenever the user \( x \) has a key to enter the scope \( v \), modelled by the access right \( x \cdot v \), in the presence of a link \( x \cdot y \), which models a direct means of communication, the user \( y \) will always manage to obtain the “key” granting access to \( v \), which then results in the access right \( y \cdot v \).

A request for a (new) channel for the transmission of (possibly confidential) data is modelled by a send action \( u(x) \) where \( x \) is a channel name which, simplifying again, is thought of as the user issuing the channel request. The receiver, corresponding to the input action \( u(y) \), then will establish direct links between \( x \) and \( y \), which is
represented by complementary forwarders $x\to y$ and $y\to x$; the described communication protocol is captured by the COMM-axiom. Note, that this axiom is essentially the same as the single reaction axiom of the calculus of explicit fusions [17]. Finally the STRUCT-rule says that reaction is closed under structural congruence.

**Example 2.3 (Key exchange)** To show the PL-calculus at work, we consider key exchange. That a user $\hat{x}$ has a key granting access to $\hat{v}$ is modelled by the access right $\hat{x}\triangleright\hat{v}$. Now suppose user $\hat{x}$ wants to exchange this key with user $\hat{y}$ and that $\hat{x}$ and $\hat{y}$ usually communicate via channel $u$, possibly a private channel as well. The PLC process $\hat{x}\triangleright\hat{v}\mid u(\hat{x})\mid u(\hat{y})$ is a possible solution to achieve this, as we have the following reactions.

$$
\begin{align*}
\hat{x}\triangleright\hat{v}\mid u(\hat{x})\mid u(\hat{y}) & \rightarrow \hat{x}\triangleright\hat{y}\mid \hat{x}\triangleright\hat{y} & \text{(COMM)} \\
\hat{y}\triangleright\hat{v}\mid \hat{x}\triangleright\hat{y} & \rightarrow \hat{y}\triangleright\hat{v}\mid \hat{x}\triangleright\hat{y} & \text{(EXCH)}
\end{align*}
$$

Hence after these two steps, user $\hat{y}$ has the key granting access to $\hat{v}$, which is modelled by the access right $\hat{y}\triangleright\hat{v}$.

Before we come to the main theme of this paper, namely secrecy, we give a short comparison of the PL-calculus on the one hand, and the calculus of explicit fusions, the fusion machine and the linear forwarder calculus on the other hand. The only properly new primitive of the PL-calculus is the access right since the fusion related calculi do not have any similar entities; however the latter calculi are not designed to reason about secrecy but only about communication via name passing.

The second theme which allows to discern the mentioned calculi concerns the mechanisms that are used to connect channels or “fuse” names. Whereas the calculus of explicit fusions addresses the issue of connection of names at the level of structural congruence, which intuitively corresponds to (irreversible) fusion of names, both the linear forwarder calculus [9] and the protected links calculus choose a “low-level” approach, and add reaction rules that “implement” the fusion of names; this “low-level” approach has the advantage, that additional reaction rules might be added to model the break down of network links, which corresponds to the removal of links between names.

Finally we would like to stress that the PL-calculus is just an example which allows to illustrate the idea of secrecy without the need to formally introduce the technical details of double pushout rewriting [6]. Hence we also omitted replication, and synchronous input and output, which only would have burdened the presentation. Moreover, for the purpose of this paper, it seemed suitable to avoid the notions of $\alpha$-equivalence and bound name.

Summarizing, the PL-calculus can be seen as a simplified version of the linear forwarder calculus with a new entity called access right. The latter allows to reason about secrecy, as demonstrated in the following, central example of the paper, which illustrates how secrecy holes are modelled in the PL-calculus.

**Example 2.4 (Secrecy hole)** A secrecy hole of a network, is a channel through which private access keys are made public. A process term $P$ contains an immediate secrecy hole, if $P$ contains a sub-term of the form $\hat{v}\triangleright\hat{w}$, i.e. if in the modelled system,
there is a publicly available key granting access to some private channel \( \bar{w} \).

Further a process \( Q \) models a system with a covert secrecy hole, if \( Q \) does not contain any sub-term of the form \( \bar{v} \triangleright \bar{w} \) but such a sub-term might arise after a number of reductions of \( P \). For example the process \( \bar{u} \triangleright \bar{w} \mid \bar{u} \triangleright \bar{v} \) does not contain any immediate secrecy hole; however we have the reaction \( (\bar{u} \triangleright \bar{w} \mid \bar{u} \triangleright \bar{v}) \rightarrow (\bar{v} \triangleright \bar{w} \mid \bar{u} \triangleright \bar{v}) \) via the \textsc{exch}-rule, and the latter contains an immediate secrecy hole. Hence, given a term \( Q \) one might want to prove that it does not contain any (covert) secrecy holes.

How the verification of secrecy may be achieved using the unfolding technique of [2], is sketched in 4.4. However we first need to recall the necessary concepts concerning transformation systems and categories that allow to model systems following the double pushout approach of [6].

3 The graphical counterpart of the PLC

In this section we give a graphical representation of the PLC; more precisely, for each process there will be a corresponding graph and vice versa. Moreover each reaction rule will be a graph transformation rule based on the double pushout approach (dpo) [6]. However, omitting the details of dpo rewriting, we give an informal presentation that nevertheless should convey the main ideas of graph transformation.

The channels or names of PLC processes will correspond to nodes in the graphical representation. All entities of the pl-calculus correspond to different kinds of edges between the nodes. An output action \( u(x) \) is represented by a send arrow \( \circ \rightarrow \bar{u} \), an input action \( u(y) \) corresponds to a receive edge \( \circ \leftarrow \bar{u} \), an access right \( x \triangleright \bar{u} \) is drawn as \( \circ \triangleleft \bar{u} \), a link \( u \triangleright \bar{w} \) becomes a connection arc \( \bar{u} \rightarrow \circ \), and the inaction is the empty graph \( \emptyset \). Finally parallel composition is achieved by union of graphs. To avoid clutter, a pair of complementary links \( u \triangleright \bar{w} \mid w \triangleright \bar{u} \) is represented by \( \bar{u} \circ \bar{w} \). For a private node \( \bar{u} \), the label \( \bar{u} \) already contains the information that this is a private node, and the gray boundary only emphasizes this fact.

Moreover, not only does each process correspond to a graph, but also each reaction rule has a corresponding transformation rule. A (linear) graph transformation rule \( q \) is essentially a pair of graphs \( L, R \) (called left- and right-hand side, respectively) with a common sub-graph \( K \) (referred to as interface), i.e. \( q = L \supseteq K \subseteq R \). Assuming that the sets of nodes and edges in rules are disjoint, and writing as if graphs were mere sets, the action of these rules or productions can be described as follows.

Suppose that the left-hand side \( L \) of a rule \( q = L \supseteq K \subseteq R \) is a sub-graph of some larger graph \( G \), i.e. \( L \subseteq G \), then the rule \( q \) first removes from \( G \) all those nodes and edges that are covered by the left-hand side \( L \) but not contained in the interface \( K \), which results in an intermediate graph \( D \subseteq G \); in a second step, those nodes and edges of \( R \) that are not contained in \( K \) are adjoined to the intermediate result \( D \), yielding a graph \( H \supseteq D \). Provided that \( R \cap G \subseteq K \) (and the inclusion \( L \subseteq G \) satisfies the so-called dangling condition [4]), the result \( H \) can be described as \( H = (G \setminus (L \setminus K)) \cup R \).

Now the graphical counterpart of the rules of the PLC is illustrated in Figure 2. Note that in this encoding of the PLC rules, one directly mentions which resources are
only used as “catalysts”, and hence remain unchanged during the the reactions. To see the correspondence between the calculus and its graphical presentation, consider the following example concerning the COMM-rule.

Example 3.1 (Communication in PLC via graphs) The protected link calculus process \( u(y) \, | \, u(x) \, | \, u(z) \) has two possibilities to evolve: either \( u(x) \) reacts with \( u(y) \) or with \( u(z) \), i.e.

\[
(y \rightarrow x \mid x \rightarrow y \mid u(z)) \leftarrow (u(y) \, | \, u(x) \, | \, u(z)) \rightarrow (u(y) \, | \, z \rightarrow x \mid x \rightarrow z)
\]

The corresponding graph transformation steps can be illustrated as follows.

A more subtle point is that it is not quite obvious what kind of graphs we have actually used. As a fact we could choose to work in the framework of typed attributed graphs of [8], or simply add additional labels to nodes that specify whether the nodes in question are private or public. However, the graphs that we use to represent PLC-processes have the characteristic property that the set of public nodes is actually a fully fledged sub-graph of the whole graph; moreover the notion of sub-graph allows for a straightforward categorical generalization, which is not the case for labels.

Hence, we will use a more abstract, alternative labelling mechanisms, which has been discussed recently within the graph transformation community and is presented in detail in [11]. The main idea of the latter work is to “mark” part of a given graph. In the present case, one might for example choose to mark only the public nodes of a graph, i.e. the objects we are working with are pairs \( \langle G, G' \rangle \) such that \( G \supseteq G' \), and here it is not important which exact notion of graph we start with, and indeed, in the next section we will replace graphs by objects of any (weakly) adhesive category.

More detailed, in the case of the graphical presentation of the PLC-calculus, let \( P \) be a PLC term and let \( [P] \) be the presentation of \( P \), which is an (edge and node labelled) graph. Then the public nodes of \( P \) form the set of free names \( fn \, P \), which is – when considered as a discrete graph, i.e. a graph without edges – a sub-graph of \( [P] \), i.e. the pair \( \langle [P], fn \, P \rangle \) satisfies \( [P] \supseteq V' \).
For the remainder of the section we write \( \langle G, G' \rangle_2 \) if \( G \) and \( G' \) are graphs, such that the inclusion \( G \supseteq G' \) holds, and call the pair \( \langle G, G' \rangle_2 \) a marked graph. Next we will supply as suitable notion of morphism between marked graphs, such that the marked part of a graph will remain marked and moreover the marked and the unmarked part will be kept distinct. This is made formal in the next definition for the case of (unlabelled multi-)graphs.

**Definition 3.2 (Graphs and morphisms, marked graphs and mappings)**

An unlabelled multi-graph is a quadruple \( G = \langle V, E, s, t \rangle \) where \( V \) is the set of nodes, \( E \) is the set of edges and \( s, t : E \rightarrow V \) are the source and target functions, respectively, which assign to each edge the source and target of the edge, respectively. W.l.o.g. we assume that nodes and edges are a pair of disjoint sets, i.e. \( E \cap V = \emptyset \); then the carrier of \( G \) can be defined as the set \( |G| := E \cup V \).

Next, a graph morphism between two graphs \( G = \langle V, E, s, t \rangle \) and \( G' = \langle V', E', s', t' \rangle \) is a pair of functions \( f = (f_V : V \rightarrow V', f_E : E \rightarrow E') \) such that the following two diagrams commute

\[
\begin{array}{ccc}
E & \xrightarrow{f_E} & E' \\
\downarrow f_V & & \downarrow f_V' \\
V & \xrightarrow{f_V} & V'
\end{array}
\quad \quad \quad \begin{array}{ccc}
E & \xrightarrow{f_E} & E' \\
\downarrow f_V & & \downarrow f_V' \\
V & \xrightarrow{f_V} & V'
\end{array}
\]

i.e. the two equations \( f_E \circ s' = s \circ f_V \) and \( f_E \circ t' = t \circ f_V \) are satisfied; such a morphism \( f \) corresponds to a unique carrier function \( |f| : |G| \rightarrow |G'| \).

Given a graph \( G = \langle V, E, s, t \rangle \), then another graph \( G' = \langle V', E', s', t' \rangle \) is a sub-graph of \( G \) if both inclusions \( G' \subseteq G \) and \( V' \subseteq V \) hold, and moreover for each edge \( e \in E' \) the two equations \( s'(e) = s(e) \) and \( t'(e) = t(e) \) hold. The fact that \( G' \) is a sub-graph of \( G \) is expressed by \( G' \subseteq G \). If \( G' \) is a sub-graph of \( G \) then the obvious inclusion morphism is written \( \iota_G : G' \rightarrow G \).

Finally, a marked graph is a pair \( G_2 = \langle G, G' \rangle_2 \) such \( G' \) is a sub-graph of \( G \), i.e. \( G' \subseteq G \). A marked graph mapping between marked graphs \( G_2 = \langle G, G' \rangle_2 \) and \( H_2 = \langle H, H' \rangle_2 \) is a pair of graph morphisms \( f_2 = (f : G \rightarrow H, f' : G' \rightarrow H') \) such that the equation \( \iota_H \circ f' = f \circ \iota_G \) is satisfied and moreover

(i) if the image of a node of \( G \) is in the marked part \( |H'| \), then the node itself is marked, i.e. for every node \( v \) of \( G \), if \( f_V(v) \in |H'| \) then \( v \in |G'| \), and

(ii) the same holds for all edges \( e \) in \( G \), i.e. \( f_E(e) \in |H'| \) implies \( e \in |G'| \).

A marked graph mapping \( f_2 = \langle f, f' \rangle \) is an inclusion mapping if \( f \) and \( f' \) are inclusion morphisms.

We will often identify a graph with its carrier and a graph morphism with the corresponding carrier function. In the next section we will present Lemma 4.1, which shows how the Conditions of item i and item ii, can be expressed succinctly using the notion of pullback in the category of graphs and graph morphisms.

Concluding this section, we remark that the graphical presentation of the PLC can be described with complete formal rigor using marked, node and edge labelled multi-graphs. However we omit these details though we encourage the interested
reader to convince himself or herself that this is indeed possible.

4 A categorical approach to secrecy

Having introduced the protected links calculus to motivate the notion of marked graph, we now set out to lift the latter notion to an abstract level, namely the so-called categories of reflected monos [11]. This is followed by suggestions for the description of secrecy related concepts using category theoretical language alone. Finally we discuss possibilities to verify secrecy properties on this abstract level.

4.1 Categories of reflected monos

To prepare the definition of the category of reflected monos, we give a categorical characterization of the category of marked graphs and mappings. For this we recall that, given a graph \( H \), a sub-graph \( H' \subseteq H \), and a morphism \( f : G \to H \) in the category of graphs and graph morphisms, the (natural choice of a) pullback of the co-span \( G \rightharpoonup f \rightharpoonup H \rightharpoonup \) is the span \( G \rightharpoonup f^{-1}(H') \rightharpoonup H' \), giving rise to the pullback square \( \begin{array}{ccc} f^{-1}(H') \subseteq H & \subseteq & H' \\ \downarrow & & \downarrow \\ G & \rightharpoonup & H' \end{array} \) where \( f^{-1}(H') = \{ h \in |G| \mid f(h) \in |H'| \} \) is the pre-image of \( |H'| \), and the graph morphism \( f \downarrow_{H'} : f^{-1}(H') \to H' \) is the co-domain restriction of \( f : G \to H \), which maps \( x \in f^{-1}(|H'|) \) to \( f \downarrow_{H'}(x) = f(x) \in |H'| \).

**Lemma 4.1 (Marked mappings as pullback squares)** Let \( G_2 = \langle G, G' \rangle_2 \) and \( H_2 = \langle H, H' \rangle_2 \) be marked graphs and \( f_2 = \langle f, f' \rangle : G_2 \to H_2 \) be a marked graph mapping. Then \( G' = f^{-1}(H') \) and \( f' = f \downarrow_{H'} \), which means that \( G \rightharpoonup f \rightharpoonup H \rightharpoonup H' \) is a pullback of \( G \rightharpoonup f \rightharpoonup H \rightharpoonup H' \), giving rise to the pullback square \( \begin{array}{ccc} G' \subseteq H & \subseteq & H' \\ \downarrow & & \downarrow \\ \end{array} \).

**Proof.** First we show that \( G' = f^{-1}(H') \). To show that \( G' \subseteq f^{-1}(H') \), let \( x \in |G'| \), then \( f(x) = f'(x) \in |H'| \), i.e. \( x \in f^{-1}(|H'|) \). To prove the converse, let \( x \in f^{-1}(|H'|) \), i.e. \( f(x) \in |H'| \), whence also \( x \in G' \) by Definition 3.2. Having shown this, the equation \( f' = f \downarrow_{H'} \) follows immediately.

Marked graphs and their mappings satisfy a certain reflection property, namely for any given mapping \( \langle f, f' \rangle : \langle G, G' \rangle_2 \to \langle H, H' \rangle_2 \), the marked part \( G' \) can be recovered from its image, i.e. the equation \( |G'| = f^{-1}(f(|G'|)) \) holds. This is also a property that might explain the name of the abstract counterpart of the category of marked graphs and their mappings.

**Definition 4.2 (Reflected Monos)** Let \( \mathbb{C} \) be a category with pullbacks along monomorphisms, i.e. for each co-span \( A \rightharpoonup f \rightharpoonup D \rightharpoonup \) with monic \( m \), a pullback span \( A \rightharpoonup m \rightharpoonup N \rightharpoonup \rightharpoonup M \) exists, yielding a pullback square \( \begin{array}{ccc} A \rightharpoonup m & \rightharpoonup & N \\ \downarrow & & \downarrow \\ A & \rightharpoonup & M \end{array} \).

Then the category of reflected \( \mathbb{C} \)-monos, written \( \text{RMon}(\mathbb{C}) \), has \( \mathbb{C} \)-monomorphisms \( A' \rightharpoonup a \rightharpoonup A \) as objects and an \( \text{RMon}(\mathbb{C}) \)-morphism \( f_{\leftrightarrow} : \langle A' \rightharpoonup a \rightharpoonup A \rangle \to \langle B' \rightharpoonup b \rightharpoonup B \rangle \) is a pair \( f_{\leftrightarrow} = \langle f : A \to B, f' : A' \to B' \rangle \) of \( \mathbb{C} \)-morphisms such that \( A \rightharpoonup a \rightharpoonup A' \rightharpoonup f' \rightharpoonup B' \) is a pullback of \( A \rightharpoonup f \rightharpoonup B \rightharpoonup b \rightharpoonup B' \), yielding a pullback square \( \begin{array}{ccc} A' \rightharpoonup f' & \rightharpoonup & B' \\ \downarrow & & \downarrow \\ A & \rightharpoonup & B \end{array} \).
4.2 Reflected monos in weakly adhesive categories

We will now recapitulate the notion of weakly adhesive category presented in [2], which provides a framework that is suitable for double pushout rewriting, and moreover is compatible with the RMon construction, a fact which will be made precise in Proposition 4.5. Weakly adhesive categories generalize adhesive categories [14] and are closely related to weak adhesive hlr categories [7]. The advantage of this weaker notion of adhesivity lays in the fact that it captures additional examples, e.g. those presented in [7], which are of practical relevance.

**Definition 4.3 (Weakly adhesive category)** A category is weakly adhesive if

(i) pullbacks along monomorphisms exist and also pushouts along monomorphisms exist, i.e. for each span \( B \leftarrow f \rightarrow A \rightarrow m \rightarrow C \) with monic \( m \), a pushout \( B \rightarrow n \rightarrow D \leftarrow g \rightarrow C \) exists, yielding a pushout square \( \begin{array}{ccc} B & \rightarrow & A \\ B' \downarrow & & A' \downarrow \\ D' & \rightarrow & C' \end{array} \);

(ii) pushouts of pairs of monomorphisms are universal (or stable under pullback), i.e. in each commutative cube over a pushout square \( \begin{array}{ccc} B & \leftarrow & A \\ B' \downarrow & & A' \downarrow \\ D' & \rightarrow & C' \end{array} \), having pullback squares as lateral faces as shown in the middle diagram in the display below, the top face is a pushout square;

(iii) pushouts along monomorphisms are mono-universal and converse mono-universal: in each commutative cube on top of a pushout square \( \begin{array}{ccc} B & \leftarrow & A \\ B' \downarrow & & A' \downarrow \\ D' & \rightarrow & C' \end{array} \) as in the left diagram in the display below, with pullback squares as back faces and the “corner”-arrows \( b \) and \( c \) monic, its top face is a pushout square if and only if the front faces are pullback squares and the morphism \( d \) is monic.

Note that adhesive categories [14], apart from having all pullbacks, satisfy the simpler (and stronger) version of Condition iii that does not contain any conditions on the vertical morphisms \( a, b, c \) and \( d \). Condition iii is equivalent to the requirement that
pushouts along monomorphisms are hereditary in the sense of [12]. Finally, the subtle difference to the weak adhesive HLR-categories of [7] is that in the definition of the latter, the vertical morphism \(d\) into the “tip” of the bottom pushout is required to be monic already in the antecedent, which implies the “top face-front faces”-equivalence. This is also the reason why the proof of Proposition 4.5 does carry over to weak adhesive HLR-categories. The exact relation among weakly adhesive, adhesive, and weak adhesive HLR categories is also discussed in [2].

An example of a category that is weakly adhesive but not adhesive and hence deserves being mentioned, is the category of undirected multi-graphs. That this category is weakly adhesive but not adhesive can be shown by adapting the results presented in [16].

**Example 4.4 (Undirected multigraphs)** An undirected multigraph is a triple \(M = \langle E, V, c : E \to V^{\oplus} \rangle\), where \(V^{\oplus}\) is the free commutative monoid over the set of vertices \(V\) and the connection function \(c\) assigns to each edge \(e \in E\) a multiset \(c(e) \in V^{\oplus}\) of adjacent vertices. Finally, given another multigraph \(M' = \langle E', V', c' : E' \to V'^{\oplus} \rangle\), a multigraph morphism \(f : M \to M'\) is a pair of functions \((f_E : E \to E', f_V : V \to V')\) such that \(f_V \circ c = c' \circ f_E\) where the homomorphism \(f_V^{\oplus} : V^{\oplus} \to E^{\oplus}\) is the freely adjoined monoid homomorphism of the function \(f_V : V \to E\).

Now we come to the main technical contribution of this paper, which says that the RMon construction yields a weakly adhesive categories when applied to a weakly adhesive category.

**Proposition 4.5 (Weakly adhesive reflected monos)** Let \(C\) be a weakly adhesive category. Then the category of reflected monos \(\text{RMon}(C)\) is weakly adhesive.

**Proof.** First one shows that a morphism \(f_\leftarrow = \langle f, f' \rangle : a \to b\) in \(\text{RMon}(C)\) is monic, if and only if both \(f\) and \(f'\) are monic in \(C\). Moreover it is straightforward to show that pullbacks along monomorphisms are constructed component-wise.

The main task of the proof consists in showing that also pushouts along monomorphisms are constructed component-wise. Given a morphism \(f_\leftarrow = \langle f, f' \rangle : a \to b\) and a monomorphism \(m_\leftarrow = \langle m, m' \rangle : a \to c\), then the candidate for the pushout directly arises from the definition of weakly adhesive categories.

It remains to show that this square satisfies the universal property of pushouts. Hence let \(\langle h, h' \rangle : b \to e\) and \(\langle k, k' \rangle : c \to e\) be morphisms such that the left one of the following \(C\)-diagrams commutes.
Now we obtain a pair of morphisms $u': D' \to E'$ and $u: D \to E$ such that the right one of these diagrams commutes, since $\{n', g\}$ are jointly epic. It remains to show that the pair $u', u$ actually defines a mediating morphism as it then is easy to show that it is unique.

To derive existence, let $E' \leftarrow \bar{u} \rightarrow \bar{D} \rightarrow D$ be a pullback of $E' \twoheadrightarrow E \leftarrow u - D$ as shown in the left one of the following diagrams.

Now there exist unique $C$-morphisms $\bar{n}: B' \to \bar{D}$ and $\bar{g}: C' \to \bar{D}$ making this $C$-diagram commute, which moreover yield pullback squares as illustrated. Then, by the definition of weakly adhesive category, the co-span $B' \rightrightarrows \bar{D} \leftrightarrow C'$ is a pushout of the span $B' \leftarrow f' \rightarrow A' \rightrightarrows C'$ as indicated in the right one of the above diagrams. Finally, by straightforward calculation, we obtain an isomorphism $i: D' \to \bar{D}$ satisfying both $u' = \bar{u} \circ i$ and $d = \bar{d} \circ i$, i.e. $\langle u, u' \rangle: d \to e$ is actually a mediating morphism.

Thus we have laid the foundations for discussing the notions of secret and secrecy on an abstract level. To illustrate the idea, we turn back to Example 2.4, and the discussion of secrecy holes in systems that are modelled using the $\textit{pl}$-calculus.
4.3 Secrecy revisited

Having established the theoretical framework, namely reflected monos in weakly adhesive categories, we are now ready to study secrecy related phenomena at a more abstract level. In particular we address the questions of how to describe secrecy holes, of when the private/public separation might be violated, and of how persistence of public information can be ensured.

As for the first point, recall that an immediate secrecy hole in a PLC process $P$ is witnessed by a sub-term of the form $\bar{v} \rhd \bar{w}$, which corresponds to the fact, that the encoding $\llbracket P \rrbracket$ as sketched in section 3 contains a sub-graph of the form $\langle \bar{v} \rhd \bar{w} \rangle$. More precisely, the graphical representation of $P$ is actually a marked graph $\langle \llbracket P \rrbracket, \text{fn} P \rangle_\exists$ and also $\langle \bar{v} \rhd \bar{w} \rangle$ is a marked graph, namely $\langle \bar{v} \rhd \bar{w} \rangle_\exists$. Moreover $P$ has an (immediate) secrecy hole if and only if the inclusion $\langle \bar{v} \rhd \bar{w} \rangle_\exists \subseteq \llbracket P \rrbracket, \text{fn} P \rangle_\exists$ holds in the category of marked graphs, which is the case if $\langle \bar{v} \rhd \bar{w} \rangle$ is a sub-graph of $\llbracket P \rrbracket$ and $\bar{v} \in \text{fn} P$.

Now, the first observation is that the marked graph $\langle \bar{v} \rhd \bar{w} \rangle_\exists$ is an object of the category of marked graphs that witnesses a secrecy violation. Hence on the abstract level, given a category $\mathbb{D}$ and a system with states being modelled as objects of the category $\mathbb{D}$, then a state $S \in \mathbb{D}$ will have a secrecy violation $V \in \mathbb{D}$ if there is a monomorphism $m: V \rightarrow S$. This leaves us the task of motivating, for the case of $\mathbb{D} = \text{RMon}(\mathbb{C})$, when an object $W \rhd \bar{w} \rightarrow W \in \text{RMon}(\mathbb{C})$ should be considered as a secrecy violation.

This leads us to the second observation, which concerns the abstract version of the private/public distinction. Given a process $P$ without an immediate secrecy hole, then, after removing all send arrows, receive edges, and connection arcs from the graphical counterpart $\llbracket P \rrbracket$, then the resulting sub-graph $\llbracket P \rrbracket^- \rhd \rightarrow \llbracket P \rrbracket^-$ is the disjoint union of the public and the private part. This is supposed to model the fact that the private information and the public information are kept distinct, and there is no means to disclose private channels.

More formally, when considering the marked graph $\langle \llbracket P \rrbracket^-, \text{fn} P \rangle$ as the inclusion $\text{fn} P \rhd \rightarrow \llbracket P \rrbracket^-$, there is a maximal sub-graph $\text{fn} P \rhd \rightarrow \llbracket P \rrbracket^-$ such that the empty graph $\emptyset$ is the only common sub-graph of $\text{fn} P$ and $\llbracket P \rrbracket^-$, i.e. the sub-graph $\text{fn} P \rhd \rightarrow \llbracket P \rrbracket^-$ is actually the pseudo-complement $[10]$ of the sub-graph $\text{fn} P \rhd \rightarrow \llbracket P \rrbracket^-$. To lift this second observation to the abstract level, note that we have the inclusion $\text{fn} P \rhd \rightarrow \llbracket P \rrbracket^-$, and hence the equation $\text{fn} P \cap \llbracket P \rrbracket^- = \text{fn} P \text{ holds}$, which in turn means that in the category of graphs we have the pullback square $\text{fn} P \llbracket \bar{w} \rhd \bar{w} \rrbracket \text{fn} P^{-}$.

Now, abstracting away, for each state $S \in \mathbb{C}$ we would like to have a publicly available part $J \rhd \rightarrow S$, which is often referred to as interface, and moreover we will require to have singled out another part $S^- \rhd \rightarrow S$, which corresponds to the secrecy relevant part, such that there is a pullback $J \leftarrow \bar{w} \rhd J \rhd \rightarrow S^-$ of $J \rhd \rightarrow S \leftarrow \bar{w} \rhd S^-$, yielding a pullback square $\bar{w} \rhd \rightarrow \bar{w} \rhd \rightarrow S^-$ with the identity on top. Now the fact that the private and public parts are kept distinct (after removing the secrecy irrelevant parts of $S$) corresponds again to existence of a pseudo-complement $\bar{J} \rhd \rightarrow S^-$ for the (sub-object described by the) monomorphism $J \rhd \rightarrow S^-$ (in the sub-object lattice over the object $S^-$).
Finally a secrecy violation is an object $w' \downarrow \sqcap \downarrow w$ such that the relevant pseudo-complement does not exist, and finally, as a fact, one has that a system state has an (immediate) secrecy hole if it contains a secrecy violation, i.e. if $w' \downarrow \sqcap \downarrow w$ models a secrecy violation and $\gamma \downarrow \sqcap \downarrow \gamma$ corresponds to a state, then a monomorphism $w' \downarrow \sqcap \downarrow w' \rightarrow \gamma \downarrow \sqcap \downarrow \gamma$ in $\text{RMon}^2(C)$ is a witness for $\gamma \downarrow \sqcap \downarrow \gamma$ having a secrecy hole.

So far we have only spoken about properties of system states, modelled as objects of categories of reflected monos, and gave suggestions of how to determine whether private and public areas are disjoint. In contrast, the third point, namely the persistency of public information, is concerned with the evolution of the system. Hence the question whether or not public information will always stay available, or may be eventually lost, concerns the transformation rules. A simple sufficient condition on a rule $l \leftarrow (i',i) \rightarrow k \rightarrow (j,j) \rightarrow r$ in a category of the form $\text{RMon}^2(C)$ that ensures persistency of public knowledge, is the requirement that $i'$ is an identity.

4.4 Verifying secrecy properties

We have argued that (at least in certain cases) it is possible to model secrecy violations as objects which have a structure that allows to disclose private information, the prime example being the marked graph $\langle \tilde{v}, \tilde{w} \rangle \subseteq \text{in}$ the context of the graphical representation of the protected links calculus (see Figure 2). In analogy, suppose that the $\text{RMon}^2(C)$-object $w' \downarrow \sqcap \downarrow w'$ models a secrecy violation where $C$ is a weakly adhesive category. Since $\text{RMon}^2(C)$ is weakly adhesive by Proposition 4.5, in principle, the results of [2] can be used to show that in a system modelled by a set of rules $\{q_n = l_n \leftarrow k_n \rightarrow r_n \mid n \in N\}$ with a start state corresponding to an object $S \in \text{RMon}^2(C)$, there is no reachable system state with a secrecy violation, i.e. the system does not have secrecy holes.

Speaking in Petri net terms, the object $w' \downarrow \sqcap \downarrow w'$ corresponds to a marking, the rules $\{q_n = l_n \leftarrow k_n \rightarrow r_n \mid n \in N\}$ with a start object $S$ in the category $\text{RMon}^2(C)$ correspond to a marked Petri net, and the fact that no reachable object contains $w' \downarrow \sqcap \downarrow w'$ corresponds to the fact that a given “bad” marking is not coverable in the marked Petri net. Further, in the long rung, the results of [2] might lead to generalizations of the methods for Petri nets and graph transformation systems, that allow to automatically verify that the “bad” marking is not coverable.

