MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

Study of Alternate Distributed Models
Release 1

MIKADO Deliverable 1.3.1

MIKADO Deliverable D1.3.1

Editor : G. BOUDOL (INRIA)

Authors : F. BARBANERA, L. BETTINI, V. BONO, M. BOREALE, G. BOUDOL, M. BUSCEMI, R. DE NICOLA, M. DEZANI-CIANCAGLINI, G.-L. FERRARI, X. GUAN, L. LOPES, A. MATOS, U. MONTANARI, R. PUGLIESE, A. RAVARA, V. SASSONE, J.-B. STEFANI, E. TUOSTO, V. VASCONCELOS, B. VENNERI

Classification : Public
Deliverable no. : D1.3.1
Reference : RR/WP1/5
Date : February 2003

Project funded by the European Community under the "Information Society Technologies" Programme (1998–2002)

Study of Distributed Models

MIKADO WP1, Deliverable D1.3.1

Gérard Boudol

February 10, 2004

Introduction

In this deliverable, we make a review of the work done by the MIKADO partners which is relevant to the WP1, Programming Model. We organize the presentation into three sections. The first one is devoted to the work on models based on the π-calculus, or using a computing style close to this calculus. There are five relevant papers here, of which four are included as part of this deliverable. The second section reports on work dealing with Ambient-like models, with two papers included in this deliverable. This section also mentions one further work, included in a deliverable of WP2. The last section deals with papers that rely on a different programming style, as found in coordination languages or synchronous programming for instance.

Distributed π-Calculi

We begin our review with the work which builds upon Milner's π-calculus. In a joint work [3] with the PROFUNDIS project, M. Boreale (from the Firenze site of MIKADO) investigates the expressiveness issue regarding some fundational calculi. This work shows that, contrarily to common belief, there is no "natural" encoding of the π-calculus into the fusion calculus of Parrow and Victor. Then M. Boreale and his co-authors introduce a new calculus which subsumes both the fusion and the π calculi. This new D-Fusion calculus is in particular more expressive than the asynchronous π-calculus, since it supports a fully abstract encoding of the guarded choice construct. With respect to the fusion calculus, D-Fusion adds the capability of creating fresh names, which is one of the most important features of the π-calculus. This work shows that there is still work to be done about the choice of primitive constructs in such basic calculi, even before considering the distribution aspect.

The work on *lsd*π by A. Matos (from the INRIA Sophia Antipolis site of MIKADO), A. Ravara, V. Vasconcelos and L. Lopes (from the Lisboa site of MIKADO) has been pursued. We recall that *lsd*π adopts the basic assumptions of the Dπ-calculus of Hennessy and Riely (see [7]), namely that communication should be purely local, and that remote communication should result form migration and local communication. This assumption has also been adopted in the proposal made by the MIKADO project for our "core programming model" [4]. In [13], the authors investigate the "network awareness principle" that was stated in [4]. More specifically, they argue that channels, in the sense of the π-calculus, should be considered as physical resources of sites. A consequence is that channel names should only have a local meaning, and that migration should then involve renaming the channels, to maintain the locality assumption. The paper [13] formalizes these ideas, and shows how to extend the usual typing of the π-calculus to the *lsd*π framework. The authors also show that the behaviour of a process in their model does not depend on its location.

The "kell-calculus" [15] has been introduced by J.-B. Stefani (from INRIA Rhône-Alpes), with several motivations. First, it aims at providing a simpler framework than the M-calculus [14] (see also [7]) for the idea of "programmable membranes". This is one of the main features that have been identified for a MIKADO core programming model (see [4]), and this is the one which is investigated also in the deliverable

D1.2.1 [5], introducing a parametric model of migration and mobility. The kell calculus does not retain the complex communication rules of the M-calculus. These are replaced with a simpler mechanism, inspired by the Seal calculus and Boxed Ambients (see [7]), where the input construct has three possible orientations, towards the outside, the inside or a sub-site of a site. The kell calculus retains from the M-calculus a form of the powerful passivation mechanism, by which a whole process is reified and can be manipulated as a datum. In the kell calculus, this is specialized to passivating the content of a domain. This construct allows, as in the M-calculus, to encode various forms of "programmable membranes" and migration behaviours. The kell calculus follows quite closely the principles that have been stated in [4], as guidelines for a MIKADO core programming model. However, it differs from the proposal made in [5] for a migration calculus in one main respect, that we explain now. The kell calculus is powerful enough to encode the migration calculus of [5]: a "membrane" $a\{M\}[P]$ in the latter is actually a pair $(\nu c)a[M \mid M' \mid c[P \mid P']]$ of nested kells, where M' and P' are uniform pieces of code in the kell calculus, encoding the **go_in** and **go_out** primitives of [5]. Then the kell calculus may be regarded as a low-level model, with respect to the migration calculus of [5]. Indeed, implementing the kell calculus would provide the basis for an implementation of the MIKADO migration calculus. However, in the kell calculus one has the ability to bypass the discipline enforced in the migration calculus. Therefore, we expect that, as usual, one has means in the low-level model to break some desirable properties (expressed for instance as equivalences) that the high-level model is supposed to enforce. This is the reason why we have chosen a high-level model for our MIKADO migration calculus.

The work [12] by M. Hennessy and J. Rathke (from the Sussex site of MIKADO, together with N. Yoshida) on Dπ is relevant to WP1, although it is not included in this deliverable (it appears in a deliverable of WP2). It introduces Safe Dπ, a non-trivial extension of Dπ, which explores the idea of a "programmable membrane" which is central to the MIKADO project. As in the migration calculus of [5] (and, for that matter, the M-calculus and the kell calculus), mobility of code is implemented in Safe Dπ using higher-order communication, and the control exercised by a domain is based on sophisticated type checking. More precisely, incoming code must pass through a specific port, the type of which will determine its subsequent behaviour. Then a port, together with its type, may be viewed as a particular instance of the MIKADO's idea of a controlling membrane.

The work [10] by X. Guan (from INRIA Sophia Antipolis) also deals with a distributed version of the π-calculus where, by contrast with Dπ, the sites are organized in a hierarchy of nested domains. The main novelty of the proposed calculus is that π-calculus channels are identified with domains (as in the Mobile Ambient calculus, see [7]). Then computing in this model amounts to communicate addresses, which are compound names indicating a path to some location. This has some similarities with channel names in *lsd*π [13], with the difference that the latter is a "flat" model, with no nesting of domains.

Ambient-like Models

An extended version of the work of M. Coppo (from Torino, as part of the Firenze site of MIKADO), M. Dezani-Ciancaglini and E. Giovannetti (both supported by the DART project) on M^3, a variant of the Mobile Ambient model, has been written (and submitted) this year [8]. The main difference between the M^3 model and the Mobile Ambients is that it replaces the problematic **open** primitive of the latter by the migration construct of Dπ, allowing for the mobility of "naked" (that is, not wrapped up into an Ambient) processes. The paper [8] is mainly concerned with types, and this is why, although relevant to the WP1 of MIKADO, it is not included in this deliverable (but it is part of a deliverable of WP2).

In [11], X. Guan introduces another variant, or more precisely a sub-calculus of the (Safe) Mobile Ambients calculus (see[7]). The starting point of the study is the encoding of the π-calculus into the pure Mobile Ambient calculus: X. Guan identifies a restricted syntax, the main categories of which are the single-threaded and immobile Ambients, which defines a model in which one can encode both the π and Dπ calculi quite easily and naturally. By contrast with the Mobile Ambients model, the resulting "wagon calculus" follows the "locality principle" stated in [4] as a requirement for our migration model.

The work by F. Barbanera, M. Bugliesi (supported by the MYTHS project), M. Dezani-Ciancaglini (sup-

ported by DART) and V. Sassone (from the Sussex site of MIKADO) on resource control in the Mobile Ambients model has been extended into a paper submitted to a journal [1]. The main innovation is the introduction, in the Mobile Ambients framework, of an explicit construct representing a resource unit, the "slot". The resulting calculus, named BOCA (for "bounded capabilities"), features capabilities for resource control, which transfer spaces between sibling ambients and from parent to child, as well as capabilities for ambient migration, which represent an abstract mechanism of resource negotiation between travelling agent and its source and destination environments. A fundamental ingredient of the calculus is a primitive which consumes space to activate processes. The combination of these features makes BOCA a suitable formalism to study the role of resource consumption, and the corresponding safety guarantees, in the dynamics of mobile systems. The important notion of private resource has guided the formulation of the refined version of the calculus, featuring named resources.

Other Models

In the MIKADO project, we also develop a model for distributed and mobile computations which is based on the concept of a tuple space of the LINDA language, and the related constructs for manipulating it, that is for reading and writing in the tuple space. This is the KLAIM programming language, which has been designed and implemented by the group of R. De Nicola at Firenze. There have been several developments related to this language during the last year. The most relevant one for WP1 of MIKADO is the extension of the language with some object oriented features, and more precisely mixin-based programming. L. Bettini, V. Bono and B. Venneri (from Firenze and Torino, supported by MIKADO and DART) have proposed in [2] an integration of objects into KLAIM which is quite simple: any value of the object-oriented sub-language (that is, an object, or a class, or a mixin) is allowed to be an element of a tuple in the tuple space. Moreover, executing a KLAIM process can be prefixed by some object-oriented computations. This work is relevant to the task of combining standard programming paradigms with distributed and mobile programming.

Another relevant work concerning KLAIM is presented in the paper [9] by R. De Nicola et al. (this is a joint work with the AGILE and PROFUNDIS projects). The motivation for this work is to provide a formal framework for a programmable quality of service. More specifically, the paper [9] introduces a refinement of the KLAIM process calculus, where the connection operations between nodes involve a quantitative information, providing a measure of the (required or provided) quality of a connection. The semantics is given by means of a labelled transition system, as usual, and also by a translation into a calculus of graph rewriting, taking the QoS attributes into account.

The last work which is included in this deliverable is a proposal by G. Boudol of a core programming language for global computing [6], based on some of the ideas of synchronous programming languages. This is intended to address the unreliable character of accessing resources in a global computing context. The synchronous programming style provides primitives for suspending a program – for lack of an event – and preempting – that is, aborting – a program when some signal is emitted. This is embedded in a fairly standard programming style, namely a core ML language supporting functional and imperative computations. In this model, a mobile agent is a thread with a memory, and a domain is simply an abstract machine running a queue of threads.

Conclusion

The classification we have adopted in presenting the work in WP1 is somewhat arbitrary. For instance, the kell calculus subsumes in a sense the Mobile Ambients model, while M^3 uses the migration construct of $D\pi$. One may also think that the LINDA model is not very different from the one of the asynchronous π-calculus: a tuple space could be regarded as a channel, with the read operation as an atomic combination of π-input and matching. More importantly, almost all the works we reported on deal with some notion of a domain (or site, or locality). As we have noted, some of them follow the principles stated in [4] as requirements for a MIKADO core programming model. Moreover, some of them are closely related to the MIKADO proposal

of a migration calculus made in [5]. Since the latter has been designed as a parametric model, that could be combined with various programming styles, one should in principle be able to merge this proposal with most of the models we have presented here.

References

[1] F. BARBANERA, M. BUGLIESI, M. DEZANI-CIANCAGLINI, V. SASSONE, *A calculus of bounded capacities*, in this deliverable, and available from the MIKADO's web site (2003).

[2] L. BETTINI, V. BONO, B. VENNERI, *O'Klaim: a coordination language with mobile mixins*, in this deliverable, and available from the MIKADO's web site. Accepted for presentation at the COORDINATION'04 Conference (2003).

[3] M. BOREALE, M. BUSCEMI, U. MONTANARI, *D-Fusion: a distinctive fusion calculus*, in this deliverable, and available from the MIKADO's web site (2003).

[4] G. BOUDOL, *Core programming model, release 0*, deliverable D1.2.0, available from the MIKADO's web site (2002).

[5] G. BOUDOL, *A parametric model of migration and mobility, release 1*, deliverable D1.2.1, available from the MIKADO's web site (2003).

[6] G. BOUDOL, *ULM, A core programming model for global computing*, in this deliverable, and available from the MIKADO's web site. An extended abstract has been accepted for presentation at the ESOP'04 Conference (2003).

[7] G. BOUDOL, I. CASTELLANI, F. GERMAIN, M. LACOSTE, *Analysis of formal models of distribution and mobility: state of the art*, deliverable D1.1.1, available from the MIKADO's web site (2002).

[8] M. COPPO, M. DEZANI-CIANCAGLINI, E. GIOVANNETTI, *The M3 paradigm: types and type inference for Ambient and process mobility*, available from the MIKADO's web site (2004).

[9] R. DE NICOLA, G.-L. FERRARI, U. MONTANARI, R. PUGLIESE, E. TUOSTO, *A formal basis for reasoning on programmable QoS*, International Symposium on Verification, LNCS 2772. In this deliverable, and available from the MIKADO's web site (2003).

[10] X. GUAN, *Towards a tree of channels*, Foundations of Global Computing Workshop, ENTCS Vol. 85. In this deliverable, and available from the MIKADO's web site (2003).

[11] X. GUAN, *From Ambients to a routing calculus*, in this deliverable (2003).

[12] M. HENNESSY, J. RATHKE, N. YOSHIDA, *SafeDpi: a language for controlling mobile code*, Tech. Rep. No. 02, University of Sussex, available from the MIKADO's web site. An extended abstract has been accepted for presentation at the FOSSACS'04 Conference (2003).

[13] A. RAVARA, A. MATOS, V. VASCONCELOS, L. LOPES, *Lexically scoping distribution: what you see is what you get*, Foundations of Global Computing Workshop, ENTCS Vol. 85. In this deliverable, and available from the MIKADO's web site (2003).

[14] A. SCHMITT, J.-B. STEFANI, *The M-calculus: a higher-order distributed process calculus*, POPL'03, available from the MIKADO's web site (2003).

[15] J.-B. STEFANI, *A calculus of kells*, Foundations of Global Computing Workshop, ENTCS Vol. 85. In this deliverable, and available from the MIKADO's web site (2003).

D-Fusion: a Distinctive Fusion Calculus

Michele Boreale[1], Maria Grazia Buscemi[2], and Ugo Montanari[2]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy.
[2] Dipartimento di Informatica, Università di Pisa, Italy.
boreale@dsi.unifi.it {buscemi,ugo}@di.unipi.it

Abstract. Fusion calculus is commonly regarded as a generalisation of pi-calculus. Actually, we prove that there is no uniform fully abstract embedding of pi-calculus into Fusion. This fact motivates the introduction of a new calculus, D-Fusion, with two binders, λ and ν. We show that D-Fusion is strictly more expressive than both pi-calculus and Fusion. The expressiveness gap is further clarified by the existence of a fully abstract encoding of mixed guarded choice into the choice-free fragment of D-Fusion.

1 Introduction

A recent trend in the design of certain distributed applications like Web services [13] or business-to-business systems [5] based on XML sees the emergence of a message-passing programming style. Languages like Highwire [4] provide, in a concurrency setting, sophisticated data structures; these allow programmers to describe and manipulate complex messages and interaction patterns. If one looks for 'foundational' counterparts of these programming languages, both the pi-calculus [6, 7] and the Fusion calculus [11], seem very promising candidates. Both of them, indeed, convey the idea of message-passing in a distilled form, and come equipped with a rich and elegant meta-theory.

The main novelty of Fusion when compared to the pi-calculus is the introduction of *fusions*. A fusion is a name equivalence that, when applied onto a term, has the effect of a (possibly non-injective) name substitution. Fusions are ideal for representing, e.g., forwarders for objects that migrate among locations [2], or forms of pattern matching between pairs of messages [4]. Computationally, a fusion is generated as a result of a synchronisation between two complementary actions, and it is atomically propagated to processes running in parallel with the active one. This happens in much the same way as, in logic programming, term substitutions resulting from a unification step on a subgoal can be forced on the other subgoals.

If compared to pi-calculus name-passing, fusions enable a more general name matching mechanism during synchronisation. However, differently from the pi-calculus, the binding mechanism of Fusion just ignores the issue of unicity of newly generated names. One of the goals of this paper is to show that this fact severely limits the expressiveness of Fusion. Overcoming this limitation calls for co-existence of two name binders, λ and ν: the former analogous to the only binder of Fusion, and the latter imposing unicity. This implies combining, in a consistent way, the somehow conflicting mechanisms of fusions and *distinctions* à la open pi-calculus [12]. The resulting *distinctive* Fusion calculus, or *D-Fusion*, is at least as expressive as pi-calculus and Fusion separately, and in fact, we strongly argue, *more* expressive than both. A more precise account of our work follows.