5 Conclusion

Based on the recently proposed reflected monos-construction of [11], we have proposed abstract, formal counterparts of secrecy related notions, that allow to reason about secret keeping in systems that are faithfully modelled by transformation systems in categories of reflected monos. To ensure that transformation systems in categories of reflected monos can be given in terms of double pushout rewriting [6], we have established that the reflected monos construction preserves weak adhesivity in the sense of [2]. Finally, we have sketched how the results of the latter work might eventually lead to automatic verification of secrecy properties, working on the abstract level of weakly adhesive categories.
Acknowledgements
I would like to thank Sander Bruggink, Barbara König, and Arend Rensink for constructive criticism and enlightening discussions about the content and the presentation of the paper.

References


A Petri Net Model of Handshake Protocols

Luca Fossati¹

Dipartimento di Informatica - Univ. di Torino, Italia

PPS - CNRS & Univ. Paris Diderot, France

Daniele Varacca

PPS - CNRS & Univ. Paris Diderot, France

Abstract
We propose a Petri net model of handshake protocols. These are asynchronous communication protocols which enforce several properties such as absence of transmission interference and insensitivity from delays of propagation on wires. We introduce the notion of handshake Petri net, a Petri net with a specific external interface. We show that the set of observable quiescent traces generated by such a net captures the properties defining a handshake protocol. Conversely we show that for any handshake protocol we can construct a corresponding net. We also study different subclasses of the model. Many examples are provided.

Keywords: Handshake protocol, Petri nets, asynchronous communications, delay-insensitivity, transmission interference.

1 Introduction

The asynchronous style of computation is characterized by several subunits acting locally, independently of each other, as opposed to the synchronous style, where a central clock disciplines everything. Working with asynchronous systems, there are a few situations one would like to avoid. One is transmission interference which may occur when two consecutive messages are sent over the same channel, with the risk of clashing into one another [13]. Another one is computation interference, where a message is delivered to an unready receiver [8,15].

One way to rule out such situations is by adopting communication protocols to enforce the desired behavior. For instance, delay-insensitive protocols guarantee that a system's behavior is independent of propagation delays over wires and of computational speeds of single units, thus preventing computation interference. Among those, we focus on the handshake protocol which requires that each message sent is followed by an acknowledgment, thus preventing transmission interference.

¹ Corresponding author: fossati@di.unito.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Thanks to its simplicity and efficiency, the handshake protocol has gained the interest of enterprises like Sun and Philips [1]. However, little research has been put forward on foundational aspects. For quite a few years the foundational research on handshake circuits, circuits obeying the handshake protocol, has relied on the model introduced by Kees Van Berkel in his PhD thesis [14]. While Van Berkel’s model will continue to be a reference for many aspects, it contains a serious shortcoming: the process composition it defines is not associative, as proved by the first author [4].

To solve this problem the first author [4] proposed a game semantics for handshake circuits which describes their composition correctly for the first time. Technically, the result was accomplished by representing handshake “behaviors” as sums of deterministic handshake strategies. The price to pay is that there are behaviors which do not fit in this representation. The crucial example is the mixer component, MIX, which will be described in Section 3.3.

This led us to look for other kinds of models. A graphical representation is probably the most natural choice for dealing with asynchronous circuits: in graphs as in circuits, composition is easy when everything else works properly. Several works have taken a similar perspective ([2], [7], …). In particular Dan Ghica developed a language for asynchronous hardware design by taking inspiration from the Geometry of Interaction and handshake circuits [5]. However his goal was to improve previous hardware design languages [14,3] and not to capture all handshake behaviors.

The model we present in this paper is based on Petri nets [10]. Petri nets are widely used as models of asynchrony, and are close to the context in which the handshake communication protocol originated [11]. However, the properties of delay-insensitivity and absence of transmission interference had not yet been formalized under a graphical representation. We call our model handshake Petri nets. We show that handshake Petri nets capture precisely the handshake protocol, in the sense that the behavior of every net is a handshake language and that every handshake language is the behavior of some net.

Plan of the paper

In Section 2 we define the notion of handshake language as set of traces (taking inspiration from [14] and [4]). In Section 3 we introduce handshake Petri nets and some of their subclasses. We put a special emphasis on deterministic behaviors, as well as on those nondeterministic behaviors which cannot be expressed as sums of deterministic components. Finally, in Section 4 we provide an interpretation of handshake Petri nets into handshake languages and we prove the correctness and completeness of this interpretation. Completeness of deterministic handshake Petri nets with respect to deterministic handshake languages will follow as a corollary.

2 The Handshake Protocol

In this section we characterize the handshake protocol in terms of languages obeying its communication discipline. We do not exactly give another trace model as, for instance, we do not define composition. We just need a yardstick against which to measure the correctness of our model. Moreover, we are only interested in the communication discipline, so we assume circuits have nonput ports (no data is exchanged
in a communication). We leave the more general case for further work.

**Definition 2.1** A **handshake structure** is a pair \( \langle P, d \rangle \), where \( P \) is a finite set of ports and the function \( d : P \rightarrow \{ \text{act, pas} \} \) determines a direction for each port, active or passive.

As we shall see, active ports are allowed to start a communication, while passive ports are initially waiting.

For the rest of this section let \( \langle P, d \rangle \) be a handshake structure and let \( \cup_{p \in P} \{ p, \bar{p} \} \) be the alphabet of messages on \( \langle P, d \rangle \). In particular, \( p \) and \( \bar{p} \) are both messages on some port \( p \)\(^2\). Two messages are independent when they are not on the same port. The function \( \lambda_P \) is defined on \( \cup_{p \in P} \{ p, \bar{p} \} \) so that \( \lambda_P(p) = - \) (input message) and \( \lambda_P(\bar{p}) = + \) (output message), for all \( p \in P \).

Let \( t \) be a trace on the alphabet of messages \( \cup_{p \in P} \{ p, \bar{p} \} \). \( t \) is a handshake trace on \( \langle P, d \rangle \) if for all \( p \in P \):

- \( t \upharpoonright \{ p, \bar{p} \} = \bar{p}p \bar{p}p \ldots \) when \( d(p) = \text{act} \);
- \( t \upharpoonright \{ p, \bar{p} \} = pp \bar{p} \bar{p} \ldots \) when \( d(p) = \text{pas} \).

We call trace each such restriction and we call request (acknowledge) the message appearing in the odd (even) positions in each thread of \( p \).

Threads induce an equivalence on traces, the homotopy relation \( \sim_P \). Given two handshake traces \( s \) and \( t \), we say that \( s \sim_P t \) when they have the same set of threads. As usual, we denote by \( [s]_\sim \) the equivalence class of trace \( s \) with respect to \( \sim \), we call \( [s]_\sim \) the position of \( s \).

Given a set of traces \( \sigma \) we write \( \sigma \leq \) for its prefix-closure. Let \( \sigma \) be a set of handshake traces, \( s \in \sigma \leq \) is passive in \( \sigma \) if and only if there is no message \( \sigma \) can output after \( s \):

\[ \forall s \cdot m \in \sigma \leq, \lambda(m) = -. \]

We write \( \text{Pas}(\sigma) \) for the set of passive traces in \( \sigma \leq \).

We define \( r_P \) as the smallest binary relation which is closed by reflexivity, transitivity and concatenation, and such that for any distinct ports \( p, q \in P \):

(i) \( p\bar{q} r_P q\bar{p} \);
(ii) \( \bar{p}q r_P \bar{q}p \);
(iii) \( pq r_P qp \)

We say that \( s \) reorders \( t \) in \( P \) if \( s r_P t \). Note that the relation \( r_P \) is not symmetric.

Let \( s \) be a handshake trace and \( p \in P \). We write \( p \bowtie_P s \) if \( sp \) is still a handshake trace. We are now ready for the definition of handshake language.

**Definition 2.2** A **handshake language** \( \sigma \) on \( \langle P, d \rangle \) is a non-empty set of finite handshake traces on \( \langle P, d \rangle \) such that:

(i) \( \text{Pas}(\sigma) \subseteq \sigma \) (closed under passive prefixes);
(ii) \( (t \in \sigma \land s r_P t) \Rightarrow s \in \sigma \) (reorder closed);
(iii) \( (s \in \sigma \leq \land p \bowtie_P s) \Rightarrow s \cdot p \in \sigma \leq \) (receptive).

---

\(^2\) One may object that the same name \( p \) is used for both the message and the port. However the context will always make clear which \( p \) we are referring to.
Note that the traces of a handshake language are finite, but the language itself may contain an infinite number of traces.

**Definition 2.3** Let $\sigma$ be a handshake language. We say that $\sigma$ is *positional* if, for finite $s, s' \in \sigma^\leq$, with $s \sim_P s'$, we have:

(i) $s \cdot t \in \sigma^\leq \Rightarrow s' \cdot t \in \sigma^\leq$;

(ii) $s \in \sigma \Rightarrow s' \in \sigma$;

We say that $\sigma$ is *deterministic* if for any distinct $p, q \in P$:

(i) $s \cdot \bar{p} \in \sigma^\leq \Rightarrow s \not\in \sigma$ (progress);

(ii) $s \cdot \bar{p} \in \sigma^\leq \land s \cdot \bar{q} \in \sigma^\leq \Rightarrow s \cdot \bar{p} \cdot \bar{q} \in \sigma^\leq$ (absence of conflict).

Positionality means that the only thing relevant in a choice is the position we are at and not the way we reached it. As for determinism: when a deterministic language $\sigma$ is able to produce an output, waiting is not an option; when there is a choice of two outputs, one choice must not exclude the other. It is not difficult to prove the following fact.

**Proposition 2.4** A deterministic language is positional.

**Examples**

Consider the handshake structures $P = \langle \{p\}, \{p \mapsto \text{pass}\} \rangle$ and $A = \langle \{p\}, \{p \mapsto \text{act}\} \rangle$, corresponding respectively to a passive and to an active port. Then, $p\bar{p}p\bar{p}p\bar{p}$ is a handshake trace on $P$ but not on $A$. The set

$$\{p\bar{p}, p\bar{p}p, p\bar{p}p\bar{p}, \ldots\}$$

is not closed under passive prefixes as it does not contain the empty string, then it is not a handshake language. Whereas both sets

$$\text{RUN}_p = \{\bar{p}, \bar{p}p, \bar{p}p\bar{p}, \ldots\} \quad \text{and} \quad \{\varepsilon, \bar{p}, \bar{p}p, \bar{p}p\bar{p}, \ldots\}$$

are handshake languages on $A$. In particular $\text{RUN}_p$ is deterministic, the other is not. The set

$$\{\bar{p}, \bar{p}p, \bar{p}p\bar{p}\}$$

is not a handshake language on $A$, because it is not receptive: after the last trace the environment is still supposed to send an acknowledge, but the language is not ready to receive it. Even $\text{RUN}_p$ is not receptive if considered with respect to a structure like $B = \langle \{p, q\}, \{p \mapsto \text{act}, q \mapsto \text{pass}\} \rangle$. One way to make it receptive with respect to $B$ is:

$$\text{REP}_{p,q} = \{\varepsilon, q\bar{p}, q\bar{p}p, q\bar{p}p\bar{p}, \ldots\}$$

this process is also called repeater since, after reception of a request on its passive port it “handshakes” indefinitely on the active. Now look at the following sets on $B$:

$$\{\varepsilon, q\bar{p}, q\bar{p}q, q\bar{p}pq, q\bar{p}pq\bar{p}, q\bar{p}pq\bar{p}p, q\bar{p}pq\bar{p}p\bar{p}, q\bar{p}pq\bar{p}p\bar{p}p, q\bar{p}pq\bar{p}p\bar{p}p\bar{p}\}$$
Neither of them is reorder-closed, then neither of them is a handshake language. For example, $\varepsilon q q p r_{(p, q)} q q p q$ but $q q p q$ is in the prefix-closure of both of the above sets, while $qqqp$ is in the prefix-closure of none. We leave it to the reader to figure out the reorder-closures of the above two sets and to show that the second’s is a handshake language while the first’s is not.

Finally, consider yet another set on $B$: 

$$\{\varepsilon, q p, q q, q q p q, q q q p, q q q q p, q q q q q p, q q q q q q q p\}$$

The reader can verify that it satisfies all the properties of a handshake language, we show that it does not satisfy those of positionality. Note that $q q p q p q$ and $q q q p p$ are two traces with the same position and that both are in the prefix-closure of the above set. However, while the first is actually an element of the set, the second is not and, conversely, while the second can be extended with $\bar{p}$, the first cannot. The language is not deterministic either since after the initial $q$ there is a mutually exclusive choice between $\bar{p}$ and $\bar{q}$.

3 Handshake Petri Nets

We assume some basic knowledge on Petri Nets, which we will use in their standard graphical representation [10]. Throughout the paper we will consider Petri nets in their unsafe version, where places are allowed to contain several tokens at the same time. This is not just for convenience. Unsafe nets are necessary to carry out our construction. We also stress that the nets we consider are in general not finite, in the sense that they may have infinitely many places and/or transitions.

Handshake Petri nets are characterized by a special “external interface” which reflects the structure of handshake ports. Let $I$ and $O$ be finite subsets of the set of transitions of a Petri net $G$. We call the triple $\langle G, I, O \rangle$ an interacting Petri net (ipn), where the transitions in $I$ are its input transitions and those in $O$ are its output transitions. A transition is external if it is an input or an output transition, internal otherwise.

An ipn $\langle G, I, O \rangle$ $t$-reduces to $\langle G', I, O \rangle$, $\langle G, I, O \rangle \xrightarrow{t} \langle G', I, O \rangle$, when the transition $t$ can fire in $G$ and the result of the firing is $G'$. We call execution of $\langle G, I, O \rangle$ any sequence of reductions starting from $\langle G, I, O \rangle$.

**Definition 3.1** An ipn $\langle G, I, O \rangle$ is a handshake Petri net (hpn) when:

- $I$ and $O$ have the same cardinality;
- input (output) transitions have exactly one incoming (outgoing) arc;
- each input transition is paired with an output transition by means of the following structure:
where the input transition is the upper one in the picture. Such a structure represents a port;

- at any time, each port contains exactly one token. In particular, when the token enables the input transition the port is passive, when it enables the output transition it is active.

Two hpn's may be composed by linking a set of ports of the first net with a set of ports of the second net. Each "link" must be between a passive and an active port and is done by adding two new places (and four new arcs) between them, as follows:

As expected, the new net will have as external transitions all and only those external transitions of the two original nets that have not been linked during the composition, and each of these inherited external transition will keep its status: inputs will stay inputs and outputs will stay outputs. It is easy to see that the composition of two hpn's is still an hpn, moreover composition of hpn's is trivially associative.

In the rest of this section we will show several examples of standard handshake components represented as handshake Petri nets. We will present each example within a specific subclass of the general model.

3.1 Handshake Marked Graphs

In the first stage we focus on marked graphs [9], which are Petri nets where each place has at most one incoming and at most one outgoing arc.

Marked graphs are significant as they allow to identify places with communication channels and in turn to represent all and only circuits which can be built out of channel, synchronization and fork operations. Moreover they have a special historical importance for the handshake protocol [12]. We call handshake marked graph a handshake Petri net which is also a marked graph.

Examples

Marked graphs represent the core of determinism. In particular they allow the representation of most deterministic handshake components: STOP, RUN, CON, SEQ, PAR, PAS, JOIN (in the notation of [14]). Two of these components are
represented below.\(^3\)

\[PAR\] (left) waits for a request on its passive port and then starts two handshakes in parallel on its active ports. Only after successful termination of both it acknowledges to the first request. \(SEQ\) (right) also waits for a request on its passive component, but then it starts its active ports in sequence, before finally acknowledging to the initial request.

The examples show that handshake marked graphs (or marked graphs in general) always react in the same way to a given stimulus. For example, \(SEQ\) always sends a request on its first active port after the reception of a request on its passive port. It can be shown that handshake marked graphs embed a particular subclass of handshake languages where each pair stimulus/response can be seen as a couple of brackets in the language and each trace becomes well-bracketed with respect to any of these couples, after a fixed number of closing brackets.

### 3.2 Deterministic Extensions

Marked graphs express only deterministic behaviors but not all deterministic behaviors are captured by marked graphs. As far as we know, no structural characterization of determinism in Petri nets exists in the literature. We propose a definition that completely characterizes determinism in the context of handshake nets.

**Definition 3.2** A (handshake) deterministic-choice graph is a (handshake) Petri net in which every place \(p\) with several outgoing edges is such that:

- Each post-transition \(t\) of \(p\) has a “guard”, a place whose only post-transition is \(t\);
- The set of all the guards of \(p\)'s post-transitions contains exactly one initial mark, so that one and only one post-transition is initially possible;
- Each of \(p\)'s post-transitions has exactly one outgoing edge to this set of guards, so that each time a transition fires another one is enabled;
- Each guard in this set has incoming edges only from \(p\)'s post-transitions, so that \(p\)'s post-transitions can never cause more than one guard to be enabled, even in the long run.

\(^3\) Although handshake Petri nets are formalized here for the first time, a similar representation for both components had already been given as far back as [11]. Actually, those pictures were an inspiration for this work.
Examples

As an example, consider $COUNT_N$ which, after reception of a request on its passive port, handshakes $N$ times on its active port. Then it acknowledges to the first request and returns to wait for an activation. In this case, the circuit needs to decide (deterministically, of course) when to acknowledge to a passive request (after $N$ handshakes on the active port). Here is the circuit $COUNT_2$, also known as $DUP$:

![Diagram of COUNT_2 circuit]

A deterministic-choice structure allows us to select each firing of a given transition and associate it to a brand new dedicated transition, as shown below:

![Diagram of selection and association]

We can do this both for an input (left) and for an output (right) transition.

3.3 General nets

In this subsection we present two examples of nondeterministic nets, the $OR$ (below left) and the $MIX$ (below right) handshake components:

![Diagram of OR and MIX handshake components]
OR has a passive and two active ports. When a request on the passive port arrives a request on either active port is sent. Acknowledge to this last request enables an acknowledge to the first one. As the picture shows, this example can be modeled by a free choice net (transitions with a shared precondition do not have any preconditions other than that).

Conversely, MIX has two passive and one active port. Each time an environment request arrives (on either passive port) MIX handshakes on its active port and after completion it acknowledges to the first request. If by the time the handshake on the active port completes the environment had sent a request on the other port, MIX chooses nondeterministically which request to acknowledge first.

This situation could not be described with a free-choice net since the choice of which request to acknowledge may not be a choice at all if the environment only made one request.

4 Soundness and Completeness

Consider an hpn \( H = (G, I, O) \), name its ports and let \( P_H \) be the set of these names. Now take \( d_H : P_H \to \{ \text{act}, \text{pas} \} \), the function which maps each port name \( p \) to the appropriate label, \( \text{act} \) if port \( p \) is active in \( H \), \( \text{pas} \) if it is passive. This allows us to define the handshake structure \( HS(H) = \langle P_H, d_H \rangle \).

Then for any port \( p \), name \( p \) (\( \bar{p} \)) its input (output) transition, and name \( \tau \) any internal transition. An execution \( t \) of \( H \) is quiescent when for no \( p \in P_H \) it can be extended as \( H \xrightarrow{t} (\tau) \xrightarrow{\bar{p}} \). We define \( HL(H) \) as the set of strings consisting of the external restriction of each quiescent execution of \( H \).

The main results of this paper, the soundness and completeness of the Petri net model, can be now formalized by the following theorems.

**Theorem 4.1** Let \( H \) be a handshake Petri net, then \( HL(H) \) is a handshake language on the handshake structure \( HS(H) \).

**Theorem 4.2** Let \( \sigma \) be a handshake language on a handshake structure \( \langle P, d \rangle \). Then there is an hpn \( H_\sigma \) such that \( HS(H_\sigma) = \langle P, d \rangle \) and \( HL(H_\sigma) = \sigma \).

The proof of soundness is a rather straightforward verification of the properties defining a handshake language. In the remaining of this section, we try to hint the proof of completeness (theorem 4.2). We must warn that the construction of \( H_\sigma \) we propose may lead to an infinite net, but let us also point out that an infinite representation is in general unavoidable. An example of language with no finite representation is the one which contains an infinite chain of (finite) traces where outputs are chosen according to a non recursive function.

Let us focus first on positional handshake languages. These languages make their choices according to the reached position, regardless of the particular interleaving followed in the execution. In the following we write \( a_i (\bar{b}_i) \) for the \( i \)th occurrence of input \( a \) (output \( \bar{b} \)). Then we can represent a choice as a pair made of a position \( \langle s \rangle \), and an output occurrence or a special symbol \( * \), where \( \langle [s]_\sim, \bar{b}_i \rangle \) expresses the choice of “playing” \( \bar{b}_i \) at \( [s]_\sim \) and \( \langle [s]_\sim, * \rangle \) the choice of doing nothing at \( [s]_\sim \).
Let $\sigma$ be a handshake language and $c$ a choice in $\sigma$. Let $t \in \sigma^\leq$, we say that $t$ allows $c$ in $\sigma$ ($t \rightarrow_{\sigma} c$) when $[t]_\sim = \text{fst}(c)$ and:

- $t \in \sigma$ if $\text{snd}(c) = \ast$;
- $t \cdot \text{snd}(c) \in \sigma^\leq$ otherwise.

We say that $t$ prevents $c$ in $\sigma$ ($t \rightarrow_{\sigma} c$) when it does not allow it.

If we consider positional strategies we see immediately that positions, rather than traces, allow choices. Moreover, since we only consider data-less communications and since outputs do not affect choices (by reordering), a position can be represented by a set of occurrences of distinct input messages, taking the last input occurrence of each thread.

Starting from the above observations and systematically using the selector structures introduced in Section 3.2 to select occurrences of input and output messages we are able to construct a handshake Petri net which corresponds to the given positional handshake language.

**Proposition 4.3** Let $\sigma$ be a positional handshake language on $\langle P, d \rangle$. Then there is an hpn $H_\sigma$ such that $HS(H_\sigma) = \langle P, d \rangle$ and $HL(H_\sigma) = \sigma$.

Since the construction associates single occurrences to transitions and since a move may occur infinitely many times, the constructed graph $H_\sigma$ is in general infinite.

In the non-positional case, reshuffling the threads of a trace may affect a choice. We first define an atomic reshuffling of a trace $t(= t' \cdot n \cdot m \cdot t''$) as a trace of the form $t' \cdot n \cdot m \cdot t''$, for $m$ and $n$ independent messages.

**Definition 4.4** Let $S$ be a set of pairs of the form $(a_i, b_j)$. $S$ is critical for a choice $c$ in a handshake language $\sigma$ just when, for all $t \in \sigma^\leq$ such that $[t]_\sim = \text{fst}(c)$,

$$\forall(a_i, b_j) \in S, t = t' \cdot b_j \cdot t'' \cdot a_i \cdot t'' \Rightarrow t \rightarrow_{\sigma} c.$$  

We write $\text{crit}_\sigma(c)$ for the set of minimal critical sets for $c$ in $\sigma$.

A similar notion is that of critical pair (the above being critical set of pairs) for $c$ in $\sigma$: a pair $(a_i, b_j)$ for which there is $s \in \sigma$ such that $s = s' \cdot b_j \cdot a_i \cdot s'' \rightarrow_{\sigma} c$ while $s' \cdot a_i \cdot b_j \cdot s'' \rightarrow_{\sigma} c$. If $(a_i, b_j)$ is a critical pair for $c$ in $\sigma$, we say that it is inverted in $t$ if and only if $t = t' \cdot b_j \cdot t'' \cdot a_i \cdot t''$.

**Lemma 4.5** Let $c$ be a choice in a handshake language $\sigma$. Let $t \in \sigma^\leq$, $[t]_\sim = \text{fst}(c)$:

$$t \rightarrow_{\sigma} c \iff \exists S \in \text{crit}_\sigma(c), \forall(a_i, b_j) \in S, t = t' \cdot b_j \cdot t'' \cdot a_i \cdot t''$$

**Proof (Sketch)** The direction right-to-left is almost immediate. For the other direction, $S$ is made of all the critical pairs which are inverted in $t$. Then we can take any trace $s$ with the same threads as $t$ and in which all the pairs of $S$ are inverted and prove that $s \rightarrow_{\sigma} c$ by induction on the number of atomic reshuffling needed to change $s$ into $t$. Note in particular that if an atomic reshuffling affects a choice, then it must consist of a critical pair which is inverted, as all the others are reorderings. So $S$ is critical, if it is not minimal we can take the minimal critical set contained in $S$ and we are done. \hfill $\square$

10
The construction of the net is a modification of the one for positional handshake languages, as we briefly sketch here. For any minimal critical set $S$ for $c$ in $\sigma$ and for all $\langle a_i, b_j \rangle \in S$ we connect transitions $a_i$, $b_j$ and $c$ as follows:

Suppose that the $j$th firing of $b$ occurs before the $i$th firing of $a$, and similarly for all the other pairs in $S$ (represented in the picture by the other incoming arcs in the precondition of transition $c$). Then $c$ is clearly prevented. Note that this scheme works for both a choice to output and a choice not to, as each choice corresponds to a transition in the graph.

The completeness theorem, specializes to several classes of nets and languages. For instance to the deterministic case.

**Theorem 4.6** Let $H$ be a handshake deterministic-choice graph, then $HL(H)$ is a deterministic handshake language on the handshake structure $HS(H)$. Conversely, if $\sigma$ is a deterministic handshake language on $\langle P, d \rangle$, there is a deterministic choice graph $H_\sigma$ such that $HS(H_\sigma) = \langle P, d \rangle$ and $HL(H_\sigma) = \sigma$.

As we mentioned, marked graphs correspond to a particular class of languages too, the *well-bracketed* ones. In this case there is even a construction yielding finite graphs. We still do not know any independent characterization of the nets that correspond to positional languages.

### 5 Conclusions

In this paper we presented a version of Petri nets, featuring a particular structure on external connections, that models the handshake protocol of communication. We showed that this model embeds the model of handshake games and strategies [4], but is more expressive.

Graphical representations as Petri nets are very close to the reality of circuits and are useful in explaining the circuits’ dynamics. Compared to trace models like games and strategies they go one level deeper: channels are unidirectional (as it usually happens in circuits) and bidirectionality can be obtained by pairing.

This higher level of intensionality brings us to reconsider the model of handshake Petri nets, which can be seen not only as a semantic model for handshake circuits, but also as their syntax. Carrying on this track we could attempt to provide a normal form for handshake Petri nets since, as we have seen, two different nets may have the same behavior.

Also, the graphical representation may drive the definition of a more standard notion of syntax in the form of a process calculus. Van Berkel already proposed the
handshake process calculus [14] (see also [6]). However, their goal was not to capture all possible handshake behaviors and a complete process calculus of handshake circuits is still wanted.

Further directions include a deeper analysis of data-exchange, as in this paper we focused especially on data-less communications. It would also be interesting to exploit those subclasses which allow finite representations of a given subset of handshake languages, as we tried to hint in Section 3.1.

Note finally that in recent times, the foundational research on the field is starting to awake again: besides [4], other efforts have been made to apply game semantics to the synthesis of HDLs for handshake circuits [5]. This strengthens our belief on the importance of the communication protocol inside the universe of asynchrony, the idea that the current model is just the core of a larger representation.

Acknowledgments

The first author would like to thank Simona Ronchi della Rocca, Felice Cardone, Pierre-Louis Curien for interesting discussions and for their support. Thanks to Gianfranco Balbo, Marco Becatti, Gianfranco Ciardo and Lucia Pomello for their advice on Petri nets.

References

A Appendix

We state two definitions which will be useful in the proofs which follow.

Definition A.1 Let $s$ and $t$ be two handshake traces on a given handshake structure, such that $s \sim t$. We define $d_{\sim}(s,t)$, the homotopy distance between $s$ and $t$ as follows:

- $d_{\sim}(s,t) = 0 \iff s = t$
- $d_{\sim}(s,t) = 1 \iff (s = s' \cdot m \cdot n \cdot s'') \land (t = s' \cdot n \cdot m \cdot s'')$, for two messages $m$ and $n$;
- In general $d_{\sim}(s,t) = k > 0$ if and only if $s \neq t$ and there is a sequence of $k + 1$ (and no less) traces $s_0, s_1, \ldots s_k$ such that $s = s_0$, $s_k = t$ and for all $0 \leq i < k$, $d_{\sim}(s_i, s_{i+1}) = 1$.

The second definition is less significant, but still useful. Let $H$ be an hpn and $t$ and $t'$ two distinct external transitions of $H$. We call $t'$ the complement of $t$, and viceversa, when $t$ and $t'$ belong to the same port.

Proof of Proposition 2.4 Let $\sigma$ be a deterministic handshake language and let $s, t \in \sigma^{\leq}$, such that $s \sim t$. Suppose that $s \cdot \bar{s} \in \sigma^{\leq}$. We show that $t \cdot \bar{s} \in \sigma^{\leq}$ by induction on $d = d_{\sim}(s,t)$:

- $d = 0$. $s = t$, then $t \cdot \bar{s} \in \sigma^{\leq}$;
- $d = l + 1$. There is a sequence $s = t_0, \ldots t_{l+1} = t$ such that $d_{\sim}(t_i, t_{i+1}) = 1$ and $t_i \cdot \bar{s} \in \sigma^{\leq}, \forall 0 \leq i \leq l$. Let $t_l = t' \cdot m \cdot n \cdot t''$ and $t_{l+1} = t' \cdot n \cdot m \cdot t''$. If $m$ is an output or $n$ an input, $t_{l+1} \cdot \bar{s} \equiv t_l \cdot \bar{s}$, which implies $t_{l+1} \cdot \bar{s} \in \sigma^{\leq}$. Let $m$ be an input and $n$ an output, then we prove $t_{l+1} \cdot \bar{s} \in \sigma^{\leq}$ by induction on the length of $\bar{s}$.

- $\bar{s} = \varepsilon$. Then $t_{l+1} \cdot \varepsilon = t_{l+1} \in \sigma^{\leq}$ by hypothesis;
- $\bar{s} = \bar{s}' a$. $t_{l+1} \cdot \bar{s}' \in \sigma^{\leq}$ by hypothesis. If $a$ is an input, $t_{l+1} \cdot \bar{s}' \cdot a \in \sigma^{\leq}$ by receptivity. Then let $a$ be an output and let $a_1, \ldots a_k$ be all the outputs that $\sigma$ can send at $t_{l+1} \cdot \bar{s}'$. Then determinism implies $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k \in \sigma^{\leq}$. Note also that if an output was possible at $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$, it would also be possible at $t_{l+1} \cdot \bar{s}'$, by reordering. Then $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$ is passive in $\sigma$. Then also $t_l \cdot \bar{s}' \cdot a_1 \ldots a_k \in \sigma$, as it reorders $t_{l+1} \cdot \bar{s}' \cdot a_1 \ldots a_k$. Then $a \in \{a_1, \ldots a_k\}$, as $a$ is an output and $t_l \cdot \bar{s}' \cdot a \in \sigma^{\leq}$. Then $t_{l+1} \cdot \bar{s}' \cdot a \in \sigma^{\leq}$.

The proof that $s \in \sigma \Rightarrow t \in \sigma$ is even simpler, as the outer induction alone will do. □

Proof of Theorem 4.1 We proceed by successive steps:

- $HL(H)$ is a set of finite handshake traces on $HS(H)$. The structure of handshake ports implies that all observable threads alternate inputs and outputs, starting with an input on passive ports and with an output on active ports. Moreover, executions are finite sequences of firings, then their external restrictions are also finite.
- $HL(H)$ is non-empty. By definition an execution is a sequence of transitions, then the empty sequence is also an execution and its external restriction is an external
trace. If the empty sequence is not quiescent in $H$ it is the prefix of a quiescent execution (as we will show in the next point).

- $HL(H)$ is closed with respect to passive prefixes. $HL(H)$ is the set of external traces of all the quiescent executions of $H$. By contradiction, suppose there is $s \in \text{Pas}(HL(H))$ which does not come from a quiescent execution of $H$. Then there is an extension of this execution which, after a sequence of internal firings, lets an output transition $\bar{o}$ fire. If this is still not quiescent we can do the same thing over again. Note however that $H$ only contains a finite number of external ports and could not continue to output indefinitely, eventually it shall stop and wait for an input, thus reaching a quiescent execution. Then $s \cdot \bar{o} \in HL(H)^\omega$ and $s$ is not passive (contradiction).

- $HL(H)$ is reorder-closed. Reorder-closedness comes as a consequence of the fact that no output transition may block any other transition but its complement and no input transition may block another input transition. (We say that $t$ blocks $t'$ in $H$ just when $H$ contains a path from $t$ to $t'$, where each place has at most one incoming arc.)

- $HL(H)$ is receptive. Handshake deterministic-choice graphs are unsafe, that means that places may contain an unlimited number of tokens. So, every time an input transition is enabled to fire it can. Note also that the enabling of an input transition depends only on the alternation with its complement output transition (besides, of course, on the presence of an incoming token). Then $HL(H)$ is receptive.

\[\square\]

**Proof of Proposition 4.3** We set up $H_o$’s external structure by providing both an input and an output transition for each port $p \in P$ and by pairing them together by means of a port structure, as showed in definition 3.1. In particular, the choice of an active or of a passive port structure is taken according to the label $d(p)$. Then we can already state $HS(H_o) = \langle P, d \rangle$. Note also that we have a specific external transition for each message in the alphabet. So, now the internal structure. The occurrence selectors defined in Section 3.2 allow us to associate a new (internal) transition to each occurrence of message: we use specific selectors for inputs as for outputs. The next step is to associate a transition to each position. Recall that a position can be represented as a set containing the last input occurrence of each thread. Then we take a new transition and we link each transition associated to any of those input occurrences into it: the link is a direct arc-place-are one. We also add a transition for each choice $c$ allowed at a given position $[s]_\omega$. In particular, if $c$ does not stand for the choice to do nothing, we link $[s]_\omega$’s transition to $c$’s transition, again by a direct arc-place-are link. Note however that $c$ might be in mutual exclusion with another choice $c'$ at $[s]_\omega$, then we need a shared precondition before the corresponding transitions. But the choice of which one to fire should be made once and for all, then this same precondition should be used in any position where the two choices are allowed and mutually exclusive (we draw several outgoing
arcs from each choice to mean this):

A special treatment is reserved for the do-nothing choice. In this case we add a transition with no outgoing arcs and put it in mutual exclusion directly with \( p \)'s transition:

When the choice to do nothing is taken, \( H_\sigma \) has to wait for another input (thus moving to a new position) before doing anything else. Let us now move to the output side, where each output occurrence \( \bar{p}_i \) might be enabled in several positions. Then we make the arcs coming from the choice of \( \bar{p}_i \) at each of these positions converge into a unique place, which will have an outgoing arc towards \( \bar{p}_i \)'s transition. One might object that so doing the same output occurrence might fire twice. But this is prevented by the selector structure (Section 3.2) which ensures that each transition associated to an occurrence of \( \bar{p} \) may fire at most once.

The construction is finally complete, now we prove by induction that \( s \in \sigma^\leq \) if and only if \( s \in HL(H_\sigma)^\leq \).

- \( s = \varepsilon \). Trivial since both \( \sigma \) and \( HL(H_\sigma) \) are handshake languages.
- \( s = s' \cdot a \). Let \( a \) be an input. Both \( \sigma \) and \( HL(H_\sigma) \) are handshake languages on the same handshake structure \( \langle P, d \rangle \). Then any direction we look, \( s'a \) must be a handshake trace on \( \langle P, d \rangle \). Since \( s' \) is a prefix of both languages, \( s'a \) is too (receptivity). Now let \( a \) be the \( i \)th occurrence of output \( \bar{p} \). \( s \bar{p} \in \sigma^\leq \) means that \( \bar{p}_i \) is allowed by the position \( [s]_{\leq} \) in \( \sigma \) and that no mutually exclusive choice has
been chosen yet. Then in $H_\sigma$, $[s]_\sim$’s transition enables the transition associated to the choice of $\tilde{p}_i$ at $[s]_\sim$. Plus, if ever there was a shared precondition among the choice of $\tilde{p}_i$ and another choice at $[s]_\sim$, we may assume that it still contains a mark since the other choice has not produced any effect so far. Then $\tilde{p}_i$’s transition may fire because it has not yet fired and because all the transitions associated to previous occurrences of $\tilde{p}$ have already fired in $s$. Then $\tilde{s}\tilde{p} \in HL(H_\sigma)^\leq$. On the other hand, $\tilde{s}\tilde{p} \in HL(H_\sigma)^\leq$ implies that the transition associated to the choice of $\tilde{p}_i$ at position $[s]_\sim$ is enabled by the transition associated to $[s]_\sim$ in $H_\sigma$. Then $[s]_\sim$ allows $\tilde{p}_i$ in $\sigma$, by definition of $H_\sigma$. Moreover, if there was another choice excluding $\tilde{p}_i$ at $[s]_\sim$ in $\sigma$, this was not chosen inside $s$. Then $\tilde{s}\tilde{p} \in \sigma^\leq$.

Now, $s$ is passive in $\sigma$ if and only if the transition $[s]_\sim$ does not have any outgoing arcs in $H_\sigma$, that is if and only if $s$ is passive in $HL(H_\sigma)$. $s$ is a non-passive trace in $\sigma$ if and only if the transition $[s]_\sim$ has a shared precondition with a transition which has no outgoing arcs in $H_\sigma$, that is if and only if $s$ is a non-passive trace in $HL(H_\sigma)$.

Proof of Theorem 4.6 The proof of the completeness part of the theorem is a simplification of the proof of Proposition 4.3. As for the the soundness part, the only properties left to prove are determinism’s two, given that we already proved the preliminary properties for Theorem 4.1. For both of them, the proof is based on the following simple observation. In a deterministic-choice graph, even if a place may have several outgoing arcs, only one of its post-transitions is actually enabled at any given time: each one has a guard and only one guard contains a mark in the initial state; successively, the firing of the enabled post-transition takes away a mark from its guard and puts it into another guard. This prevents any situation of confusion, so that once a transition is enabled it will stay enabled until it fires.

Also, given two executions $ex'$ and $ex''$, we can define an execution $ex$ which completes $ex'$ with those firings which occur in $ex''$ and not in $ex'$ itself. We show how to do this by providing a constructive algorithm which gradually deletes the two original strings $ex'$ and $ex''$ while writing $ex$. We initially set $ex$ to the empty string. If at a given time $ex'' = a \cdot u''$ and $ex' = u' \cdot a \cdot v'$, where $a$ does not appear in $u'$, we append $a$ to $ex$ while removing it from the two original strings. So that $ex'$ becomes $u'v'$ and $ex''$ becomes $a \cdot u''$. If $a$ does not appear in $ex'$ we remove it from $ex''$ and we append it at the end of $ex'$. If $ex'' = \varepsilon$, we append what is left of $ex'$ at the end of $ex$. Eventually, $ex$ will consist of all the firings of $ex'$ (possibly in a different order) followed by the firings of $ex''$ that were not there in $ex'$. About the order of the firings, note that if $ex'$ and $ex''$ had the same external trace, $ex$ would still have that external trace.

Now, let $\tilde{s}\tilde{p} \in HL(H)^\leq$. Recall that $\tilde{s}\tilde{p}$ is the prefix of an external trace of a quiescent execution of $H$. Then $\tilde{s}\tilde{p}$ is also the external trace of a prefix of a quiescent execution, then the external trace of an execution $ex'\tilde{p}_i$ of $H$. We prove the first property by induction on the length of $ex'$. If it is empty, then it is not quiescent because $\tilde{p}$ extends it. Moreover, any sequence of internal transitions can be completed with $\tilde{p}_i$, as explained above. If $ex' = u \cdot a$, let $v$ be a sequence of internal

\footnote{Recall also that all deterministic handshake languages are positional (Prop. 2.4).}
firings such that \( u \cdot v \) is an execution. Then \( v \) can be completed with \( a \cdot \bar{p} \) to obtain \( v' \cdot \bar{p} \), a sequence which extends \( u \). Then \( s \notin HL(H) \). For the second property, let also \( s\bar{q} \in HL(H)^\leq \) and let \( ex''\bar{q} \) be an execution whose external trace is \( s\bar{q} \). Just as above we can interleave \( ex'\bar{p} \) and \( ex''\bar{q} \) so to obtain execution \( ex \) whose external trace is \( s\bar{p}\bar{q} \in HL(H)^\leq \). \( \square \)

**Proof of Lemma 4.5** The direction right-to-left is almost immediate: if there exists such a critical set, by definition \( t \rightarrow^\sigma c \). Then suppose that \( t \rightarrow^\sigma c \). Let \( S \) be the set of all critical pairs for \( c \) in \( \sigma \) which are inverted in \( t \) and let \( s \) be a handshake trace with the same threads as \( t \)\(^5\) and in which all pairs of \( S \) are inverted. We prove by induction on \( d = d_\prec(s, t) \) that \( s \rightarrow^\sigma c \).

- \( d = 0 \). \( s = t \), then \( s \rightarrow^\sigma c \);
- \( d = l + 1 \). There is a sequence \( t = t_0, \ldots, t_{l+1} = s \) such that \( d_\prec(t_i, t_{i+1}) = 1 \) and \( t_i \rightarrow^\sigma c_i \), \( \forall 0 \leq i \leq l \). By contradiction assume \( t_{l+1} \rightarrow^\sigma c \). Let \( t_l = t' \cdot m \cdot n \cdot t'' \) and \( t_{l+1} = t' \cdot m \cdot n \cdot t'' \). If \( m \) is an input or \( n \) an output, \( t_l \) \( \triangleright t_{l+1} \). Then since \( t_{l+1} \rightarrow^\sigma c \), also \( t_l \rightarrow^\sigma c \). Contradiction. Then \( m \) is an output, say the \( i \)-th occurrence of \( b_i \) and \( n \) is an input, say the \( i \)-th occurrence of \( a \). By definition \( \langle a_i, b_j \rangle \) is a critical pair and since the sequence \( t_0, \ldots, t_{l+1} \) is minimal by definition of \( d_\prec \), \( \langle a_i, b_j \rangle \in S \).

But \( \langle a_i, b_j \rangle \) is not inverted in \( s \): contradiction!

Then \( S \) is critical for \( c \) in \( \sigma \). If \( S \) is not minimal we just need to take the minimal critical set contained in \( S \) and we are done. \( \square \)

**Proof of Theorem 4.2** We already described the general construction of \( H_\sigma \) for \( \sigma \) positional (proof of proposition 4.3) as well as its extension to the non-positional case (end of section 4). Lemma 4.5 justifies this extended construction by telling us that an “exception” to positionality has all the pairs of a critical set inverted and, vice versa, if a trace has all the pairs of a critical set inverted, then it is an exception to positionality. Then the proofs that \( s \in \sigma^\leq \iff s \in HL(H_\sigma)^\leq \) and \( s \in \sigma \iff s \in HL(H_\sigma) \) are just adaptations of the corresponding proofs that we gave for proposition 4.3. \( \square \)

---

\(^5\) To be more precise we should take \( s \) from a larger set, where the number of input occurrences in each thread of \( s \) is equal to the number of input occurrences in the corresponding thread of \( t \). This allows a thread of \( s \) to differ from the corresponding thread of \( t \) by an output occurrence. However reorder-closeness implies that outputs do not affect choices, so that we can assume \( s \) and \( t \) have exactly the same threads.
Making the unobservable, unobservable

Julian Rathke
ecs, University of Southampton

Paweł Sobociński ¹
ecs, University of Southampton

Abstract

Behavioural equivalences of many various calculi for modelling distributed systems differ significantly because the properties which can be observed through interaction depend heavily upon their mode of communication. A typical approach to describing the semantics of communicating processes is to provide a labelled transition system (LTS) which captures the interaction potential of the individual processes within a larger system. In many cases, a natural rendering of this LTS leads to too fine a semantics as unobservability of certain communications is not accounted for.

We propose that a standard approach to augmenting LTSs allows morally unobservable communications to actually be modelled as unobservables in the semantics. This approach derives from a rule initially given by Honda and Tokoro to account for unobservability of reception in the asynchronous π-calculus. We examine the implications of adding such rules to LTS with respect to the proving behavioural equivalences for various synchronisation mechanisms.

Keywords: Please list keywords from your paper here, separated by commas.

1 Introduction

Semantic descriptions of processes in the process algebra/calculus tradition have come to be dominated by the use of labelled transition systems (LTSs) to describe the nature of the interaction contributed by each participant in a communication. This approach was inspired by early work of Plotkin [11] on structural operational semantics and developed by Milner as a means of providing such semantics for CCS [8]. This development, namely the addition of labels, has an interesting artefact. Labelled transition systems now not only give structural definitions of reduction but also allow programs to be compared for equivalence without the need for examining their behaviour within all enclosing contexts. The dual purpose nature of these transition systems does have an unfortunate side-effect in that the labelled transitions which are useful to describe structurally defined reduction relations do not always coincide with observability properties of language with respect to program

¹ Research partially supported by EPSRC grant EP/D066565/1.

©2008 Published by Elsevier Science B. V.
equivalence. This problem is particularly acute when the underlying communication mechanism between processes is anything less than fully synchronous. In a nutshell, the natural looking structural LTSs for a given language may yield too fine an equivalence relation if the labels are used for comparing programs.

A well-known example of this lies in the unobservability of reception in languages featuring asynchronous communication. This issue is studied thoroughly in [5,1] in the setting of the asynchronous π-calculus in which it is demonstrated that (weak) bisimilarity on the structural LTS for the π-calculus does not correctly capture a notion of (weak) observable equivalence in this language. There are two different, but related, approaches to resolving this problem: [1] proposes a variant definition of bisimilarity while [5] proposes an additional, non-structural labelled transition rule to model the unobservability of reception. For technical reasons concerning uniformity of definition and difficulties in establishing transitivity of the variant bisimilarity, we prefer the latter approach by Honda and Tokoro [5]. Their innovation was to blur the observability of reception actions in π-calculus by augmenting the standard LTS with (essentially) the following Honda Tokoro (HT) style rule:

\[ P \xrightarrow{a?b} P \parallel a!b \]

This rule allows a process which genuinely performs a receive action \( P \xrightarrow{a?b} P' \) to be compared with a process who needn’t be able to perform any actions, let alone a receive action. By doing this we see that the contextually unobservable receive actions in the asynchronous world actually become unobservable in their LTS model.

In this paper we pursue the idea of Honda and Tokoro by generalising their approach of using additional LTS rules to guarantee the correct unobservability properties in a structurally defined LTS. We do this by identifying how we can routinely add HT rules to an LTS without compromising the soundness and completeness of its associated (bi)similarity with respect to a simple notion of contextual equivalence. We demonstrate this approach in three different scenarios: fully asynchronous, output asynchronous and fully synchronous communication in CCS-like calculi. Furthermore, we discuss how to specialise the resulting LTSs by only adding the necessary additional unobservability rules – in some calculi some actions are inherently observable and do not require a HT rule. This analysis is useful because it results in more manageable LTSs.

It is important to state that this paper does not address the issue of characterising when particular actions are or are not contextually unobservable, as studied in [9]. The focus here is on the general applicability and correctness of the unobservability rules.

2 Structural rules, LTSs and (bi)simulations

We need to recall the standard notion of a labelled transition system. In all of our examples the set of states coincides with the set of processes of the calculus at hand.

**Definition 2.1** Suppose that \( A \) is a set of observations and \( S \) is a set of states. An \((A,S)\)-labelled transition system (LTS) is a ternary relation \( T \subseteq S \times A \times S \). We
will write \( s \overset{\alpha}{\rightarrow} s' \) for \((s, \alpha, s') \in \mathcal{T}\).

Traditionally, LTSSs defined by a set of SOS rules have been used to:

\begin{itemize}
\item define the reduction relation compositionally by considering the structure of terms;
\item define the possible “interactions” and reason about the resulting (labelled) preorders/ equivalences;
\item use labelled preorders/equivalences to characterise the observational preorder/equivalence.
\end{itemize}

There is a natural tension between the first and the third point. The first may require us to observe the intensional structure of the terms in order to characterise the possible reductions. The third requires us to characterise the structure that can be observed. Examples of and techniques for relaxing this tension are the remit of this paper.

Our LTSS will normally be defined using SOS rules. Given an LTS \( \mathcal{T} \) and a rule \( \psi \), let \( \psi(\mathcal{T}) \) be the LTS obtained by adding the extra transitions which can be derived using (possibly several applications of) \( \psi \) from the existing transitions of \( \mathcal{T} \). This is easily defined as a least fixed point. Seen as an endofunction on the class of LTSS, \( \psi \) is idempotent.

This process can be extended to sets of rules \( \Psi = \{ \psi_i \}_{i \in I} \). As before, given an LTS \( \mathcal{T} \), let \( \Psi(\mathcal{T}) \) denote the smallest extension of \( \mathcal{T} \) which contains all those transitions that follow from the derivations that use the rules in \( \Psi \). When we speak about the LTS defined by a set of rules \( \Psi \), we mean the LTS \( \Psi(\emptyset) \) where \( \emptyset \) is the empty LTS.

**Definition 2.2** Given two \((A,S)\)-LTSS \( \mathcal{T}_1 \) and \( \mathcal{T}_2 \), a simulation from \( \mathcal{T}_1 \) to \( \mathcal{T}_2 \) is a binary relation \( R \subseteq S \times S \) such that if \( P R Q \) and \( P \overset{\alpha}{\rightarrow} P' \) in \( \mathcal{T}_1 \) then \( \exists Q' \) such that \( Q \overset{\alpha}{\rightarrow} Q' \) in \( \mathcal{T}_2 \) and \( P' R Q' \). A simulation \( R \) is a bisimulation iff \( R^{-1} \) is a simulation.

Given an LTS \( \mathcal{T} \), \( \preceq_\mathcal{T} \) is similarity, the largest simulation, while \( \sim_\mathcal{T} \) is bisimilarity, the largest bisimulation.

### 3 Full asynchrony

Consider the following simple language \( \mathcal{L}_f \) of finite processes generated by:

\[ P ::= 0 \mid a! \mid a? \mid P \parallel Q \mid \tau \]

We can think of a typical term as a textual representation of a “soup” of processes \([2]\) which interact with each other. We thus ask that parallel composition behave appropriately: a process is an equivalence class of terms quotiented by the smallest congruence in which \( \parallel \) is associative, commutative and has 0 as identity.

The execution semantics (i.e. what a program does) is captured by the smallest (unlabelled) transition system \( \rightarrow \) that contains \( \tau \rightarrow 0 \) as well as \( a! \parallel a? \rightarrow 0 \) (for any \( a \)) and is closed under parallel composition. We will refer to the edges in the above transition system as *interactions* or *reductions*. 

Assume that an interaction has an observable effect, for concreteness let us say that each interaction generates a certain amount of heat which can be detected by the observer’s sophisticated sensors. The external observer cannot see the internals of a process (the identity and number of the atoms) but can experiment with it only by:

- introducing new ingredients into the soup;
- measuring the change in heat, i.e. observing interactions.

Our most controversial assumption is that time can be slowed down to such an extent that no two reductions can ever happen truly concurrently, that is, the observer can observe the reductions one at a time. This is normally referred to as interleaving.

For example, the observer can distinguish between $0$ and $a!$; the experiment runs as follows: the observer introduces $a?$ into the both the soups and observes that while the first soup stays at constant temperature, the second soup gets a little bit hotter.

The above ingredients induce a canonical preorder and equivalence on the set of processes: the preorder being the largest (reduction) simulation which is a precongruence (wrt to $\parallel$) and the equivalence being the largest reduction bisimulation which is a congruence. For us, these two relations capture the power of an external observer.

**Definition 3.1** Reduction precongruence is the largest relation $\preceq$ which:

(i) is a simulation with respect to reduction, i.e. $P \preceq Q$ and $P \rightarrow P'$ then there exists $Q'$ such that $Q \rightarrow Q'$ and $P' \preceq Q'$;

(ii) is stable wrt $\parallel$, i.e. $P \preceq Q$ then for any $R$ we have $P \parallel R \preceq Q \parallel R$.

**Definition 3.2** Reduction congruence is the largest relation $\simeq$ which:

(i) satisfies conditions (i) and (ii) of Definition 3.1;

(ii) is symmetric.

In the following we will attempt to reason about the contextually defined reduction (pre)congruence by characterising the possible experiments as labelled transitions; the procedure (initial state, experiment, final state) will be a transition in an LTS. Roughly, this means that we are in effect replacing “contexts” of Definitions 3.1 and 3.2 with “labels” in a LTS. The goal is to use labelled preorders and equivalences, such as bisimulation, as proof methods for reasoning about reduction (pre)congruence.

The first kind of experiment is the simplest, the observer does not need to do anything and can observe a rise in temperature due to the presence of a $\tau$ within the term. Let us call this experiment $\tau$. We can characterise inductively some of the processes for which this experiment is successful:

\[
\begin{align*}
\tau \rightarrow 0 & \quad (\text{TAU}) \\
\frac{P \rightarrow P'}{P\parallel Q \rightarrow P'\parallel Q} & \quad (\tau \parallel) 
\end{align*}
\]

Another experiment is the following: the observer introduces an output on $a$ ($a!$) and observes that there is a rise in temperature. Let us name (slightly counterintuitively)
this kind of experiment $a\?$?. The reason for this label is that this is the capability (input) that has been observed. Also this kind of experiment can be characterised inductively:

$\begin{align*}
a? & \xrightarrow{a?} 0 & (\text{In}) \\
\end{align*}$

$\begin{align*}
P \xrightarrow{a?} P' & \xrightarrow{P\parallel Q \xrightarrow{a?} P\parallel Q} (\text{In}) \\
\end{align*}$

Dually, the observer can introduce an input to observe an output.

$\begin{align*}
a! & \xrightarrow{a!} 0 & (\text{Out}) \\
\end{align*}$

$\begin{align*}
P \xrightarrow{a!} P' & \xrightarrow{P\parallel Q \xrightarrow{a!} P\parallel Q} (\text{Out}) \\
\end{align*}$

Finally, if the observer’s experiments lead her to conclude that there is an input on $a$ in soup $A$ and an output on $a$ in soup $B$, then a heating should be observed in the combined soup without additional ingredients. Note that the label is the same as our first experiment: this is because, from the observer’s point of view, the same (empty) process is provided to $P \parallel Q$ for a change in heat to be observed.

$\begin{align*}
P \xrightarrow{a?} P' & \xrightarrow{P\parallel Q \xrightarrow{a?} P\parallel Q} \quad \text{(Comm)} \\
\end{align*}$

Let $\Phi_f \overset{\text{def}}{=} \{(\tau), (\tau\parallel), (\text{In}), (\text{In}\parallel), (\text{Out}), (\text{Out}\parallel), (\text{Comm})\}$. Consider the lts $C_f \overset{\text{def}}{=} \Phi_f(\emptyset)$. The following lemma shows that our intuition of this labelled transition system as a “logbook” of certain experiments is correct: for each kind of label $\alpha$ there is a process $\chi_\alpha$ which is added by the experimenter to observe a reduction.

Lemma 3.3 Let $\chi_\alpha = a?$, $\chi_a = a!$ and $\chi_\tau = 0$. Then $P \xrightarrow{\alpha} P'$ implies $P \parallel \chi_\alpha \rightarrow P'$.

For $\tau$-labelled transitions, the other direction also holds.

Lemma 3.4 $P \xrightarrow{\tau} P'$ iff $P \rightarrow P'$.

Indeed, reasoning on the lts is sound wrt to observation.

Lemma 3.5 (Soundness of $C_f$) $\not\sim C \subseteq \not\sim$ and $\sim C \subseteq \sim$.

Proof. It suffices to show that $\not\sim$ is a reduction simulation and stable under parallel composition. It is a reduction simulation because it is a $\tau$-simulation and $\tau$s coincide with reductions (Lemma 3.4). To show that it is stable under $\parallel$, it suffices to show that $\{(P \parallel Q, Q \parallel R) \mid P \not\sim Q\}$ is a simulation. This is easily checked by cases, the most interesting being $P \parallel R \xrightarrow{\tau} P' \parallel R'$ where, for some $a$, $P \xrightarrow{a!} P'$ and $R \xrightarrow{a?} R'$. Then, by assumption, $\exists Q', Q \xrightarrow{a!} Q' \land P' \not\sim Q'$. But then $Q \parallel R \xrightarrow{\tau} Q' \parallel R'$ which is in the relation. The proof for the case of bisimulation and reduction congruence proceeds similarly.

However, the reverse inclusion (completeness) does not hold. The lts allows us to distinguish processes which are observationally not distinguishable. For example:

Lemma 3.6 Let $P_1 \overset{\text{def}}{=} a? \parallel a$! and $P_2 \overset{\text{def}}{=} \tau$. Then $P_1 \not\sim C P_2$ but $P_1 \not\sim C P_2$.

Proof. Clearly $P_1 \not\sim C P_2$ since $P_1 \xrightarrow{a!} a?$ which cannot be matched by $P_2$. 

5
We will show that \( \mathcal{R} \overset{\text{def}}{=} \{(P_1 \parallel R, P_2 \parallel R) \mid R \text{ a process}\} \cup \Delta \) is closed under reduction and stable under parallel composition. Suppose that \( P_1 \parallel R \rightarrow Q \). Then:

(i) \( R \rightarrow R' \) and \( Q = P_1 \parallel R' \). Then also \( P_2 \parallel R \rightarrow P_2 \parallel R' \) and \( Q \mathcal{R} (P_2 \parallel R') \) by construction;

(ii) \( P_1 \rightarrow 0 \) and \( Q = R \). But also \( \tau \rightarrow 0 \) so \( \tau \parallel R \rightarrow R \);

(iii) \( R = a! \parallel R' \) and \( P_1 \parallel R \rightarrow a! \parallel R' = R \). But also \( \tau \parallel R \rightarrow R \).

(iv) \( R = a? \parallel R' \) which is similar to the previous case. \( \square \)

We also have \( P_2 \preceq P_1 \) and, moreover, \( P_1 \simeq P_2 \); the two processes are reduction congruent. In fact, already \( a! \preceq P_2 \) and \( a? \preceq P_2 \), although, in each of these cases the other direction does not hold since \( P_2 \) can reduce. Of course, neither \( a? \preceq a! \) nor \( a! \preceq a? \).

We have shown that there is a mismatch between our logbook (the \( \text{LTS} \)) and the actual power of experiments. This can be expressed succinctly by noting that the converse of Lemma 3.3 does not hold and, more than this, no characterisation of the labels by contexts is possible. This is implied by the following result.

**Theorem 3.7** If, for an \( \text{LTS} \ X \), there exist processes \( \chi_\alpha \) such that

\[
P \overset{\alpha}{\longrightarrow}_X P' \text{ iff } P \parallel \chi_\alpha \rightarrow P'
\]

then \( \preceq \subseteq \preceq_X \) and \( \simeq \subseteq \simeq_X \).

**Proof.** We will show that \( \preceq \) is a \((X)\)-simulation. Suppose that \( P \preceq Q \) and \( P \overset{\alpha}{\longrightarrow} P' \). Then \( P \parallel \chi_\alpha \rightarrow P' \) and since \( \preceq \) is stable under \( \parallel \) and closed under reduction we have that there exists \( Q' \) such that \( P' \preceq Q' \) and \( Q \parallel \chi_\alpha \rightarrow Q' \). But, by assumption, the final part implies that \( Q \overset{\alpha}{\longrightarrow} Q' \). The same reasoning shows that \( \simeq \) is a bisimulation. \( \square \)

**Corollary 3.8** There do not exist \( \chi_\alpha \) such that \( P \overset{\alpha}{\longrightarrow} P' \) iff \( P \parallel \chi_\alpha P' \).

**Proof.** If such a set existed then by Theorem 3.7 we would have \( \preceq \subseteq \preceq_C \) which we know is not true by the conclusion of Lemma 3.6. \( \square \)

What has gone wrong? A moment’s reflection about the processes of Lemma 3.6 confirms that, for instance, our idea that \( P \overset{a!}{\longrightarrow} P' \) tests for the presence of an output (and dually, an input) on \( a \) in \( P \) is unimplementable. Indeed, the experimenter which provides an \( a? \) process and observes an interaction can conclude that either \( P \) has an output on \( a \) or that a reduction was already possible in \( P \). The following rules\(^2\) take into account the possibility of the latter:

\[
\frac{P \overset{\alpha}{\longrightarrow} P'}{P \overset{a?}{\longrightarrow} P'[a!]} \quad \text{(InHT)} \quad \frac{P \overset{\alpha}{\longrightarrow} P'}{P \overset{a!}{\longrightarrow} P'[a?]} \quad \text{(OutHT)}
\]

The presence of the extra component in the results is just the process that the experimenter provided, the reduction was already present in \( P \) and so this additional

\(^2\) These rules are named after Honda and Tokoro’s rule for asynchronous \( \pi \)-calculus, although here they are modified for strong equivalences.
The process was not consumed. Letting $\Psi_f \overset{\text{def}}{=} \{\text{InHT}, \text{OutHT}\}$, we will consider the LTS $\mathcal{HT}_f \overset{\text{def}}{=} \Psi_f \mathcal{C}_f$. We shall sometimes refer to those transitions which are in $\mathcal{HT}$ but not in $\mathcal{C}$ as Honda-Tokoro transitions.

The following lemma shows the relationship between the $a!$- and $a?$-labelled transitions in the two LTSs.

**Lemma 3.9** For $\alpha \in \{a!, a?\}$:

$$P \overset{\alpha}{\rightarrow}_{\mathcal{HT}} P' \iff P \overset{\alpha}{\rightarrow}_{\mathcal{C}} P' \lor (P \overset{\tau}{\rightarrow} P'' \land P' = P'' \parallel \chi_{\alpha})$$

The $\tau$-labelled transitions are unchanged. We use these observations to show that simulation (bisimulation) on $\mathcal{HT}_f$ remains sound for reduction precongruence (congruence).

**Lemma 3.10 (Soundness of $\mathcal{HT}_f$)**

$$\preceq_{\mathcal{HT}} \subseteq \scaled\preceq_{\mathcal{HT}} \subseteq \sim_{\mathcal{HT}} \subseteq \simeq_{\mathcal{HT}}$$

**Proof.** We need to show that $\preceq_{\mathcal{HT}}$ is stable under $\parallel$. This is done by showing that $\{ (P \parallel R, Q \parallel R) \mid P \preceq_{\mathcal{HT}} Q \}$ is a simulation. One of the two symmetric interesting cases is again $P \parallel R \overset{\tau}{\rightarrow} P' \parallel R'$ for $P \overset{a!}{\rightarrow}_{\mathcal{C}} P'$ and $R \overset{a?}{\rightarrow}_{\mathcal{C}} R'$. Note that $R = R' \parallel a?$. We have, by assumption, that $\exists Q' . Q \overset{a!}{\rightarrow} Q'$ such that $P' \preceq Q'$. Now, using the conclusion of Lemma 3.9, either $Q \overset{\tau}{\rightarrow} Q'' \parallel R = Q'' \parallel a? \parallel R' = Q' \parallel R'$. □

The rules of $\Psi_f$ are designed so that the ‘if’ direction of the following holds, provided that the ‘only if’ direction holds.