The binding mechanism of pi-calculus generalises that of λ-calculus in several ways. Input prefix binds like λ, and name passing takes place in pi-calculus in a way typical of functional

Research supported in part by FET Global projects *PROFUNDIS* and *MIKADO*.

programming, i.e., formal names are assigned their actual counterpart. The *restriction* binder ν, however, is very different from λ, as a restricted name can be exported (extruded), with the guarantee that it will never be identified to anything else. Open pi-calculus [12] takes an important step forward over the original proposal, allowing for input actions whose formal parameters can be instantiated, hence 'fused', in a 'lazy' fashion – at any time when needed for synchronisation. In order to preserve the intended semantics of restriction, however, newly extruded names are kept distinct from 'old' names, using special relations called distinctions.

Fusion calculus is presented in [11] as a more uniform and more expressive version of pi-calculus. The main idea is to decompose input prefix $a(x)$ into a binder (x) and a prefix $a\langle x\rangle$. In the polyadic case, matching between the input list and the output list of variables induces name unification, i.e. a fusion. The latter is propagated across processes, but one or more binders can be used to control the scope (i.e. propagation) of the fusion. Thus one achieves both a perfect symmetry between input and output and a more general name passing mechanism.

At first sight, Fusion is more general than pi-calculus. And, indeed, in [11] it is stated that the pi-calculus transition system can be embedded into Fusion's, provided that one identifies restriction (νx) with the (x) binder of Fusion.

Our first move is to argue that this embedding breaks down if comparing the two calculi on the basis of abstract semantics. We prove that no 'uniform' encoding exists of pi-calculus into Fusion that preserves any 'reasonable' behavioural equivalence (at least as fine as trace equivalence). Here 'uniform' means homomorphic with respect to parallel composition and name substitution, and mapping (νx) to (x), and preserving (a subset of) weak traces. As hinted before, the ultimate reason is that in Fusion all names are like logical variables, i.e., their unification always succeeds.

The above failure motivates the introduction of a new calculus, D-Fusion, with two binders, λ and ν: the first generalises input prefix, and the second models restriction. Also, any issue of symmetry between input and output is preempted, since we have just one kind of prefix (no polarisation); polarised prefixes can be easily encoded, though. As in open pi-calculus, in D-Fusion distinctions are employed to constrain possible fusions. In logical terms, this corresponds to allow unification not only among variables, but also among variables and dynamically generated constants (that is, ν-extruded names). However, unification fails whenever one tries to identify two distinct constants, or to identify a 'recent' constant with an 'old' variable. We show that the additional expressive power achieved in this way is relevant. Both pi-calculus and Fusion can be uniformly encoded into D-Fusion. Moreover, the combined mechanism of restriction and unification yields additional expressive power: it allows to express a form of pattern matching which cannot be expressed in the other two calculi. As a consequence, we prove, D-Fusion cannot be uniformly encoded neither into Fusion, nor into pi-calculus.

Next, the gap between D-Fusion and Fusion/pi-calculus is explored from a more concrete perspective. First, we exhibit a simple security protocol and a related *correlation* property that are readily translated into D-Fusion. The property breaks down if uniformly translating the protocol into Fusion. The failure is illuminating: in Fusion, one has no way of declaring unique fresh names to correlate different messages of the protocol.

Palamidessi has shown [9, 10] that nondeterministic *guarded choice* cannot be simulated in the choice ($+$) -free pi-calculus in a fully abstract way, while preserving any 'reasonable' semantics. This is due to the impossibility of atomically performing, in the absence of $+$, an external synchronisation and an internal exclusive choice among a number of alternatives. We prove that in D-fusion, under mild typing assumptions, guarded choice can actually be simulated in a fully abstract way in the choice-free fragment. The encoding preserves a reasonable semantics in the sense of [9, 10]. Informally, branches of a choice are represented as concurrent

processes. Synchronisation is performed in the ordinary way, but it forces a fusion between a global λ-name and a ν-name local to the chosen branch. Excluded branches are atomically inhibited, since any progress would lead them to fusing two distinct ν-names.

The rest of the paper is organised as follows. Section 2 contains the proof that the pi-calculus cannot be uniformly encoded into Fusion. In Section 3 we introduce the D-Fusion calculus and in Section 4 we give a notion of bisimulation for the calculus. In Section 5 we show that the D-Fusion calculus is more expressive than both pi-calculus and Fusion. We further explore this expressiveness gap in Sections 6 by means of an example and in Section 7 by encoding the mixed guarded choice into the choice-free calculus. Section 8 contains a brief overview of some related work and a few concluding remarks. Appendices A and B contain a few most technical definitions.

2 Fusion and Pi

The aim of this section is illustrating the difference between pi-calculus and Fusion, and to show that the former cannot be uniformly encoded in the latter.

The crucial difference between the pi-calculus and Fusion shows up in synchronisations: in Fusion, the effect of a synchronisation is not necessarily local, and is regulated by the scope of the binder (x). For example, an interaction between $\overline{u}v.P$ and $ux.Q$ will result in a fusion of v and x. This fusion will also affect any further process R running in parallel, as illustrated by the example below:

$$R \,|\, \overline{u}v.P \,|\, ux.Q \xrightarrow{\{x=v\}} (R \,|\, P \,|\, Q)[v/x].$$

The binding operator (x) can be used to limit the scope of the fusion, e.g.:

$$R \,|\, (x)\left(\overline{u}v.P \,|\, ux.Q\right) \xrightarrow{\tau} R \,|\, (P \,|\, Q)[v/x].$$

where τ denotes the identity fusion. For a full treatment of pi-calculus and Fusion we refer to [7] and to [11], respectively.

Below, we show that there is no 'reasonably simple' encoding of the pi-calculus Π into Fusion $\mathcal{F}$. We focus on encodings $[\![\cdot]\!]$ that have certain compositional properties and preserve (a subset of) weak traces. As to the latter, we shall only require that moves of P are reflected in $[\![P]\!]$, not vice-versa. We also implicitly require that the encoding preserves arity of I/O actions (the length of tuples carried on each channel). This is sometimes not the case for process calculi encodings; however, it is easy to relax this condition by allowing names of $[\![P]\!]$ to carry *longer* tuples. We stick to the simpler correspondence just for notational convenience. Note that the encoding presented in [11] does satisfy our criterion. We shall assume here the standard pi-calculus late operational semantics [7] and, for the purpose of our comparison, we shall identify the late input pi-action $a(\widetilde{x})$ with the Fusion input action $(\widetilde{x})\,a\widetilde{x}$.

Definition 1. *A translation* $[\![\cdot]\!] : \Pi \to \mathcal{F}$ *is* uniform *if for each* $P, Q \in \Pi$ *it holds that:*

- *for each trace of actions s not containing bound outputs,* $P \stackrel{s}{\Longrightarrow}$ *implies* $[\![P]\!] \stackrel{s}{\Longrightarrow}$;
- $[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$;
- *for each y,* $[\![(\nu y)\,P]\!] = (y)[\![P]\!]$;
- *for each substitution* σ, $[\![P\sigma]\!] = [\![P]\!]\sigma$.

The next proposition generalises an example from [11]. Below, we fix an arbitrary Π-equivalence included in trace equivalence, $\sim_\Pi$, and an arbitrary $\mathcal{F}$-equivalence which is included in trace equivalence *and* is preserved by parallel composition, $\sim_\mathcal{F}$ (like, e.g., hyper-equivalence of [11]).

Proposition 1. *There is no uniform translation* $[\![\cdot]\!] : \Pi \to \mathcal{F}$ *such that for each* $P, Q \in \Pi$*:*

$$P \sim_\Pi Q \text{ implies } [\![P]\!] \sim_{\mathcal{F}} [\![Q]\!].$$

PROOF: Suppose that there exists such a translation $[\![\cdot]\!]$. Let P and Q be the following two pi-agents:

$$P \;=\; (\nu u, v)\,(\,\overline{a}\langle u, v\rangle \,|\, \overline{u}|v.\overline{w}) \qquad Q \;=\; (\nu u, v)\,\big(\,\overline{a}\langle u, v\rangle \,|\, (\,\overline{u}.(v.\overline{w}) + v.(\overline{u}|\overline{w})\,)\big).$$

Obviously, $P \sim_\Pi Q$ (e.g. they are strongly late bisimilar). Suppose $[\![P]\!] \sim_{\mathcal{F}} [\![Q]\!]$. Let $R = a(x,y).(\overline{c}x|cy)$ and A and B be as follows:

$$\begin{array}{l} A = [\![P]\!]|R = (u,v)([\![\overline{a}\langle u,v\rangle]\!]\,|\,[\![\overline{u}]\!]\,|\,[\![v.\overline{w}]\!])\,|\,R \\ B = [\![Q]\!]|R = (u,v)([\![\overline{a}\langle u,v\rangle]\!]\,|\,[\![\overline{u}.(v.\overline{w}) + v.(\overline{u}|\overline{w})]\!])\,|\,R. \end{array}$$

Since $\sim_{\mathcal{F}}$ is preserved by $|$, A and B are $\sim_{\mathcal{F}}$-equivalent. By uniformity of the encoding, it is easy to check that $A \stackrel{\overline{w}}{\Longrightarrow}$. On the other hand, a careful case analysis shows that $B \stackrel{\overline{w}}{\not\Longrightarrow}$. This is a contradiction. □

3 The Distinctive Fusion Calculus, D-Fusion

Syntax We consider a countable set of names $\mathcal{N}$ ranged over by $a, b, \ldots, u, v, \ldots, z$. We write $\widetilde{x}$ for a finite tuple $x_1, \ldots, x_n$ of names. The set $\mathcal{DF}$ of D-Fusion *processes*, ranged over by $P, Q, \ldots$, is defined by the syntax:

$$P ::= \mathbf{0} \;\big|\; \alpha.P \;\big|\; P|P \;\big|\; P+P \;\big|\; [x=y]P \;\big|\; !P \;\big|\; \lambda x P \;\big|\; (\nu x)\,P$$

where *prefixes* α are defined as $\alpha ::= a\widetilde{v}$. The occurrences of x in $\lambda x P$ and $(\nu x)\,P$ are *bound*, thus notions of *free names* and *bound names* of a process P arise as expected and are denoted by $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$, respectively. The notion of *alpha-equivalence* also arises as expected. In the rest of the paper we will identify alpha-equivalent processes.

Note that we consider one kind of prefix, thus ignoring polarities. However, a sub-calculus with polarities can be easily retrieved, as shown at the end of this section.

The main difference from Fusion is the presence of two distinct binding constructs, λ and ν. The λ-abstraction operator corresponds to the binding construct of Fusion and generalises input binding of the pi-calculus. The restriction operator (ν) corresponds to the analogous operator of the pi-calculus: it allows a process to create a fresh, new name that will be kept distinct from other names.

Operational Semantics For R a binary relation over $\mathcal{N}$, let $R^\star$ denote the reflexive, symmetric and transitive closure of R with respect to $\mathcal{N}$. We use σ, σ' to range over substitutions, i.e. finite partial functions from $\mathcal{N}$ onto $\mathcal{N}$. Domain and co-domain of σ, denoted $\mathrm{dom}(\sigma)$, $\mathrm{cod}(\sigma)$ are defined as expected. We denote by $t\sigma$ the result of applying σ onto a term t. Given a set/tuple of names $\widetilde{x}$, we define $\sigma_{|\widetilde{x}}$ as $\sigma \cap (\widetilde{x} \times \mathcal{N})$, and $\sigma_{-\widetilde{x}}$ as $\sigma - (\widetilde{x} \times \mathcal{N})$.

Definition 2 (fusions). *We let* $\phi, \chi, \ldots$ *range over* fusions*, that is total equivalence relations on* $\mathcal{N}$ *with only finitely many non-singleton equivalence classes. We let:*

- $\mathrm{n}(\phi)$ *denote* $\{x : x \,\phi\, y \text{ for some } y \neq x\}$*;*
- τ *denote the identity fusion (i.e.,* $\mathrm{n}(\tau) = \emptyset$*);*

- $\phi + \psi$ *denote the finest fusion which is coarser than* ϕ *and* ψ*, that is* $(\phi \cup \psi)^\star$*;*
- ϕ_{-z} *denote* $(\phi - (\{z\} \times \mathcal{N} \cup \mathcal{N} \times \{z\}))^\star$*;*
- $\{x = y\}$ *denote* $\{(x,y)\}^\star$*;*
- $\phi[x]$ *denote the equivalence class of* x *in* ϕ*.*

A fusion ϕ arises as the result of equating two lists of names in a synchronisation. In general, names in the original lists might be either λ-abstracted or ν-extruded, or free. Informally, the effect of ϕ should be that of a substitution, mapping equivalent names onto representatives. However, in the presence of ν-, λ-abstracted and free ϕ-equivalent names, some care is needed to ensure that the representative be chosen appropriately. This concept is made precise in the next definition.

Definition 3 (induced substitutions). *Let* $\widetilde{x}$ *and* $\widetilde{k}$ *be two disjoint tuples of names and* ϕ *be a fusion such that for any two distinct* $k_1, k_2 \in \widetilde{k}$*, not* $(k_1 \,\phi\, k_2)$*. We say that a substitution* σ *is* induced by $\lambda\widetilde{x}$*,* $(\nu\widetilde{k})$ and ϕ*, if* $\mathrm{dom}(\sigma) = \mathrm{n}(\phi)$ *and for each equivalence class* E *of* ϕ*,* σ *maps all names in* E *onto one and the same name* $y \in E$ *such that:*

- *if* $\widetilde{k} \cap E = \{h\}$ *then* $y = h$*;*
- *if* $\widetilde{k} \cap E = \emptyset$ *and* $E - \widetilde{x} \neq \emptyset$ *then* $y \in E - \widetilde{x}$*.*

For example, suppose ϕ has only two non-singleton equivalence classes $\{x, w, k\}$ and $\{y, h\}$, then a substitution σ induced by λxy, (νk) and ϕ is the one mapping x, w, k to k and y, h to h. It is important to stress that, by definition, no induced σ exists if for distinct $k_1, k_2 \in \widetilde{k}$ it holds $k_1 \,\phi\, k_2$. Also note that in general there may be more than one induced substitution. E.g. $\{a = b\}$ (with empty $\lambda\widetilde{x}$ and $(\nu\widetilde{k})$) induces both $[a/b]$ and $[b/a]$. We introduce the labelled transition system for D-Fusion.

Definition 4 (labelled transition system). *The transition relation* $P \xrightarrow{\mu} Q$*, for* μ *a label of the form* $(\nu\widetilde{x})\,\lambda\widetilde{y}\,a\widetilde{v}$ (action) *or of the form* $(\nu\widetilde{x})\,\phi$ (effect)*, is defined in Table 1.*

Some notations for actions and effects. The bound names of μ are written $\mathrm{bn}(\mu)$ and are defined as expected, while $\mathrm{subj}(\mu)$ and $\mathrm{obj}(\mu)$ denote the subject and object part of μ, if μ is an action, otherwise they both denote conventional value '$-$'. Moreover, $\mathrm{n}(\mu)$ denotes all names in μ. We use abbreviations such as $\mathrm{n}(\phi, \mu)$ to denote $\mathrm{n}(\phi) \cup \mathrm{n}(\mu)$ and $(\nu z)\,\mu$ for $(\nu\widetilde{x}z)\,\phi$, if $\mu = (\nu\widetilde{x})\,\phi$. Furthermore, we shall identify actions and effects up to reordering of the tuple $\widetilde{x}$ in $(\nu\widetilde{x})$ and $\lambda\widetilde{x}$.