**Lemma 3.11** Let $\chi_{\alpha}$ be defined as in Lemma 3.3. Then

$$P \overset{\alpha}{\rightarrow}_{\mathcal{HT}} P' \iff P \parallel \chi_{\alpha} \rightarrow P'$$

**Proof.** Lemma 3.4 takes care of $\tau$. ($\Rightarrow$) is implied by Lemma 3.3 and the fact that if $P \rightarrow P'$ then $P \parallel \chi_{\alpha} \rightarrow P' \parallel \chi_{\alpha}$. For ($\Leftarrow$) if $P \parallel \chi_{\alpha} \rightarrow P'$ then either the $\chi_{\alpha}$ is consumed in which case $P \overset{a}{\rightarrow}_{\mathcal{C}} P'$ or it is not in which case $P \overset{\alpha}{\rightarrow}_{\mathcal{HT}} P'$. □

The above is enough to derive completeness.

**Corollary 3.12 (Completeness of $\mathcal{HT}_f$)**

$$\preceq_{\mathcal{HT}} \subseteq \scaled\preceq_{\mathcal{HT}} \subseteq \sim_{\mathcal{HT}} \subseteq \simeq_{\mathcal{HT}}$$

**Proof.** Consequence of the conclusions of Lemma 3.11 and Theorem 3.7. □

## 4 Asynchrony

Here we will consider a language $\mathcal{L}_a$ where, an $a?$ capability can guard another process.

$$P ::= 0 \mid a! \mid a?P \mid P \parallel Q \mid \tau P$$

The semantics is captured by the smallest transition system that contains $a! \parallel a?P \rightarrow P$ as well as $\tau P \rightarrow P$ and is closed by parallel composition.
As before, we have the experiment for the \( \tau \) prefix where nothing needs to be provided in order to observe a heating:

\[
\tau P \xrightarrow{\text{(Tau)}} P \\
\frac{P \xrightarrow{\text{(Tau)\|}} P'}{P\|Q \xrightarrow{\text{(Tau\|)\|}} P'\|Q}
\]

Also, the observer can experiment with a process by providing an output,

\[
\frac{a?P \xrightarrow{\text{(In)\|}} P}{P\|Q \xrightarrow{\text{(In\|)\|}} P'\|Q}
\]

The experiment for outputs is more involved because our language now allows more powerful tests: the observer introduces an input on \( a \) followed by some other process \( R \) of the observer’s choosing (\( a?R \)) and observes a rise in temperature consistent with one interaction. We will denote this kind of experiment \( a! \downarrow R \). The inductive presentation is the following:

\[
\frac{a! \downarrow R \xrightarrow{\text{(Out)\|}} R}{P\|Q \xrightarrow{\text{(Out\|)\|}} P'\|Q}
\]

The final rule is needed to characterise those internal reductions which come from two interacting parallel components:

\[
\frac{P \xrightarrow{a?} P' \quad Q \xrightarrow{a!} Q'}{P\|Q \xrightarrow{(\text{Comm})} P'\|Q'}
\]

Let \( \Phi_a \stackrel{\text{def}}{=} \{(\text{Tau}), (\text{Tau\|}), (\text{In}), (\text{In\|}), (\text{Out}), (\text{Out\|}), (\text{Comm})\} \) and \( C_a = \Phi_a(\emptyset) \). The following is the counterpart of Lemma 3.3 for our current setting.

**Lemma 4.1** Let \( \chi_{a! \downarrow R} = a?R \), \( \chi_{a?} = a! \), \( \chi_\tau = 0 \). Then \( P \xrightarrow{a} P' \) implies \( P\|\chi_a \rightarrow P' \).

It is also easy to check that the conclusion of Lemma 3.4 holds. We easily obtain soundness; the proof of the following is essentially the same as the proof of Lemma 3.5.

**Lemma 4.2 (Soundness of \( C_a \))** \( \preceq_C \subseteq \preceq \) and \( \sim_C \subseteq \simeq \).

As in the fully synchronous case, simulation on our LTS is too strong: there exist processes which are distinguished by similarity but which are not distinguished by observational precongruence.

**Lemma 4.3** Let \( P_1 \stackrel{\text{def}}{=} a?a! \) and \( P_2 \stackrel{\text{def}}{=} \tau \). Then \( P_1 \sim P_2 \) but \( P_1 \not\sim_C P_2 \).

**Proof.** Again, it is easy to see that \( P_1 \not\sim_C P_2 \) as \( P_1 \xrightarrow{a!} a! \) which cannot be matched by \( P_2 \).

---

3 The motivation for this notation is our work on deriving structural LTSs from reduction rules [12, 13]. Roughly, the ‘\( \downarrow \)’ in the label separates the information provided by the process (here an output capability) from the data provided by the environment (here \( R \)).
On the other hand \( \mathcal{R} \) is closed under reduction and stable under parallel composition: Suppose that \( P_1 \parallel R \rightarrow Q \). Then:

(i) the reduction happened in \( R \), in which case \( Q = P_1 \parallel R' \), and \( P_2 \parallel R \rightarrow P_2 \parallel R' \);

(ii) \( R = a! \parallel R' \) and \( P_1 \parallel R \rightarrow a! \parallel R' = R \). But also \( \tau \parallel R \rightarrow R \).

\[ \square \]

To obtain completeness, we can again generate a new LTS \( \mathcal{HT}_a \) from \( C_a \) by applying rules:

\[ \frac{P \xrightarrow{a} P'}{P \xrightarrow{a} P'|a!} \text{ (InHT)} \]

\[ \frac{P \xrightarrow{a} P'}{P \xrightarrow{a?R \parallel R' \parallel P'' \parallel a?R} \text{ (OutHT)}} \]

**Corollary 4.4 (Completeness of \( \mathcal{HT}_a \))** \( \subseteq \subseteq \mathcal{HT} \) and \( \sim \subseteq \sim \mathcal{HT} \).

**Proof.** The presence of the Honda-Tokoro rules allows us to establish the counterpart to Lemma 3.11 and completeness follows from Theorem 3.7. \( \square \)

The fact that we are applying both (InHT) and (OutHT) should come as a surprise. In fact, in the fully asynchronous case, it was intuitively clear that both the labels should be unobservable. Here, while the \( a? \) transition should be unobservable, our intuition tells us that \( a! \) should be observable. The crucial observation is that while (InHT) really does make the inputs unobservable, the (OutHT) does not make outputs unobservable, it only accounts for the fact that the experiment for \( a! \) can fail. In fact, in the fully asynchronous setting we had \( R \subseteq \mathcal{HT} \tau \) and this does not hold here:

**Example 1** \( a! \not\subseteq \mathcal{HT} \tau \).

**Proof.** \( a! \xrightarrow{a\parallel \tau} \tau \). The \( \tau \) process must match this with the Honda-Tokoro transition \( \tau \xrightarrow{a\parallel \tau} a?\tau \). But clearly \( \tau \not\subseteq \mathcal{HT} a?\tau \), since the first process can do a \( \tau \) labelled transition. \( \square \)

Indeed, \( \mathcal{HT} \) remains sound.

**Lemma 4.5 (Soundness of \( \mathcal{HT}_a \))** \( \subseteq \mathcal{HT} \subseteq \subseteq \) and \( \sim \subseteq \sim \).

**Proof.** We need to show that \( \subseteq \) is stable under \( \| \). This is done by showing that \( \mathcal{R} = \{ (P \parallel R, Q \parallel R) \mid \exists Q' \mid R \xrightarrow{\tau} Q' \parallel R \} \) is a simulation.

The interesting case is \( P \parallel R \xrightarrow{\tau} P' \parallel R' \) for \( P \xrightarrow{a\parallel 0} P' \) and \( R \xrightarrow{a\parallel \tau} R' \). Suppose that \( Q \) does not have an output on \( a \) available and will be forced to match the \( a!\parallel 0 \) transition with a Honda-Tokoro transition.

We know that, for some \( S, R'' \), \( R = R'' \parallel a?S \) and \( R' = R'' \parallel S \). The key observation is that \( P \xrightarrow{a\parallel S} P' \parallel S \). Then \( Q \) will have to match this with a Honda-Tokoro transition, hence \( \exists Q' \mid Q \xrightarrow{\tau} Q' \parallel Q \xrightarrow{a\parallel S} \parallel a?S \parallel Q' \). But then \( Q \parallel R \xrightarrow{\tau} Q' \parallel R = Q' \parallel a?S \parallel R'' \). But \( P' \parallel R' = P' \parallel S \parallel R'' \), hence by \( \parallel \), we remain in \( \mathcal{R} \).

Similar reasoning goes through for the case of bisimilarity and reduction congruence. \( \square \)
In Section 6 we will show that, when reasoning about reduction congruence, the rule \((\text{OutHT})\) is not actually necessary.

5 Synchrony

We can recycle our results for the synchronous language \(\mathcal{L}_s\), where both inputs and outputs guard other processes:

\[
P ::= 0 \mid a!P \mid a?P \mid P \parallel Q \mid \tau P
\]

The reduction relation \(\rightarrow\) is the smallest relation which, for any \(P, Q\), contains \(a!P \parallel a?Q \rightarrow P \parallel Q\) as well as \(\tau P \rightarrow P\) and is closed under parallel composition. In this case, our \(\mathcal{C}_s\) LTS is generated by:

\[
\begin{align*}
\frac{a?P \xrightarrow{\tau} P}{P \parallel a!R \xrightarrow{\tau} P \parallel R} & \hspace{1cm} (\text{In}) \\
\frac{P \xrightarrow{\tau} P'}{P\parallel Q \xrightarrow{\tau} P\parallel Q'} & \hspace{1cm} (\text{Out}) \\
\frac{a!P \xrightarrow{\tau} P'}{P\parallel a?R \xrightarrow{\tau} P\parallel a?R} & \hspace{1cm} (\text{Comm}) \\
\frac{\tau P \xrightarrow{a?R} P'}{P\parallel Q \xrightarrow{\tau} P\parallel Q'} & \hspace{1cm} (\text{Tau}) \\
\frac{P \parallel Q \xrightarrow{\tau} P\parallel Q}{P\parallel Q \xrightarrow{\tau} P\parallel Q'} & \hspace{1cm} (\text{Tau})
\end{align*}
\]

The set \(\Psi_s\) of Honda-Tokoro rules is the set containing the two rules:

\[
\begin{align*}
\frac{P \xrightarrow{\tau} P'}{P \parallel a?R \xrightarrow{\tau} P\parallel a?R} & \hspace{1cm} (\text{InHT}) \\
\frac{P \parallel a!R \xrightarrow{\tau} P'}{P \parallel a!R \xrightarrow{\tau} P'} & \hspace{1cm} (\text{OutHT})
\end{align*}
\]

and we automatically obtain a sound and complete LTS (for both reduction precongruence and reduction congruence) \(\mathcal{H}T_s = \Psi_s(\mathcal{C}_s)\).

Just as in the asynchronous case, when reasoning about reduction congruence we can be more efficient. In fact, the results of the proceeding section imply that the transitions generated by the rules in \(\Psi_s\) are not actually necessary for completeness and, in fact, bisimilarity on \(\mathcal{C}_s\) is already sound and complete for reduction congruence.

6 Refining rules

We have shown that closing wrt \((\text{InHT})\) and \((\text{OutHT})\) an automatic way of obtaining an LTS on which similarity characterises reduction precongruence and indeed bisimilarity characterises reduction congruence. It is clear that this procedure is essentially the same in our three settings and is sufficient for completeness. What we have not addressed in closing the LTSs with these rules is whether it was necessary to do so. Indeed, for the synchronous language one would expect that both input and output actions are observable and that there should be no need for addition of the \(\mathcal{H}T\) rules in that setting. This is indeed the case when considering bisimilarity and reduction congruence and in this section we shall demonstrate how we can safely remove some of the \(\mathcal{H}T\) rules in certain circumstances.

To start, consider the language \(\mathcal{L}_a\) of asynchronous communication \(\text{(cf. Section 4)}\). Interestingly, although the exclusion of the \((\text{OutHT})\) rule breaks completeness
of the LTS for similarity, this is not the case for bisimilarity. To show this requires more work but the reward is a refined LTS in the sense that bisimulations in specific cases can be made smaller, and therefore, reasoning about reduction congruence is easier.

The key to showing the redundancy of (Out-HT) for the asynchronous language lies in the fact that strong output actions are preserved by reduction congruence as given by the following result, which is similar in spirit to Theorem 2 of [9]. However, apart from the proof, the notion of observational equivalence is different (ours is “dynamic” in the sense of [10]).

**Theorem 6.1** Suppose that \( a! Q \parallel R \simeq R \parallel R' \) for some \( R' \).

**Proof.** Omitted. \(\square\)

Indeed, let \( \sim_{\text{in-HT}} \) denote bisimilarity over the LTS given by extending with only rule (in-HT). The characterisation of reduction congruence does not break:

**Theorem 6.2** \( \sim_{\text{in-HT}} = \simeq \).

**Proof.** It is easy to check that soundness (\( \sim_{\text{in-HT}} \subseteq \simeq \)) still holds. For the reverse inclusion, completeness, we establish that \( \simeq \) is in fact a bisimulation relation. Suppose then that \( P \simeq Q \) and that \( P \xrightarrow{\alpha} P_0 \). The interesting case is \( \alpha = a!R \). Choose a name \( c \) which is fresh for both \( P \) and \( Q \) and let \( \chi \overset{\text{def}}{=} c! \parallel a?c? ; P \) must be able to engage in output on \( a \) so

\[
P \parallel \chi \rightarrow (P'' \parallel c! \parallel c?) \rightarrow P'
\]

for some \( P', P'' \) where \( P_0 = P' \parallel R \). We see that because \( P \simeq Q \), we must also have some \( Q', Q'' \) and

\[
Q \parallel \chi \rightarrow Q'' \rightarrow Q'
\]

such that \( (P'' \parallel c! \parallel c?) \simeq Q'' \) and \( P' \simeq Q' \). The freshness of \( c \) and Theorem 6.1 tells us that \( c \) is not in \( P'' \) and thus \( Q' \) cannot be of the form \( c! \parallel Q''' \) for any \( Q''' \). But this means that the \( c! \) in \( \chi \) must have been consumed. The only possibility for this is if \( Q'' = Q' \parallel c! \parallel c? \) and this could have only arisen if \( Q = a! \parallel Q' \). From this we can see that \( Q \xrightarrow{\alpha R} Q' \parallel R \) and, by congruence of \( \simeq \) we have \( P' \parallel R \simeq Q' \parallel R \) as required. \(\square\)

The key property used in the above proof is the preservation of strong output actions given by Theorem 6.1. This property is a useful one for characterising the observability of particular actions and has in fact been exploited in the literature in the form of barbed equivalences [9,6]. In fact, as a corollary of Theorem 6.1 we can easily check that reduction barbed congruence in the finite asynchronous calculus with output barbs coincides with our reduction congruence.

For the synchronous language \( L_s \) it is in fact possible to prove that bisimilarity over \( \Phi_s(\emptyset) \) without the addition of any Honda Tokoro rules is already complete for reduction congruence. The proof of this follows the same line as the proofs of Theorems 6.1 and 6.2 but for both input and output actions separately. The point is that it is sound to blindly add the HT rules to \( \Phi_s(\emptyset) \) in this setting and we
automatically obtain completeness. We do not need to know whether inputs and outputs are observable or not.

The $\mathcal{HT}$ rules certainly increase the size of the LTS and introduce some undesirable properties such as infinite branching. Indeed, when $\mathcal{HT}$ rules can be avoided, one often obtains a more useful LTS in the sense that bisimulations are easier to construct. We finish this section with a trivial but illustrative example to show the impact of the (removal of the) rule $(\text{outHT})$ on reasoning with bisimilarity. Consider the processes of $\mathcal{L}_a$

$$P \overset{\text{def}}{=} a! \parallel a?0 \quad \text{and} \quad Q \overset{\text{def}}{=} \tau 0$$

and suppose that we wish to demonstrate that $P \not\simeq Q$. Theorem 6.2 would immediately distinguish these processes as there is no $\sim_{\text{inHT}}$ which relates these due to the presence of an output action on $a$ in $P$ which is clearly absent from $Q$. Compare this the use of the $\sim_{\text{HT}}$ relation: consider how $Q$ could match $P$'s initial move but there can be no subsequent matching transition from $a?c!$.

The interesting point in the above example is that there is a non-trivial use of the continuation process in the experiment $a!c!$. Indeed, these continuation processes are crucial to the soundness of our systematic approach of extending LTSs with $\mathcal{HT}$ rules. Note that in the more traditional approach to LTS semantics that one finds in, say, [8], there is no room for specifying the continuation process in an experiment — effectively it is always just the nil process. For the synchronous language this turns out to be sufficient as no $\mathcal{HT}$ rules are necessary, however, what is significant that, if the continuation processes are restricted to be the nil process, it is actually unsound to add them. The previous example suffices to demonstrate this because $P$ and $Q$ would in fact become bisimilar if we restricted to $a!0$ labels and admitted rule $(\text{outHT})$. Because it is unsound to add the $\mathcal{HT}$ rules for observable actions in such a restricted setting, then in order to obtain completeness, one is forced to characterise observability in order to ascertain which of the $\mathcal{HT}$ rules need to be added.

7 Conclusions and related work

Sewell’s paper [16] about the derivation of LTS has stimulated considerable interest (e.g. [7,14,15,4]) in the relationship between labelled transition systems and underlying reduction semantics. Our simple “fully asynchronous” language (cf. Section 3) was considered already by Sewell but observability was not taken into account in his derived LTS. Bonchi [3] has also considered this example: reduction congruence agrees with the closely related concept of saturated semantics. In this paper, we did not consider the derivation process as such but it is usually the case that such derivations (cf. [12,13]) yield LTSs which are sound but not complete, here we have illustrated a method by which one “completes” them. This fits in with our general research programme that aims at developing techniques (such as derivation of LTSs) which are applicable across several languages for concurrency.
Acknowledgement.

We thank the anonymous referees for their helpful remarks, which have helped us to improve the presentation of the paper.

References


Security policies enforcement using finite edit automata

Danièle Beauquier a,1  Joëlle Cohen a,2  Ruggero Lanotte b,3

a LACL
University Paris-Est
CRETEIL, FRANCE

b Dipartimento di Scienze della cultura, Politiche e dell’Informazione
Università dell’Insubria
COMO, ITALIA

Abstract

Edit automata have been introduced by J.Ligatti et al. as a model for security enforcement mechanisms which work at run time. In a distributed interacting system, they play a role of monitor that runs in parallel with a target program and transforms its execution sequence into a sequence that obeys the security property. In this paper we characterize security properties which are enforceable by finite edit automata, i.e. edit automata with a finite set of states. We prove that these properties are a sub-class of \( \infty \)-regular sets. Moreover given an \( \infty \)-regular set \( P \), one can decide in time \( O(n^2) \) whether \( P \) is enforceable by a finite edit automaton (where \( n \) is the number of states of the finite automaton recognizing \( P \)) and we give an algorithm to synthesize the controller.

Keywords: Controller, finite edit automata, security.

1 Introduction

Security enforcement mechanisms are used to prevent violation of a policy which must guarantee protection of an extensible system and its user. Web browsers which upload and run applets programs or a database that allows users to submit their own queries have to ensure that the behavior of the system is not dangerous. This goal can be reached by means of a program monitor which enforces the security policy.

We restrict ourselves to enforcement mechanisms which work at run time in parallel with the program under control.

Schneider [Sch00, HMS06] defined the first formal model of program monitor and studied what properties are enforceable with respect to this model. Ligatti and al.
Beauquier Cohen and Lanotte [LBW05a, LBW05b, BLW02] propose a more general model based on edit automata. The monitor is not only able to interrupt a program execution in the case when it violates the security policy but it also can modify its behavior using suppression and insertion mechanisms.

Our goal is to characterize policies which can be enforced by edit automata having limited capabilities namely a finite memory. In [TTD08] a family of edit automata named Bounded History Automata is introduced and policies enforceable by these automata are characterized, but the framework is different, the input alphabet and the set of states are not necessarily finite. Our results depend crucially on the finiteness of these two parameters.

The next section is devoted to definitions. Section 3 gives general properties of edit automata, in particular the fact that any recursive security policy is enforceable by an edit automaton. In section 4 we study the power of finite edit automata. The behavior of a finite edit automaton is analyzed as well as the structure of the enforced policy. The main result is given in section 5: security policies enforceable by a finite edit automaton are exactly $\infty$-regular properties which are memory bounded. It is also proved that if the policy $P$ is given by its finite automaton with $n$ states, one can decide in $O(n^2)$ whether $P$ is enforceable by a finite edit automaton and synthesize the controller in the positive case. We conclude in the last section with an example to illustrate the obtained results.

2 Basic notions

An execution $\sigma$ is a finite sequence of actions $a_1a_2\ldots a_n$. With $|\sigma|$ we denote the length $n$ of $\sigma$. We use the notation $A^*$ (resp. $A^\omega$) to denote the set of all finite length (resp. infinite length) sequences of actions on a system with finite action set $A$. Let $A^\infty = A^* \cup A^\omega$. The symbol $\epsilon$ denotes the empty sequence. We use the notation $\sigma[i]$ to denote the $i$-th action in the sequence. The notation $\sigma[..i]$ denotes the prefix of $\sigma$ of length $i$, and $\sigma[i+1..]$ denotes the corresponding suffix. When $\tau$ is a proper prefix of $\sigma$ we write $\tau < \sigma$ and we write $\tau \leq \sigma$ to denote the fact that $\tau < \sigma$ or $\tau = \sigma$.

An ultimately periodic sequence is an infinite sequence of the form $uv^\omega$, where $u, v \in A^*$, $v \neq \epsilon$.

For $\sigma \in A^\infty$ let us denote $\text{Pref}(\sigma)$ the set of prefixes of $\sigma$, and for a set $P \subset A^\infty$, $\text{Pref}(P)$ denotes the set $\{u \in A^\infty \mid u \in \text{Pref}(\sigma)\text{ for some } \sigma \in P\}$. A security policy $P$ is a subset of $A^\infty$ such that $\epsilon \in P$.

We denote $P_{\text{fin}}$ the set $P \cap A^*$ and $P_{\text{inf}}$ the set $P \cap A^\omega$.

For $X \subset A^*$, the limit of $X$ denoted $\overline{X}$ is the set of infinite sequences which have infinitely many prefixes in $X$.

An edit automaton $A$ is a deterministic finite or countably infinite state machine $(Q, i, \delta)$ that is defined with respect to some system with action set $A$. $Q$ is the set of automaton states, $i$ is the initial state, and, $\delta: Q \times A \rightarrow Q \times (A \cup \{\epsilon\})$, is the transition function. We require that $\delta$ be Turing Machine computable. If $\delta(q, a) = (q', b)$, and $b \neq \epsilon$, the transition is an insertion step: $a$ is observed on the input but not consumed and $b$ is produced on the output. We will write the
transition \( q \xrightarrow{a} b \rightarrow q' \).

If \( \delta(q, a) = (q', \epsilon) \), the transition is a suppression step: \( a \) is read (consumed) on the input and nothing is produced on the output. We will write the transition \( q \xrightarrow{a} \epsilon \rightarrow q' \).

A configuration is a pair \((\sigma, q) \in A^\infty \times Q\).

We define a labeled relation \( \rightarrow \) on the set of configurations as follows:

\[
(\alpha, q) \xrightarrow{\beta} (\sigma, q') \text{ if } q \xrightarrow{\alpha} b \rightarrow q' \text{ is an insertion step in } A.
\]

Observe that an insertion is not possible if the input is empty.

\[
(\alpha, q) \xrightarrow{\epsilon} (\sigma, q') \text{ if } q \xrightarrow{\alpha} \epsilon \rightarrow q' \text{ is a suppression step in } A.
\]

A computation of the edit automaton is a finite or infinite sequence

\[
(\sigma_0, q_0) \xrightarrow{b_1} (\sigma_1, q_1) \xrightarrow{b_2} ... (\sigma_n, q_n) \xrightarrow{b_{n+1}} (\sigma_{n+1}, q_{n+1}) ... .
\]

In such a computation, on the input \( \sigma_0 \) starting from state \( q_0 \), the edit automaton produces the output \( b_1 b_2 ... b_n \) and the piece of the input which is read after \( n \) steps is the prefix \( \sigma' \) of \( \sigma_0 \) such that \( \sigma_0 = \sigma' \sigma_n \).

The reflexive and transitive closure of \( \rightarrow \) is denoted \( \rightarrow^* \). For \( \beta \in A^\omega \) we write \( (\alpha, q) \xrightarrow{\beta \omega} (\sigma, q) \) if from configuration \((\alpha, q)\) there is no more suppression and the output is \( \beta \). More precisely, for every \( n \) there is a computation \((\alpha, q) \xrightarrow{\beta[n]} (\sigma, q_n)\).

In \[LBW05a\], several types of enforcements are defined. As it is done in \[LBW05b\] we limit our study to effective enforcement. An effective enforcement preserves soundness and transparency. An enforcement mechanism of a policy \( P \) is sound when it ensures that outputs always obey \( P \). Soundness is the main goal of an enforcement. It is transparent if it preserves the executions that already obey \( P \). Notice that without transparency, any policy can be enforced in a trivial way just by outputing the empty sequence for example without reading the input. The formal definition is given below.

An edit automaton \( A \) effectively enforces a policy \( P \subset A^\infty \) if

\[
\text{E1 } \forall \sigma \in A^\infty, \ T_A(\sigma) \in P \text{ (soundness)}
\]

\[
\text{E2 } \forall \sigma \in P, T_A(\sigma) = \sigma. \text{ (transparency)}
\]

In \[LBW05b\] a characterization of properties enforceable by edit automata is given:

**Theorem 2.1** There exists an edit automaton \( A \) that effectively enforces a security property \( P \) iff

\[
\bullet \forall \sigma \in A^\omega \quad \text{iff} \quad \sigma \in P_{\text{fin}} \quad \text{iff} \quad \sigma \in \overline{P}_{\text{fin}} \quad \text{or} \\
\bullet \exists \sigma' < \sigma \forall \tau > \sigma' \tau \in P \implies \tau = \sigma \text{ and the existence and actions of } \sigma \text{ are}
\]

3
Beauquier Cohen and Lanotte

computable from \( \sigma' \)

- the membership problem is decidable for \( P_{1in} \).

3 Properties of edit automata

By transitivity of \( \mapsto_{\star} \) we have the following proposition on the monotonicity of \( T \).

**Proposition 3.1** Let \( A \) be an edit automaton. If \( \sigma \leq \tau \), then \( T_A(\sigma) \leq T_A(\tau) \).

The next Lemma describes the possible outputs of a finite input, and the possible inputs for a finite output.

**Lemma 3.2** Let \( A \) be an edit automaton that effectively enforces a policy \( P \).

(i) If \( T_A(\sigma) = \tau \) and \( \sigma \in A^* \) then
- either \( \tau \in A^* \) and \( (\sigma, s) \mapsto_{\tau} (\epsilon, q) \) for some state \( q \)
- or \( \tau \) is infinite and the computation from \( (\sigma, s) \) is infinite and from some step it contains only insertions.

(ii) If \( T_A(\sigma) = \tau \) and \( \tau \in A^\omega \) then
- either \( \sigma \in A^* \) and \( (\sigma, s) \mapsto_{\tau} (\epsilon, q) \) for some state \( q \)
- or \( \sigma \) is infinite and there exists a prefix \( \sigma_1 \) of \( \sigma = \sigma_1\sigma_2 \) such that
  \( (\sigma_1, s) \mapsto_{\tau} (\epsilon, q) \) and from \( (\sigma_2, q) \) the computation contains only suppressions.

(iii) if \( \sigma \in P_{fin} \) then \( (\sigma, s) \mapsto_{\sigma} (\epsilon, q) \) for some state \( q \) and the last step of this computation is a suppression step.

**Proof.** 1. Let \( \sigma \in A^* \) and \( T_A(\sigma) = \tau \). The number of suppression steps in the computation from \( (\sigma, s) \) is therefore finite and bounded by \( |\sigma| \).

Suppose that \( \tau \in A^* \). Then the number of insertion steps in the computation from \( (\sigma, i) \) is also finite. Let us consider the last insertion step in the computation with input \( \sigma \):

\( (\sigma, s) \mapsto_{\tau} (a\sigma', q_1) \mapsto_{b} (a\sigma', q_2) \) where \( (a\sigma', q_1) \mapsto_{b} (a\sigma', q_2) \) is the last insertion step.

From configuration \( (a\sigma', q_2) \) only suppression steps occur. Moreover \( A \) is complete therefore there exists a state \( q \) such that \( (a\sigma', q_2) \mapsto_{\tau} (\epsilon, q) \). Concatening the two computations we get \( (\sigma, s) \mapsto_{\tau} (\epsilon, q) \).

Suppose now that \( \tau \in A^\omega \). Here the number of insertion steps in the computation from \( (\sigma, s) \) is infinite. Since there is no possible computation from any configuration \( (\epsilon, q) \), the last suppression step is of the form \( (a\sigma', q_1) \mapsto_{\tau} (\sigma', q_2) \) for some non empty suffix \( \sigma' \) of \( \sigma \) and some states \( q_1, q_2 \). It follows that from configuration \( (\sigma', q_2) \) there are only insertion steps and infinitely many.

2. the second point is proved in a symmetric way.

3. if \( \sigma \in P_{fin} \) then \( T_A(\sigma) = \sigma \) and from point 2. of the lemma we get \( (\sigma, s) \mapsto_{\sigma} (\epsilon, q) \) for some state \( q \). The last step of this computation cannot be an insertion step because the input is empty at the end of this step. \( \Box \)

**Lemma 3.3** If \( A \) is an edit automaton effectively enforcing \( P \), then for any \( \sigma \in A^\infty \):

\( (\sigma' < \sigma \text{ and } \sigma' \in P) \implies \sigma' \leq T_A(\sigma) \).
Lemma 3.4 Let $A$ be an edit automaton effectively enforcing $P$ let $\sigma \in P$ and $\tau_1, \tau_2 \in \text{PreIm}(\sigma)$ such that $\tau_1 \neq \tau_2$. If $\tau$ is a prefix of $\tau_1$ and $\tau_2$ such that $\sigma < \tau$, then $T_A(\tau) = \sigma$.

Proof.
First of all we note that by definition of $\text{PreIm}(\sigma)$, there exist $a_1, a_2 \in A$ such that $\tau_1 a_1$ and $\tau_2 a_2$ are in $P$. We note also that, since $\tau_1 \geq \tau > \sigma$ and $\tau_1 \in \text{PreIm}(\sigma)$, we have that, for any $\gamma$ such that $\tau_1 \geq \gamma > \sigma$, it holds that $\gamma \notin P$. Hence $T_P = \sigma$.

Therefore by Lemma 3.3, $\sigma \leq T_A(\tau)$. Hence either $T_A(\tau) = \sigma$ or $\sigma < T_A(\tau)$.