The rules in Table 1 deserve some explanation. As mentioned, we have two kinds of labels, actions and effects. Apart from the absence of polarities, *actions* are governed by rules similar to those found in pi-calculus. The main difference is that on the same action one can find both ν- and λ-extruded names. On the other hand, *effects* are similar to those found in Fusion. A major difference is that our effects can also ν-extrude names. An effect $(\nu\widetilde{x})\,\phi$ can be created as a result of a communication (rule Com), and be propagated across parallel components, until a λ that binds a fused name z is encountered (rule λ-Open$_f$). At that point, the corresponding substitution $\sigma_{|z}$ is applied onto the target process and z is discarded from the fusion (the result is ϕ_{-z}). Any ν-extruded name $w \in \widetilde{x}$ which is equivalent to z must be ν-closed back, provided that no free name is in that equivalence class. In rule Com, handling of λ- and ν-extruded names can be explained similarly, but note that those λ-extruded names $\widetilde{z}$ that are not instantiated (by $\sigma_{|\widetilde{y}'\widetilde{y}''}$) must be λ-closed (this happens when in some equivalence class all names are λ-abstracted). Finally, note that the side condition '$\phi[z] \cap \widetilde{x} = \emptyset$' in rule ν-Open prevent effects from equating two distinct ν-extruded names.

$$\text{(Act)}\ \ \alpha.P \xrightarrow{\alpha} P \qquad \text{(Match)}\ \ \frac{P \xrightarrow{\mu} Q}{[a=a]P \xrightarrow{\mu} Q} \qquad \text{(Sum)}\ \ \frac{P_1 \xrightarrow{\mu} Q}{P_1 + P_2 \xrightarrow{\mu} Q}$$

$$\text{(ν-Pass)}\ \ \frac{P \xrightarrow{\mu} Q}{(\nu z)P \xrightarrow{\mu} (\nu z)Q}\ \ z \notin \mathrm{n}(\mu) \qquad \text{(λ-Pass)}\ \ \frac{P \xrightarrow{\mu} Q}{\lambda z P \xrightarrow{\mu} \lambda z Q}\ \ z \notin \mathrm{n}(\mu)$$

$$\text{(ν-Open)}\ \ \frac{P \xrightarrow{\mu} Q}{(\nu z)P \xrightarrow{(\nu z)\mu} Q}\ \ \begin{cases} z \in \mathrm{n}(\mu) \\ \mu \text{ an action implies } z \neq \mathrm{subj}(\mu) \\ \mu = (\nu \widetilde{x})\,\phi \text{ implies } \phi[z] \cap \widetilde{x} = \emptyset \end{cases}$$

$$\text{(λ-Open$_a$)}\ \ \frac{P \xrightarrow{(\nu \widetilde{x})\lambda \widetilde{y} a \widetilde{v}} Q}{\lambda z P \xrightarrow{(\nu \widetilde{x})\lambda \widetilde{y} z a \widetilde{v}} Q}\ \ z \in \widetilde{v} - (\{a\} \cup \widetilde{x}\widetilde{y})$$

$$\text{(λ-Open$_f$)}\ \ \frac{P \xrightarrow{(\nu \widetilde{x})\phi} Q}{\lambda z P \xrightarrow{(\nu \widetilde{x}')\phi_{-z}} (\nu \hat{w})\,(Q\sigma_{|z})}\ \ \begin{cases} z \in \mathrm{n}(\phi) \\ \sigma \text{ induced by } \lambda z,\ (\nu \widetilde{x}) \text{ and } \phi \\ \widetilde{x}' = \widetilde{x} \cap \mathrm{n}(\phi_{-z}) \\ \hat{w} = \widetilde{x} - \widetilde{x}' \end{cases}$$

$$\text{(Com)}\ \ \frac{P_1 \xrightarrow{(\nu \widetilde{x}')\lambda \widetilde{y}' a \widetilde{u}} Q_1 \quad P_2 \xrightarrow{(\nu \widetilde{x}'')\lambda \widetilde{y}'' a \widetilde{v}} Q_2}{P_1 | P_2 \xrightarrow{(\nu \widetilde{x})\,\{\widetilde{v}=\widetilde{u}\}_{-\widetilde{y}'\widetilde{y}''}} (\nu \widetilde{w})\lambda \widetilde{z}((Q_1|Q_2)\sigma_{|\widetilde{y}'\widetilde{y}''})}\ \ \begin{cases} |\widetilde{u}| = |\widetilde{v}| \\ \sigma \text{ induced by } \lambda \widetilde{y}'\widetilde{y}'',\ (\nu \widetilde{x}'\widetilde{x}'') \text{ and } \{\widetilde{v} = \widetilde{u}\} \\ \widetilde{z} = \widetilde{y}'\widetilde{y}'' \cap \mathrm{cod}(\sigma_{|\widetilde{y}'\widetilde{y}''}) \\ \widetilde{x} = \widetilde{x}'\widetilde{x}'' \cap \mathrm{n}(\{\widetilde{v} = \widetilde{u}\}_{-\widetilde{y}'\widetilde{y}''}) \\ \widetilde{w} = \widetilde{x}'\widetilde{x}'' - \widetilde{x} \end{cases}$$

$$\text{(Par)}\ \ \frac{P \xrightarrow{\mu} Q}{P|R \xrightarrow{\mu} Q|R} \qquad \text{(Rep)}\ \ \frac{P|!P \xrightarrow{\mu} Q}{!P \xrightarrow{\mu} Q}$$

Symmetric rules for (Sum) and (Par) are not shown. Usual conventions about freshness of bound names apply.

Table 1. Actions and effects transitions in D-Fusion.

Let us illustrate the rules with some examples. We shall write $\{\widetilde{x}=\widetilde{y}\}.P$ for $(\nu c)\,(c\widetilde{x}|c\widetilde{y}.P)$ (for a fresh name c).

Example 1.

1. Let $P=(\nu x)\,(\nu c)\,(cx.P_1\,|\,cy.P_2)$. The interaction between $cx.P_1$ and $cy.P_2$ will result into a fusion $\{x=y\}$, that causes x to be extruded:

$$(\nu x)\,(\nu c)\,(cx.P_1\,|\,cy.P_2)\xrightarrow{(\nu x)\,\{x=y\}}(\nu c)\,(P_1\,|\,P_2).$$

   Now consider $Q=\lambda y\,P$. The effect of λ-abstracting y in P is that of removing ϕ and getting the induced substitution $[x/y]$ applied onto the continuation:

$$\lambda y\,(\nu x)\,(\nu c)\,(cx.P_1\,|\,cy.P_2)\xrightarrow{\tau}(\nu x)\,((\nu c)\,(P_1\,|\,P_2)[x/y]).$$

   Note that our operational rules are preserved by a form of structural equivalence. For instance, $(\nu x)\,\lambda y\,(\nu c)\,(cx.P_1\,|\,cy.P_2)$ has the same τ-transition as Q.
2. Another example is as follows (P below is some continuation):

$$\lambda z\,(\nu w)\,\{z,a=w,b\}.P\xrightarrow{\{a=b\}}(\nu w)\,P[w/z],\quad\text{but}\quad\lambda z\,(\nu w)\,\{z,z=w,b\}.P\xrightarrow{(\nu w)\,\{w=b\}}P[w/z].$$

Encoding I/O polarities We can encode polarities as follows:

$$\overline{c}\langle\widetilde{v}\rangle.P=(\nu x)\,\lambda y\,c\widetilde{v}xy.P\qquad c\langle\widetilde{v}\rangle.P=(\nu x)\,\lambda y\,c\widetilde{v}yx.P$$

for some chosen fresh x and y. The position of the ν-name x forbids fusions between actions with the same polarity and, hence, communication. For instance, the process $\overline{c}\langle\widetilde{v}\rangle.P|\overline{c}\langle\widetilde{u}\rangle.Q$ has no τ-transition, since the latter would force the fusion of two distinct ν-names, which is forbidden by the operational rules. We denote by $\mathcal{DF}^{\mathrm{p}}$, *polarised D-Fusion*, the subset of $\mathcal{DF}$ in which every prefix can be interpreted as an input or output, in the above sense.

4 Bisimulation

Like in the case of Fusion, a 'natural' semantics of D-Fusion is required to be closed under substitutions. However, one should be careful in keeping ν-extruded names distinct from others (λ-extruded and free). To this purpose, we introduce below a concept of *distinction*, akin to [12]. Distinctions keep track of ν-extruded names, for which fusions should be forbidden. Differently from [12], however, our distinctions keep track of λ-extruded names as well, and of the order of their extrusion relative to ν-extruded names. The idea is to treat ν-extruded names as constants, while allowing a fusion between two λ-extruded names, or a fusion of a 'recent' λ-extruded name to an 'old' ν-extruded name. Technically, we find it convenient to model distinctions as sequences.

Definition 5 (distinctions). *A* distinction *D is a sequence of labelled names of the form $x_1^{e_1}\cdots x_n^{e_n}$, with labels $e_i\in\{\nu,\lambda\}$ and x_i pairwise distinct, for $1\le i\le n$. We let $\lambda(D)$ (resp. $\nu(D)$) denote the set of λ-labelled (resp. ν-labelled) names in D and let $\mathrm{n}(D)=\lambda(D)\cup\nu(D)$. Given a substitution σ and a distinction $D=x_1^{e_1}\cdots x_n^{e_n}$, we say that σ* respects *D, written $\sigma\vdash D$, if for each x_i^{ν} in D it holds that: $x_i\sigma=x_i$ and moreover for each x_j^{λ} in D, $x_j\sigma=x_i$ implies $j>i$. We let $D\sigma$ denote the result of applying σ to D and then erasing all duplicated (i.e. following the leftmost) occurrences of names in the sequence. Given a fusion ϕ and a distinction D, we say that ϕ and D* induce *σ, written $\phi,D\rightsquigarrow\sigma$, if σ is a substitution induced by $\lambda\widetilde{x}$, $(\nu\widetilde{k})$ and ϕ, where $\widetilde{x}=\lambda(D)$ and $\widetilde{k}=\nu(D)$, and σ respects D.*

Example 2. Consider $D = x_0^{\lambda} \cdot x_1^{\nu} \cdot x_2^{\lambda} \cdot x_3^{\lambda} \cdot {x_4}^{\nu}$ and $\phi = \{x_1, x_2, x_4 = x_2, x_3, x_5\}$. We have that $\phi, D \rightsquigarrow \sigma$, where σ maps x_1, x_2, x_3 to x_1, and x_5 to x_4: in fact, we have that $\sigma \vdash D$. Note that $D\sigma = x_0^{\lambda} \cdot x_1^{\nu} \cdot x_4^{\nu}$. On the other hand, if one considers D and $\{x_4 = x_0\}$ then there is no σ' induced by this effect and D: indeed, σ' should have to map x_0 to x_4, but this implies $\sigma' \not\vdash D$.

We use the following abbreviation: $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)} P'$ means that either $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}\alpha} P'$ and $\phi = -$, or $P \xrightarrow{(\nu\tilde{x})\phi} P'$ and $\alpha = -$ (here $-$ is just a conventional 'null' value). Moreover, we stipulate that $-, D \rightsquigarrow \mathtt{id}$, where $\mathtt{id}$ is the identity substitution.

Definition 6 (open bisimulation). *A set $\mathcal{R} = \{R_D\}_D$ of process relations indexed by distinctions is an* indexed simulation *if for each D, whenever $P\ R_D\ Q$ and $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)} P'$ and $\phi, D \cdot \tilde{x}^{\nu} \rightsquigarrow \sigma$, then a transition $Q \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)} Q'$ exists such that $P'\sigma\ R_{D'}\ Q'\sigma$, where $D' = D\sigma \cdot \tilde{x}^{\nu} \cdot \tilde{y}^{\lambda}$.*

$\mathcal{R}$ is an indexed bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1} = \{R_D^{-1}\}_D$ are indexed simulations.* Open bisimulation*, $\sim$, is the largest indexed bisimulation preserved by respectful substitutions, i.e.: for each σ and D, if $P \sim_D Q$ and σ respects D then $P\sigma \sim_{D\sigma} Q\sigma$.*

When no confusion arises, we write $P \sim Q$ for $P \sim_{\varepsilon} Q$, where ε is the empty distinction.

Proposition 2 (basic properties of $\sim_D$). *Let P and Q be two processes. Then:*

- *$P \sim_D Q$ and $x \notin \mathrm{n}(D)$ imply $\lambda x P \sim_D \lambda x Q$.*
- *$P \sim_D Q$ implies $(\nu x) P \sim_{D-x} (\nu x) Q$.*

Prefix, parallel composition, matching, sum and replication operators preserve $\sim_D$.

Example 3. 1. An example of 'expansion' for parallel composition is as follows:

$$((\nu k)\, ak.ak)|av \ \sim\ (\nu k)\, ak.(ak.av + av.ak + \{k = v\}) \ +\ av.((\nu k)\, ak.ak) \ +\ (\nu k)\, \{k = v\}.ak.$$

Note that, after the extrusion of k, free name v can *still* be fused to k. On the contrary, if we consider a distinction D and let $v \in \mathrm{n}(D)$, no fusion at all is possible:

$$((\nu k)\, ak.ak)|av \ \sim_D\ (\nu k)\, ak.(ak.av + av.ak) \ +\ av.((\nu k)\, ak.ak).$$

2. The following two examples show the effect of fusing a λ-abstracted name with a free name and with another λ-abstracted name, respectively:

$$\lambda v\, \{k = v\}.P \ \sim\ \tau.P[k/v] \qquad (\lambda k, v)\, \{k = v\}.P \ \sim\ \lambda k\, \tau.P[k/v].$$

The definition of open bisimulation contains an implicit universal quantification over substitutions, which makes it not handy. In Appendix A we give a symbolic characterisation of open bisimulation, akin to [12].

5 Expressiveness of D-Fusion

Below, we provide the two obvious uniform and fully abstract translations from Π and $\mathcal{F}$ to $\mathcal{DF}$. The definition of uniformity can be extended to the case of encodings from $\mathcal{F}/\Pi$ into $\mathcal{DF}$ in the obvious way: in particular, by requiring that (x) and (νx) be mapped to λx and (νx), respectively.

Definition 7 (**$[\![\cdot]\!]_\pi$**). *The translation $[\![\cdot]\!]_\pi : \Pi \to \mathcal{DF}$ is defined by extending in the expected homomorphic way the following clauses:*

$$[\![\overline{a}\langle x\rangle.P]\!]_\pi = \overline{a}\langle x\rangle.[\![P]\!]_\pi \qquad [\![a(x).P]\!]_\pi = \lambda x\, a\langle x\rangle.[\![P]\!]_\pi \qquad [\![(\nu x)\,P]\!]_\pi = (\nu x)\,[\![P]\!]_\pi\,.$$

For any distinction $D = x_1^{e_1} \cdots x_n^{e_n}$, we can generate an equivalent Π-distinction D_π (that is, an irreflexive, symmetric relation on $\mathcal{N}$, [12]) by setting $x_i\, D_\pi\, x_j$ iff: either ($e_i = e_j = \nu$ and $i \neq j$) or ($e_i = \nu$ and $e_j = \lambda$ and $j < i$) or ($e_j = \nu$ and $e_i = \lambda$ and $i < j$). It is reasonable to consider here only Π-distinctions that can be generated in this way. For any such Π-distinction D_π, let us denote by $[\![D_\pi]\!]$ a chosen $\mathcal{DF}$-distinction that generates D_π. Let $\sim^{\mathrm{o}}_{D_\pi}$ denote open D_π-bisimilarity over Π ([12]).

Proposition 3. *$P \sim^{\mathrm{o}}_{D_\pi} Q$ iff $[\![P]\!]_\pi \sim_{[\![D_\pi]\!]} [\![Q]\!]_\pi$.*

We turn now our attention to Fusion.

Definition 8 (**$[\![\cdot]\!]_f$**). *The translation $[\![\cdot]\!]_f : \mathcal{F} \to \mathcal{DF}$ is defined by extending in the expected homomorphic way the following clauses:*

$$[\![\overline{a}\langle x\rangle.P]\!]_f = \overline{a}\langle x\rangle.[\![P]\!]_f \qquad [\![a\langle x\rangle.P]\!]_f = a\langle x\rangle.[\![P]\!]_f \quad [\![(x)P]\!]_f = \lambda x\,[\![P]\!]_f$$

Let $\sim^{\mathrm{he}}$ denote hyper-equivalence over $\mathcal{F}$ (see [11]).

Proposition 4. *$P \sim^{\mathrm{he}} Q$ iff $[\![P]\!]_f \sim [\![Q]\!]_f$.*

Most interesting, we now show that D-Fusion cannot be uniformly encoded into Π. The intuitive reason is that, in D-Fusion, the combined use of fusions and restrictions allows one to express a form of input with pattern matching. This is not possible in Π, at least without breaking atomicity of certain actions (e.g. choice). To show this fact, we restrict our attention to polarised D-Fusion, $\mathcal{DF}^{\mathrm{p}}$.

Given $P \in \Pi$ and a trace of actions s, let us write $P \stackrel{\hat{s}}{\Longrightarrow}$ if $P \stackrel{s'}{\Longrightarrow}$ for some trace s' that exhibits the same sequence of subject names, with the same polarity, as s (e.g., $s = a\langle\widetilde{x}\rangle \cdot \lambda\widetilde{y}\,\overline{b}\langle\widetilde{v}\rangle$ and $s' = a\langle\widetilde{z}\rangle \cdot \overline{b}\langle\widetilde{w}\rangle$). The reference semantics for Π is again the late operational semantics.

Definition 9. *A translation $[\![\cdot]\!] : \mathcal{DF}^{\mathrm{p}} \to \Pi$ is* uniform *if for each $P, Q \in \mathcal{DF}^{\mathrm{p}}$:*

- *for each trace s, $P \stackrel{s}{\Longrightarrow}$ implies $[\![P]\!] \stackrel{\hat{s}}{\Longrightarrow}$;*
- *$[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$;*
- *for each y, $[\![(\nu y)\,P]\!] = (\nu y)\,[\![P]\!]$;*
- *for each substitution σ, $[\![P\sigma]\!] = [\![P]\!]\sigma$.*

Below, we denote by $\sim_{\mathcal{DF}^{\mathrm{p}}}$ any fixed equivalence over $\mathcal{DF}^{\mathrm{p}}$ which is contained in trace semantics (defined in the obvious way), and by $\sim_\Pi$ any fixed equivalence over Π which is contained in trace equivalence.

Proposition 5. *There is no uniform translation $[\![\cdot]\!] : \mathcal{DF}^{\mathrm{p}} \to \Pi$ such that $\forall P, Q \in \mathcal{DF}^{\mathrm{p}}$:*

$$P \sim_{\mathcal{DF}^{\mathrm{p}}} Q \Rightarrow [\![P]\!] \sim_\Pi [\![Q]\!].$$

PROOF: Suppose that there exists such a translation $[\![\cdot]\!]$. Let us consider the following two $\mathcal{DF}^{\mathrm{p}}$-processes P and Q:

$$P = (\nu c,k,h)\,(c\langle k\rangle.\overline{a}|c\langle h\rangle.\overline{b}|\overline{c}\langle k\rangle) \qquad Q = \tau.\overline{a}.$$

It holds that $P \sim Q$ in $\mathcal{DF}^{\mathrm{p}}$: the reason is that, in P, synchronisation between prefixes $c\langle h\rangle$ and $\overline{c}\langle k\rangle$, which carry different *restricted* names h and k, is forbidden (see rule ν-OPEN). Thus P can only make $c\langle k\rangle$ and $\overline{c}\langle k\rangle$ synchronise, and then perform $\overline{a}$. Thus, $P \sim_{\mathcal{DF}^{\mathrm{p}}} Q$ holds too.

On the other hand, due to uniformity, it is easily checked that $[\![P]\!] \stackrel{\overline{b}}{\Longrightarrow}$, while $[\![Q]\!] \stackrel{\overline{b}}{\not\Longrightarrow}$ (the proof of the latter relies on $b \notin \mathrm{fn}(Q)$ and on the uniformity with respect to substitutions). Thus $[\![P]\!] \not\sim_{\Pi} [\![Q]\!]$. □

Of course, it is also true that D-Fusion cannot be uniformly encoded into $\mathcal{F}$, as this would imply the existence of a uniform fully abstract encoding from Π to $\mathcal{F}$, which does not exist (Proposition 1).

The conclusion is that there is some expressiveness gap between D-Fusion on one side and the other two calculi on the other side, at least, as far as our simple notion of uniform encoding is concerned. This gap is further explored by means of more elaborate examples in the next two sections.

6 Example: Correlation

This example aims at illustrating the gap between D-Fusion and Fusion from a more concrete perspective. Consider the following simple protocol. An agent A asks a trusted server S for two keys, to be used to access two distinct services (e.g. A might be a proxy requiring remote connections on behalf of two different users). Communication between A and S takes place over an insecure public channel, controlled by an adversary, but it is protected by encryption and challenge-response nonces. Informally, the dialogue between A and S is as follows:

$$\begin{array}{l} 1.\ A \to S : n \\ 2.\ S \to A : \{n,k\}_{k_S} \\ 1'.\ A \to S : n' \\ 2'.\ S \to A : \{n',k'\}_{k_S} \end{array}$$

Here $\{\cdot\}_{(\cdot)}$ is symmetric encryption and k_S is a secret master key shared by A and S. A simple property of this protocol is that A should never receive k and k' in the wrong order (k' and then k), even in case S accepts new requests before completing old ones. Indeed, nonces n and n' are intended to help A to contrast attackers replying old, compromised keys or trying to get distinct sessions confused. In other words, nonces do *correlate* each request to S with the appropriate reply of S.

Below, we show that the above small protocol and the related ordering property can be readily translated and verified in D-Fusion. Next, we show that the property breaks down when (uniformly) translating the protocol into Fusion.

D-Fusion Encryption is not a primitive operation in D-Fusion. However, in the present case, it is sensible to model an encrypted message $\{n,k\}_{k_S}$ as an output action $\overline{k_S}\langle n,k\rangle$: only knowing the master key k_S, and further specifying a session-specific nonce, it is possible to acquire the key k (similarly for $\{n',k'\}_{k_S}$, of course). Thus, assuming A concludes the protocol with a

conventional 'commit' action and that p is the public channel, A, S and the whole protocol P might be specified as follows (below, we abbreviate $\lambda\widetilde{x}\,p\langle\widetilde{x}\rangle.X$ as $p(\widetilde{x}).X$):

$$\begin{aligned} A &= (\nu n)\left(\overline{p}\langle n\rangle|\lambda y\, k_S\langle n,y\rangle.(\nu n')\,(\overline{p}\langle n'\rangle|\lambda y'\, k_S\langle n',y'\rangle.\overline{commit}\langle y,y'\rangle)\right)\\ S &= p(x).\left(\overline{k_S}\langle x,k\rangle|p(x').\overline{k_S}\langle x',k'\rangle\right)\\ P &= (\nu k_S)\,(A|S). \end{aligned}$$

Let A_{spec} be the process defined like A, except that the $\overline{commit}\langle y,y'\rangle$ action is replaced by $\overline{commit}\langle k,k'\rangle$, and let $P_{\text{spec}} = (\nu k_S)\,(A_{\text{spec}}|S)$. The property that A should never receive k and k' in the wrong order is stated as: $P \sim P_{\text{spec}}$.

Informally, equivalence holds true because the second input action in A/A_{spec}, that is $\lambda y'\, k_S\langle n',y'\rangle$, can only get synchronised with the second output action in P, that is $\overline{k_S}\langle x',k'\rangle$. In fact, n' can be extruded only *after* x has been received, hence the appropriate distinction will forbid fusion of x and n'. The equivalence is formally shown by exhibiting the appropriate symbolic indexed bisimulation, composed basically by all pairs of derivatives P' and Q' s.t. $P \stackrel{s}{\Longrightarrow} P'$ and $Q'$is obtained from P' by replacing $\overline{commit}\langle y,y'\rangle$ with $\overline{commit}\langle k,k'\rangle$, with distinctions as given by the bound names in s.

Fusion Suppose P^f and P^f_{spec} are obtained by some uniform encoding of P and P_{spec} above into Fusion. It is not difficult to show that P^f can be 'attacked' by an adversary R that gets n and n' and fuse them together, $R = p(x).\left(\overline{p}\langle x\rangle|p(y).\overline{p}\langle y\rangle\right)$. Formally, for $\alpha = \overline{commit}\langle k',k\rangle$,

$$P^f|R \stackrel{\alpha}{\Longrightarrow} \text{ and, thus, } P^f|R \not\sim^{\text{he}} P^f_{\text{spec}}|R,$$

which proves that P^f and P^f_{spec} are not hyper-equivalent.

This example illustrates the difficulty of modelling fresh, indistinguishable quantities (nonces) in Fusion. This makes apparent that Fusion is not apt to express security properties based on correlation.

7 Encoding guarded choice

In this section we show how the combined mechanisms of fusions and restrictions can be used to encode different forms of guarded choice *via* parallel composition, in a clean and uniform way. Informally, different branches of a guarded choice will be represented as concurrent processes. The encodings add pairs of extra names to the object part of each action: these extra names are used as 'side-channels' for atomic coordination among the different branches.

Below we make use of the concept of sorting system, which has been introduced in polyadic pi-calculus to discipline the use of channels [6]. We adopt a slightly extended notion, intended to also control 'patterns' that can arise in the object part of actions. In a sorting system $\mathbf{S}$, the set of names is partitioned into a family of countable *sorts*, denoted as $S, S', \ldots$. Moreover, there is a sorting function that maps each sort S to a *pattern* $(x_1 : S_1, \ldots, x_n : S_n)$, written also $\mathbf{S}:\ S \mapsto \widetilde{x} : \widetilde{S}$, with the x_i's not necessarily distinct: the intended meaning is that names of sort S should only carry tuples $\widetilde{v}$ of type $\widetilde{S}$ s.t. $x_i = x_j$ implies $v_i = v_j$. We denote by $\sim^{\text{S}}$ the bisimilarity induced by $\mathbf{S}$: the intuition is that actions not obeying the above requirement on patterns are discarded; a formal definition of $\sim^{\text{S}}$ is given in Appendix B. Note that $\sim^{\text{S}}$ is in general more generous than ordinary open bisimilarity $\sim$. For example, assuming $a \in S \mapsto (x : S', x : S')$, then $(\nu n)\,\lambda x\, axn.P \sim^{\text{S}} (\nu n)\,\lambda x\, anx.P \sim^{\text{S}} (\nu n)\, ann.P[n/x]$ and $(\nu n)\,(\nu m)\, amn.P \sim^{\text{S}} \mathbf{0}$.

Example 4. Consider the guarded choice $A = \lambda x\,(\nu n)\,a\langle xn\rangle.P + \lambda x\,(\nu m)\,a\langle xm\rangle.Q$. Its intended 'parallel' implementation is the process:

$$B = \lambda x\Big(\,(\nu n)\,a\langle xn\rangle.P\,|\,(\nu m)\,a\langle xm\rangle.Q\,\Big)$$

(here, $x,n,m \notin \mathrm{fn}(a,P,Q)$). Assume now a sorting discipline by which messages travelling on channel a must carry two identical names (i.e. $a \in S \mapsto (x : S', x : S')$). In B, the parallel component that first consumes any such message, forces fusion of x either to n or to m, and consequently inhibits the other component. E.g.:

$$\lambda u\,\overline{a}\langle uu\rangle|B \xrightarrow{\tau}\sim\ (\nu n)\,(P\,|\,(\nu m)\,a\langle mn\rangle.Q) \quad\sim\quad P\,|(\nu n,m)\,a\langle mn\rangle.Q.$$

Under the mentioned sorting assumption, $(\nu m,n)\,a\langle mn\rangle.Q$ is equivalent to $\mathbf{0}$, because there is no way of fusing m and n. Thus the process on the right of $\sim$ is equivalent to P. In other words, choice between P and Q has been resolved atomically.

The above line of reasoning can be formalised in two ways. One way is considering bisimilarity induced by the above sorting, say $\sim^{\mathrm{S}}$, which only takes into account transitions that obey the sorting discipline (the rightmost two names in the object part of $a/\overline{a}$-actions are the same). The other way is keeping standard bisimilarity $\sim$, while inserting processes inside a 'firewall' that filters out a-messages not respecting the given sorting. The latter can be easily defined in D-Fusion relying on 'non-linear' inputs:

$$\mathsf{F}_{a,a'}[\cdot] = (\nu a')\,\Big(\,\lambda z\,azz.\overline{a'}\langle zz\rangle|[\cdot]\,\Big)\,.$$

We state the result in both forms below.

Proposition 6. *Let A and B be as in Example 4.*

1. $A \sim^{\mathrm{S}} B$;
2. *Let A' and B' be the processes obtained from A and B, respectively, by replacing the outermost occurrences of a with a fresh a'. Then $\mathsf{F}_{a,a'}[A'] \sim \mathsf{F}_{a,a'}[B']$.*

Note that the result above exploits in a crucial way features of both Fusion (non-linearity of input actions, in the firewall, and sharing of input variable x, in B) and of D-Fusion (restricted input).

Proposition 6 can be generalised to fully abstract encodings of different forms of guarded choice. For the sake of simplicity, we will state the results in terms of sorted bisimilarity. We believe the results can also be stated in terms of untyped bisimilarity, at the cost of breaking uniformity of the encoding and of introducing more sophisticated forms of 'firewalls'. We examine two cases, input-guarded choice and mixed choice.

Input-guarded (ig) choice Let us fix, as a source language the fragment of polarised D-Fusion with guarded choice, $\mathcal{DF}^{\mathrm{p,ig}}$. In this language, input prefix and summation $+$ are replaced by input-guarded choice $\sum_{i\in i} a_i\langle\widetilde{x_i}\rangle.P_i$. We also assume a sorting system $\mathbf{S}$ on this source language. The target language is the fragment of polarised D-Fusion with no form of summation. The relevant clauses of the encoding are:

$$[\![\sum_{i\in i} a_i\langle\widetilde{x_i}\rangle.P_i]\!]_{\mathrm{ig}} = \lambda z\,\Pi_{i\in I}(\nu n)\,\lambda\widetilde{x_i}\,a_i\langle\widetilde{x_i}zn\rangle.[\![P_i]\!]_{\mathrm{ig}} \qquad [\![\overline{a}\langle\widetilde{v}\rangle.P]\!]_{\mathrm{ig}} = \lambda z\,\overline{a}\langle\widetilde{v}zz\rangle.[\![P]\!]_{\mathrm{ig}}\,,$$

where $\Pi_{i\in I}X_i$ denotes the parallel composition of all X_i's. The encoding acts as a homomorphism over the remaining operators. Below, **S1** indicates the sorting induced by the translation, imposing that the two extra object names introduced by the encoding be the same (i.e., if **S**: $S \mapsto \widetilde{x} : \widetilde{S'}$, then **S1**: $S \mapsto (\widetilde{x} : \widetilde{S'}, y : S_0, y : S_0)$, for a fresh y and a new sort S_0). The corresponding open bisimilarity is denoted by $\sim^{\mathrm{S1}}$. The proof of the following theorem is straightforward, given that there is a 1-to-1 correspondence between moves of P and moves of $[\![P]\!]_{\mathrm{ig}}$, under the sortings **S** and **S1**.

Theorem 1 (full abstraction for ig choice). *Let $P, Q \in \mathcal{DF}^{\mathrm{p,ig}}$. It holds that $P \sim^{\mathrm{S}} Q$ if and only if $[\![P]\!]_{\mathrm{ig}} \sim^{\mathrm{S1}} [\![Q]\!]_{\mathrm{ig}}$.*

Of course, the above theorem also yields a fully abstract encoding of input-guarded choice for pi-calculus, which may be viewed as a sub-calculus of $\mathcal{DF}^{\mathrm{p,ig}}$.

Mixed choice in a sorted pi-calculus As a source language we fix here a sorted version of polyadic pi-calculus [6] with 'mixed' choice, Π^{mix}. In this language, prefixes and $+$ are replaced by mixed summation, $\sum_{i\in I} a_i(\widetilde{x_i}).P_i + \sum_{j\in J} \overline{b_j}\langle \widetilde{v_j}\rangle.Q_j$. A sorting system **S** is fixed on this language. The target language is again the fragment of polarised D-Fusion with no summation at all. The encoding is a bit more complex than in the previous case, as it implies adding *two* pairs of extra names to coordinate different branches. The relevant clause is:

$$\begin{array}{l}[\![\sum_{i\in I} a_i(\widetilde{x_i}).P_i + \sum_{j\in J} \overline{b_j}\langle v_j\rangle.Q_j]\!]_{\mathrm{mix}} \quad = \\ (\lambda z,u)\,(\,\Pi_{i\in I}(\nu n)\,\lambda\widetilde{x_i}\,a_i\langle \widetilde{x_i} z n u u\rangle.[\![P_i]\!]_{\mathrm{mix}} \;\mid\; \Pi_{j\in J}(\nu n)\,\overline{b_j}\langle \widetilde{v_j} u u z n\rangle.[\![Q_j]\!]_{\mathrm{mix}}\,).\end{array}$$

Note that the relative positions of ν-names correctly forbid communication between branches of opposite polarities within the same choice (no 'incestuous' communication, according to the terminology of [8]). The encoding acts as a homomorphism over the remaining operators of Π^{mix}.

Below, $\sim^{\mathrm{S}}$ indicates sorted open bisimilarity in Π^{mix}. We denote by **S2** the sorting induced by the above translation imposing that the components of each pair of extra names introduced by the encoding be the same (i.e., if **S**: $S \mapsto \widetilde{x} : \widetilde{S'}$, then **S2**: $S \mapsto (\widetilde{x} : \widetilde{S'}, y : S_0, y : S_0, z : S_0, z : S_0)$, where S_0 is a new sort and y and z are fresh). The corresponding bisimilarity is written $\sim^{\mathrm{S2}}$.