We prove by contradiction that $T_A(\tau) = \sigma$.

Let us suppose that $\sigma < T_A(\tau)$. Since $\tau_1 a_1 \in P$ we have that $T_A(\tau_1 a_1) = \tau_1 a_1$.

By definition of $T_A$, we have that $T_A(\tau) \in P$, but we have noticed that, for any $\gamma$ such that $\sigma < \gamma \leq \tau_1$, it holds that $\gamma \notin P$. Hence $T_A(\tau) = \tau_1 a_1$.

Similarly we can prove that $T_A(\tau) = \tau_2 a_2$. Hence $\tau_1 a_1 = \tau_2 a_2$ implying that $\tau_1 = \tau_2$ that is a contradiction by hypothesis.

\[ \square \]

4 Finite edit automata

A finite edit automaton is an edit automaton with a finite set of states. Our goal is to characterize properties enforceable by a finite edit automaton. We briefly recall some definitions about regular sets of finite or infinite sequences. For more details see [PP04].

4.1 Regular sets of finite or infinite sequences

A deterministic finite automaton on an alphabet $A$ is a tuple $A = (Q, A, s, F, \delta)$, where $Q$ is a finite set of states, $s$ is the initial state, $F$ the set of terminal states, and $\delta : Q \times A \rightarrow Q$ is a partial transition function. We write $q \xrightarrow{a} q'$ if $\delta(q, a) = q'$. A finite sequence $u \in A^*$ is recognized (or accepted) by $A$ if $s \xrightarrow{a_1} \ldots \xrightarrow{a_n} q'$ and $q' \in F$. The set of sequences recognized by $A$ is denoted $L(A)$. The automaton $A$ is pruned if every state $q$ is reachable from $s$ and $q$ can reach at least one state of $F$.

A set $L \subset A^*$ is regular if there exists a deterministic finite automaton $A$ such that $L = L(A)$.
A deterministic Muller automaton on an alphabet $A$ is a tuple $A = (Q, A, s, F, \delta)$, where $Q$ is a finite set of states, $s$ is the initial state, $F \subseteq 2^Q$ the family of sets of infinitely repeated states, and $\delta : Q \times A \rightarrow Q$ is a partial transition function. An infinite sequence $u \in A^\omega$ is recognized by $A$ if there is an infinite run of $A$ with input $u$ whose set of infinitely repeated set of states belongs to $F$.

A set $L \subseteq A^\omega$ is $\omega$-regular if there exists a deterministic Muller automaton such that $L = L(A)$.

A set $F \subseteq Q$ is alive if there is at least one run from $s$ whose set of infinitely repeated set of states is equal to $F$.

The automaton $A$ is pruned if
- every set of $F$ is alive
- every state $q$ is reachable from $s$ and $q$ can reach at least one state of one set of $F$.

A set $P \subseteq A^\infty$ is $\infty$-regular if $P_{\text{fin}}$ is regular and $P_{\text{inf}}$ is $\omega$-regular.

Clearly a set $P \subseteq A^\infty$ is $\infty$-regular iff there exists a generalized Muller automaton $A = (Q, A, s, F, F, \delta)$ such that $(Q, A, s, F, \delta)$ recognizes $P_{\text{fin}}$ and $(Q, A, s, F, \delta)$ recognizes $P_{\text{inf}}$.

The generalized Muller automaton $A$ is pruned if
- every state $q$ is reachable from $s$ and $q$ can reach at least one state of $F$ or one set of $F$.
- every set of $F$ is alive.

In the next two subsections we study the properties of $P_{\text{fin}}$ and $P_{\text{inf}}$ for a property $P$ enforced by a finite edit automaton.

### 4.2 Properties of $P_{\text{fin}}$

**Lemma 4.1** If there exists a finite edit automaton $A$ that effectively enforces a security policy $P$ then $P_{\text{fin}}$ is regular.

**Proof.** We give only a sketch of the proof. Let $q$ be a state in $A$. We define $L_q$ as the set of finite sequences $v \in A^*$ such that there exists a finite sequence $u$ and a computation $(u, s) \overset{u}{\rightarrow} (\epsilon, q)$ (i.e. $T_A(u) = v$).

If $A$ enforces the policy $P$, from Lemma 3.2(iii) we have $P_{\text{fin}} = \bigcup_{q \in Q} L_q$.

Proving that $L_q$ is regular will imply that $P_{\text{fin}}$ is regular.

It is easy to construct a finite automaton $A_q$ that accepts $L_q$.

**Proposition 4.2** If there exists a finite edit automaton $A$ that effectively enforces a security policy $P$ then for any $\sigma \in P_{\text{fin}}$, it holds that $\text{PreIm}(\sigma)$ is a finite set.

**Proof.** By contradiction suppose that $\text{PreIm}(\sigma)$ is an infinite set for some $\sigma \in P$. Recall that all sequences of $\text{PreIm}(\sigma)$ have $\sigma$ as prefix. Since the alphabet $A$ is finite, there is an action $a$ and $c$ such that $\sigma a \in \text{PreIm}(\sigma)$ and $\sigma a a \in P$.

For each of these sequences $\alpha$, using Lemma 3.2(iii), we have:

$(\sigma, s) \overset{\sigma}{\rightarrow} (\epsilon, q)$ and $(\sigma a, s) \overset{\sigma a}{\rightarrow} (\epsilon, q_0)$ for some states $q, q_0$, and the last steps of these two computations are a suppression step.

Thus $(\sigma a, s) \overset{\sigma}{\rightarrow} (\alpha a, q) \overset{\alpha a}{\rightarrow} (\epsilon, q_0)$. 

6
Let us analyse the second part of this computation. There is a step where the input contains only the last action $a$, it means that $\sigma \alpha$ has been suppressed from the input. From Lemma 3.4, since $\sigma \alpha \not\in P$ the output is $\sigma$.

So we have:

$$(\sigma \alpha a, s) \xrightarrow{\sigma} (\alpha a, q) \xrightarrow{\epsilon} (a, q') \xrightarrow{\alpha a} (a', \epsilon) \xrightarrow{\epsilon} (\epsilon, q)$$

Because the set of states of $A$ is finite there is three states $q_1, q_2, q_3$ and infinitely many sequences $\alpha$ such that

$$(\alpha a, q) \xrightarrow{\epsilon} (a, q_1) \xrightarrow{\alpha a} (a, q_2) \xrightarrow{\epsilon} (\epsilon, q_3). \quad (\ast)$$

Therefore, for fixed $q, q_1, q_2$, we have for infinitely many $\sigma$:

$$(\alpha a, q) \xrightarrow{\epsilon} (a, q_1) \xrightarrow{\alpha a} (a, q_2).$$

Since the alphabet $A$ is finite, there is a sequence $\sigma_0$ of length greater than $n$, where $n$ is the number of states of $A$ satisfying $(\ast)$.

Thus there is a repeated state in the part $(a, q_1) \xrightarrow{\alpha a} (a, q_2)$. But in this computation there are only insertions, so if a state is repeated in this part, since the input does not change and is equal to $a$, it implies that the controller which his deterministic will make insertions for ever and will never realize the suppression of $a$ in the last part. Contradiction.

A finite automaton $A$ is simple if every cycle in $A$ contains at least one state of $F$.

**Lemma 4.3** If $P_{\text{fin}}$ is a regular set recognized by a deterministic pruned finite automaton $A$ then

the set $\text{PreIm}(\sigma)$ is finite for any $\sigma \in P_{\text{fin}}$ iff $A$ is simple.

**Proof.** The "if" part is proved by contradiction. Suppose there is a cycle in $A$ without final states. Then there is a path $s \xrightarrow{w_1} q_1 \xrightarrow{w} q \xrightarrow{u} q_2$ where

- $q_1, q_2$ are final states,
- $v$ is the label of the non-final cycle,
- there is no final state except $q_1, q_2$ on the path $q_1 \xrightarrow{wv} q_2$.

Then $w_1wv^*w_2 \subset \text{PreIm}(w_1)$ that contradicts Proposition 4.2.

Conversely, let $u \in P_{\text{fin}}$ and $s \xrightarrow{u} q$ be the path labeled by $u$ starting from $s$. There are finitely many reachable final states from final state $q$. A sequence $\tau$ in $\text{PreIm}(u)$ labels a path where $q$ is the last final state of the path, because $u$ is its longest prefix that belongs to $P_{\text{fin}}$. Moreover, this path can be extended up to a final state $q'$ such as $s \xrightarrow{u} q \xrightarrow{\tau} q'$. As there are no cycles between $q$ and $q'$, the number of such sequences is finite. \qed

**4.3 Properties of $P_{\text{inf}}$**

From Theorem 2.1, we know that if a security policy $P$ is effectively enforceable by a finite edit automaton $A$ then $P_{\text{fin}} \subset P_{\text{fin}}$. Let us study now the infinite sequences $\sigma$ of $P_{\text{inf}}$ which are not in $P_{\text{fin}}$. The next Lemma proves that the computation of such a sequence can be decomposed in three parts, the first part reads and outputs $\sigma_F$, in the second part the controller consumes a piece of the input and outputs nothing, in the last part, the input no
longer changes and the controller makes infinitely many insertions.

**Lemma 4.4** If a finite edit automaton effectively enforces a security policy $P$ then every $\sigma \in P_{inf} \setminus \overline{P_{fin}}$ can be written in a unique way $\sigma_1\alpha\beta$ such that:

- $\sigma_1$ is the longest finite prefix of $\sigma$ in $P_{fin}$
- $(\sigma_1\alpha\beta, s) \xrightarrow{\sigma_1} (\alpha\beta, q) \xrightarrow{\epsilon} (\beta, q')$ for some $q, q'$
- $(\beta, q') \xrightarrow{\alpha\beta} \omega$ and $\beta$ is ultimately periodic
- $\sigma_1\alpha \notin \text{Pref}(P_{fin})$

**Proof.** Let $P' = P_{inf} \setminus \overline{P_{fin}}$. Consider $\sigma \in P'$. Let $\sigma_1$ be the longest finite prefix of $\sigma$ such that $\sigma_1 \in P_{fin}$. The sequence $\sigma_1$ exists since $\sigma \notin \overline{P_{fin}}$ and $\epsilon \in P$. Then using Lemma 3.2 there is a computation $(\sigma_1, s) \xrightarrow{\sigma_1} (\epsilon, q)$ where the last computation step is a suppression one.

Let $\sigma = \sigma_1\sigma'$. So one has $(\sigma_1\sigma', s) \xrightarrow{\sigma_1} (\sigma', q) \quad (1)$.

Since $\sigma \in P$, for every $\beta \in A^*$ such that $\sigma_1\beta < \sigma$ we have a computation $(\sigma, s) \xrightarrow{\sigma_1\beta} (\sigma'', q'') \quad (2)$ for some $\sigma'', q''$.

Now, using the determinism of the controller, computation (1) is a prefix of computation (2) Then from $(\sigma', q)$, there must be an insertion step in order to output $\beta$.

Let then $\sigma' = \alpha\sigma''$ where $\alpha$ is the longest finite prefix of $\sigma'$ on which $A$ produces only suppressions ($\alpha$ can be the empty sequence). We have the computations $(\sigma', q) \xrightarrow{\epsilon} (\sigma'', q') \xrightarrow{b} (\sigma'', q_1)$ where $(\sigma'', q') \xrightarrow{b} (\sigma'', q_1)$ is the first insertion step after $(\sigma', q)$.

Suppose now that there is a suppression step after this insertion step and again consider the first one: we have $\sigma' = \alpha\sigma''$ and states $q_1, q_2$ such that there is a computation $(\sigma, s) \xrightarrow{\sigma_1\alpha} (\sigma', q) \xrightarrow{\epsilon} (\sigma'', q') \xrightarrow{\alpha'} (\sigma'', q_1)$ and the next step is a suppression one. It follows that $T_A(\sigma_1\alpha) = \sigma_1\alpha'$. Hence we have $\sigma_1\alpha' \in P_{fin}$ and $\sigma_1 < \sigma_1\alpha' \leq \sigma$ whereas $\sigma_1$ is the longest finite prefix of $\sigma$ that belongs to $P$: contradiction.

Therefore in the computation from $(\sigma', q)$, once there is an insertion step, there are always insertion steps. Let $q$ be the state from which only insertions steps occur in computation from $(\sigma', q)$. Let then $\sigma = \sigma_1\alpha\beta$ with computation $(\sigma, s) \xrightarrow{\sigma_1\alpha} (\alpha\beta, q) \xrightarrow{\epsilon} (\beta, q')$. Moreover since there is no more suppression step, the computation of length $n$ from $(\beta, q')$ produces as output the prefix of length $n$ of $\alpha\beta$: $(\beta, q') \xrightarrow{[\alpha\beta][n]} (\beta, q_n)$ for some state $q_n$. But $A$ has a finite set of states, and the input does not change from $(\beta, q')$ thus there exists integers $n_1 < n_2$ such that $q_{n_1} = q_{n_2}$ and from $q_{n_1}$ the output is periodic, so from $(\beta, q')$ the output is ultimately periodic.

We have proved that any infinite sequence in $P'$ is of the form $\sigma_1\alpha\beta$ where $\sigma_1 \in P_{fin}$, $\alpha$ as input corresponds to a sequence of suppression steps and $\beta$ is an ultimately periodic sequence.

Remark that for such a sequence $\sigma$ in $P'$, the set $\text{Pref}(\sigma) \cap \text{Pref}(P_{fin})$ is finite. Indeed if it was not the case there would exist a prefix $\gamma$ of $\sigma$ and $\gamma' \in A^*$ such that $\gamma = \sigma_1\alpha\gamma'$ with $\beta' < \beta$ and $\gamma\gamma' \in P_{fin}$. Then we should have $T(\gamma\gamma') = \gamma\gamma'$. But
on the other hand \( T(\gamma \gamma') = T(\sigma_1 \alpha \beta' \gamma') = \sigma \). A contradiction. We deduce that there is a longest prefix of \( \sigma \) in \( \text{Pref}(P_{\text{fin}}) \) and this prefix is of the form \( \sigma_1 \alpha' \) where \( \alpha' \leq \alpha \).

We are now in position to give an \( \omega \)-regular expression of \( P_{\text{inf}} \setminus \overset{\longrightarrow}{P_{\text{fin}}} \)

**Proposition 4.5** If a finite edit automaton effectively enforces a security policy \( P \) then \( P_{\text{inf}} \setminus \overset{\longrightarrow}{P_{\text{fin}}} \) is of the form \( \cup_{j \in J} R_j \beta_j \) where

- \( J \) is finite
- for every \( j \in J \), \( R_j \) is regular
- for every \( j \in J \), \( \beta_j \) is an ultimately periodic sequence in \( A^\omega \)
- for every \( j \in J \), \( R_j \cap \text{Pref}(P_{\text{fin}}) = \emptyset \)
- \( R_i \cap \text{Pref}(R_i) = \emptyset \) for \( i \neq j \)
- for every \( u < v \) with \( u \not\in \text{Pref}(P_{\text{fin}}) \) and \( v \in R_j \) we have \( |v| - |u| \leq K \) where \( K \) is the number of states of \( A \).

As a consequence \( P_{\text{inf}} \) is \( \omega \)-regular.

**Proof.** Let \( P' = P_{\text{inf}} \setminus \overset{\longrightarrow}{P_{\text{fin}}} \), then \( \sigma \) has the form \( \sigma_1 \alpha_1 \beta_1 \sigma_2 \) satisfying properties of Lemma 4.4. Remark that the set \( \text{Pref}(\sigma) \cap \text{Pref}(P_{\text{fin}}) \) is finite. Indeed if it is not finite then there exists a prefix \( \gamma \) of \( \sigma \) and \( \gamma' \in A^* \) such that \( \gamma = \sigma_1 \alpha \beta' \) with \( \beta' < \beta_1 \) and \( \gamma' \in P_{\text{fin}} \). Then we should have the following : \( T_A(\gamma \gamma') = \gamma' \). But we have using Lemma 4.4, \( T_A(\gamma \gamma') = T_A(\sigma_1 \alpha_1 \beta' \gamma') = \sigma \). Contradiction.

We deduce that there is a longest prefix of \( \sigma \) in \( \text{Pref}(P_{\text{fin}}) \) and this prefix is of the form \( \sigma_1 \alpha' \) where \( \alpha' \leq \alpha_1 \).

Now we focus on the set \( E \) of finite sequences \( \alpha \sigma \) for all \( \sigma \) in \( P' \) and we prove that \( E \) is finite.

Recall that each \( \sigma \in P' \) is written in a unique way \( \sigma_1 \alpha_1 \beta_\sigma \) such that the computation starting in \( (\sigma_1 \alpha_1 \beta_\sigma, s) \) is as follows:

\[
(\sigma_1 \alpha_1 \beta_\sigma, s) \xrightarrow{\sigma_1} (\alpha_1 \beta_\sigma, q_\sigma) \xrightarrow{\epsilon} (\beta_\sigma, q'_\sigma) \xrightarrow{\alpha_\sigma \beta_\sigma} \omega.
\]

If \( E \) is infinite, there exists an infinite sequence \( \alpha \) such that every prefix \( \alpha[..p] \) of \( \alpha \) is also a prefix of some \( \alpha_\sigma_\rho \) in \( E \) for some \( \sigma_\rho \in P' \). Since the set of states of \( A \) and the alphabet \( A \) are finite, there exist an action \( a \) and states \( q \) and \( q' \) such that for all \( \sigma_\rho \) we have \( q_\sigma = q, q'_\rho = q' \) and \( a \) is the first letter of \( \beta_\sigma \).

Thus for all \( \sigma_\rho \) we have:

\[
(\alpha \sigma \beta_\sigma, q) \xrightarrow{\epsilon} (\beta_\sigma, q') \xrightarrow{\alpha_\sigma \beta_\sigma} \omega.
\]

But the computation \( (\beta_\sigma, q') \xrightarrow{\alpha_\sigma \beta_\sigma} \omega \) depends only on \( q' \) and the first letter of \( \beta_\sigma \). Thus all the outputs \( \alpha_\sigma \beta_\sigma \) are equal and so necessarily equal to \( \alpha \). The sequence \( \alpha \) is ultimately periodic \( \alpha = \alpha_1 \alpha_2^\omega \). There exists \( p \) large enough such that \( \alpha_\rho = \alpha_1 \alpha_2^K \alpha_\rho' \), where \( K \) is the number of states of \( A \). Then the computation \( (\alpha_\sigma \beta_\sigma, q) \xrightarrow{\epsilon} (\beta_\sigma, q') \) has some repeated state in the following way:

\[
(\alpha_1 \alpha_2^K \alpha_\rho', q) \xrightarrow{\epsilon} (\alpha_2^K \alpha_\rho' \beta_\sigma, q_1) \xrightarrow{\epsilon} (\alpha_2^K \alpha_\rho' \beta_\sigma, q_1) \xrightarrow{\epsilon} (\beta_\sigma, q').
\]

On the other hand \( \alpha_1 \beta_\rho = \alpha_1 \alpha_2^\omega \). So from configuration \( (\alpha_1 \beta_\rho, q_1) \) the computation makes only suppressions for ever. It contradicts the fact that \( (\beta_\sigma, q') \xrightarrow{\alpha_\sigma \beta_\sigma} \omega \).
Since $\sigma \in P'$, we deduce that $\sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3 \in P'$ for any positive integer $k$.

Then we must have for any positive integer $k$:

- $T_A(\sigma_1 \alpha \alpha_3) = \sigma$
- $T_A(\sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3) = \sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3$

Besides that we have $T_A(\sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3) = \sigma$.

Hence for any positive integer $k$ we have $\sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3 = \sigma_1 \alpha_1 \alpha^k \alpha_2 \alpha_3$.

We deduce that $\sigma = \sigma_1 \alpha_1 \alpha^\omega$. In that case $T_A(\sigma) = \sigma_1$. Contradiction.

We have proved that $E$ is finite and the number of sequences in $E$ is bounded by $K^2 |\mathcal{A}|$.

Now we make precise the set $P'$. From Lemma 4 $P_{1 \infty}$ is regular. Let $L'_q = \{ u | (u, s) \xrightarrow{v} (\epsilon, q) \text{ for some } v \in \mathcal{A}^* \}$. From $A$ it is easy to construct a finite automaton which recognizes $L'_q$. Consider $L'_q = L'_q \cap P_{1 \infty}$ Clearly, for each $u \in L'_q$ we have $(u, s) \xrightarrow{u} (\epsilon, q)$. And $L'_q$ is a regular set.

Let $a \in A$. We define the set of states $Q_a = \{ q \in Q \exists q' \in Q \delta(q, a) = (q', b) \}$.

For $q \in Q_a$ we can notice that there is exactly one infinite computation from $(a^\gamma, q)$ for any finite or infinite sequence $\gamma$; this computation performs only insertion steps and the output for this computation is ultimately periodic as proved in Lemma 7. Let us denote $\beta_{q, a}$ this ultimately periodic sequence.

Let $q$ be a state of $A$ and $q'$ be a state of $Q_a$ for a non-empty $Q_a$. Let $F_{q, q', a}$ be the set of sequences $\alpha$ in $A^*$ whose length is less than $K$ such that there is a computation $(\alpha, q) \xrightarrow{\epsilon^*_s} (\epsilon, q')$ and such that $\alpha a < \beta_{q, a}$.

The set $F_{q, q', a}$ has at most one sequence. Indeed, if $\alpha$ and $\alpha'$ are two different sequences in $F_{q, q', a}$, since $\alpha$ and $\alpha'$ are prefixes of $\beta_{q, a}$ one has $\alpha < \alpha'$ (or the converse). It follows that $\alpha' = \alpha a u$ for some $u$.

Besides we have also the computations $(\alpha, q) \xrightarrow{\epsilon^*_s} (\epsilon, q')$ and $(\alpha', q') \xrightarrow{\epsilon^*_s} (\epsilon, q')$. From the first one we deduce the computation $(\alpha a u, q) \xrightarrow{\epsilon^*_s} (au, q')$. From the second one we deduce the computation $(\alpha a u, q') \xrightarrow{\epsilon^*_s} (\epsilon, q')$. But from $(au, q')$ there are only insertions steps that is in contradiction with this last computation. Let us denote $\alpha_{q, q', a}$ the unique sequence in $F_{q, q', a}$ when $F_{q, q', a}$ is not empty. And we can set :

$$\beta_{q, a} = \alpha_{q, q', a} \alpha_{q, q', a}$$

for $q, q'$ such that $Q_a \neq \emptyset$ and $F_{q, q', a} \neq \emptyset$. Moreover one can prove that $|\alpha_{q, q', a}| < K^2$, otherwise in the last part of the computation when there are only insertion steps, the repetitive part would begin inside the production of $\alpha_{q, q', a}$ and $\alpha_{q, q', a}$ would contain at least $K$ times the period. But in that case the computation starting from $(\alpha_{q, q', a} \alpha_{q, q', a}^\omega, q)$ would be made of suppressions for ever. A contradiction.

We have then

- $|\alpha_{q, q', a}| < K^2$
- $\beta_{q, a}$ is ultimately periodic.

We define the sets $R_{q, q', a} = L_q \alpha_{q, q', a} a$.

We have proved that $P' \subset \bigcup_{q, a} R_{q, q', a} \beta_{q, a}$. By construction we have clearly also $\bigcup_{q, a} R_{q, q', a} \beta_{q, a} \subset P_{1 \infty}$. Moreover an infinite sequence in some $R_{q, q', a}$ cannot belong to $P_{1 \infty}$ since $R_{q, q', a} \cap \text{Pref}(P_{1 \infty}) = \emptyset$. 

10
Hence \( P' = \cup_{q,q',a} R_{q,q',a} \beta_{q',a} \).

We have to prove that sets \( R_{q,q',a} \) satisfy

- \( R_{q,q',a} \cap \text{Pref}(P_{1in}) = \emptyset \),
- they are mutually disjoints,
- they are regular
- \( R_{q,q',a} \cap \text{Pref}(R_{q_1,q_1',b}) = \emptyset \) for any \( (q, q', a) \neq (q_1, q_1', b) \).

- The first property follows from Lemma 7.
- Let \( u \in R_{q,q',a} \cap R_{q_1,q_1',b} \). Then we have \( u = v \alpha_a a = w \omega_b b \) with \( v, w \in P_{1in} \) such that there are computations
  \[
  (v, s) \xrightarrow{\epsilon, q} (w, \epsilon, q_1) \xrightarrow{\epsilon, q} (a, q') \xrightarrow{\epsilon, q_1} (b, q_1').
  \]

Firstly \( a = b \) clearly holds.

Suppose now \( v < w \). Then we have \( w = v \alpha' a \) where \( \alpha' \leq \alpha_a \) and there is a computation \( (w, q_0) \xrightarrow{\epsilon, q} (\alpha', q) \xrightarrow{\epsilon, q_2} (\epsilon, q_2) \) for some \( q_2 \). Hence \( v = w \) and moreover \( \alpha_a = \alpha_b \).

- \( R_{q,q',a} \) has been proved to be regular.
- Let \( u \in R_{q,q',a} \cap \text{Pref}(R_{q_1,q_1',b}) \). Then on one hand we have \( u = v \alpha_a a = w \omega_b b \) with \( v \in P_{1in} \) with a computation \( (u, q_0) \xrightarrow{\epsilon, q} (a, q') \) with a suppression last step. On the other hand we have a sequence \( u' \) such that \( uu' = v \alpha_a a u' \in R_{q_1,q_1',b} \). Thus there must be the computation \( (uu', q_0) \xrightarrow{w} (b, q_1') \). Hence we have \( (uu', q_0) \xrightarrow{w} (b, q_1') \). But from \( (uu', q_0) \) there are only insertion steps. It follows that \( u' = \epsilon \) and \( u \in R_{q_1,q_1',b} \). As aforementioned we deduce \( (q, q', a) = (q_1, q_1', b) \). □

**Definition 4.6** A policy \( P \) is memory bounded if \( P \) is of the form \( P = P_{1in} \cup P_{1in} \cup_{j \in J} R_j \beta_j \) where

- \( \epsilon \in P \)
- \( J \) is finite
- \( P_{1in} \subset A^* \) is recognized by a simple finite automaton
- for every \( j \in J, R_j \) is regular
- for every \( j \in J, \beta_j \) is an ultimately periodic sequence in \( A^\omega \)
- for every \( j \in J, R_j \cap \text{Pref}(P_{1in}) = \emptyset \)
- \( R_j \cap \text{Pref}(R_i) = \emptyset \) for \( i \neq j \)
- there exists a constant \( K \) such that for every \( u < v \) with \( u \notin \text{Pref}(P_{1in}) \) and \( v \in R_j \mid v - |u| \leq K \).

From Proposition 4.5 we obtain:

**Theorem 4.7** If a security policy \( P \) is enforced by a finite edit automaton, then \( P \) is memory bounded.

We intend now to characterize the generalized Muller automata which recognize properties that are memory bounded.
Definition 4.8 A pruned generalized Muller automaton $A$ is simple if:

S0. $s \in F$

S1. each cycle encounters $F$ or a set of $\mathcal{F}$

S2. the restriction of the automaton $A$ to each set $F_i$ in $\mathcal{F}$ which has no state in $F$ is an elementary cycle $C_i$

S3. there is no edge from a state in $C_i$ to a state not in $C_i$ for every $i$

S4. each alive set $G$ such that $G \cap F \neq \emptyset$ belongs to $\mathcal{F}$.

Proposition 4.9 Given a pruned generalized Muller automaton $A$, $L(A)$ is memory bounded iff $A$ is simple.

Proof. Let us suppose that $A$ is simple. Let $P$ be the property recognized by $A$.

Because of S0, $\epsilon \in P$. Because of condition S4 we have $P_{\text{fin}} \subset P_{\text{inf}}$. Let $P' = P \setminus (P_{\text{fin}} \cup P_{\text{inf}})$. Let $\mathcal{F}'$ be the family of sets $F_i$ in $\mathcal{F}$ disjoint from $F$. Let us remark that from properties S2 and S3 one can deduce that for distinct $F_i$ and $F_j$ in $\mathcal{F}'$, cycles $C_i$ and $C_j$ are disjoint. Let us call this property, property S5. For each $G$ in $\mathcal{F}'$, and each $q \in G$ let $R_q$ be the set of finite sequences recognized by $A$ in a computation which starts in the initial state and stops in state $q$ without running through $G$ before the last state $q$. Let $\beta_q$ be the periodic infinite sequence recognized by $A$ in a computation which starts in $q$. Because of properties S2 and S3, $\beta_q$ is unique.

Clearly $P' = \bigcup_{G \in \mathcal{F}', q \in G} R_q \beta_q$ where $\mathcal{F}'$ is the set of $F_i$ in $\mathcal{F}$ disjoint of $F$.

By construction, $R_q \cap P_{\text{fin}} = \emptyset$, and from property S3, $R_q \cap \text{Pref}(P_{\text{fin}}) = \emptyset$. Because of properties S3 and S5, $R_q \cap \text{Pref}(R_{q'}) = \emptyset$. At last, for two different $q$ and $q'$, there is no common sequence to $R_q \beta_q$ and $R_{q'} \beta_{q'}$ because either $q$ and $q'$ are not in the same $G$ of $\mathcal{F}'$ and the computation of $\sigma$ cannot have two distinct infinitely repeated sets, or $q$ and $q'$ are in the same $G$ of $\mathcal{F}'$ and the first state of $G$ reached in the computation of $\sigma$ is unique.

So if $A$ satisfies properties S0 – S4, then $P'$ has the required form.

Conversely, if property S0 is not satisfied then $\epsilon \notin P$. If S4 is not satisfied, then $P_{\text{fin}} \notin P_{\text{inf}}$. If property S2 is not satisfied then $P'$ contains infinite sequences which are not ultimately periodic. Thus $P'$ cannot have the required form.

Let us suppose that $A$ satisfies properties S0, S2, S4 but not property S3. There exists a cycle $C_i$ in $\mathcal{F}'$ which has an outgoing edge. In that case, since the automaton is pruned, this edge can be extended either in a path which reaches $F$, or in a path which reaches another cycle $C_j$. Because of Proposition 2 and Lemma 5 $C_i$ cannot reach $F$. So it reaches another cycle $C_j$.

For any constant $K > 0$ one can build an infinite path with a piece $p$ larger than $K$ and larger than the number of states of $A$ inside $C_i$ and reaching after that $C_j$ for ever. Let $\sigma$ the infinite sequence labeling this infinite path. Invoking the determinism of the automaton, the piece $p$ cannot contain a period of the ultimately periodic part of $\sigma$. So if $P'$ has the required form, $\sigma$ belongs to some $R_i \beta_i$, but the piece $p$ corresponds to prefixes of $\sigma$ which are not in $\text{Pref}(P_{\text{fin}})$ and the length of $p$ is larger than $K$. A contradiction. We have proved that if $A$ satisfies property S2 but not property S3 then $P'$ cannot have the required form.
Suppose now that $A$ satisfies properties $S_0$, $S_2$, $S_3$ and $S_4$ but not property $S_1$. It means that there exists a cycle $C$ from which one can reach either $F$, or some $C_i$. Then along the same lines as in the previous case $P'$ cannot have the required form.

**Proposition 4.10** Given a pruned generalized Muller automaton $A$ with $n$ states, one can decide in time $O(n^2)$ whether $L(A)$ is memory bounded.