Theorem 2 (full abstraction for mixed choice). *Let $P, Q \in \Pi^{\mathrm{mix}}$. It holds that $P \sim^{\mathrm{S}} Q$ if and only if $[\![P]\!]_{\mathrm{mix}} \sim^{\mathrm{S2}} [\![Q]\!]_{\mathrm{mix}}$.*

In a pi-calculus setting, it is well-known that mixed choice cannot be encoded into the choice-free fragment, if one requires the encoding be uniform and preserve a 'reasonable semantics' [9, 10, 8]. The theorem above shows that, under mild typing conditions, pi-calculus mixed choice *can* be implemented into the choice-free fragment of D-Fusion. The encoding is uniform, deadlock- and divergence-free, and preserves a 'reasonable semantics'. This is yet another evidence of the expressiveness gap between D-Fusion and pi-calculus.

8 Conclusions and Future Work

We have proposed the D-Fusion calculus, an extension of the fusion calculus where two distinct binders coexist, one analogous to the (x) binder in fusion, the other imposing name freshness. We have shown that D-Fusion is more expressive than both Fusion and pi-calculus.

Our expressiveness results seem to suggest that an efficient distributed implementation of D-Fusion might be nontrivial to design. This design would certainly involve the introduction of a distributed model of the calculus, including, e.g., explicit fusions [3] for broadcasting fusions asynchronously, and primitives for handling explicit localities. We leave this task for future work. For the time being, we just note that distributed implementations of pi/fusion-like calculi do exist (e.g., the fusion machine of [2]) and may represent a good starting point for building a distributed implementation of D-Fusion.

In [1] the synchronisation mechanism of the pi-calculus is extended to allow for polyadic synchronisation, where channels are vectors of names. The expressiveness of polyadic synchronisation, matching and mixed choice is compared and it is shown how the degree of synchronisation of a calculus increases its expressive power.

We plan to extend the D-Fusion calculus by generalising name fusions to substitutions over an arbitrary signature of terms. We believe that the extended D-Fusion would be strictly more expressive that Logic Programming, the intuition being that restriction (creation of new fresh names) cannot be modelled in LP.

Finally, it would also be interesting to study a coalgebraic model for D-Fusion. A coalgebraic framework would present several advantages. For instance, morphisms between coalgebras enjoy the property of "reflecting behaviours" and thus they allow to characterise bisimulation equivalences in more abstract terms as kernels of morphisms.

References

1. M. Carbone and S. Maffeis. On the Expressive Power of Polyadic Synchronisation in Pi-Calculus. To appear in *Nordic Journal of Computing*.
2. P. Gardner, C. Laneve, and L. Wischik. The fusion machine (extended abstract). In *Proc. of CONCUR '02*, LNCS 2421. Springer-Verlag, 2002.
3. P. Gardner and L. Wischik. Explicit Fusions. *Theoretical Computer Science*. To appear.
4. L. G. Meredith, S. Bjorg, and D. Richter. Highwire Language Specification Version 1.0. Unpublished manuscript.
5. Microsoft Corp. Biztalk Server - `http://www.microsoft.com/biztalk`.
6. R. Milner. The Polyadic pi-Calculus: a Tutorial. Technical Report, Computer Science Dept., University of Edinburgh, 1991.
7. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.
8. U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.
9. C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus. In *Conf. Rec. of POPL'97*, 1997.
10. C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.
11. J. Parrow and B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *Proc. of LICS'98*. IEEE Computer Society Press, 1998.
12. D. Sangiorgi. A Theory of Bisimulation for the pi-Calculus. *Acta Informatica*, 33(1): 69-97, 1996.
13. World Wide Web Consortium (W3C) - `http://www.w3.org/TR/wsdl12`.

A Symbolic Bisimulation

We define an alternative notion of bisimulation, akin to the 'efficient characterisation' of [12], and prove that it is equivalent to the 'concrete' bisimulation, defined in Section 4. The new formulation is based on symbolic transitions of the form $P \xrightarrow{\mu}_{\chi} Q$, where χ is a logical condition necessary for the transition to take place. For technical convenience, we shall represent conditions as fusions. The defining clauses for $\xrightarrow{\mu}_{\chi}$ are obtained from those in Table 1 by simply attaching the appropriate χ. In particular, action prefix is defined as $\alpha.P \xrightarrow{\alpha}_{\tau} P$, while communication and matching rules are defined as follows:

$$(\textsc{Match}_S)\ \frac{P \xrightarrow{\mu}_{\chi} Q}{[a=b]P \xrightarrow{\mu}_{\chi+\{a=b\}} Q} \qquad (\textsc{Com}_S)\ \frac{P_1 \xrightarrow{(\nu\tilde{x}')\lambda\tilde{y}'\,a\tilde{u}}_{\chi_1} Q_1 \quad P_2 \xrightarrow{(\nu\tilde{x}'')\lambda\tilde{y}''\,b\tilde{v}}_{\chi_2} Q_2}{P_1|P_2 \xrightarrow{(\nu\tilde{x})\,\{\tilde{v}=\tilde{u}\}_{-\tilde{y}'\tilde{y}''}}_{\chi_1+\chi_2+\{a=b\}} (\nu\tilde{w})\lambda\tilde{z}((Q_1|Q_2)\sigma_{|\tilde{y}'\tilde{y}''})},$$

where the side condition of $\textsc{Com}_S$ is the same as in rule $\textsc{Com}$. The other rules just propagate χ, with the proviso that λ- and ν- extruded names do not occur in χ. Given fusions ϕ and ψ, we write $\phi \vDash \psi$ if $\psi \subseteq \phi$.

Definition 10 (symbolic bisimulation). *A set $\mathcal{R} = \{R_D\}_D$ of process relations indexed by distinctions is an* indexed symbolic simulation *if whenever $P\ R_D\ Q$ and $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)}_{\chi} P'$ and $(\phi+\chi), D\cdot\tilde{x}^{\nu} \rightsquigarrow \sigma$, then a transition $Q \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\beta,\psi)}_{\chi'} Q'$ exists such that:*

1. $\chi \vDash \chi'$,
2. $\chi \vDash \{\mathrm{subj}(\alpha) = \mathrm{subj}(\beta)\}$ *and* $\chi \vDash \{\mathrm{obj}(\alpha) = \mathrm{obj}(\beta)\}$,
3. $\phi+\chi = \psi+\chi$,
4. $P'\sigma\ R_{D'}\ Q'\sigma$, *where* $D' = D\sigma\cdot\tilde{x}^{\nu}\cdot\tilde{y}^{\lambda}$.

$\mathcal{R}$ is a indexed symbolic bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1} = \{R_D^{-1}\}_D$ are indexed open simulations. The largest indexed symbolic bisimulation is written $\sim^{\mathrm{symb}}$.*

Theorem 3. *For each P, Q and D, $P \sim_D Q$ if and only if $P \sim_D^{\mathrm{symb}} Q$.*

PROOF: For any ϕ and σ, let $\phi\sigma$ denote the fusion $\{(x\sigma, y\sigma) \mid x\,\phi\,y\}^{\star}$, and let the *fusion induced by* σ (seen as a binary relation) be defined as $\sigma^{\star}$. The proof of the theorem is based on the following operational correspondences:

1. $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)}_{\chi} P'$ implies there is a transition $P\sigma \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha\sigma,\phi\sigma)} P'\sigma$, for each substitution σ induced by χ.
2. $P\sigma \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\beta,\psi)} P_1$ implies there is a fusion χ induced by σ and a transition $P \xrightarrow{(\nu\tilde{x})\lambda\tilde{y}(\alpha,\phi)}_{\chi'} P'$ s.t. $\chi \vDash \chi'$, $\beta = \alpha\sigma$ and $\psi = \phi\sigma$ and $P_1 = P'\sigma$.

□

Example 5. Fusions and conditions play pretty different roles in bisimulation semantics. As an example, let $P = [a=b](ca \mid \overline{c}b)$ and $Q = ca \mid \overline{c}b$. Both in P and Q the interaction between ca and $\overline{c}b$ results in a fusion $\{a=b\}$, but $P \not\sim Q$ since P can perform an action only under the condition $\chi = \{a=b\}$, i.e. $P \xrightarrow{\{a=b\}}_{\{a=b\}} \mathbf{0}$ while $Q \xrightarrow{\{a=b\}}_{\tau} \mathbf{0}$.

B Sorting

We adapt the concept of sorting from pi-calculus, and slightly extend it in order to also control 'patterns' that can arise in the object part of actions. In a sorting system **S**, the set of names is partitioned into a family of countable *sorts*, denoted as $S, S', \ldots$. Moreover, there is a sorting function that maps each sort S to a *pattern* $(x_1 : S_1, \ldots, x_n : S_n)$, written also $\mathbf{S} : \ S \mapsto \widetilde{x} : \widetilde{S}$, with the x_i's not necessarily distinct: the intended meaning is that names of sort S should only carry tuples $\widetilde{v}$ of type $\widetilde{S}$ s.t. $x_i = x_j$ implies $v_i = v_j$.

Like in polyadic pi-calculus, one can devise typing systems which guarantee that well-typed processes will never perform 'illegal' (in the above sense) actions. However, we are primarily interested in defining the open bisimilarity $\sim^{\mathrm{S}}$ naturally induced by a sorting system **S**. Let us say that a substitution σ is **S**-*respectful* if, for each x, $x\sigma$ and x belong to the same sort. Similarly, we say a fusion ϕ is **S**-respectful if each equivalence class is entirely included in one sort (hence any substitution induced by ϕ is respectful). In a **S**-sorted D-Fusion, for a given sorting system **S**, we assume that in every prefix $a\widetilde{v}$ each v_i belongs to the appropriate sort, and we *only consider* **S**-*respectful substitutions and fusions*. For $\alpha = a\widetilde{v}$ an action, where $a \in S \mapsto \widetilde{x} : \widetilde{S'}$, the fusion induced by α, denoted ϕ_α, is defined as $\sum_{x_i,\, x_j \in \widetilde{x}, x_i = x_j} \{v_i = v_j\}$ (here 'Σ' denotes sum of fusions defined in Def. 2). Furthermore, we shall denote by $\equiv$ equivalence over actions induced by the law: $\lambda x \mu \equiv \mu$ if $x \notin \mathrm{n}(\mu)$.

With the above conventions and notations, the definition of $\sim^{\mathrm{S}}$ is obtained from Definition 6 by making the clauses of actions and effects explicitly distinct, and changing the clause for *actions*. This results in the following definition. Recall that all fusions and substitutions are required to be **S**-respectful.

Definition 11 (sorted open bisimulation). *A set $\mathcal{R} = \{R_D\}_D$ of process relations indexed by distinctions is an* sorted indexed simulation *if for each D, whenever $P\ R_D\ Q$*

- *if $P \xrightarrow{(\nu\widetilde{x})\,\lambda\widetilde{y}\,a\widetilde{v}} P'$ and $\phi_{a\widetilde{v}}, D \cdot \widetilde{x}^{\nu} \cdot \widetilde{y}^{\lambda} \rightsquigarrow \sigma$ then a transition $Q \xrightarrow{(\nu\widetilde{x})\,\lambda\widetilde{y'}\,a\widetilde{v'}} Q'$ exists such that $\phi_{a\widetilde{v'}}, D \cdot \widetilde{x}^{\nu} \cdot \widetilde{y}^{\lambda} \rightsquigarrow \sigma'$ and $\lambda\widetilde{y}\,(a\widetilde{v}\sigma) \equiv \lambda\widetilde{y'}\,(a\widetilde{v'}\sigma')$, $\sigma_{-\widetilde{y}} = \sigma'_{-\widetilde{y'}}$ and $P'\sigma\ R_{D'}\ Q'\sigma'$, where $D' = (D \cdot \widetilde{x}^{\nu} \cdot \widetilde{y}^{\lambda})\sigma$.*
- *$P \xrightarrow{(\nu\widetilde{x})\,\phi} P'$ and $\phi, D \cdot \widetilde{x}^{\nu} \rightsquigarrow \sigma$, then a transition $Q \xrightarrow{(\nu\widetilde{x})\,\phi} Q'$ exists such that $P'\sigma\ R_{D'}\ Q'\sigma$, where $D' = D\sigma \cdot \widetilde{x}^{\nu}$.*

$\mathcal{R}$ is a sorted indexed bisimulation *if both $\mathcal{R}$ and $\mathcal{R}^{-1} = \{R_D^{-1}\}_D$ are sorted indexed simulations.* Sorted open bisimulation*, $\sim^{\mathrm{S}}$, is the largest sorted indexed bisimulation preserved by respectful substitutions, i.e.: for each σ and D, if $P \sim^{\mathrm{S}}_D Q$ and σ respects D then $P\sigma \sim^{\mathrm{S}}_{D\sigma} Q\sigma$.*

We write $P \sim^{\mathrm{S}} Q$ for $P \sim^{\mathrm{S}}_{\varepsilon} Q$, where ε is the empty distinction.

Electronic Notes in Theoretical Computer Science 85 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume85.html 19 pages

Lexically scoped distribution: what you see is what you get

António Ravara [a] Ana G. Matos [b] Vasco T. Vasconcelos [c]
Luís Lopes [d]

[a] *CLC and Dep. of Mathematics, IST, Technical University of Lisbon, Portugal.*
[b] *INRIA, Sophia-Antipolis, France.*
[c] *Dep. of Informatics, Faculty of Sciences, University of Lisbon, Portugal.*
[d] *Dep. of Computer Science, Faculty of Sciences, University of Porto, Portugal.*

Abstract

We define a lexically scoped, asynchronous and distributed π-calculus, with local communication and process migration. This calculus adopts the network-awareness principle for distributed programming and follows a simple model of distribution for mobile calculi: a lexical scope discipline combines static scoping with dynamic linking, associating channels to a fixed site throughout computation. This discipline provides for both remote invocation and process migration. A simple type system is a straightforward extension of that of the π-calculus, adapted to take into account the lexical scope of channels. An equivalence law captures the essence of this model: a process behavior depends on the channels it uses, not on where it runs.

1 Introduction

Current hardware developments in network technology, namely high-bandwidth, low-latency networks and wireless communication, has opened new prospects for mobile computation, while at the same time introducing new problems that need to be addressed at the software level. The fundamental problem stems from the lack of a formal background on which to assert the correctness of a given system specification. Thus, adequate theoretical modeling of distributed mobile systems is required to produce provably correct software specifications and to reason about distributed computations.

A natural and simple framework to study such systems is Dpi, the distributed π-calculus of Hennessy and Riely [4,3] that extends the π-calculus [6,7] by distributing processes over a (flat) network of *localities* — named sites where computation happens. Communication only takes place within localities (to avoid global synchronization), but processes may migrate between locations.

Lexically scoped programming. To motivate the importance of lexical scope in programming languages in general, consider the following function written in (a variant of) Pascal.

```
function f (): Integer;
  var x: Integer := 1;
  function g () : Integer;
    begin g := x end;
  begin f := x + g () end;
```

A *programmer* would write the body of function g knowing that x is global in g, and would develop function f keeping in mind that x is local to f, and that the x of g and that of f denote the same variable. Programmers have been doing this kind of reasoning for decades, both in imperative (Algol, Pascal) and in functional (ML, Haskell) languages. It is intuitive, and accumulated experience has shown the concept to be right. Furthermore, an unoptimized *compiler* would assign to variable x a memory location at the activation record for function f. To evaluate expressions in the body of a function, the generated code must read the value of x: if in f, it performs a local operation (reading from the data for the current activation); if in g, it first finds where f's activation is.

The good news is that we do not need to abandon these ideas when moving into a distributed setting, even in the presence of code migration. While in the above example the pertinent question is "which function does this variable belong to?", in a distributed setting we will ask "which locality does this channel belong to?". Under a lexical scope regime, the answer must be clear in the *syntax*. Therefore, we adhere to the *lexical scope in a distributed context* of Obliq [2], which is here understood as the discipline under which the locality of channels is fixed throughout computation and can be determined by straightforward code inspection. This principle must not be disturbed by computation, in particular by code mobility and dynamic linking. The paradigm allows us to *a priori* view channels as physical resources of sites, unlike Dpi where channels are network-wide identifiers that only *a posteriori* (via a type system) get located in sites.

Lexically scoped distributed π-calculus. *lsd*π is a variant of Dpi where channel location can be inferred from the syntax of networks. The lexical scope discipline is obtained by the following simple rule: *compound channels (such as* $a@s$*) belong to the site where they are explicitly located (*s*); while simple channels are (implicitly) located at the site where they occur.* To accommodate process mobility, (free) channels are renamed via a translation function that adapts the syntax of the migrated process to its new location. A form of *dynamic linking* (of a channel to a site) is obtained via a $(\nu\, a)$ binding, for the site that will allocate channel a will only be known at runtime.

In *lsd*π *what you see is what you get*, since the location of channels are clear in the program text, and the syntax plainly expresses the location of the resources used, even in presence of mobile code.

Dpi and *lsdπ*. Consider a network where we declare a channel x within some site f, ask for the process x?()P to migrate to another site g, and in parallel launch a receptor x?()Q located at f. In an untyped version of Dpi, the network and the one obtained by a reduction step appear as follows.

```
f [new x                          new x@f
  go g. x?()P |                   g [x?()P] |
  x?()Q]                          f [x?()Q]
```

From the preceding discussion on the Pascal program, we are interested in asking "where does channel x belong to?" (or "where is x to be allocated?"). Analyzing the new x part, one would expect channel x to belong to site f, but at the level of the subnetworks for g and for f we cannot conclude that. The type system for Dpi identifies the receptor x at f with that at g, and, since x is bound at f, rejects the network.

In general, as we have said, by inspecting the name of a free channel within a site and name of the site, the physical location of a channel can be known at all times. This is achieved by allowing both simple (a) and compound ($a@s$) channels, which reflect two possible views: local (process-level), and global (network-level). In $lsd\pi$, the above left network is indeed typable, and reduces to:

```
new x@f
g [x@f?()P'] |
f [x?()Q]
```

where it is clear (in each of the three lines) that channel x still belongs to site f. Process P' (obtained from P by a translation on names) reflects the fact that P has migrated from f: a free simple channel y (implicitly located at f) becomes attached to its site, denoted y@f; all the remaining channels remain unchanged.

Explicitly located input/output processes (receptors x@f?(y)P and messages x@f!⟨v⟩), absent in every proposal of distributed process calculi to date, obviate the need for a migration primitive like go (written go s.P in [1] and in [4]). It suffices to attach to such processes the behavior "migrate toward the home site": a remote message or receptor at channel x@f migrate to f, in this way obtaining the required "local status". Hence, an input/output process can perform two types of reduction, depending on the format of the prefix: it either communicates locally, or it migrates to the site where it is located.

Contributions. The lexical scope discipline allows for a straightforward extension of the simple type system of the π-calculus, assigning the usual channel types to channels, and record types (maps from channels to channel types) to sites, guarantying the absence of arity mismatches in input/output.

From the standard notion of asynchronous bisimulation, this discipline yields a simple equivalence law, whereby a process behavior depends only on the names it uses (and not on where it runs).

Specifically, our contributions are: (1) a distributed asynchronous π-calculus that provides for local communication, remote invocation, and weak mobility, in a lexical scope regime; (2) a precise syntactic treatment of channels as resources of sites, which ensures that each channel belongs to a unique, lexically defined, site; (3) a simple type system revealing the site of each channel; and (4) a notion of equivalence, inducing a fundamental law.

Related work. Dpi is an explicitly typed calculus. The processes accepted by the type system are similar in many respects to $lsd\pi$ processes. Our conjecture is that what Dpi achieves with a type system, what we guarantee with our semantics. Free names and substitution in Dpi are just like in π (hence the channels are global), but the type system is quite elaborate, being difficult to recognize what is a well-typed process. The semantics of $lsd\pi$ is more complex than that of π (to get channels as resources of sites), but the type system is fairly simple.

Klaim [8,9] is a formalism combining asynchronous higher-order process calculus with the Linda coordination paradigm. It supports process mobility and distribution and provides mechanisms for security. Although communication is not channel-based (else via distributed tuple spaces), in what respects the scoping discipline, $lsd\pi$ is similar to Klaim: both use static scoping to uniquely resolve identifiers at communication-time, and both support dynamic linking at migration-time.

Outline. The next two sections introduce the syntax and the operational semantics of the calculus. Section 4 presents the type system, and Section 5 the behavioral equivalence. Section 6 concludes the paper.

2 Syntax

Consider a countable set $\mathcal{C}$ of *simple channels* a, b, c, x, y, z, and a countable set $\mathcal{S}$ of *sites* s, r, t, such that the two sets are disjoint. Compound channels — pairs channel-site, like $a@s$ — form *located channels*, designating a channel a at site s, belonging to the set $\mathcal{C}@\mathcal{S} \stackrel{\text{def}}{=} \{a@s \mid a \in \mathcal{C} \wedge s \in \mathcal{S}\}$. Sets $a@\mathcal{S}$ and $\mathcal{C}@s$ are defined analogously as $\{a@s \mid s \in \mathcal{S}\}$ and $\{a@s \mid a \in \mathcal{C}\}$.

Let $u, v, \ldots$ stand for both simple and located channels, henceforth collectively called *channels*. Take $\tilde{x}$ as a sequence of pairwise distinct simple channels, and $\tilde{v}$ as a sequence of channels. Let $|\tilde{v}|$ denote the length of the sequence $\tilde{v}$, and let $\{\tilde{v}\}$ denote the set of the elements in the sequence $\tilde{v}$. Furthermore, let n, m stand for both sites and for channels, henceforth collectively called *names*, and belonging to the set $\mathcal{N} \stackrel{\text{def}}{=} \mathcal{C} \cup \mathcal{S} \cup \mathcal{C}@\mathcal{S}$. Finally, let g, h stand for both sites and compound channels, henceforth collectively called *global names*.

The grammars in Figure 1 define the languages of names, of processes, and of networks. *Receptors*, $u?(\tilde{x})P$, and *messages*, $u!\langle\tilde{v}\rangle$, are the basic processes in the calculus. A receptor is an input-guarded process. A message targets

$$
\begin{array}{rrcl}
\textit{Simple channels,} & a, b, c, x, y, z & \in & \mathcal{C} \qquad \textit{Sites,} \quad r, s, t \in \mathcal{S} \\
\textit{Channels,} & u, v & ::= & a \mid a@s \\
\textit{Globals,} & g, h & ::= & a@s \mid s \\
\textit{Names,} & n, m & ::= & a \mid a@s \mid s \\
\textit{Processes,} & P, Q & ::= & \mathbf{0} \mid (P \mid Q) \mid (\nu\, n)\, P \mid u!\langle \tilde{v} \rangle \\
 & & \mid & u?(\tilde{x})P \mid u?*(\tilde{x})P \\
\textit{Networks,} & N, M & ::= & \mathbf{0} \mid (N \mid M) \mid (\nu\, g)\, N \mid s[P] \\
\textit{Entities,} & X, Y, Z & ::= & P \mid N
\end{array}
$$

Fig. 1. Syntax of $lsd\pi$

a name u and carries a sequence of channels $\tilde{v}$. Prefixes of receptors and of messages, may occur as compound channels, but this is just a consequence of the local/remote form that all channels might exhibit. As to the objects of communication, notice that although parameters are simple channels only, arguments can be channels. We call this *global substitution*: channels are to be seen as identifiers, while parameters are "blind" to the structure of the value they receive.

The remaining constructors are fairly standard in name-passing process calculi: $u?*(\tilde{x})P$ denotes a persistent receptor; *inaction* $\mathbf{0}$, denotes the terminated process; $(P \mid Q)$ denotes the *parallel composition* of processes; and $(\nu\, n)\, P$ denotes the *restriction of the scope* of the name n to the process P.

The basic unit of *networks* is a process running at a given site, $s[P]$. Again, the remaining constructors are standard: *inaction* denotes the empty network; the parallel composition of networks, $(N \mid M)$, denotes a merge of networks; and $(\nu\, g)\, N$ represents the restriction of the scope of a global to a network. Name restriction is limited to globals at network level, for local channels make no sense outside the scope of a site in a lexical scope regime. Furthermore, if channels are to be understood as resources of sites, it is reasonable to limit the creation of local channels to within sites.

We will generally refer to processes and networks as *entities*; let $\mathcal{X}$ denote the set of all entities, ranged over by X, Y, Z. As usual in process calculi, for some $m \geq 0$, we abbreviate $(\nu\, n_1) \cdots (\nu\, n_m)\, P$ to $(\nu\, \widetilde{n})\, P$ and $(\nu\, g_1) \cdots (\nu\, g_m)\, N$ to $(\nu\, \tilde{g})\, N$, and consider that the operator 'ν' binds tighter than the operator '$|$'. Furthermore, in $(X \,|\, Y)$ we omit the parenthesis when the meaning is clear.

Finally, we do not allow the communication of site names. We believe this restriction does not result in any loss of expressive power.

X	$\mathsf{fn}(X)$	n	$\mathsf{subj}(n)$	$\mathsf{obj}(n)$
$\mathbf{0}$	$\{\}$	a	$\{a\}$	$\{a\}$
$(X \mid Y)$	$\mathsf{fn}(X) \cup \mathsf{fn}(Y)$	$a@s$	$\{a@s\}$	$\{a@s, s\}$
$(\nu\, n)\, X$	$(\mathsf{fn}(X) \cup \mathsf{obj}(n)) \setminus \mathsf{subj}(n)$	s	$\mathcal{C}@s \cup \{s\}$	$\{s\}$
$u?(\tilde{x})P$	$\mathsf{fn}(P) \setminus \{\tilde{x}\} \cup \mathsf{obj}(u)$			
$u!\langle\tilde{v}\rangle$	$\mathsf{obj}(u\tilde{v})$			
$s[P]$	$\mathsf{fn}(P)@s \cup \{s\}$			

Fig. 2. Subjects, objects and free names

3 Operational semantics

This section describes the reduction semantics of *lsd*π, starting from the definition of the free names, substitution, through structural congruence, ending in reduction.

Free names in networks and in processes. The definition of free names establishes the basic principles of the lexical scope discipline. Presence of compound channels inherently introduces binding subtleties, and to implement a lexical scope regime on names, we cannot rely on the definition of free names used in Dpi.

It becomes useful to distinguish two central concepts regarding name bindings: *subjects* of a name n, $\mathsf{subj}(n)$, those that are caught by a restriction on n; and *objects* of a name n, $\mathsf{obj}(n)$, those which can catch n by restriction. In calculi with simple names only we find that $\mathsf{subj}(n) = \{n\} = \mathsf{obj}(n)$. Here, the definition must be refined.

The rules in the right table in Figure 2 inductively define the sets of *subjects* and of *objects of names*, extended to sets and to sequences of names in the expected way. Subjects and objects are just two sides of the same binding relation. We should be able to say interchangeably that m is bound by n, or that m is a subject of n, that is n is an object of m. In fact, $m \in \mathsf{subj}(n) \Leftrightarrow n \in \mathsf{obj}(m)$. This property allows us to say n *binds* m, $n \hookrightarrow m$, if $m \in \mathsf{subj}(n)$, or equivalently, if $n \in \mathsf{obj}(m)$.

The rules in the left table in Figure 2 inductively define $\mathsf{fn}(X)$, the set of *free names* in processes and in networks, considering $n@s$ to be n if $n = g$, and assuming that this operator extends to sequences and to sets of names. The case of the persistent receptor is, in many ways, analogous to the simple receptor. Hereafter, we omit the treatment of this case when it is analogous to that of the simple receptor.

Substitution. Channels and sites occupy different positions in structured names, so substitution is not a total function on names. We are mainly interested in its application to the operations performed within the calculus,

restricting our attention to three kinds of substitutions of m by n: *name replacement*, for *change of bound names*, if either $m, n \in \mathcal{C}, m, n \in \mathcal{S}$, or $m, n \in \mathcal{C}@s$ for some s; *name instantiation*, for *communication*, if $m \in \mathcal{C}$ and $n \in (\mathcal{C} \cup \mathcal{C}@\mathcal{S})$; *name translation*, for *migration*, if $m \in \mathcal{C}$ and $n \in m@\mathcal{S}$.

Name substitution must be defined in such a way that when applied to processes or to networks, the correspondence between their binders and bound names does not change. However, some binder properties which are true in calculi like λ or π — as $\mathsf{subj}(n) = \{n\} = \mathsf{obj}(n)$ — no longer stand in this setting, so one must generalize the old intuitions: (1) *A binder on a name n binds, in a process within its scope, the free occurrences of that name n.* But $a@s$ is bound in $(\nu\, s)\, a@s!\langle\rangle$, so we say more generally that the set of names bound by n is $\mathsf{subj}(n)$. (2) *If name n is changed by a substitution $\{m_2/m_1\}$ then $n = m_1$ and the result is m_2.* But s is free in $a@s!\langle\rangle$, so we say more generally that n is changed by $\{m_2/m_1\}$ if it is a subject of m_1, and the change affects components of n, namely elements of $\mathsf{obj}(n)$.

Substitution may be defined for $lsd\pi$ in the usual way if these generalizations are taken into account. Following Hindley and Seldin [5], a *substitution on names and entities* is a partial function $\mathcal{X}\{\mathcal{N}/\mathcal{N}\} \mapsto \mathcal{X}$, inductively defined by the rules in Figure 3.

The case $s[P]\{b@s/a@s\}$ incorporates the fact that while outsiders see free channels as being uniformly remote, *locally* they might take both the compound or the simple form. Therefore, when crossing the site boundary, this substitution must be unfolded in two. Let $\{\tilde{v}/\tilde{x}\}$ denote a set of simultaneous substitutions, defined as usual.

Structural congruence. We define a static binary relation on processes and networks — *structural congruence* — as the least congruence relation containing the rules in Figure 4.

Rule S-ALPHA uses an alpha-congruence relation, denoted $\equiv_\alpha$, defined in the usual way from the definition of substitution.

The rules in the top of Figure 4 (down to SP-GC are adapted from the standard rules of the π-calculus [6,7]. Differences reside mainly in the side conditions, resulting directly from the fact that name creation introduces free names, which are themselves subject to bindings. Rule SN-GC garbage collects inactive sites and channel restrictions. The same is true for SP-GC, which is intended for processes. Rule S-RESO controls the capture of sites in compound channel creations by site restrictions. However, two channel name creations may always commute.

The rules S-ROUT and S-SCOS) are inspired on those proposed by Hennessy and Riely for Dpi [4], but are adapted to a lexical scope regime: Rule SN-ROUT describes how processes within a site may be split or aggregated. Rule SN-SCOS depicts how a site boundary affects the view over a name restriction: outsiders always see channel creations as remote, so their compound and simple forms are indistinguishable (up to name capture) to them, while local viewers differentiate those two forms of local channel creation.

$$(n)\{m_2/m_1\} \stackrel{\text{def}}{=} \begin{cases} m_2 & \text{if } n = m_1 \\ a@m_2 & \text{if } n = a@m_1 \\ n & \text{if } m_1 \not\rightarrow n \end{cases}$$

$$\mathbf{0}\{m_2/m_1\} \stackrel{\text{def}}{=} \mathbf{0}$$

$$(X \mid Y)\{m_2/m_1\} \stackrel{\text{def}}{=} X\{m_2/m_1\} \mid Y\{m_2/m_1\}$$

$$((\nu\, n)\, X)\{m_2/m_1\} \stackrel{\text{def}}{=} \begin{cases} (\nu\, n)\, X & \text{if (1)} \\ (\nu\, n\{m_2/m_1\})X\{m_2/m_1\} & \text{if } \neg(1) \wedge (2) \\ (\nu\, n')X\{n'/n\}\{m_2/m_1\} & \text{if } \neg(1) \wedge \neg(2) \wedge (3) \end{cases}$$

$$(u?(\tilde{x})P)\{m_2/m_1\} \stackrel{\text{def}}{=} \begin{cases} u\{m_2/m_1\}?(\tilde{x})P & \text{if (4)} \\ u\{m_2/m_1\}?(\tilde{x})P\{m_2/m_1\} & \text{if } \neg(4) \wedge (5) \\ u\{m_2/m_1\}?(\tilde{x}')P\{y/x_i\}\{n/m\} & \text{if } \neg(4) \wedge \neg(5) \wedge (6) \end{cases}$$

$$(u!\langle v_1 \ldots v_n\rangle)\{m_2/m_1\} \stackrel{\text{def}}{=} u\{m_2/m_1\}!\langle v_1\{m_2/m_1\} \ldots v_n\{m_2/m_1\}\rangle$$

$$(s[P])\{m_2/m_1\} \stackrel{\text{def}}{=} \begin{cases} s[P\{b/a\}\{b@s/a@s\}] & \text{if } m_1 = a@s \wedge m_2 = b@s \\ s\{m_2/m_1\}[P\{m_2/m_1\}] & \text{otherwise} \end{cases}$$

1. $n \hookrightarrow m_1$
2. $n \not\rightarrow m_2 \vee m_1 \notin \mathsf{fn}(X)$
3. n' is fresh
4. $\exists_{x_i \in \{\tilde{x}\}}.x_i \hookrightarrow m_1$
5. $(\forall_{x_i \in \{\tilde{x}\}}.x_i \not\rightarrow m_2) \vee m_1 \notin \mathsf{fn}(P)$
6. y is fresh and $\exists_{x_i \in \{\tilde{x}\}}.\tilde{x}' = \tilde{x}\{y/x_i\}$

Fig. 3. Substitution on names, processes and networks

The rules SN-MIGO, SN-MIGI and SN-RMIGI are specific to a lexical scope regime of channels. They make use of the fact that a channel which is explicitly located in the site where it occurs is in fact a local channel, and so it might as well take its simple format. This law is only needed at communication time [1], so it regards only channels which appear in the top-level of a process.