**Proof.** Here is the algorithm:

- Check that $s \in F$
- Compute the set $C$ of terminal strongly connected components of $A$
- Compute the set $C'$ of terminal strongly connected components which do not intersect $F$
- Check whether there is only one infinite path from one state of each component in $C'$
- Compute the set of states $D$ not in $F$ that can reach $F$
- Check that the set of paths that reach $F$ from each state in $D$ is finite.

4.4 Characterization of policies effectively enforced by a finite edit automaton

In this subsection we give the reverse part of Theorem 4.7.

**Theorem 4.11** Given a pruned generalized Muller automaton $A$ recognizing a security policy $P$ which is memory bounded, one can build an edit automaton which effectively enforces $P$.

**Proof.** From Proposition 4.9 the automaton $A$ is simple. We now build the edit automaton which enforces $P$. Let us first describe informally the behavior of this controller. The states of the automaton $A$ can be divided into three parts: $F$, $I$ the set of internal states not in $F$ but that can reach $F$, and the set $O$ for the other ones. Let an input $u_1u_2...u_k$ in $P_{\text{fin}}$ which has $k+1$ prefixes in $P_{\text{fin}}$, namely $\epsilon, u_1, u_1u_2, ...u_1u_2...u_k$. The controller reads $u_1$ except its last letter and memorizes it, then observes the last letter of $u_1$ and writes $u_1$, and finally reads the last letter of $u_1$, the controller processes in the same way for $u_2, ...u_k$. The reason why the controller does not read immediately the last letter of $u_1$ is that after this reading, the output must be $u_1$, so the controller must write entirely $u_1$ before the end of the reading of $u_1$. Sequences in $P_{\text{fin}} \cup P_{\text{fin}}$ are processed in this way. For infinite sequences in $P \setminus (P_{\text{fin}} \cup P_{\text{fin}})$, as long as the prefix is in $\text{Pref}(P_{\text{fin}})$ the treatment is as before, but when the input is no longer in $\text{Pref}(P_{\text{fin}})$, which happens when the automaton $A$ enters a state in $O$ then the controller reads the input and memorizes it until a final cycle is reached. When a final cycle is reached the controller stops the reading and writes the memorized factor followed by the periodic final part. For any sequence $\sigma$ not in $P$, the behavior of the controller at the beginning is the same, but as soon as the automaton $A$ cannot read a letter the controller stops the writing and reads the input up to the end. So the output is the longest prefix of $\sigma$ that is in $P_{\text{fin}}$. We give now the detailed transitions of the controller. Let
$A = (Q, A, s, F, \delta)$. The set of states $Q$ can be divided into three parts: $F$ the set of final states, $I$ the set of states not in $F$ but that can reach $F$, and $O$ the set of other states. We divide $O$ into $O_i$ the set of states in $O$ which do not belong to a set in $F$ and $O_f$ the complement. Due to the fact that $A$ is simple, each state $q$ of $O_f$ defines a unique sequence $a_q u_q$ which is the label of the elementary cycle starting in $q$.

The set of states of the edit automaton is: $Q = Q \cup \hat{Q} \times A \cup (Q \cup \bar{Q}) \times A^{\leq n} \cup \{t\}$. Transitions are:

- $q \xrightarrow{a} q'$ if $q \in F$ and $\delta(q, a) = q' \notin F$
- $q \xrightarrow{a} \hat{q}'$ if $q \in F$ and $\delta(q, a) = q' \in F$
- $\hat{q} \xrightarrow{a} q$ if $q \in F$
- $q_u \xrightarrow{a} q'_u$ if $\delta(q, a) = q' \in F \cup O_f$

The controller memorizes the factor it will write later if the input is admissible.

- $q_{bu} \xrightarrow{a} \hat{q}'_u$ if $\delta(q, a) = q' \in F \cup O_f$
- $\bar{q}_{bu} \xrightarrow{a} \hat{q}_u$ if $q \in F \cup O_f$
- $\hat{q} \xrightarrow{a} \hat{q}$ if $q \in F \cup O_f$

The controller writes the factor it has memorized.

- $\hat{q} \xrightarrow{a} q$ if $q \in F$ The controller ends the reading of a sequence in $P_{\text{fin}}$.
- $\hat{q} \xrightarrow{a} \hat{q}'$ if $q \in O_f$ and $\delta(q, b) = q' \in O_f$

The controller writes an infinite periodic sequence and no longer reads any letter.

- $q_a \xrightarrow{a} t$ and $q \xrightarrow{a} t$ if there is no transition $\delta(q, a)$
- $t \xrightarrow{a} t$ for every $a \in A$.

The input is not in $P$, its longest prefix in $P$ has been written, the end of the input is read without writing anything.

From Theorems 4.7 and 4.11 we get:

**Theorem 4.12** A security policy is effectively enforceable by a finite edit automaton iff it is memory bounded.

and from Proposition 4.10:

**Theorem 4.13** Given a pruned generalized Muller automaton $A$ with $n$ states, one can decide in $O(n^2)$ whether $L(A)$ is effectively enforceable by a finite edit automaton.

## 5 An example

In the following example the set of actions is $A = \{0, 1, a, \sharp\}$ where

- 0 is an action for opening a session
- 1 is an action for closing a session
• $a$ is an action that is allowed to be done only outside a session

• ♯ is an interruption that can be used to end processing while a user intends to have a forbidden behavior.

The policy we consider here is:

$$P = \text{Pref}(X) \cup X0^\omega$$

where $X = ((01)^* \cup a)^*$.

The “normal” behavior is represented by $\text{Pref}(X) \cup X$. When an attempt to execute action $a$ inside a session, namely just after an opening action 0 the process is interrupted by an infinite sequence of ♯ actions that corresponds to $X0^\omega$.

An edit automaton that enforces $P$ is shown below. It interrupts any attempt of running action $a$ when a session is opened but not closed. Any irrelevant opening or closing action is suppressed as well as irrelevant interruptions. An insertion step $q \xrightarrow{a|a} q'$ followed by a suppression step $q' \xrightarrow{a|\epsilon} q''$ is compressed in one single transition $q \xrightarrow{a|a} q''$.

Discussion

In this paper we have characterized the policies that can be enforced by finite edit automata. These policies are a subclass of $\infty$-regular policies. Moreover, we provide an algorithm which constructs the program monitor from an automaton that recognizes the policy.

Finite transducers [Ber79] are a classical notion very close to edit automata. A finite transducer on an alphabet $A$ is defined by its finite set of states $Q$, an initial state $s$ and a set of transitions $\delta \subset Q \times A \times \{\epsilon\} \times Q \cup Q \times \{\epsilon\} \times A \times Q$. Transitions $(q, a, \epsilon, q')$ (read transitions) represent a suppression of $a$ on the input and nothing is written on the output, transitions $(q, \epsilon, b, q')$ (write transitions) correspond to an input unchanged and a $b$ is written on the output. Determinism implies there are no two different read transitions from the same state and the same read action, no two different write transitions from the same state and if there is a read action form a state there is no write action from the same state. A finite transducer is complete if from every state where a write transition is impossible there is a read transition
for every letter in input. Thus the difference between deterministic complete finite
transducers (dcft) and edit automata is very minimal, the write transitions in dcft
do not depend on a future input (the input can empty) but only on the current
state contrary to edit automata. One can prove and it is not surprising that finite
edit automata and dcft enforce the same class of policies.

Our future work will be done in several directions. A natural question is whether
one can decide if a given edit automaton enforces the set of sequences of its output.
We will solve this question positively at least in the case when the edit automaton is
finite. Secondly, we plan to explore the power of pushdown edit automata. At last,
we intend to distinguish actions of different types. Actually in practice there are
some limitations about the power of the controller. Some actions are unsuppressible
by the controller or uninsertable. So it may be of interest to consider a specification
that takes into account this feature.

References

[HMS06] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement
[LBW05b] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program
[PP04] D. Perrin and J.E. Pin. *Infinite Words Automata, Semigroups, Logic and Games*, volume 141
2000.
[TTD08] C. Talhia, N. Tawbia, and M. Debbabib. Execution monitoring enforcement under memory-
Composition of Web services: algorithms and complexity

Philippe Balbiani
Institut de recherche en informatique de Toulouse
Université Paul Sabatier
Toulouse, France

Fahima Cheikh
Institut de recherche en informatique de Toulouse
Université Paul Sabatier
Toulouse, France

Guillaume Feuillade
Institut de recherche en informatique de Toulouse
Université Paul Sabatier
Toulouse, France

Abstract
Composition of services is necessary for realizing complex tasks on the Web. It has been characterized either as a plan synthesis problem or as a software synthesis problem: given a goal and a set of Web services, generate a composition of the Web services that satisfies the goal. We propose algorithms for performing automated Web service composition. We also examine the composition of services from the perspective of computational complexity.

Keywords: Service composition, controller synthesis, computational complexity.

1 Introduction
The development of service oriented architectures for implementing distributed software systems demands that organizations make their abilities accessible via the Internet through Web service interfaces. In most cases, Web services are nothing more than elementary components in a client-server architecture. Their importance lies in the fact that we can compose them to create complex business processes.

1 Email: Philippe.Balbiani@irit.fr
2 Email: Fahima.Cheikh@irit.fr
3 Email: Guillaume.Feullade@irit.fr

©2008 Published by Elsevier Science B. V.
Composition of Web services involves multifarious difficulties and requires to formally define the semantics of the input services. If one tries to compose complex business processes from given input services then plan synthesis algorithms from artificial intelligence or software synthesis algorithms from computer science can be employed. There exists many approaches to the composition problem [11]. Characterizing Web service composition as a plan synthesis problem forces us to devise algorithms tackling incomplete information and uncertain effects. Different automated techniques have been proposed to solve the composition/plan problem [14,15,16]. Nevertheless, their computational complexity has not been investigated in details. Characterizing Web service composition as a software synthesis problem compels us to devise algorithms working with behavioural descriptions given in terms of automata. Different automated techniques have been proposed to solve the composition/software problem [3,4,6]. Nevertheless, their completeness rests on syntactical restrictions that prevent them from being fully applicable.

Although services might be considered as non autonomous agents which know only about themselves, service oriented architectures and multi-agent systems share many characteristics [8]. To illustrate the truth of this, one has only to mention the fact that several researchers have recently advocate the use of Web service technology to build multi-agent systems accessible through the Web [10] or the use of multi-agent-based coalition formation approaches for Web service composition [12]. In this paper, we propose algorithms for performing automated Web service composition. We also examine the composition of services from the perspective of computational complexity.

The section-by-section breakdown of the paper is as follows. Section 2 recalls the notion of finite automata and establishes the concept of Web service. In section 3, basic definitions are given and preliminary results are proved. These definitions and these results will be used in great depth in the remaining sections. Section 4 introduces the composition problem: given a goal and a set of Web services, generate a composition of the Web services that satisfies the goal. In section 5, we examine the composition of services from the perspective of computational complexity. Two ways of solving the composition problem are presented in section 6. In section 7, we talk about some open problems.

2 Web services as finite automata

In this section, the notion of finite automata is recalled and the concept of Web service is established.

2.1 Finite automata

Let $\Sigma$ be a finite set of actions. A finite automaton over $\Sigma$ is a structure $A = (S, \Delta, s^{in})$ where $S$ is a finite set of states, $\Delta$ is a function

$\Delta: S \times \Sigma \rightarrow 2^S,$

$s^{in} \in S$ is an initial state. For all $\Sigma' \subseteq \Sigma$, the relation $\rightarrow_{A}^{\Sigma'} \subseteq S \times S$ describes how the finite automaton can move from one state to another in 1 step under some
action in $\Sigma'$. It is defined formally as follows: $s \xrightarrow{\Sigma'}_A t$ iff there exists $a \in \Sigma'$ such that $t \in \Delta(s, a)$. Furthermore, let $\rightarrow^{\Sigma'}_A$ be the reflexive transitive closure of $\rightarrow^{\Sigma'}_A$.

For all $\Sigma' \subseteq \Sigma$, we shall say that $A$ loops over $\Sigma'$ iff for all $a \in \Sigma'$, $\rightarrow^{\{a\}}_A = Id_S$.

2.2 Products

Let $A_1 = (S_1, \Delta_1, s_1^{in})$ and $A_2 = (S_2, \Delta_2, s_2^{in})$ be finite automata over $\Sigma$. By $A_1 \otimes A_2$, we denote the asynchronous product of $A_1$ and $A_2$, i.e. the finite automaton $A = (S, \Delta, s^{in})$ over $\Sigma$ such that $S = S_1 \times S_2$, $\Delta$ is the function defined by

- $(t_1, t_2) \in \Delta((s_1, s_2), a)$ iff either $t_1 \in \Delta_1(s_1, a)$ and $t_2 = s_2$ or $t_1 = s_1$ and $t_2 \in \Delta_2(s_2, a)$,

$s^{in} = (s_1^{in}, s_2^{in})$. By $A_1 \times A_2$, we denote the synchronous product of $A_1$ and $A_2$, i.e. the finite automaton $A = (S, \Delta, s^{in})$ over $\Sigma$ such that $S = S_1 \times S_2$, $\Delta$ is the function defined by

- $(t_1, t_2) \in \Delta((s_1, s_2), a)$ iff $t_1 \in \Delta_1(s_1, a)$ and $t_2 \in \Delta_2(s_2, a)$,

$s^{in} = (s_1^{in}, s_2^{in})$.

2.3 Bisimulations

Let $A_1 = (S_1, \Delta_1, s_1^{in})$ and $A_2 = (S_2, \Delta_2, s_2^{in})$ be finite automata over $\Sigma$. For all $\Sigma' \subseteq \Sigma$, a relation $Z \subseteq S_1 \times S_2$ such that $(s_1^{in}, s_2^{in}) \in Z$ is called a bisimulation between $A_1$ and $A_2$ modulo $\Sigma'$, notation $Z : A_1 \leftrightarrow A_2 (\Sigma')$, iff the following conditions are satisfied for all $(s_1, s_2) \in Z$ and for all $a \in \Sigma \setminus \Sigma'$:

- for all $t_1 \in S_1$, if $s_1 \xrightarrow{\Sigma'}_A \circ \xrightarrow{\{a\}}_A \circ \xrightarrow{\Sigma'}_A t_1$ then there exists $t_2 \in S_2$ such that $s_2 \xrightarrow{\Sigma'}_A \circ \xrightarrow{\{a\}}_A \circ \xrightarrow{\Sigma'}_A t_2$ and $(t_1, t_2) \in Z$.

Furthermore, for all $\Sigma' \subseteq \Sigma$, if there is a bisimulation between $A_1$ and $A_2$ modulo $\Sigma'$ then we write $A_1 \leftrightarrow A_2 (\Sigma')$.

2.4 Web services

Let $\Pi$ be a finite set of channels. Following the line of reasoning suggested by [3,4,6], we model Web services on finite automata with input and output. Web services communicate by sending asynchronous messages through channels. Communication through channels can be assumed to be reliable so that messages, once they are sent, do not get lost during their transmission. In this paper, for simplicity, we abstract from message contents and we consider that channels cannot contain, at all times, more than 1 message. Formally, a Web service over $\Pi$ and $\Sigma$ is a finite automaton over $\{!, ?\} \times \Pi \cup \Sigma$. For all $\pi \in \Pi$, the send action $!\pi$ consists of adding a message at channel $\pi$ whereas the receive action $?\pi$ consists of taking away a message at channel $\pi$. The action $!\pi$ can be executed provided the channel is not full (i.e. $\pi$ must contain exactly 0 message) whereas the action $?\pi$ can be executed provided the channel is not empty (i.e. $\pi$ must contain exactly 1 message). This motivates the
following definition. Let \( A = (S, \Delta, s^\text{in}) \) be a finite automaton over \((\{!, \?\} \times \Pi) \cup \Sigma\). By \( FA(A) \), we denote the finite automaton \( A' = (S', \Delta', s'^\text{in}) \) over \((\{!, \?\} \times \Pi) \cup \Sigma\) of exponential size such that \( S' = S \times 2^\Pi \), \( \Delta' \) is the function defined by

- \( (t, Q) \in \Delta'(s, !\pi) \) iff \( t \in \Delta(s, !\pi), Q = P \cup \{\pi\}, \pi \notin P \),
- \( (t, Q) \in \Delta'(s, ?\pi) \) iff \( t \in \Delta(s, ?\pi), Q = P \setminus \{\pi\}, \pi \in P \),
- \( (t, Q) \in \Delta'(s, a) \) iff \( t \in \Delta(s, a), Q = P \),

\( s'^\text{in} = (s^\text{in}, \emptyset) \). Remark that one can construct \( FA(A) \) in exponential time. Take the case of \( A \), the finite automaton from figure 1. Then \( FA(A) \) is the finite automaton from figure 2.

\[
\begin{align*}
\text{Fig. 1.} & \\
\text{Fig. 2.} &
\end{align*}
\]

3 Basic definitions and preliminary results

In this section, basic definitions are given and preliminary results are proved. These definitions and these results will be used in great depth in the remaining sections.

3.1 Basic definitions

It is convenient to take a finite set \( \Pi^o \) of channels such that \( (\Sigma \cup \Pi) \cap \Pi^o = \emptyset \) and \( \text{Card}(\Pi) = \text{Card}(\Pi^o) \) and to use a bijection \( \pi \mapsto \pi^o \) from \( \Pi \) to \( \Pi^o \). By \( L^o \), we mean the finite automaton \( A' = (S', \Delta', s'^\text{in}) \) over \((\{!, \?\} \times \Pi^o) \) such that \( S' = \{0\} \), \( \Delta' \) is the function defined by

- \( \Delta'(0, !\pi^o) = \{0\} \) and \( \Delta'(0, ?\pi^o) = \{0\} \),
- \( s'^\text{in} = 0 \). Let \( A = (S, \Delta, s^\text{in}) \) be a finite automaton over \((\{!, \?\} \times (\Pi \cup \Pi^o)) \cup \Sigma\). By \( \text{Del}^o(A) \), we denote the finite automaton \( A' = (S', \Delta', s'^\text{in}) \) over \((\{!, \?\} \times \Pi) \cup \Sigma\) such that \( S' = S \), \( \Delta' \) is the function defined by

- \( \Delta'(s, !\pi) = \Delta(s, !\pi) \cup \Delta(s, !\pi^o) \) and \( \Delta'(s, ?\pi) = \Delta(s, ?\pi) \cup \Delta(s, ?\pi^o) \),
- \( \Delta'(s, a) = \Delta(s, a) \),
\( s^{in'} = s^{in} \). By \( FA^o(A) \), we denote the finite automaton \( A' = (S', \Delta', s^{in'}) \) over \((\{!, ?\} \times (\Pi \cup \Pi'))\cup \Sigma \) of exponential size such that \( S' = S \times 2^\Pi \), \( \Delta' \) is the function defined by

- \((t, Q) \in \Delta'((s, P), !\pi) \) iff \( t \in \Delta(s, !\pi), Q = P \cup \{\pi\}, \pi \not\in P \) and \((t, Q) \in \Delta'((s, P), ?, \pi) \) iff \( t \in \Delta(s, ?, \pi), Q = P \setminus \{\pi\}, \pi \in P, \)
- \((t, Q) \in \Delta'((s, P), !\pi^o) \) iff \( t \in \Delta(s, !\pi^o), Q = P \cup \{\pi\}, \pi \not\in P \) and \((t, Q) \in \Delta'((s, P), ?, \pi^o) \) iff \( t \in \Delta(s, ?, \pi^o), Q = P \setminus \{\pi\}, \pi \in P, \)
- \((t, Q) \in \Delta'((s, P), a) \) iff \( t \in \Delta(s, a), Q = P, \)

\( s^{in'} = (s^{in}, \emptyset) \). Remark that one can construct \( FA^o(A) \) in exponential time. Let \( \mathcal{A} = (S, \Delta, s^{in}) \) be a finite automaton over \( \{!, ?, \} \times \Pi \). By \( Ren^o(A) \), we denote the finite automaton \( A' = (S', \Delta', s^{in'}) \) over \((\{!, ?\} \times (\Pi \cup \Pi'))\cup \Sigma \) such that \( S' = S \), \( \Delta' \) is the function defined by

- \( \Delta'(s, !\pi) = \{s\} \) and \( \Delta'(s, ?, \pi) = \{s\}, \)
- \( \Delta'(s, !\pi^o) = \Delta(s, !\pi) \) and \( \Delta'(s, ?, \pi^o) = \Delta(s, ?, \pi), \)
- \( \Delta'(s, a) = \{s\}, \)

\( s^{in'} = s^{in} \). Obviously, \( Ren^o(A) \) loops over \((\{!, ?, \} \times \Pi) \cup \Sigma \).

### 3.2 Preliminary results

Now, we present some useful lemmas.

**Lemma 3.1** Let \( \mathcal{A}_1 = (S_1, \Delta_1, s_1^{in}) \) be a finite automaton over \((\{!, ?, \} \times \Pi) \cup \Sigma \) and \( \mathcal{A}_2 = (S_2, \Delta_2, s_2^{in}) \) be a finite automaton over \((\{!, ?, \} \times \Pi) \). Then, \( FA^o(A_1 \otimes A_2) \) is isomorphic to \( Del^o(FA^o(A_1 \otimes L^o) \times Ren^o(A_2)) \).

**Proof.** States in \( FA^o(A_1 \otimes A_2) \) are of the form \(((s_1, s_2), P)\) with \( s_1 \in S_1, s_2 \in S_2 \) and \( P \subseteq \Pi \) whereas states in \( Del^o(FA^o(A_1 \otimes L^o) \times Ren^o(A_2)) \) are of the form \(((s_1, 0), P), s_2)\) with \( s_1 \in S_1, P \subseteq \Pi \) and \( s_2 \in S_2 \). Obviously, the bijection \(((s_1, s_2), P) \mapsto (((s_1, 0), P), s_2)\) is an isomorphism from \( FA^o(A_1 \otimes A_2) \) to \( Del^o(FA^o(A_1 \otimes L^o) \times Ren^o(A_2)) \). \( \Box \)

**Lemma 3.2** Let \( \mathcal{A}_1 = (S_1, \Delta_1, s_1^{in}) \) be a finite automaton over \((\{!, ?, \} \times \Pi) \cup \Sigma \) and \( \mathcal{A}_2 = (S_2, \Delta_2, s_2^{in}) \) be a finite automaton over \((\{!, ?, \} \times (\Pi \cup \Pi')) \cup \Sigma \) looping over \((\{!, ?, \} \times \Pi) \cup \Sigma \). Then, \( Del^o(FA^o(A_1 \otimes L^o) \times A_2) \) is isomorphic to \( FA^o(A_1 \otimes Del^o(L^o \times A_2)) \).

**Proof.** States in \( Del^o(FA^o(A_1 \otimes L^o) \times A_2) \) are of the form \(((s_1, 0), P), s_2)\) with \( s_1 \in S_1, P \subseteq \Pi \) and \( s_2 \in S_2 \) whereas states in \( FA^o(A_1 \otimes Del^o(L^o \times A_2)) \) are of the form \(((s_1, (0, s_2)), P)\) with \( s_1 \in S_1, s_2 \in S_2 \) and \( P \subseteq \Pi \). Obviously, the bijection \(((s_1, 0), P), s_2) \mapsto (((s_1, (0, s_2)), P) \) is an isomorphism from \( Del^o(FA^o(A_1 \otimes L^o) \times A_2) \) to \( FA^o(A_1 \otimes Del^o(L^o \times A_2)) \). \( \Box \)

**Lemma 3.3** Let \( \mathcal{A}_1 = (S_1, \Delta_1, s_1^{in}) \) be a finite automaton over \((\{!, ?, \} \times \Pi) \cup \Sigma \) and \( \Pi' \subseteq \Pi \). Then, one can construct in polynomial time a modal \( \mu \)-calculus formula \( f(A_1, \Pi') \) over \((\{!, ?, \} \times (\Pi \cup \Pi')) \cup \Sigma \) of polynomial size such that for all finite automata \( A_2 = (S_2, \Delta_2, s_2^{in}) \) over \((\{!, ?, \} \times (\Pi \cup \Pi')) \cup \Sigma, A_1 \models Del^o(A_2) (\{!, ?, \} \times \Pi') \) iff \( A_2 \models f(A_1, \Pi') \).

5
There exists a finite automaton with input and output. Take the case of services into a complex business process that can simulate the given finite automaton plays the role of the Web service that will combine and coordinate the available Web services into a complex business process that can simulate the given finite automaton. 

Now, we are ready to announce the first result of this paper:

5 Lower bound

\[ \text{Theorem 3.4} \]

Let \( A = (S_A, \Delta_A, s_A) \) and \( B = (S_B, \Delta_B, s_B) \) be finite automata over \( (\{!, ?\} \times \Pi) \cup \Sigma \) and \( \Pi' \subseteq \Pi \). Then, the following conditions are equivalent:

(i) There exists a finite automaton \( C \) over \( \{!, ?\} \times \Pi \) such that \( FA(A) \rightleftharpoons FA(B \otimes C) \) \((\{!, ?\} \times \Pi')\).

(ii) There exists a finite automaton \( C \) over \( (\{!, ?\} \times (\Pi \cup \Pi')) \cup \Sigma \) looping over \( (\{!, ?\} \times \Pi) \cup \Sigma \) and such that \( FA(A) \rightleftharpoons Del^\circ (FA(B \otimes L^\circ) \times C) \) \((\{!, ?\} \times \Pi')\).

(iii) There exists a finite automaton \( C \) over \( (\{!, ?\} \times (\Pi \cup \Pi')) \cup \Sigma \) looping over \( (\{!, ?\} \times \Pi) \cup \Sigma \) and such that \( FA^\circ (B \otimes L^\circ) \times C \models f(FA(A), \Pi') \).

4 Composition of Web services

This section considers issues that arise when addressing the task of combining and coordinating a set of Web services. We assume the process of Web service composition to be goal oriented: given a goal and a set of Web services, generate a composition of the Web services that satisfies the goal. According to [14,15,16], goals are conditions on the behaviour of the composition that can be expressed in the EaGLe language. In this approach, service composition boils down to the task of combining and coordinating the available Web services into a complex business process satisfying the given condition. According to [3,4,6], goals are finite automata with input and output, i.e. Web services as defined in section 2.4. In this approach, service composition boils down to the task of combining and coordinating the available Web services into a complex business process that can simulate the given finite automaton with input and output. In this paper, we automate composition as defined in the second approach. This brings us to the following decision problem:

- \( CP \): given a finite set \( \Sigma \) of actions, a finite set \( \Pi \) of channels, finite automata \( A = (S_A, \Delta_A, s_A) \) and \( B_1 = (S_{B_1}, \Delta_{B_1}, s_{B_1}) \), \( B_2 = (S_{B_2}, \Delta_{B_2}, s_{B_2}) \), ... , \( B_n = (S_{B_n}, \Delta_{B_n}, s_{B_n}) \) over \( (\{!, ?\} \times \Pi) \cup \Sigma \) and \( \Pi' \subseteq \Pi \), determine whether there exists a finite automaton \( C = (S_C, \Delta_C, s_C) \) over \( \{!, ?\} \times \Pi \) such that \( FA(A) \rightleftharpoons FA(B_1 \otimes \ldots \otimes B_n \otimes C) \) \((\{!, ?\} \times \Pi')\).

In \( CP \), \( A \) plays the role of the given finite automaton with input and output and \( B_1 \), \( B_2 \), ... , \( B_n \) play the role of the available Web services. As for the finite automaton \( C \), it plays the role of the Web service that will combine and coordinate the available Web services into a complex business process that can simulate the given finite automaton with input and output. Take the case of \( A, B_1, B_2 \), the finite automata from figure 3. Then the finite automaton \( C \) from figure 4 is such that \( FA(A) \rightleftharpoons FA(B_1 \otimes B_2 \otimes C) \) \((\{!, ?\} \times \{\pi_1, \pi_1', \pi_2, \pi_2'\})\).

5 Lower bound

Now, we are ready to announce the first result of this paper:

\( CP \) is \( \text{EXPTIME} \)-hard.
Let $\Sigma$ be a finite set of actions. A Petri net over $\Sigma$ is a structure of the form $N = (P, T, F, l)$ where $P$ is a finite set of places, $T$ is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a relation, $l$ is a function

- $l: T \rightarrow \Sigma$.

For all $t \in T$, let $t^* \subseteq P$ be the set of all $p \in P$ such that $t F p$. By $FA(N)$, we denote the finite automaton $A' = (S', \Delta', u^{in'})$ over $\Sigma$ such that $S' = 2^P$, $\Delta'$ is the function defined by

- $v' \in \Delta'(u', a)$ iff there exists $t \in T$ such that $l(t) = a$, $t^* \subseteq u'$ and $v' = (u' \setminus t) \cup t^*$, $u^{in'} = \emptyset$. Let us consider the following decision problem:

- $PN$: given a finite set $\Sigma$ of actions and Petri nets $N = (P_N, T_N, F_N, l_N)$, $O = (P_O, T_O, F_O, l_O)$ over $\Sigma$, determine whether $FA(N) \leftrightarrow FA(O) (\emptyset)$.

Seeing that $PN$ is $EXPTIME$-hard [9], it suffices to reduce $PN$ to $CP$ in order to demonstrate that $CP$ is $EXPTIME$-hard. Given a finite set $\Sigma$ of actions and Petri nets $N = (P_N, T_N, F_N, l_N)$, $O = (P_O, T_O, F_O, l_O)$ over $\Sigma$, we are asked whether $FA(N) \leftrightarrow FA(O) (\emptyset)$. The instance $\rho(\Sigma, N, O)$ of $CP$ that we construct is given by the finite set $\Sigma^e$ of actions, the finite set $\Pi$ of channels, the finite automata $\mathcal{A} = (S_A, \Delta_A, s_A^0)$ and $\mathcal{B} = (S_B, \Delta_B, s_B^0)$ over $\{!, ?\} \times \Pi$ and $\Pi' \subseteq \Pi$ defined by

- $\Sigma^e = \Sigma \cup \{a_1^e, a_2^e, a_3^e, a_4^e\}$ where $a_1^e, a_2^e, a_3^e$ and $a_4^e$ are new actions,
- $\Pi = P_N \cup P_O$,
- $\Pi' = P_N \cup P_O$,
- $\mathcal{A}$ is the finite automaton from figure 5,
- $\mathcal{B}$ is the finite automaton from figure 6.
In order to understand how the flower-form parts of $A$ and $B$ are defined, the reader is invited to consult figure 7. This completes the construction. Obviously, $\rho$ can be computed in logarithmic space. Moreover, $FA(N) \leftrightarrow FA(O)$ iff there exists a finite automaton $C = (S_C, \Delta_C, s_0^{C})$ over $\{!, ?\} \times \Pi$ such that $FA(A) \leftrightarrow FA(B \otimes C)$ $((\{!, ?\} \times \Pi'))$. Hence, $CP$ is $EXPTIME$-hard.
6 Upper bound

Whether $CP$ is in $EXPTIME$ or not is not known to us. Now, we are ready to announce the second result of this paper:

$CP$ is in $2EXPTIME$.

6.1 A $2EXPTIME$ decision procedure based on controller synthesis

Let us consider the following decision problem:

- $CS$: given a finite set $\Sigma$ of actions, a finite set $\Pi$ of channels, a finite automaton $A = (S_A, \Delta_A, s_A^0)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ and a modal $\mu$-calculus formula $\phi$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, determine whether there exists a finite automaton $B = (S_B, \Delta_B, s_B^0)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ looping over $\{(!, ?) \times \Pi \}\cup \Sigma$ and such that $A \times B \models \phi$.