Name translation. When migrating a process P previously running in r, the simple channels of P (which were once local) will become remote references. Therefore, they must be located accordingly — they become explicitly located at r — while all other channels remain unchanged.

[1] See rule RP-COMM in Figure 5.

$$
\begin{array}{lrl}
\text{[S-ALPHA]} & X \equiv Y & \text{if } X \equiv_\alpha Y \\
\text{[S-ASSO]} & ((X \mid Y) \mid Z) \equiv (X \mid (Y \mid Z)) & \\
\text{[S-COMM]} & (X \mid Y) \equiv (Y \mid X) & \\
\text{[S-NEUT]} & (X \mid \mathbf{0}) \equiv X & \\
\text{[S-SCOP]} & ((\nu\, n)\, X) \mid Y \equiv (\nu\, n)\, (X \mid Y) & \text{if } \mathsf{subj}(n) \cap \mathsf{fn}(Y) = \emptyset \\
\text{[S-RESO]} & (\nu\, n)\, (\nu\, m)\, X \equiv (\nu\, m)\, (\nu\, n)\, X & \text{if } n \not\rightarrow m \text{ and } m \not\rightarrow n \\
\text{[SP-GC]} & (\nu\, \widetilde{n})\, \mathbf{0} \equiv \mathbf{0} & \text{if } \mathsf{fn}((\nu\, \widetilde{n})\, \mathbf{0}) = \emptyset \\
\text{[SN-GC]} & (\nu\, \widetilde{n})\, s[\mathbf{0}] \equiv \mathbf{0} & \text{if } \mathsf{fn}((\nu\, \widetilde{n})\, s[\mathbf{0}]) = \emptyset \\
\text{[SN-ROUT]} & (s[P] \mid s[Q]) \equiv s[(P \mid Q)] & \\
\text{[SN-SCOS}_1\text{]} & (\nu\, g)\, s[P] \equiv s[(\nu\, g)\, P] & \text{if } g \not\rightarrow s \wedge g \notin \mathcal{C}_{@s} \\
\text{[SN-SCOS}_2\text{]} & (\nu\, a@s)\, s[P] \equiv s[(\nu\, a@s)\, P] & \text{if } a \notin \mathsf{fn}(P) \\
\text{[SN-SCOS}_3\text{]} & (\nu\, a@s)\, s[P] \equiv s[(\nu\, a)\, P] & \text{if } a@s \notin \mathsf{fn}(P) \\
\text{[SN-MIGO]} & s[a@s!\langle \tilde{v} \rangle] \equiv s[a!\langle \tilde{v} \rangle] & \\
\text{[SN-MIGI]} & s[a@s?(\tilde{x})P] \equiv s[a?(\tilde{x})P] & \\
\text{[SN-MIGR]} & s[a@s?*(\tilde{x})P] \equiv s[a?*(\tilde{x})P] &
\end{array}
$$

Fig. 4. Structural congruence on processes and networks

It should be clear that some compound channels will become local after migration. Their simplification to simple channels is only necessary at communication time, and is dealt with by the already mentioned structural congruence rules SN-MIGO, SN-MIGI and SN-RMIGI.

Reduction. The rules in Figure 5 inductively define the *reduction relation* on processes and networks.

Axiom RP-COMM, standard in the π-calculus [6,7], only applies to simple communication channels, i.e., local ones. Whichever the site the process occurs in, we know that it has allocated the communication channel. This is why we can say communication is local. Rule RP-RCOMM reveals that a replicated input is a persistent resource.

The axioms RN-MIGO and RN-MIGI, to which we add RN-RMIGI, are those proposed by Vasconcelos *et al.* for DiTyCO [13], and only make sense in a lexical scope setting. Whenever an output/input process is prefixed over a remote channel, we know it requires a remote resource to perform its communication. Therefore, in order to obtain the status of "local", the code simply

$$\text{[RP-COMM]} \quad a?(\tilde{x})P \mid a!\langle\tilde{v}\rangle \to P\{\tilde{v}/\tilde{x}\}$$

$$\text{[RP-RCOMM]} \quad a?*(\tilde{x})P \mid a!\langle\tilde{v}\rangle \to a?*(\tilde{x})P \mid P\{\tilde{v}/\tilde{x}\}$$

$$\text{[RN-MIGO]} \quad r[a@s!\langle\tilde{v}\rangle] \to s[(a@s!\langle\tilde{v}\rangle)\sigma_r] \qquad r \neq s$$

$$\text{[RN-MIGI]} \quad r[a@s?(\tilde{x})P] \to s[(a@s?(\tilde{x})P)\sigma_r] \quad r \neq s$$

$$\text{[RN-RMIGI]} \quad r[a@s?*(\tilde{x})P] \to s[(a@s?*(\tilde{x})P)\sigma_r] \quad r \neq s$$

$$\text{[RN-SITE]} \quad \frac{P \to Q}{s[P] \to s[Q]}$$

$$\text{[R-CONT]} \quad \frac{X \to Y}{E[X] \to E[Y]}$$

$$\text{[R-STR]} \quad \frac{X \equiv X' \quad X' \to Y' \quad Y' \equiv Y}{X \to Y}$$

$$E \quad ::= \quad [] \quad | \quad (E \mid X) \quad | \quad (\nu\, n)\, E$$

$$P\sigma_r \stackrel{\text{def}}{=} P\{a_1@r/a_1, \ldots, a_n@r/a_n\} \text{ where } \mathsf{fn}(P) \cap \mathcal{C} = \{a_1, \ldots, a_n\}$$

Fig. 5. Reduction rules

migrates to the site the communication channel belongs to (which is explicit in the name of the channel). A name translation is then performed upon the code by means of σ_r, so as to reflect the new site where it is running: channels that previously were simple (thus implicitly located) are now composed with the site from where the code came from.

It is important to notice that no new free name is created during reduction.

Proposition 3.1 (Reduction preserves free names) *If $X \to Y$, then $\mathsf{fn}(X) \supseteq \mathsf{fn}(Y)$.*

Dynamic linking. A channel which is created under an input prefix cannot be used until that input has been "consumed". In particular, the site where local channels are to be created may result from some computation [2]. We support this view with an example that shows how to create local channels "anywhere" in the network by using dynamic linking. Consider a server with address a at site s providing some application, while requiring some resources (say a new channel b). It is possible to define these resources as being local to

[2] Notice though, that once the local channel is created, the site it is associated to will remain fixed.

$$\text{TS-Chl}\ \Gamma \vdash a@s{:}\Gamma(s)(a) \qquad \text{TS-Chs}\ \Gamma \vdash a{:}\Gamma(\hbar)(a) \qquad \text{TS-Uni}\ \frac{\Gamma \vdash \widetilde{v_1{:}\gamma_1} \quad \Gamma \vdash \widetilde{v_2{:}\gamma_2}}{\Gamma \vdash v_1\widetilde{v_2{:}\gamma_1\gamma_2}}$$

$$\text{TP-Out}\ \frac{\Gamma \vdash \widetilde{v{:}\gamma} \quad \Gamma \vdash u{:}Ch(\widetilde{\gamma})}{\Gamma \vdash u!\langle\tilde{v}\rangle} \qquad \text{TP-Inp}\ \frac{\Gamma \vdash \widetilde{x{:}\gamma} \quad \Gamma \vdash u{:}Ch(\widetilde{\gamma}) \quad \Gamma \vdash P}{\Gamma \setminus \tilde{x}@\hbar \vdash u?(\tilde{x})P}$$

$$\text{TP-Ress}\ \frac{\Gamma \vdash P}{\Gamma \setminus a@\hbar \vdash (\nu\, a)\, P} \qquad \text{T-Resl}\ \frac{\Gamma \vdash X}{\Gamma \setminus a@s \vdash (\nu\, a@s)\, X}$$

$$\text{T-Resn}\ \frac{\Gamma \vdash X}{\Gamma \setminus s \vdash (\nu\, s)\, X} \qquad \text{T-Par}\ \frac{\Gamma \vdash X \quad \Gamma \vdash Y}{\Gamma \vdash (X \mid Y)}$$

$$\text{TN-Net}\ \frac{\Gamma \vdash P}{\Gamma\{s/\hbar\} \vdash s[P]} \qquad \text{T-Nil}\quad \Gamma \vdash \mathbf{0}$$

Fig. 6. Typing names, processes and networks

the site that downloads the application:

$$s[a?(x)x?()((\nu\, b)\, P)] \mid r[a@s!\langle c\rangle \mid c!\langle\rangle)] \mid t[a@s!\langle c\rangle \mid c!\langle\rangle)]\,.$$

In fact, the above process can reduce both to $r[(\nu\, b)\, P]$ or to $t[(\nu\, b)\, P]$ (if $x \notin \mathsf{fn}(P)$). By maintaining both simple and located forms of channels, we are able to express the creation of a local name (the above b) in a site which will be determined only at run time.

4 The type system

This section presents a type system for *lsd*π. The system is a straightforward extension of that for the simply typed π-calculus [12] and it is a simplified form of the one in Amadio *et al.* for $d\pi_1^r$ [1], adapted to deal with the lexical scope of channels. The main result of this section ensures the preservation of typability by reduction, a property usually know as *subject reduction*. From there it is simple to prove a type safety corollary, ensuring the absence of arity mismatch in communication.

Types for channels are those of the simply typed π-calculus, $Ch(\gamma_1, \ldots, \gamma_n)$, describing a channel capable of carrying a series of channels of types $\gamma_1, \ldots, \gamma_n$. Types for sites only capture the types of its free channels: if a_1 to a_n of types γ_1 to γ_n are the free channels of site s, then we assign to site s the type $\{a_1{:}\gamma_1, \ldots, a_n{:}\gamma_n\}$.

The types of $lsd\pi$ are a subset of those of $d\pi_1^r$: simply remove the located type — a form of existential type — unnecessary in this setting due to name instantiation. The type system of $lsd\pi$ is syntax-oriented, though it considers a special site — $\hbar$, for "here" — which designates the current site when typing the occurrences of simple channels in processes. Due to the motto '*simple channels are local*, "here" corresponds always to the same site. The "site variable" is instantiated at the network level, or disposed of if the simple channels is bound. At the end of the typing procedure, all channels are explicitly associated to one site.

Definition 4.1 (Types) *Consider the finite sets* $\{s_1, \ldots, s_n\} \subseteq \mathcal{S} \uplus \{\hbar\}$ *and* $\{a_1, \ldots, a_n\} \subseteq \mathcal{C}$ *, with* $n \geq 0$ *and where the designated elements of each sets are pairwise distinct.*

$$\begin{array}{rrcl} \textit{Channel types,} & \gamma & ::= & \mathrm{Ch}(\gamma_1, \ldots, \gamma_n) \\ \textit{Site types,} & \varphi & ::= & \{a_1{:}\gamma_1, \ldots, a_n{:}\gamma_n\} \\ \textit{Typings,} & \Gamma & ::= & \{s_1{:}\varphi_1, \ldots, s_n{:}\varphi_n\} \end{array}$$

A site type is a map from channels into channel types; a typing is a map from sites into site types. If Γ is a typing, then $\Gamma \setminus s$ removes s from the domain of Γ; $\Gamma \setminus a@s$ removes a from the domain of $\Gamma(s)$; and $\Gamma\{s/t\}$ replaces s by t in the domain of Γ, provided that channels common to t and to s have the same types. The *union of typing assumptions* $\Gamma + \Delta$ is defined pointwise, if for all $s \in \mathsf{dom}(\Gamma)$ and for all $a \in \Gamma(s) \cap \Delta(s)$ we have $\Gamma(s)(a) = \Delta(s)(a)$, as:

$$(\Gamma + \Delta)(s) \stackrel{\text{def}}{=} \begin{cases} \Gamma(s), & \text{if } s \in \mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Delta) \text{ or } \Gamma(s) = \Delta(s), \\ \Gamma(s) \cup \Delta(s), & \text{if } s \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) \text{ and } \Gamma(s)(a) = \Delta(s)(a), \\ \Delta(s), & \text{if } s \in \mathsf{dom}(\Delta) \setminus \mathsf{dom}(\Gamma). \end{cases}$$

$$\Gamma\{s/t\} \stackrel{\text{def}}{=} \Gamma \setminus t + \{s{:}\Gamma(t)\}$$

The rules in Figure 6 inductively define the type system of $lsd\pi$. It uses two kinds of judgments: $\Gamma \vdash \widetilde{v{:}\gamma}$ asserts that Γ types (simple and located) channels $\tilde{v}$ with types $\widetilde{\gamma}$, according to the typing assumption Γ; $\Gamma \vdash X$ says that process or network X conforms to the typing assumption Γ.

The special site $\hbar$ plays a very important role: it allows delaying decisions when typing processes. If in a process a channel with subject a occurs both in a simple and in a compound form, we assume the simple one located "*here*" (at $\hbar$), and the compound one located at s. Dpi type systems ([1,4]) commit too soon, assuming the same site. For instance, the following judgments hold.

$$\emptyset \vdash_s (\nu\, a@s)\, a!\langle\rangle \qquad \text{and} \qquad \emptyset \vdash_s (\nu\, a)\, a@s!\langle\rangle$$

These judgments, however, should not hold in our type system, since, contrary to what happens in Dpi, the binders do not capture the channel subject of the process. Therefore, the valid judgments are:

$$\{\hbar{:}\{a{:}Ch()\}\} \vdash (\nu\, a@s)\, a!\langle\rangle \qquad \text{and} \qquad \{s{:}\{a{:}Ch()\}\} \vdash (\nu\, a)\, a@s!\langle\rangle$$

This distinction is also useful when deciding if a process should be rejected: is $a@s?()P \mid a!\langle v\rangle$ a communication error? it depends on the site where the process is running and on the channels being the same. Thus, a process like $(\nu\, a@s)\,(a@s?()P \mid a!\langle v\rangle)$ is well behaved, and our type system accepts it[3]. In short, the special site $\hbar$ is the key for a lexically scoped type system.

The type system enjoys subject-reduction, a result which depends on the preservation of typability by appropriate substitutions.

Lemma 4.2 (Substitution lemma) *Let Υ be a sequence of name replacements or name instantiations. Then, $\Gamma \vdash X$ implies $\Gamma\Upsilon \vdash X\Upsilon$.*

Proof. By induction on the derivation of $\Gamma \vdash X$. It is useful to prove that $\Gamma \vdash \widetilde{v{:}\gamma}$ implies $\Gamma\Upsilon \vdash \widetilde{v\Upsilon{:}\gamma}$ by using induction on the length of Υ and $\tilde{v}$, and a case analysis on names. □

We show now that the type system enjoys subject-reduction. Recall that we omit the treatment of the replicated input, as it is very similar to the case of the simple input.

Theorem 4.3 (Subject reduction) *If $\Gamma \vdash X$ and $X \to Y$, then $\Gamma \vdash Y$.*

Proof. The proof consists of inductions on the derivations of the reduction step. As usual, we use a lemma stating that alpha and structural congruence also preserve typability. The base cases in the derivation of $P \to Q$ are the process axioms, for which we use Lemma 4.2 and typing rules T-PAR, TP-INPS, TP-OUTS and TP-RINPS. The base cases for the derivation of $X \to Y$ are the migration axioms. We use the result for processes, Lemma 4.2, the definition of the translation function, as well as typing rules TN-NET, TP-INPL, TP-OUTL and TP-RINP. The cases of the induction steps are straightforward. □

Type safety follows.