The language of modal $\mu$-calculus cannot express the fact that a finite automaton over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ loops over $\{(!, ?) \times \Pi \}\cup \Sigma$. That is why Arnold et al. [2] extend it in such a way that looping becomes expressible. This extension is called modal-loop $\mu$-calculus. It consists in associating with each $\theta \in \{(!, ?) \times \Pi \}\cup \Sigma$ a proposition $\lambda_\theta$ whose interpretation is that a state $s$ of a finite automaton $A = (S, \Delta, s^0_\theta)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, $A \models \lambda_\theta$ iff $\Delta(s, \theta) = \{s\}$. Thus, one can construct in polynomial time a modal-loop $\mu$-calculus formula $g(\Pi, \Sigma)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ of polynomial size such that for all finite automata $A = (S, \Delta, s^0_\theta)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, $A \models g(\Pi, \Sigma)$ iff $A$ loops over $\{(!, ?) \times \Pi \}\cup \Sigma$. Moreover, given a finite automaton $A = (S_A, \Delta_A, s_A^0)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, Arnold et al. [2] show how to construct in polynomial time a modal $\mu$-calculus formula $\phi/A$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ of polynomial size such that for all finite automata $B = (S_B, \Delta_B, s_B^0)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, $A \times B \models \phi$ iff $B \models \phi/A$. Hence, $CS$ is equivalent to the following decision problem:

- given a finite set $\Sigma$ of actions, a finite set $\Pi$ of channels, a finite automaton $A = (S_A, \Delta_A, s_A^0)$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$ and a modal $\mu$-calculus formula $\phi$ over $\{(!, ?) \times (\Pi \cup \Pi^o)\} \cup \Sigma$, determine whether $\phi/A \wedge g(\Pi, \Sigma)$ is satisfiable.

Now, take a finite set $\Sigma$ of actions, a finite set $\Pi$ of channels, finite automata $A = (S_A, \Delta_A, s_A^0)$ and $B_1 = (S_{B_1}, \Delta_{B_1}, s_{B_1}^0)$, ..., $B_n = (S_{B_n}, \Delta_{B_n}, s_{B_n}^0)$ over $\{(!, ?) \times \Pi \}\cup \Sigma$ and $\Pi \subseteq \Pi$. To determine whether there exists a finite automaton $C = (S_C, \Delta_C, s_C^0)$ over $\{(!, ?) \times \Pi \}$ such that $FA(A) \hookrightarrow FA(B_1 \otimes \ldots \otimes B_n \otimes C)$, we consider the following algorithm:

(i) Compute $\phi = f(FA(A), \Pi')$.

(ii) Compute $B' = FA^\omega(B_1 \otimes \ldots \otimes B_n \otimes L^\omega)$.

(iii) Compute $\phi' = \phi/\Pi' \wedge g(\Pi, \Sigma)$.

(iv) If $\phi'$ is satisfiable then return the value $true$ else return the value $false$.

By theorem 3.4, the above algorithm returns the value $true$ iff there exists a finite automaton $C$ over $\{(!, ?) \times \Pi \}$ such that $FA(A) \hookrightarrow FA(B_1 \otimes \ldots \otimes B_n \otimes C)$ $\{(!, ?) \times \Pi'\}$. It can be implemented in double exponential time.
6.2 A 2EXPTIME decision procedure based on filtration

Let us consider the following decision problem:

- **FIL**: given a finite set $\Sigma$ of actions, a finite set $\Pi$ of channels, a finite automaton $A = (S_A, \Delta_A, s^0_A)$ over $\{!, ?\} \times \Pi \cup \Sigma$, a finite automaton $B = (S_B, \Delta_B, s^0_B)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ and $\Pi' \subseteq \Pi$, determine whether there exists a finite automaton $C = (S_C, \Delta_C, s^0_C)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping over $\{!, ?\} \times \Pi \cup \Sigma$ and such that $A \xleftarrow{\text{FIL}} \text{Del}^0(B \times C) (\{!, ?\} \times \Pi)$.

Suppose that we are given a finite set $\Sigma$ of actions, a finite set $\Pi$ of channels, a finite automaton $A = (S_A, \Delta_A, s^0_A)$ over $\{!, ?\} \times \Pi \cup \Sigma$, a finite automaton $B = (S_B, \Delta_B, s^0_B)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ and $\Pi' \subseteq \Pi$. Let $C = (S_C, \Delta_C, s^0_C)$ be a finite automaton over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping over $\{!, ?\} \times \Pi$ and such that $A \xleftarrow{\text{FIL}} \text{Del}^0(B \times C) (\{!, ?\} \times \Pi')$. Hence, there exists a bisimulation $Z$ between $A$ and $\text{Del}^0(B \times C)$ modulo $\{!, ?\} \times \Pi'$ such that $s^m_A Z (s^m_B, s^m_C)$. Let $\equiv \subseteq S_C \times S_C$ be the binary relation such that for all $s^1_C, s^2_C \in S_C$,

- $s^1_C \equiv s^2_C$ iff for all $s_A \in S_A$ and for all $s_B \in S_B$, $s_A Z (s_B, s^1_C)$ iff $s_A Z (s_B, s^2_C)$.

Note that $\equiv$ is an equivalence relation. Let $s_C \in S_C$. The set of all states in $S_C$ equivalent to $s_C$ modulo $\equiv$, in symbols $\equiv s_C$, is called the equivalence class of $s_C$ in $S_C$ modulo $\equiv$ as its representative. The set of all equivalence classes of $S_C$ modulo $\equiv$, in symbols $S_C/\equiv$, is called the quotient set of $S_C$ modulo $\equiv$. Suppose that $C^f = (S_C, \Delta_{C^f}, s^0_{C^f})$ is the finite automaton over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping over $\{!, ?\} \times \Pi$ and such that $S^f = S_C/\equiv$, $\Delta_{C^f}$ is the function such that

- $| t_C' | \in \Delta_{C^f}(| s_C |, !\pi^o)$ iff for all $s_A \in S_A$ and for all $s_B, t_B \in S_B$, if $t_B \in \Delta_B(s_B, !\pi^o)$ and $s_A Z (s_B, s_C)$ then there exists $t_A \in S_A$ such that $t_A \in \Delta_A(s_A, !\pi^o)$ and $t_A Z (t_B, t_C')$.

$s^m_{C^f} = | s^m_C |$. Then $C^f$ is called the greatest filtration of $C$ through $A$ and $B$. Let $Z^f \subseteq S_A \times (S_B \times S_{C^f})$ be the binary relation such that for all $s_A \in S_A$ and for all $(s_B, | s_C |) \in S_B \times S_{C^f}$, $s_A Z^f (s_B, | s_C |)$ iff $s_A Z (s_B, s_C)$. It is a simple matter to check that $Z^f$ satisfies $A \xleftarrow{\text{Fil}} \text{Del}^0(B \times C^f) (\{!, ?\} \times \Pi')$. For our purpose, the crucial property of the greatest filtration is that the following conditions are equivalent:

- there exists a finite automaton $C = (S_C, \Delta_C, s^0_C)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping over $\{!, ?\} \times \Pi \cup \Sigma$ and there exists a relation $Z \subseteq S_A \times (S_B \times S_C)$ such that $Z$:
  - $A \xleftarrow{\text{FIL}} \text{Del}^0(B \times C) (\{!, ?\} \times \Pi')$,
  - there exists a finite automaton $C = (S_C, \Delta_C, s^0_C)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping over $\{!, ?\} \times \Pi \cup \Sigma$ and there exists a relation $Z \subseteq S_A \times (S_B \times S_C)$ such that $Z$:
    - $A \xleftarrow{\text{FIL}} \text{Del}^0(B \times C) (\{!, ?\} \times \Pi')$ and
    - $(C_1)$ $S_C \subseteq 2^{|\text{Card}(S_A) \times \text{Card}(S_B)|}$,
    - $(C_2)$ $t_C \in \Delta(C(s_C, !\pi^o))$ iff for all $s_A \in S_A$ and for all $s_B, t_B \in S_B$, if $t_B \in \Delta_B(s_B, !\pi^o)$ and $(s_A, s_B) \in S_C$ then there exists $t_A \in S_A$ such that $t_A \in \Delta_A(s_A, !\pi^o)$ and $(t_A, t_B) \in t_C$,
    - $(C_3)$ $s_A Z (s_B, s_C)$ iff $(s_A, s_B) \in S_C$.

Hence, we can give a simple algorithm for solving **FIL**:

(i) For each finite automaton $C = (S_C, \Delta_C, s^0_C)$ over $\{!, ?\} \times (\Pi \cup \Pi^0) \cup \Sigma$ looping
over \(((\text{!}, ?) \times \Pi) \cup \Sigma\) and such that \((C_1)\) and \((C_2)\) are satisfied, determine whether \(Z: A \leftrightarrow Del^\pi(\mathcal{B} \times \mathcal{C})\) \(((\text{!}, ?) \times \Pi')\) where \(Z \subseteq S_A \times (S_B \times S_C)\) is the relation satisfying \((C_3)\).

(ii) If one of these calls returns the value \(true\) then return the value \(true\) else return the value \(false\).

Not surprisingly, the above algorithm returns the value \(true\) iff there exists a finite automaton \(C = (S_C, \Delta_C, s_C^{in})\) over \(((\text{!}, ?) \times (\Pi \cup \Pi^o)) \cup \Sigma\) looping over \(((\text{!}, ?) \times \Pi) \cup \Sigma\) and such that \(A \leftrightarrow Del^\pi(\mathcal{B} \times \mathcal{C})\) \(((\text{!}, ?) \times \Pi')\). Seeing that determining whether \(A \leftrightarrow Del^\pi(\mathcal{B} \times \mathcal{C})\) \(((\text{!}, ?) \times \Pi')\) can be done in polynomial time [1], it follows immediately that it can be implemented in exponential time. Now, take a finite set \(\Sigma\) of actions, a finite set \(\Pi\) of channels, finite automata \(A = (S_A, \Delta_A, s_A^{in})\) and \(B_1 = (S_{B_1}, \Delta_{B_1}, s_{B_1}^{in}), \ldots, B_n = (S_{B_n}, \Delta_{B_n}, s_{B_n}^{in})\) over \(((\text{!}, ?) \times \Pi) \cup \Sigma\) and \(\Pi' \subseteq \Pi\). To determine whether there exists a finite automaton \(C = (S_C, \Delta_C, s_C^{in})\) over \(((\text{!}, ?) \times \Pi)\) such that \(FA(A) \leftrightarrow FA(B_1 \otimes \ldots \otimes B_n \otimes C)\), we consider the following algorithm:

(i) Compute \(A' = FA(A)\).
(ii) Compute \(B' = FA^o(B_1 \otimes \ldots \otimes B_n \otimes L^o)\).
(iii) If there exists a finite automaton \(C = (S_C, \Delta_C, s_C^{in})\) over \(((\text{!}, ?) \times (\Pi \cup \Pi^o)) \cup \Sigma\) looping over \(((\text{!}, ?) \times \Pi) \cup \Sigma\) and such that \(A' \leftrightarrow Del^\pi(\mathcal{B'} \times \mathcal{C})\) \(((\text{!}, ?) \times \Pi')\) then return the value \(true\) else return the value \(false\).

By theorem 3.4, the above algorithm returns the value \(true\) iff there exists a finite automaton \(C\) over \((\text{!}, ?) \times \Pi\) such that \(FA(A) \leftrightarrow FA(B_1 \otimes \ldots \otimes B_n \otimes C)\) \(((\text{!}, ?) \times \Pi')\). It can be implemented in double exponential time.

7 Conclusion and open problems

We have presented a framework in which Web services are described as message passing automata. Deterministic algorithms that check a composition’s existence and return one if it exists have been proposed. In order to ensure their termination in a finite number of steps, we have characterized the computational complexity (EXPTIME-hardness and membership in 2EXPTIME) of the composition problem. Our main results are that \(CP\) is EXPTIME-hard and \(CP\) is in 2EXPTIME. An interesting (and still open) question is to evaluate the exact complexity of Web service composition. Variants of \(CP\) can be considered as well. For instance, one may consider that the given automata are deterministic or that the channels they use can contain more than 1 message at a time. Take another example: one may replace “bisimulation” by “trace equivalence”. What is the complexity of Web service composition in this case? In other respects, we have not considered which message is actually sent/received when performing a messaging action. To enrich our formalism that way, we may augment each send/receive action with an additional first-order term indicating what kind of message is exchanged. Henceforth, a message exchange action consists of a channel \(\pi\) and a first-order term \(t\) which indicate that a message of the form \(t\) is sent or received through channel \(\pi\). For which classes of messages is Web service composition decidable? When this problem is decidable, how complex is it?
References


On symbolic semantics for name-decorated contexts
(extended abstract)

Andrea Bracciali, Roberto Bruni, Alberto Lluch Lafuente

Department of Computer Science, University of Pisa, Italy

Abstract

Under several regards, various of the recently proposed computational paradigms are open-ended, i.e. they may comprise components whose behaviour is not or cannot be fully specified. For instance, applications can be distributed across different administration domains that do not fully disclose their internal business processes to each other, or the dynamics of the system may allow reconfigurations and dynamic bindings whose specification is not available at design time. While a large set of mature design and analysis techniques for closed systems have been developed, their lifting to the open case is not always straightforward. Some existing approaches in the process calculi community are based on the need of proving properties for components that may hold in any, or significantly many, execution environments. Dually, frameworks describing the dynamics of systems with unspecified components have also been presented. In this paper we lay some preliminary ideas on how to extend a symbolic semantics model for open systems in order to deal with name-based calculi. Moreover, we also discuss how the use of a simple type system based on name-decoration for unknown components can improve the expressiveness of the framework. The approach is illustrated on a simple, paradigmatic calculus of web crawlers, which can be understood as a term representation of a simple class of graphs.

Keywords: Open Systems, Abstract Semantics, Nominal Calculi, Type Systems.

1 Introduction

Concurrent and distributed systems are more and more becoming open environments where components, agents or services interact one with another by dynamically establishing connections. For instance, in service oriented architectures, computational resources may be accessed through temporary interactive sessions. Such open-interaction environments, subject to the dynamical binding of their components, may result into systems being partially defined even at run-time. Describing and analysing the behaviour of such systems in presence of incomplete information clearly appears more difficult than the analysis of closed interactive systems, already recognised as a challenging problem in its own.

Open computational environments have been first addressed in terms of execution contexts, for instance in order to determine the (minimal) execution context where the com-
putation of a component may exhibit some desired properties. In the semantical approach of [Sew98], the possible transitions of a component are labelled with information characterising those contexts in which behavioural equivalence enjoys congruence properties (relevant to allow modular reasoning). Then, several other authors have proposed different symbolic semantics [LM00,SS03,SS05a,SS05b,KSS05,BGMS05,EK06] so as not considering all the possible contexts, because universal quantification can seriously impair verification techniques. These semantics carry abstract representations of the minimal contexts necessary for components to evolve. Here the term “symbolic” reminds the attempt of defining suitably abstract representations that can finitely represent universal classes of components and contexts. The issue of avoiding universal closure of contexts finds its dual formulation in avoiding universal closure with respect to pluggable components.

In [BBB07], jointly with Paolo Baldan, a general methodology for analysing the behaviour of open systems modelled as contexts \( C[X_1, \ldots, X_n] \), i.e. open terms of suitable process calculi have been proposed. Variables of open terms represent holes where other contexts and components \( p \), i.e. closed terms, can be dynamically plugged in. The operational semantics of contexts is given by means of a symbolic transition system (STS), where states are contexts and transitions are labelled by modal formulae characterising the structure that a component must possess or the actions it must be able to perform in order to enable a symbolic transition. Symbolic transitions are of the form:

\[
C[X_1, \ldots, X_n] \xrightarrow{\varphi_1, \ldots, \varphi_n} a D[Y_1, \ldots, Y_m]
\]

The corresponding closed system \( C[p_1, \ldots, p_n] \) can perform a transition labelled with \( a \), whenever each component \( p_i \) satisfies the corresponding formula \( \varphi_i \). The target state will be a suitable instance of \( D[Y_1, \ldots, Y_m] \), where process variables \( Y_1, \ldots, Y_m \) appear in formulae \( \varphi_1, \ldots, \varphi_n \). The logic where the formulae \( \varphi_i \) live and the notion of satisfaction are targeted to the process calculus under study. Starting from the rules defining a calculus, a constructive procedure based on unification distills a (sound and complete) standard STS.

Given an STS, several behavioural equivalences can be defined directly over contexts, amongst which we mention strict and loose bisimilarities. The former is a straight extension of the ordinary bisimilarity with exact matching of transition labels, while the latter is obtained by relaxing the requirements when comparing formulas during the bisimulation game. In order to abstract from internal computations, symbolic counterparts of weak bisimilarity have been defined. They are called strict and loose weak symbolic bisimilarity (denoted \( \approx_s \) and \( \approx_l \), respectively). All these equivalences are correct approximations of their universal counterparts. Differently from other approaches the STS semantics preserves the openness of the system during its evolution, thus allowing dynamic instantiation to be accounted for in the semantics.

By integrating ideas from [BBB02,BBB07] and [BLMT08,BLM08,BLMT07], we are interested in the development of a flexible semantic framework for open systems that admits a graphical counterpart. In this extended abstract we report on an ongoing development of the STS theory aimed at accounting for calculi with an explicit treatment of names, à la \( \pi \)-calculus. Names broadly represent references of a possibly reconfigurable interconnection network amongst components. Consequently, the extended theory may be adapted also to other representation formalisms, such as the hierarchical graphs considered in [BLM08], where names can be used to account for the hierarchy.
In order to make the framework more flexible, drawing inspiration from [BLMT08], we introduce a type discipline for open systems which prescribes processes, contexts and variables can be composed together. Types fix some kind of basic interface, allowing or disallowing the use of certain names within the corresponding well-typed processes.

We present our type framework with the help of a web crawling scenario, modelled with a simple nominal calculi, where names stand for references to web pages and processes offer an abstract representation for web crawlers and pages. In this first exploration the use of names is limited (for instance we do not deal with restriction operators), but we believe it is still sufficient to illustrate the relevance of the proposed approach. We define crawlers with different policies and confront them with a non-fully specified network. By adopting a suitable symbolic equivalence we can test the different behaviours over a symbolic semantics. The needed extensions to the theory of STS are discussed. For the sake of readability and generality, the actors of our scenario are also illustrated as graphs, where processes and names play the role of edges and connections, and operational rules that of graph transformations. A further advantage of this graphical presentation is to make evident that interfaces can dynamically evolve, e.g. crawlers expose the web addresses they know and such knowledge is increased during their exploration activity.

We show how global properties of the network can be enforced by imposing type restrictions to unknown network components. Types constrain the pages that the unknown part of the network is enforced to contain and the list of links that the network can point to. In particular, we shall concentrate on valid networks, where no broken link is allowed. Such type restrictions have to be updated according to the symbolic transitions that make the overall system evolve. Consequently, standard subject reduction results have to be rethought in this dynamical open context. One of the benefits of considering type restrictions in our example is that, while crawlers can be distinguished in arbitrary networks, their behaviour is equivalent in networks of type valid.

Summarising, the main objective of this extended abstract is twofold: (i) to define typed extensions of the STS symbolic semantics for nominal calculi, and (ii) to use a type discipline for unknown components to derive suitable abstract equivalences. We remark that our types are inspired by graphical models of process calculi and that, for the first time, it is shown a significant abstract equivalence based on loose weak symbolic bisimilarity.

This paper is structured as follows. Section 2 overviews the basics of STS. Section 3 describes our simple web crawling scenario: a simple nominal calculus over which we apply the STS theory. Section 4 introduces name-decorated types in the STS approach. Section 5 draws some conclusions and outlines future developments.

2 Background

The main concepts about STS and associated symbolic behavioural equivalences are briefly recalled. A detailed presentation can be found in [BBB07].

For mere illustration purposes, we introduce for this section a simple process calculus, called Tick. The processes of the Tick calculus are defined by the syntax and operational rules in Figure 1, where \( \ell \) ranges over a fixed set of labels \( \Lambda \), \( \tau \in \Lambda \) is a distinguished label and \( a \) ranges over \( \Lambda - \{ \tau \} \). Tick processes consist of lists of actions which can
be performed sequentially. The hiding operator \((a)\) allows to hide action \(a\), which then shows up as silent action \(\tau\) at the top level. For example, the Tick process \((a)(b)\ c\ a\ 0\) can perform a series of two steps: \((a)(b)\ c\ a\ 0\ \rightarrow\ (a)(b)\ a\ 0\ \rightarrow\ (a)(b)\ 0\). Note that to avoid confusion with the positioning of labels in symbolic transitions, we put the action label on the lower-right of the arrow and not above it.

2.1 Processes, Contexts and Formulas

We distinguish between processes (ranged over by \(p,q,...\)), i.e. closed terms of a process calculus, and contexts (ranged by \(C[X_1,...,X_n], D[X_1,...,X_n],...\)), i.e. terms of the calculus that may contain variables. For the sake of readability, we consider only single-holed contexts \(C[X]\), where \(X\) is the variable occurring in the context. Processes are often considered up to some suitable structural congruence \(\equiv\) but in our example we will not.

An operational and abstract semantics of contexts, can be defined as a symbolic transition system, whose states are contexts and whose labels encode the structural and behavioural conditions that components should fulfil to enable the move, according to the following principles: (1) abstracting from components not playing an active role in the transition; (2) specifying the active components as less as possible; and (3) making assumptions both on the structure and on the behaviour of the active components.

Labels consist of formulae of a suitable logic, \(\phi,\psi,...\) comprising both temporal and spatial modalities in the style of [CC01,CG00] and depend on the specific calculus considered. Each formula represents the set of processes that fulfil it. A possible temporal formula is \(\lozenge a\phi\), satisfied by the processes that can fulfil \(\phi\) after having performed an \(a\) labelled transition (\(p \models \lozenge a\phi\) if \(\exists q. p \rightarrow a q \land q \models \phi\)). Spatial formulae are about the algebraic structure of a term, so that, for instance, \(p \models f(\phi)\) if \(\exists q. p \equiv f(q)\) and \(q \models \phi\), where \(f\) is one of the operators of the calculus. Thus, each component \(p\) can also be regarded as a (purely spatial) formula.

To gain some insights regarding the choice of the logic, note that an instance \(C[p]\) of a given context \(C[X]\), in order to perform a transition, must match the left-hand side of the conclusion of a semantical rule. This might impose the component \(p\) to have a certain structure, hence the need of inserting the spatial operators \(f \in \Sigma\) in the logic, where \(\Sigma\) denotes the signature of the calculus under consideration. Furthermore, the premises of the matched rule must be satisfiable. Such premises may require component \(p\) to be able to exhibit some behaviour, i.e. to perform a certain transition. Hence the logic includes also temporal operators \(\lozenge a\) expressing the capability of performing action \(a\).

Labels must also be able to express no constraints over unspecified components of contexts, for instance when they do not take active part in the transition or in order to avoid unnecessarily tight constraints over components. This is achieved by including variables
as formulas of the logic which are fulfilled by any process. For instance, the formula $\varphi a X$ is satisfied by any process which is able to perform an action $a$, i.e. by any process $p$ such that $p \rightarrow a q$ for some $q$. Variables in formulae will be later used to identify the continuation, or residual, of a process after it has exhibited the capabilities and structure imposed by the formula. For instance, whenever $p \models \varphi a X$ and thus $p \rightarrow a q$, the variable $X$ in the formula $\varphi a X$, identifies the continuation $q$. We say that $p$ satisfies $\varphi$ with residual $q$, written $p \models \varphi; q$, when $p \models \varphi[q/X]$, for $X$ being the only process variable of $\varphi$. Symbol $;$ is also used for formulae composition such that $\varphi; \psi$ is an alias for $\varphi[q/X]$ (for $X$ being the only process variable of $\varphi$).

2.2 Symbolic Transition Systems

An STS $S$ is a set of symbolic transitions

$$C[X] \overset{D[\ell]}{\rightarrow} a D[Y]$$

The variable names in contexts are not relevant, while the correspondence between each variable $X$ in the source and its residual $Y$ in the target, as expressed by the formula $\varphi$ in which the residual may occur, is relevant.

For $S$ to provide an abstract view of a given process calculus we require some additional properties enforcing the correspondence with the ground transitions over components. Intuitively, whenever $C[X] \overset{D[\ell]}{\rightarrow} a D[Y]$, the context $C$, if instantiated with any component satisfying $\varphi$, must be able to perform action $a$ and become a suitable instance of $D$. More precisely, for any component $q$ such that $p \models \varphi; q$, the component $C[p]$ can perform $a$ becoming $D[q]$ (soundness). Analogously, any ground transition on components $C[p] \rightarrow a q$ should have a suitable symbolic counterpart with source $C[X]$ (completeness).

A constructive procedure for determining a correct and complete STS has been defined (see [BBB07]). It relies on unification for defining the constraints over unknown components of a coordinator according to the structure of semantical rules. It can be straightforward implemented in Prolog for a large class of calculi. An overview of the construction will be given in Section 3.1.

Example 1 Let $C[X]$ denote an arbitrary context in Tick. Then the STS consisting of the following (schema of) symbolic transitions is sound and complete:

$$(a_1) \ldots (a_n) a. C[X] \overset{Y_\tau}{\rightarrow} (a_1) \ldots (a_n) C[Y] \quad (a_1) \ldots (a_n) \ell. C[X] \overset{Y_\ell}{\rightarrow} (a_1) \ldots (a_n) C[Y]$$

where $n \geq 0$, $a \in \{a_1, \ldots, a_n\}$ and $\ell \not\in \{a_1, \ldots, a_n\}$. Intuitively, either the hole does nothing and the rest of the context is able to execute an action according to (hide) or (lift) (leftmost transitions), or the hole itself is able to perform an action (rightmost transitions).

For example, the contexts $(a) (b) a. X$ and $(a) (b) X$ have the transitions

$$(a) (b) a. X \overset{Y_\tau}{\rightarrow} (b) Y \quad \text{and} \quad (a) (b) X \overset{Y_\ell}{\rightarrow} (b) Y$$

for $\ell \not\in \{a, b\}$ and $\alpha \in \{a, b\}$. \qed
2.3 Strong Symbolic Bisimilarities

Given a process calculus, several observational equivalences can be defined on top of its operational semantics given in terms of a labelled transition system (LTS). We focus on bisimilarity, by far the most popular equivalence due to its suitability to support modular reasoning and efficient model checking techniques. We start recalling ground bisimilarity.

**Definition 2** (\(\sim\)) A strong bisimulation is a symmetric relation \(\sim\) over processes such that if \(p \sim q\), then for any transition \(p \xrightarrow{\alpha} p'\) a component \(q'\) and a transition \(q \xrightarrow{\alpha} q'\) exist such that \(p' \sim q'\). We denote by \(\sim\) the largest bisimulation, called strong bisimilarity or just bisimilarity.

A natural way of lifting equivalences from ground processes to contexts consists of considering all possible closed instances of the contexts, so that \(C[X] \equiv_u D[X]\) if and only if \(\forall p, C[p] \equiv D[p]\). However, universal quantification makes verification hard when not unfeasible. Moreover, such a bisimilarity works with a complete, although potentially infinite, specification of the system future behaviour, i.e. all its possible instantiations. This may not be appropriate when dealing with open systems. Informally speaking, the instant in which information becomes available seems to have a role in distinguishing the behaviour of different contexts.

**Definition 3** (\(\sim_u\)) A symmetric relation \(\sim\) over the set of contexts \(C\) is a strict symbolic bisimulation if for any two contexts \(C[X]\) and \(D[X]\) such that \(C[X] \equiv D[X]\), for any transition

\[
C[X] \xrightarrow{\phi} a C'[Y]
\]

there exists a transition \(D[X] \xrightarrow{\phi} D'[Y]\) such that \(C'[Y] \equiv D'[Y]\). The largest strict symbolic bisimulation is an equivalence relation \(\sim_u\) called strict symbolic bisimilarity.

For instance, referring to the calculus \(\text{Tick}\), we can show that \((a)\ (b)\ (C)\ (a)\ X\), since the symbolic moves for the contexts (see Example 1) are of the kind

\[
(a)\ (b)\ X \xrightarrow{\alpha Y} (a)\ Y \quad (b)\ (a)\ X \xrightarrow{\alpha Y} (b)\ Y
\]

where \(\ell = \alpha\) if \(\alpha \not\in \{a, b\}\) and \(\ell = \tau\), otherwise.

For a sound and complete STS we have \(\sim_u \Rightarrow \sim_{ul}\), but the converse does not hold in general. As mentioned, open processes that are equivalent under strict symbolic bisimilarity are ensured to be equivalent under universal closure but the vice-versa may not hold.

A non-trivial relaxation in the presence of spatial formulae regards the requirement of exact matching between the formulae labels: a transition can be simulated by another transition with weaker spatial constraints on the residuals.

**Definition 4** (\(\sim_{ul}\)) A symmetric relation \(\sim\) over the set of contexts \(C\) is a loose symbolic bisimulation if for any pair of contexts \(C[X]\) and \(D[X]\) such that \(C[X] \equiv D[X]\), for any transition

\[
C[X] \xrightarrow{\phi} a C'[Y]
\]

a transition \(D[X] \xrightarrow{\psi} D'[Z]\) and a spatial formula \(\psi\) exists such that \(\phi = \psi; \psi'\) and \(C'[Y] \equiv D'[\psi']\). The greatest loose bisimulation \(\sim_{ul}\) is called loose symbolic bisimilarity.
For sound and complete STS it holds $\sim_s \Rightarrow \sim_1 \Rightarrow \sim_u$. We note that $\sim_1$ is not guaranteed to be an equivalence relation, since it may fail to be transitive in some “pathological” situations (see the example in [BBB05]). In such cases, its transitive closure $(\sim_1)^*$ should be considered as the relevant equivalence.

2.4 Weak Symbolic Bisimilarities

Many calculi, in particular those representing distributed systems, present silent actions $\tau$ that model internal (non-observable) computations. In such cases, strong bisimilarity is too fine, and weak bisimilarity $\approx$, which abstracts away non-observable transitions during the simulation game, provides a more meaningful equivalence. We denote by $\approx_u$ its counterpart over contexts defined by universal closure, and we present a straight weak extension of symbolic bisimilarities.

The relations $\overset{\phi}{\Rightarrow}_a$ and $\overset{\phi}{\Rightarrow}$ represent in a single transition, called weak (symbolic) transition, a sequence of symbolic transitions with at most one observable action or none, respectively. Formula $\phi$, labelling the weak transitions, arises as the composition of the formulae labelling each single step. Then $C[X] \overset{\phi}{\Rightarrow} D[Y]$ if $C[X] \overset{\phi_1}{\Rightarrow} \tau \overset{\phi_2}{\Rightarrow} \cdots \overset{\phi_n}{\Rightarrow} \tau D[Y]$, with $\phi = \phi_1; \cdots; \phi_h$ and $h \geq 0$. Analogously, $C[X] \overset{\phi}{\Rightarrow}_a D[Y]$ stands for $C[X] \overset{\phi_1}{\Rightarrow} \tau \cdots \overset{\phi_k}{\Rightarrow} \tau \overset{\phi_k+1}{\Rightarrow} a \overset{\phi_{k+1}}{\Rightarrow} \tau \cdots \overset{\phi_h}{\Rightarrow} \tau D[Y]$. In the following we let $\overset{\phi}{\Rightarrow}_\ell$ denote $\overset{\phi}{\Rightarrow}$ if $\ell = \tau$ and $\overset{\phi}{\Rightarrow}_\ell$ otherwise.

**Definition 5 ($\approx_s$)** A symmetric relation $\overset{\phi}{\Rightarrow}$ on contexts is a strict weak symbolic bisimulation if for all contexts $C[X]$, $D[X]$ with $C[X] \overset{\phi}{\Rightarrow} D[X]$ we have

- if $C[X] \overset{\phi}{\Rightarrow} C'[Y]$ then $D[X] \overset{\psi}{\Rightarrow}_\ell D'[Y]$ and $C'[Y] \overset{\psi'}{\Rightarrow}_\ell D'[Z]$.

The largest strict weak symbolic bisimulation $\approx_s$ is an equivalence relation called strict weak symbolic bisimilarity (it holds $\approx_s \Rightarrow \approx_u \Rightarrow \approx_u$).

The contexts $(a) a$. $X$ and $(a) X$ of the Tick calculus are not strict bisimilar, but they are weak strict bisimilar. Roughly, this happens because the symbolic move $(a) a$. $X \overset{\psi}{\Rightarrow}_\ell (a) X$ can be weakly simulated by $(a) X$ by remaining idle.

Finally, a loose weak symbolic bisimilarity can be defined, abstracting on silent actions and releasing constraints over formula correspondence.

**Definition 6 ($\approx_l$)** A symmetric relation $\overset{\phi}{\Rightarrow}$ on contexts is a loose weak symbolic bisimulation if for all contexts $C[X]$, $D[X]$ with $C[X] \overset{\phi}{\Rightarrow} D[X]$ we have

- if $C[X] \overset{\phi}{\Rightarrow}_\ell C'[Y]$ then $D[X] \overset{\psi}{\Rightarrow}_\ell D'[Z]$ and a spatial formula $\psi'$ exists such that $\phi = \psi; \psi'$ and $C'[Y] \overset{\psi'}{\Rightarrow}_\ell D'[\psi'].$

The largest loose weak symbolic bisimulation $\approx_l$ is called loose weak symbolic bisimilarity.
3 Scenario: Web Crawlers

Web crawlers (also known as bots, spiders or scutters) are programs that systematically browse the web to gather (and even produce) data. Prominent examples include useful applications such as those used to feed search engines (e.g. Googlebot), and spambots that collect email addresses or post forums with malicious purposes (e.g. spamming or phishing).

Crawlers start their search with a seed of pages and maintain a list of visited pages. Known pages are examined to extract their links and add them to the list of pages to visit (the crawling frontier). Crawlers follow certain policies that regard page selection or if and how frequently pages are revisited. Such policies have an impact on the performance of a site and in particular on its performance: a non polite crawler with a high frequency of page request can overload the web server.

Some protocols exist that aim at harmonising the collaboration between crawlers and sites. For instance, robot exclusion and inclusion protocols (e.g. the de-facto standards robots.txt and sitemaps, respectively) are used by web sites to inform crawlers of links to be excluded and included in their spidering activity. Crawlers are free to respect or not such protocols. On the other hand, web servers can sometimes distinguish crawlers from human browsers (e.g. based on navigation speed or patterns) and thus control whether protocols are being respected or violated.

We consider a scenario in which crawlers adhere to different policies that depend on the level of trust in the information available from the net, viz. their propensity to check the validity of links. A scrupulous crawler checks the existence (e.g. requesting the page header only) of a page before deciding to examine it (i.e. downloading it completely) and before communicating the page to its (possibly remote) database. A cautious crawler moves (i.e. changes target page) in a similar way, but does not check the page existence when communicating the url of a page to its database. A rash crawler checks nothing, i.e. it assumes the existence of pages that it communicates or tries to examine. All three kinds of crawler are able to examine an existing page. For the sake of simplicity we restrict to static networks: no page is added or removed during crawling activities.

Each kind of crawler has a different impact on a web server performance: a scrupulous crawler performs more page requests than the the cautious one, which, in his turn, performs more requests than the rash one.

We model such scenario with a simple name-based calculus where crawler agents \( c \) operate on a web of links \( \text{link}(x, y) \). We assume denumerable sets of channel names (ranged by \( a, b, \ldots \)) and of site addresses (ranged by \( x, y, z, w, \ldots \)) are available. The web system \( s \) may be empty or comprise crawlers, links and their composition:

\[
\begin{align*}
s & ::= 0 \mid c \mid \text{link}(x, y) \mid s \mid s
\end{align*}
\]

Sites are seen just as collections of links with the same origin. If the collection is empty we say the site is missing, it is valid otherwise. If the target of a link is a missing site, then the link is called broken.

A crawler is an autonomous agent that can visit sites, learn new site addresses and communicate them to its database on a given channel. We distinguish three kinds of crawlers...
c(a, x, ˜y) | link(x, z) | s \rightarrow_{\tau} c(a, x, ˜y + z) | link(x, z) | s

(i)

Figure 2. Textual (i) and graphical (ii) representation of LEARN rules where c \in \{rash, cautious, scrupulous\}.

c ::= rash(a, x, ˜y) | cautious(a, x, ˜y) | scrupulous(a, x, ˜y)

where a is the channel for communicating site addresses, x is the current site address of the crawler and ˜y is the set of site addresses the crawler has already learnt (but not necessarily valid or visited). We let ˜y denote the set \{y_1, ..., y_n\} and write ˜y + x for the set \{y_1, ..., y_n, x\} and ˜y - y_i for the set \{y_1, ..., y_{i-1}, y_{i+1}, ..., y_n\}.

The operational semantics is given by few (unconditional) rewrite rules, see Figures 2–4, assuming that parallel composition is associative, commutative and with identity 0. The rules are parametric w.r.t. a generic system s and w.r.t. suitable site addresses x, ˜y, z, w and reference channel a for the crawler.

The rules are accompanied by a self-explanatory visual notation that is reminiscent of a graphical interpretation of process calculi (see e.g. [Gad07,FHL+05]): names are represented as nodes of type ◦ and • for channels and pages, respectively, crawlers and links as hyper-edges (rounded boxes) and their arguments (names used) are indicated by tentacles of various types. More precisely, the first argument of a crawler (e.g. the address of its database) is indicated by an upwards concave tentacle, the second one (the current site) by a bar-ended tentacle and the set of visited sites by arrowed tentacles. For links the arrowed tentacle indicates the target and the plain one represents the source. In our intuitive notation, items in the left- and right-hand side are identified by their position and we remark that a graph rewriting reading of the rules should be understood with matchings not being injective, i.e. two different rule nodes can be matched with the same actual node (e.g. learning of known pages is allowed).

Any crawler can learn new site addresses by looking at the links departing from its current site. The corresponding rules are identical for the three different kind of crawlers and abstract away the actual interaction that would take place in concrete crawlers (rules LEARN). The graphical representation makes evident that the interface of the crawler agent may be enlarged by the acquisition of a new site address.

Any crawler can move to new sites (rules MOVE). In particular, rash crawlers move eagerly around the web, to any target they have learnt; cautious and scrupulous crawlers move only to valid sites. The graphical representations show the two different policies used by the crawlers and make evident the swap of names in the interface of the crawler.
A second difference in the considered policies lies in the observations crawlers can make (rules OBS): rash and cautious communicate any site addresses they know; scrupulous crawlers communicate only site addresses they are currently examining.

### 3.1 Symbolic Transitions

A (sound and complete) symbolic transition system for our calculus is simply obtained by taking as symbolic transitions for each context \( C[X] \) all the transitions resulting from the possible (most general) unifications with the left hand sides of each rewrite rule, where \( s, x, \tilde{y}, z, w, a \) are seen as (fresh) variables. More precisely, if \( L[s] \rightarrow_{\alpha} R[s] \) is a rewrite rule (for a suitable label \( \alpha \), possibly the silent one), and \( \theta \) is a most general unifier between \( L[s] \) and \( C[X] \), then we have the transition

\[
C[X] \xrightarrow{\theta(X)} \alpha \theta(R[s])
\]

where \( \theta(X) \) denotes the term substituted for \( X \) by the substitution \( \theta \) (which with a slight abuse of notation can be directly interpreted as a spatial formula) and \( \theta(R[s]) \) inductively applies the substitution \( \theta \) to the variables in \( R[s] \) (recall that \( \theta(L[s]) = \theta(C[X]) \)).

For instance, considering the context \( \text{rash}(a, x, \tilde{y} + z) \mid X \) and the LEARN rule of Fig. 2, we obtain a unifier \( \theta \) that unifies \( X \) with \( \text{link}(x, z) \mid s \). The resulting symbolic transition is the topmost of Fig. 5.

Unification is considered up to associativity, commutativity and identity of parallel composition (see [BBB07]). We also require an exact matching for non-process variables \( x, \tilde{y}, z, w, a \) appearing in the rules, i.e. \( \theta \) must substitute them with actual values.
\[ c(a, x, \tilde{y}) \mid s \rightarrow_{az} c(a, x, \tilde{y}) \mid s \quad \text{with } z \in \tilde{y} + x \]

\[ \text{scrupulous}(a, x, \tilde{y}) \mid s \rightarrow_{ax} \text{scrupulous}(a, x, \tilde{y}) \mid s \]

(i)

(ii)

Figure 4. Textual (i) and graphical (ii) representation of OBS rules where \( c \in \{ \text{rash, cautious} \} \) and \( c' = \text{scrupulous} \).

In the following we shall often focus on the three open processes \( R[X] \), \( K[X] \) and \( S[X] \) defined below:

\[ R[X] \overset{\text{def}}{=} \text{rash}(a, x, \emptyset) \mid X \]
\[ K[X] \overset{\text{def}}{=} \text{cautious}(a, x, \tilde{y}) \mid X \]
\[ S[X] \overset{\text{def}}{=} \text{scrupulous}(a, x, \tilde{y}) \mid X \]

Some of the symbolic transitions for \( R[X] \), \( K[X] \) and \( S[X] \) obtained with this technique are depicted in Fig. 5. In particular, the first transition is obtained from rule LEARN for rash contexts, the second one from rule MOVE, the next two from rule OBS, and so on. Other transitions, needed for determining a complete STS regard the presence of crawlers in holes and are not considered here.

3.2 Abstract Semantics

A natural question that emerges is: under which situation can the different crawlers exhibit essentially the same abstract behaviour? If we consider weak bisimilarities then it is evident that \( \text{rash}(a, x, \tilde{y}) \mid s \) and \( \text{cautious}(a, x, \tilde{y}) \mid s \) are equivalent for any given system \( s \). Indeed even if they follow different movement policies both communicate all the addresses they gather (valid or not). Instead, it is possible to find suitable networks that distinguish scrupulous crawlers from rash and cautious crawlers with the same knowledge. For instance, consider the processes \( \text{rash}(a, x, \emptyset) \mid \text{link}(x, y) \) and \( \text{scrupulous}(a, x, \emptyset) \mid \text{link}(x, y) \). The latter will be able to communicate only the valid site \( x \), while the former can communicate also the missing site \( y \). It follows from the considerations above that \( R[X] \approx_u K[X] \),
sy still, the loose bisimilarity approximates universal closure weak bisimilarity.

Yes, because even if abstract move of K both contexts behave bisimilarly in terms of pages observed. Indeed we know that R transitions relative to the M

However, the situation changes when we consider the coarser equivalence

It follows that when we consider symbolic semantics, the situation is slightly different. In fact, it might be the case that certain silent moves for K[X] require the presence of some links as hypothesis, while this is not the case for R[X]. This is evident when comparing the two transitions relative to the MOVE rules for R[X] and K[X] (from Fig. 5):

R[X] Y→ rash(a, yi, ŷ + x - yi) | Y
K[X] link(yi,z)Y→ cautious(a, yi, ŷ + x - yi) | link(yi, z) | Y (for any z)

It follows that R[X] ≈g K[X] but this is not a desirable result, when considering that both contexts behave bisimilarly in terms of pages observed. Indeed we know that R[X] are equivalent K[X] under universal closure weak bisimilarity.

However, the situation changes when we consider the coarser equivalence ≈l, according to which the symbolic move of K[X] can be simulated by the less constraining (more abstract) move of R[X]. On the other hand, K[X] loosely simulate R[X]? The answer is yes, because even if K[X] has no transition that can be used to simulate the silent step

R[X] Y→ rash(a, yi, ŷ + x - yi) | Y

still, K[X] can just stay idle. Thus while R[X] ≈g K[X] we have R[X] ≈l K[X]. In words, the loose bisimilarity approximates universal closure weak bisimilarity.

The situation is slightly different when considering K[X] and S[X], because S[X] cannot observe yi without first moving to yi, thus requiring the site to be valid, while K[X] can observe it anyway. S[X] can only communicate yi as:

S[X] link(yi,z)Y→ scrupulous(a, yi, ŷ + x - yi) | link(yi, z) | Y Y→aiyi ...
Hence, we have that cautious and scrupulous are not equivalent under loose weak bisimilarity but neither they are under universal weak bisimilarity. Indeed, it can be shown that the behaviour of a cautious crawler subsumes that of a scrupulous crawler by showing that $K[X]$ loosely simulates $S[K[X]]$. In words a context with a cautious crawler behaves like a context with both a cautious and a scrupulous crawler.

4 Typed Symbolic Transition Systems

In the previous section we saw that some crawlers can exhibit different behaviours depending on the network of pages they operate on. Now suppose that we are given some guarantees about the holes that appear in a context, like the fact that $R[X], K[X]$ and $S[X]$ represent valid networks, in the sense that they contain valid site addresses only. Then, we would expect that $R[X], K[X]$ and $S[X]$ are all equivalent as they are all able to observe the same pages in the same order. Indeed, we would like to consider them to be equivalent under a variant of universal weak bisimilarity that takes into account the set of valid holes, rather than any possible system. Unfortunately, we saw in the previous section that our loose equivalence $\approx_l$ does distinguish cautious and rash from scrupulous in the general case.

In this section we propose a technique for stipulating some guarantees over the holes and for manipulating the symbolic transitions under such guarantees in order to account for an equivalence coarser than $\approx_l$. We show the technique at work on our case study and then try to distill some general guidelines for making it applicable in general.

4.1 Typing

The first step is defining a suitable type system for terms. Here we consider a type system based on the page addresses with a particular type that stands for valid networks.

In particular types take the form $T_{\tilde{d},\tilde{p}}$, where $\tilde{d}$ is the set of addresses that must correspond to valid sites in the system and $\tilde{p}$ is the set of addresses that can be pointed by a system to without being necessarily valid within the system itself. We require $\tilde{d} \subseteq \tilde{p}$ since it does make little sense to forbid a site to point to pages it guarantees to exist. We define our types formally as follows.

**Definition 7 (Typed Systems)** A system $s$ has type $T_{\tilde{d},\tilde{p}}$, written $s : T_{\tilde{d},\tilde{p}}$, iff:

- for any $x \in \tilde{d}$ there exists $y$ such that $s$ contains link$(x, y)$;
- for any link $(x, y)$ in $s$ such that $y \notin \tilde{p}$ then it must be the case that link$(y, z)$ is in $s$ for some $z$.

A system $s$ is called valid if $s : T_{\emptyset,\emptyset}$.

In words, if we let $\tilde{r}$ be the set of pages of a system $s$ and call $\tilde{d}$ the subset of $r$ formed by those pages that the system guarantees to exist, then $\tilde{p}$ are pages that can be in $\tilde{d}$ or in any other system. These are pages that the system is allowed to point. Any pointed page outside $\tilde{p}$ must necessarily be provided by the system. In sum, a system cannot point to
a page outside \( \tilde{p} \) non existing in any system but is allowed to point to pages in \( \tilde{d}, \tilde{p} \setminus \tilde{d} \) (possibly outside \( \tilde{s} \)) and a page not in \( \tilde{p} \) but actually part of the system \( \tilde{s} \).

It is easy to see that our types are ordered as stated in the following lemma, that basically expresses that the set of requirements imposed by a type can be relaxed.

**Lemma 8** If \( s : T_{\tilde{d},\tilde{p}} \), then for any \( x, y \) we have \( s : T_{\tilde{d} - x, \tilde{p} + y} \).

Clearly for any system \( s \) we can find sets of addresses \( \tilde{d}, \tilde{p} \) such that \( s \) is of type \( T_{\tilde{d},\tilde{p}} \). In addition, the most precise type we can assign to a system \( s \) is \( T_{\tilde{d},\tilde{p}} \) and for any other type \( T_{\tilde{e},\tilde{q}} \) such that \( s : T_{\tilde{e},\tilde{q}} \) we have \( \tilde{e} \subseteq \tilde{d} \) and \( \tilde{p} \subseteq \tilde{q} \).

It should be evident that the presence of crawlers does not influence the typing of a system, which depends just on links. Moreover, as the rewrite rules cannot change the set of links in the system, it follows that the typing enjoys subject reduction.

**Lemma 9 (Subject Reduction)** If \( s : T_{\tilde{d},\tilde{p}} \) and \( s \to_\alpha s' \), then \( s' : T_{\tilde{d},\tilde{p}} \).

The following theorem is particularly relevant for the application of our technique.

**Theorem 10** For any site \( s \), and site addresses \( \tilde{d}, \tilde{p} \) with \( \tilde{d} \) non empty it holds that \( s : T_{\tilde{d},\tilde{p}} \) iff there are \( x \in \tilde{d}, y \notin \tilde{p}, z \notin \tilde{p} \) such that

- \( s \equiv \text{link}(x,y)|s' \) and \( s' : T_{\tilde{d} - x,\tilde{p}} \);
- or \( s \equiv \text{link}(x,z)|s' \) and \( s' : T_{\tilde{d} - x + z,\tilde{p} + z} \).

We observe that Theorem 10 establishes a logical equivalence between the type predicate \( \L_\cdot : T_{\tilde{d},\tilde{p}} \) and the disjunction of spatial formulas with typed holes, namely \( \text{link}(x,y)\L_\cdot : T_{\tilde{d} - x,\tilde{p}} \) and \( \text{link}(x,z)\L_\cdot : T_{\tilde{d} - x + z,\tilde{p} + z} \) (where address variables are existentially quantified as in Theorem 10).

### 4.2 Decorated Variables

The second step towards our typed STS is the decoration of process variables with typing information, so to consider well-typed contexts only.

A decorated variable takes the form \( X : T_{\tilde{d},\tilde{p}} \). It represents a hole that can be filled only with systems \( s \) of the corresponding types, i.e. such that \( s : T_{\tilde{d},\tilde{p}} \).

**Definition 11 (Typed Contexts)** We say that \( C[X : T_{\tilde{d},\tilde{p}}] \) has type \( T_{\tilde{e},\tilde{q}} \), written \( C[X : T_{\tilde{d},\tilde{p}}] : T_{\tilde{e},\tilde{q}} \) iff for any \( s : T_{\tilde{d},\tilde{p}} \) then \( C[s] : T_{\tilde{e},\tilde{q}} \). A context \( C[X : T_{\tilde{d},\tilde{p}}] \) is called valid if \( C[X : T_{\tilde{d},\tilde{p}}] : T_{\emptyset,\emptyset} \).

The following lemma guarantees that typed contexts can be typed. The notion of most precise type for \( C[X : T_{\tilde{d},\tilde{p}}] \) can then be lifted smoothly.

**Lemma 12** For any \( C[X : T_{\tilde{d},\tilde{p}}] \) we can find a type \( T_{\tilde{e},\tilde{q}} \) such that \( C[X : T_{\tilde{d},\tilde{p}}] : T_{\tilde{e},\tilde{q}} \).

In addition it easy to state how to restrict the type of a hole while preserving the type of its context. This is formalised as follows.
LEMMA 13  For any $z$ and $y \in \bar{p} - \bar{d}$, if $C[X : T_{d,\bar{p}}] : T_{\bar{e},\bar{q}}$ then $C[X : T_{d+z,\bar{p}-y+z}] : T_{\bar{e},\bar{q}}$.

For what concern our contexts $R[X]$, $K[X]$ and $S[X]$, we are interested in considering valid systems w.r.t. the names initially known by the crawlers, hence we can restrict to $R[X : T_{\bar{y}+x,\bar{y}+x}], K[X : T_{\bar{y}+x,\bar{y}+x}]$ and $S[X : T_{\bar{y}+x,\bar{y}+x}]$ which are all of type $T_{\bar{0},\bar{0}}$, i.e. valid.

4.3 Typed Universal Equivalence

The third step is to refine the universal weak bisimilarity $\approx_u$ according to the type decoration of the variables: we say that $C[X : T_{d,\bar{p}}]$ is universally weak bisimilar to $D[X : T_{d,\bar{p}}]$, written $C[X : T_{d,\bar{p}}] \approx_u D[X : T_{d,\bar{p}}]$, if for any $s : T_{d,\bar{p}}$ we have $C[s] \approx D[s]$.

LEMMA 14  For any type $T_{d,\bar{p}}$ and any contexts $C[X]$ and $D[X]$ such that $C[X] \approx_u D[X]$ we have $C[X : T_{d,\bar{p}}] \approx_u D[X : T_{d,\bar{p}}]$.

Note that the overall types of $C[X : T_{d,\bar{p}}]$ and $D[X : T_{d,\bar{p}}]$ are not considered and might be even different.

From the Lemma above, it follows that $R[X : T_{\bar{y}+x,\bar{y}+x}] \approx_u K[X : T_{\bar{y}+x,\bar{y}+x}]$. Moreover, from the notion of typed systems we can expect that $K[X : T_{\bar{y}+x,\bar{y}+x}] \approx_u S[X : T_{\bar{y}+x,\bar{y}+x}]$, but the proof of $K[X : T_{\bar{y}+x,\bar{y}+x}] \approx_u S[X : T_{\bar{y}+x,\bar{y}+x}]$ requires universal closure w.r.t. all systems $s : T_{\bar{y}+x,\bar{y}+x}$.

4.4 Decorated Symbolic Transitions

The last and fourth step is to exploit symbolic equivalences to conclude that $K[X : T_{\bar{y}+x,\bar{y}+x}] \approx_u S[X : T_{\bar{y}+x,\bar{y}+x}]$, i.e. that all three crawlers are equivalent in valid networks. Unfortunately, we have already seen that $K[X] \not\approx_t S[X]$. However, our idea is to exploit the logical equivalence exposed by Theorem 10 to give $S[X]$ the possibility of simulating the unmatched transition (see Section 3.2)

$$K[X] \xrightarrow{\bar{y}_a y_i} K[Y]$$

We notice that all symbolic transitions carry as formula just some spatial information. In general, given the kind of rewrite rules under consideration, such spatial labels can take one of the below forms:

(a) $C[X] \xrightarrow[Y][\alpha]{Y} D[Y]$

(b) $C[X] \xrightarrow[c][Y][\alpha]{Y} D[Y]$

(c) $C[X] \xrightarrow[\text{link}(x,y)]{Y}[\alpha] D[Y]$

(d) $C[X] \xrightarrow[c][\text{link}(x,y)]{Y}[\alpha] D[Y]$

For the forms (a) and (b) (observations $Y$ and $c \mid Y$, respectively) we just keep the decoration assigned in the source, resulting in decorated transitions

(a') $C[X : T_{d,\bar{p}}] \xrightarrow[Y][\alpha]{Y} D[Y : T_{d,\bar{p}}]$

(b') $C[X : T_{d,\bar{p}}] \xrightarrow[c][Y][\alpha]{Y} D[Y : T_{d,\bar{p}}]$

For the forms (c) and (d) (observations $\text{link}(x,y) \mid Y$ and $c \mid \text{link}(x,y) \mid Y$) we exploit Theorem 10 to derive a proper decoration for $Y$. We show what happens for $\text{link}(x,y) \mid Y$, but the other case is entirely analogous.

(c1) $C[X : T_{d,\bar{p}}] \xrightarrow[\text{link}(x,y)]{Y}[\alpha] D[Y : T_{d-x,\bar{p}}]$ if $y \in \bar{p}$

(c2) $C[X : T_{d,\bar{p}}] \xrightarrow[\text{link}(x,y)]{Y}[\alpha] D[Y : T_{d-x+y,\bar{p}+y}]$ if $y \notin \bar{p}$

15
The decorated symbolic transitions for $K[X : T_{\tilde{y} + x, \tilde{y} + x}]$ are in Fig. 6, where we let $z \not\in \tilde{y} + x$.

Note that is not important to decorate $Y$ also in the labels, because they are matched exactly, and given that the decoration of $X$ is known, that of $Y$ follows unambiguously.

We define a new notion of bisimilarity, called decorated loose weak bisimilarity $\approx_d$.

**Definition 15 ($\approx_d$).** Two contexts $C[X : T_{a, b}]$ and $C'[X : T_{a', b}]$ are decorated loose weak bisimilar if it is possible to find a symmetric relation $\tau$ (decorated loose weak bisimulation) such that whenever $C[X : T_{a, b}] \mathrel{\tau} C'[X : T_{a', b}]$ we have that for each transition $C[X : T_{a, b}] \xrightarrow{\phi \alpha} D[Y : T_{c, d}]$ the following holds:

(i) $\phi \neq Y$ and there exists a (weak) decorated symbolic transition $C'[X : T_{a', b}] \xrightarrow{\psi \hat{\alpha}} D'[Z : T_{\hat{c}, \hat{d}}]$ and a spatial formula $\psi'$ such that $\phi = \psi; \psi'$ and $D[Y : T_{c, d}] \mathrel{\tau} D'[\psi']$.

(ii) $\phi = Y$ and

i) either $C'[X : T_{a', b}] \xrightarrow{Y \alpha} D'[Y : T_{d, d}]$ and $D[Y : T_{d, d}] \mathrel{\tau} D'[Y : T_{d, d}]$,

ii) or for any $x \in d$, $y \in \hat{p}$ and $z \not\in \hat{p}$ it holds that:

- $C'[X : T_{a', b}] \xrightarrow{\text{link}(x, y) \alpha} D'[Y : T_{d-x, d}]$ with $D[\text{link}(x, y) \mid Y : T_{d-x, d}] \mathrel{\tau} D'[Y : T_{d-x, d}]$ and

- $C'[X : T_{a', b}] \xrightarrow{\text{link}(x, z) \alpha} D'[Y : T_{d-x+z, d}]$ with $D[\text{link}(x, z) \mid Y : T_{d-x+z, d}] \mathrel{\tau} D'[Y : T_{d-x+z, d}]$ where $\hat{\alpha}$ stands for label $\alpha$ if $\alpha \neq \tau$ and no label otherwise.

Note that we exploit again Theorem 10 to give the possibility of simulating the silent formula $Y$ when the hole has type $T_{a, b}$ by considering separately the two cases that are exposed by Theorem 10.

Let us now return to our goal of showing the equivalence of crawlers in valid networks. This result is obtained by showing that $S[X : T_{\tilde{y} + x, \tilde{y} + x}] \approx_d K[X : T_{\tilde{y} + x, \tilde{y} + x}]$. Indeed, we
protocols. For instance, the sitemap index can be seen at http://www.di.unipi.it/~lafuente/ice08

For the convenience of the reader we have implemented our scenario and made it avail-

4.5 Scenario Implementation

can be simulated by the symbolic moves

\[
\begin{align*}
\mathcal{S}[X : T_{\hat{y} + x\hat{y} + x}] & \xrightarrow{\text{link}(y_i, z)} \text{scrupulous}(a, y_i, \hat{y} + x - y_i) \mid \text{link}(y_i, z) \mid Y : \\
T_{\hat{y} + x - y_i, \hat{y} + x} & \text{for } z \in \hat{y} + x, \text{ and} \\
\mathcal{S}[X : T_{\hat{y} + x\hat{y} + x}] & \xrightarrow{\text{link}(y_i, z)} \text{scrupulous}(a, y_i, \hat{y} + x - y_i) \mid \text{link}(y_i, z) \mid Y : \\
T_{\hat{y} + x - y_i, z, \hat{y} + x + z} & \text{for } z \notin \hat{y} + x.
\end{align*}
\]

In fact, we have also:

\[
\begin{align*}
\mathcal{K}[\text{link}(y_i, z) \mid Y : T_{\hat{y} + x - y_i, \hat{y} + x}] & \approx_d \text{scrupulous}(a, y_i, \hat{y} + x - y_i) \mid \text{link}(y_i, z) \mid Y : \\
& T_{\hat{y} + x - y_i, \hat{y} + x},
\end{align*}
\]

and

\[
\begin{align*}
\mathcal{K}[\text{link}(y_i, z) \mid Y : T_{\hat{y} + x - y_i, z, \hat{y} + x + z}] & \approx_d \text{scrupulous}(a, y_i, \hat{y} + x - y_i) \mid \text{link}(y_i, z) \mid Y : \\
& T_{\hat{y} + x - y_i, z, \hat{y} + x + z} \text{ for } z \notin \hat{y} + x.
\end{align*}
\]

In conclusion all three crawlers are equivalent in valid networks according to the deco-
rated bisimilarity introduced in this paper and this is a nice result in the illustrating scenario
because we know that in a valid network one can freely chose the desired policy with the
guarantee of obtaining the same (observable) behaviour.

4.5 Scenario Implementation

For the convenience of the reader we have implemented our scenario and made it avail-
able at http://www.di.unipi.it/~lafuente/ice08. The web page proposes a simple game where players should find out the crawling policy according to observations only. While deduction is possible in missing sites, in valid sites (as shown in this paper) it is all a matter of guessing and having luck on one’s side.

We remark that the site types \( T_{d, \hat{d}} \) we use are related to typical inclusion and exclusion
protocols. For instance, the sitemap index can be seen as \( d \), i.e. the list of pages whose existence a site guarantees, while the robots.txt file would be pages in \( \hat{d} \setminus d \) that reside on the site, i.e. the list of pages that a site asks not to visit.

In our scenario the motivation under the site asking crawlers not to visit certain pages
is that they are not guaranteed to exist and not because they contain information the site
would prefer not to be crawled, which is the typical intention of robots.txt.

Thus, in our implementation we call this file mightmiss.txt. The polite crawler offered
there behaves like the scrupulous one, but exploits the information in that file to perform
less page existence checks, thus lowering the server’s load.

We believe that one could apply our technique to establish new crawling protocols
or enrich existing ones. For instance, web sites can exhibit their type and, based on it
and desired behaviour, crawlers can decide which policy (e.g. rash, cautious, scrupulous,
polite) to apply in order to perform a polite, efficient and correct crawling activity.
5 Final remarks

We have performed a first step towards the treatment of names and types in STS, our approach to the specification and reasoning of open systems. Our work has been illustrated with a simple nominal calculus, inspired by a web crawling scenario. We have shown how the usual equivalence notion of STS is too fine grained, in the sense that it does distinguish between web crawlers one expects to be equivalent in some networks. We have thus defined a suitable (name-decorated) type system, that allows us, e.g. to constrain an unknown network to be valid, i.e. to not contain any broken link. Based on such types, a new variant of bisimilarity have been defined. According to this notion, all three considered crawlers are equivalent for valid networks.

The presented work should be understood as a first step towards the quite ambitious goal of having more general equivalences, e.g. based on types defined by structural induction.

As future work we plan to generalise our technique to prominent nominal calculi (e.g. the $\pi$-calculus) and to deepen in the relationship with graph transformation approaches dealing with types and unspecified graph parts (e.g. [BLMT08]). More precisely, we would like to focus on service oriented calculi (e.g. [BBNL08]) where the notion of hole and type naturally resemble services and their specifications.

References