Theorem 4.4 (Type Safety) *If $N \equiv s[a!\langle\tilde{v}\rangle \mid a?(\tilde{x})P \mid M]$ and $\Gamma \vdash N$, then $|\tilde{v}| = |\tilde{x}|$, and similarly for the replicated input.*

5 Behavioral equivalence

The aim of this section is the behavioral characterization of a lexically scoped approach to distributed mobile calculi. We present a standard (early) labeled transition system, and based on it we define a standard asynchronous

[3] Dpi type systems reject it since the channel is the same.

α	$\mathsf{fn}(\alpha)$	$\mathsf{bn}(\alpha)$
τ	$\emptyset$	$\emptyset$
$u?\langle\tilde{v}\rangle$	$\{u, \tilde{v}\}$	$\emptyset$
$(\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle$	$\{u\} \cup \{\tilde{v}\} \setminus \mathsf{subj}(\widetilde{n})$	$\mathsf{subj}(\widetilde{n})$

Fig. 7. Free and bound names in actions

bisimulation. We then prove an important law of lexically scoped distributed settings: a process behavior depends on the names it uses, not on where it runs. Using it, we show how to simulate a general go primitive, and a code optimization for process migration.

Labeled transition system. Consider two kinds of actions: those of processes, which are naturally like π-process actions; and those of networks, obtained from processes' actions by explicitly locating all its channels. The following grammar defines the set of *actions*, considering $\forall n \in \{\widetilde{n}\}.n \not\rightarrow u$.

$$\alpha ::= \tau \;\mid\; u?\langle\tilde{v}\rangle \;\mid\; (\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle$$

The subject of an action, $\mathsf{subj}(\alpha)$, is a partial function that uses the homonym notion over names, defined in Figure 2:

$$\mathsf{subj}(u?\langle\tilde{v}\rangle) = \mathsf{subj}((\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle) \stackrel{\text{def}}{=} \mathsf{subj}(u)\,.$$

Extending this notion to sequences of names in the expected way, the rules in the Figure 7 inductively define the sets of *free names*, $\mathsf{fn}(\alpha)$, and of *bound names*, $\mathsf{bn}(\alpha)$, in a action α. Let $\mathsf{names}(\alpha) \stackrel{\text{def}}{=} \mathsf{fn}(\alpha) \cup \mathsf{bn}(\alpha)$.

The channel translation function σ_s used in the migration axioms of the reduction relation is also used to locate all channels of an action.

$$\alpha\sigma_s \stackrel{\text{def}}{=} \begin{cases} \tau, & \text{if } \alpha = \tau\,; \\ u@s?\langle\tilde{v}@s\rangle, & \text{if } \alpha = u?\langle\tilde{v}\rangle\,; \\ (\nu\,\widetilde{n}@s)\,u@s!\langle\tilde{v}@s\rangle, & \text{if } \alpha = (\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle\,. \end{cases}$$

The rules in the Figure 8, together with rules symmetric to R-PAR and RP-COM, inductively define an *early labeled transition system* for $lsd\pi$. The system is standard, apart from rule RN-SITE that makes the location of all channels explicit for the network level.

Asynchronous bisimulation. We adapt the standard definition of an asynchronous bisimulation to $lsd\pi$ networks (cf. Sangiorgi and Walker [11]).

Definition 5.1 (Asynchronous bisimulation) *A symmetric binary relation $\mathcal{R}$ on networks is a* strong bisimulation*, or simply a bisimulation if, whenever $N\mathcal{R}M$:*

$$\text{[LRP-OUT]} \quad a!\langle\tilde{v}\rangle \xrightarrow{a!\langle\tilde{v}\rangle} \mathbf{0}$$

$$\text{[LRP-INP]} \quad a?(\tilde{x})P \xrightarrow{a?\langle\tilde{v}\rangle} P\{\tilde{v}/\tilde{x}\}$$

$$\text{[LRN-MIGO]} \quad r[a@s!\langle\tilde{v}\rangle] \xrightarrow{\tau} s[(a@s!\langle\tilde{v}\rangle)\sigma_r] \qquad r \neq s$$

$$\text{[LRN-MIGI]} \quad r[a@s?(\tilde{x})P] \xrightarrow{\tau} s[(a@s?(\tilde{x})P)\sigma_r] \qquad r \neq s$$

$$\text{[LRN-RMIGI]} \quad r[a@s?*(\tilde{x})P] \xrightarrow{\tau} s[(a@s?*(\tilde{x})P)\sigma_r] \qquad r \neq s$$

$$\text{[LRP-RCOMM]} \quad \frac{a?(\tilde{x})P \xrightarrow{\alpha} Q}{a?*(\tilde{x})P \xrightarrow{\alpha} a?*(\tilde{x})P \mid Q}$$

$$\text{[LRP-COMM]} \quad \frac{P \xrightarrow{(\nu\,\widetilde{n})\,a!\langle\tilde{v}\rangle} P' \quad Q \xrightarrow{a?\langle\tilde{v}\rangle} Q'}{P \mid Q \xrightarrow{\tau} (\nu\,\widetilde{n})\,(P' \mid Q')} \quad \mathsf{subj}(\widetilde{n}) \cap \mathsf{fn}(Q) = \emptyset$$

$$\text{[LR-RES]} \quad \frac{X \xrightarrow{\alpha} X'}{(\nu\,n)\,X \xrightarrow{\alpha} (\nu\,n)\,X'} \qquad \mathsf{subj}(n) \notin \mathsf{names}(\alpha)$$

$$\text{[LR-EXT]} \quad \frac{X \xrightarrow{\alpha} X'}{(\nu\,n)\,X \xrightarrow{(\nu\,n)\,\alpha} X'} \qquad \mathsf{subj}(n) \in \mathsf{fn}(\alpha) \setminus \mathsf{subj}(\alpha)$$

$$\text{[LR-PAR]} \quad \frac{X \xrightarrow{\alpha} X'}{X \mid Y \xrightarrow{\alpha} X' \mid Y} \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Y) = \emptyset$$

$$\text{[LR-ALPHA]} \quad \frac{X \equiv_\alpha X' \quad X' \xrightarrow{\alpha} Y' \quad Y' \equiv_\alpha Y}{X \xrightarrow{\alpha} Y} \qquad \text{[LRN-SITE]} \quad \frac{P \xrightarrow{\alpha} Q}{s[P] \xrightarrow{\alpha\sigma_s} s[Q]}$$

Fig. 8. Early labeled transition system of *lsdπ*

(i) $N \xrightarrow{\tau} N'$ *implies* $\exists M'$ $(M \xrightarrow{\tau} M'$ *and* $N'\mathcal{R}M')$.

(ii) $N \xrightarrow{((\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle)\sigma_s} N'$ *with* $\mathsf{subj}(\widetilde{n}\sigma_s) \cap \mathsf{fn}(M) = \emptyset$ *implies* $\exists M'$ $(M \xrightarrow{((\nu\,\widetilde{n})\,u!\langle\tilde{v}\rangle)\sigma_s} M' \wedge N'\mathcal{R}M')$.

(iii) $N \xrightarrow{(u?\langle\tilde{v}\rangle)\sigma_s} N'$ *implies* $\exists M'$ $((M \xrightarrow{(u?\langle\tilde{v}\rangle)\sigma_s} M'$ *and* $N'\mathcal{R}M') \vee (M \xrightarrow{\tau} M' \wedge N'\mathcal{R}(M' \mid s[u!\langle\tilde{v}\rangle]))$.

We denote by $\sim$ *the largest bisimulation. Two networks* N *and* M *are* strongly bisimilar*, if there is a strong bisimulation* $\mathcal{R}$ *such that* $N\mathcal{R}M$.

The following proposition is a basic property of a bisimilarity relation.

Proposition 5.2 *Strong bisimilarity contains the structural congruence relation and is a congruence relation.*

The lexical scoping law. The main result of this section is an equivalence law capturing the lexical scope nature of *lsdπ*: a process behaves similarly when running at site s or at site r, as long as simple channels are instantiated

according to the environment they were meant for: that is the purpose of the translation function σ_s.

We say that P *is absolute* if $\mathsf{fn}(P) \cap \mathcal{C} = \emptyset$, i.e., if P has only located channels. A simple channel is *relative* to the site where it appears. Note that $P\sigma_s$ is an absolute process, irrespective of s. The following result says that the name translation that occurs at migration time does not change the behavior of the network, for it merely instantiates the simple local channels, which are relative to the site they appear in.

Proposition 5.3 $s[P] \sim s[P\sigma_s]$.

This law lies at the basis of several properties of *lsd*π networks. To show some of those properties, we introduce a weak form of bisimulation. Let a weak transition be $\stackrel{\alpha}{\Rightarrow} \stackrel{\text{abv}}{=} \stackrel{\tau}{\rightarrow}^* \stackrel{\alpha}{\rightarrow} \stackrel{\tau}{\rightarrow}^*$, if $\alpha \neq \tau$, and let $\stackrel{\tau}{\Rightarrow} \stackrel{\text{abv}}{=} \stackrel{\tau}{\rightarrow}^*$. Replacing everywhere in the Definition 5.1, strong transitions with weak transitions, one gets the notion of *weak bisimulation*. The largest weak bisimulation is weak bisimilarity, denoted by $\approx$, which is also a congruence.

The behavior of an absolute process depends on the names it uses, not on where it runs.

Theorem 5.4 *If P is absolute, then $s[P] \approx r[P]$.*

Migration does not change the behavior of the process; it simply advances computation.

Corollary 5.5 $s[P] \approx r[P\sigma_s]$

Assume a new production $\mathsf{go}\, r.P$ in the grammar of processes, which migrates a process P from site s to site r, according to the axiom.

$$s[\mathsf{go}\, r.P] \stackrel{\tau}{\rightarrow} r[P\sigma_s]$$

The results below are easy to establish using the previous laws.

Proposition 5.6

(i) $s[\mathsf{go}\, r.P] \approx s[(\nu\, a@r)\,(a@r!\langle\rangle \mid a@r?()P)]$, *where* $a@r \notin \mathsf{fn}(P)$.

(ii) $s[a@r?(\tilde{x})P] \approx s[(\nu\, c)\,(a@r?(\tilde{x})c!\langle\tilde{x}\rangle \mid c?(\tilde{y})P)]$, *where* $|\tilde{x}|=|\tilde{y}|$ *and* $c \notin \mathsf{fn}(P)$.

These last properties are examples of possible applications of the basic law in proving the correctness of encodings or of code optimizations.

6 Conclusions and future work

This paper presents *lsd*π, a calculus for distribution and mobility which proposes a lexical scope discipline for channels as an alternative to the established approach that assumes all channels as global. We have contributed with the theoretical treatment, stemming from the general lexical scope principles. It was not a trivial work, since basic aspects like binder properties, ranging from the definition of free names to its coherence with the operational semantics,

had to be reconsidered and redesigned. The result of this task is presented in a technical report [10], but here we have simplified the technicalities significantly: the withdrawal of syntactic restrictions and the pruning of the name translation associated to the migration primitive introduced in [13].

We have shown that the premise under which channels are the resources of sites is tightly weaved in the semantics of the language, and hope to have given elucidative examples and intuitions. Finally, a further evidence of the potentialities of the calculus is an equivalence law stating that a process behavior depends on the names it uses, not on where it runs. We foresee a wide range of applications of the law, but herein show only two of them: on comparing the expressive power of distributed calculi; and on designing code optimizations.

We find three main areas of future work to elaborate on the foundations presented here.

(i) To carry on work on the comparison between a lexical scope approach and an approach that considers channels as global network resources. The closest calculus to $lsd\pi$ is $d\pi_1^r$ (an asynchronous and receptive version of Dpi), but still several differences in the design choices of the calculi lead to technical difficulties in obtaining an absolute comparison. Therefore, we aim at comparing the *consequences* of adopting one scheme or another, on two of which we have ongoing work.

Sites as first class citizens. We have confronted ourselves with the question *Why pass sites?*, for it seems that lexical scope obviates the need of introducing such feature — instead of revealing the name of a site, giving in fact access to all of its resources (current and future), one can send the subset of its channels, to whom one intends to give access (like Dpi does via typing). We believe this choice does not result in serious loss of expressiveness, and is a research topic to pursue.

Migration mechanisms. While Dpi uses an explicit migration primitive, in $lsd\pi$ migration is triggered by the necessity of using a remote channel. We have presented a simulation of the migration primitive in $lsd\pi$, although the other direction has not yet been studied.

(ii) We believe lexically scoped distribution can be further shown worthwhile in various crucial subjects like implementation, security, and programming design. In particular, we envisage to control unrestricted migration (cf. [4]), and are currently working on a type system for doing that. We intend to investigate the security advantages of not passing sites, for we expect this choice to allow us to obtain granularity in the control of the information disclosed by a site at the language level.

(iii) At a time where effort is being put in the establishment of a common core programming model for global computation, we do not seek to present a complete calculus that would satisfy all the desirable requirements for such a challenging goal. Instead, we aim at a simple calculus which con-

centrates on the study of the name restriction discipline, arguing that lexically scoped distribution is a good option, since it allows to treat channels as the resources of locations. The work in this paper can constitute a foundation for generalizations and variations of the language, paving the way for further research making use of this discipline.

Acknowledgement

This work was partially supported by the Portuguese Fundação para a Ciência e a Tecnologia (via CLC, the project MIMO, POSI/CHS/39789/2001, and scholarships POCTI/SFRH/BPD/6782/2001 and POSI/SFRH/BD/7100/20-01), by the EU FEDER (via CLC) and the EU IST proactive initiative FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100). We thank Gérard Boudol, Ilaria Castellani, Matthew Hennessy and Francisco Martins, as well as the anonymous referees, for their comments.

References

[1] Roberto M. Amadio, Gérard Boudol, and Cédric Lhoussaine. The receptive distributed π-calculus. Rapport de Recherche 4080, INRIA Sophia-Antipolis, 2000.

[2] Luca Cardelli. A language with distributed scope. In ACM, editor, *POPL'95: 22nd Annual ACM Symposium on Principles of Programming Languages (San Francisco, CA, U.S.A.)*, pages 286–297. ACM Press, 1995.

[3] Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In Andrew Gordon, editor, *FOSSACS'03: 6th International Conference on Foundations of Software Science and Computation Structures (Warsaw, Poland)*, volume 2620 of *Lecture Notes in Computer Science*, pages 282–298. Springer-Verlag, 2003. Available as Technical Report COGS 2002:1, University of Sussex, U. K., 2002.

[4] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.

[5] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[6] Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, U. K., 1991.

[7] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992. Available as Technical Reports ECS-LFCS-89-85 and ECS-LFCS-89-86, University of Edinburgh, U. K., 1989.

[8] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[9] Rocco De Nicola, GianLuigi Ferrari, Rosario Pugliese, and Betti Veneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[10] António Ravara, Ana G. Matos, Vasco T. Vasconcelos, and Luís Lopes. A lexically scoped distributed π-calculus. DI/FCUL TR 02–4, Department of Computer Science, University of Lisbon, 2002.

[11] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[12] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In Eike Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 524–538. Springer-Verlag, 1993.

[13] Vasco T. Vasconcelos, Luís Lopes, and Fernando Silva. Distribution and mobility with lexical scoping in process calculi. In *HLCL'98*, volume 16 (3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

URL: http://www.elsevier.nl/locate/entcs/volume85.html 21 pages

A Calculus of Kells

Jean-Bernard Stefani [a]

[a] *INRIA Rhône-Alpes,*
655, Avenue de l'Europe, 38334 Saint-Ismier Cedex, France
E-mail: jean-bernard.stefani@inrialpes.fr

Abstract

This paper introduces the Kell calculus, a new process calculus that retains the original insights of the M-calculus (local actions, higher-order processes and programmable membranes) in a much simpler setting. The calculus is shown expressive enough to provide a direct encoding of several recent distributed process calculi such as Mobile Ambients and the Distributed Join calculus.

1 Introduction

The calculus of Mobile Ambients [5] has received much attention in the past five years, as witnessed by the numerous variants that have been proposed to overcome some of its perceived deficiencies: Safe Ambients (SA) [12], Safe Ambients with passwords [14], Boxed Ambients (BA) [3], Controlled Ambients (CA) [20], New Boxed Ambients (NBA) [4], Ambients with process migration ($\mathbf{M}^3$) [7].

Mobile Ambients, unfortunately, are difficult to implement in a distributed setting. Consider, for instance, the reduction rule associated with the `in` capability in the original Mobile Ambients:

$$n[\texttt{in}\, n.P \mid Q] \mid m[R] \rightarrow m[R \mid n[P \mid Q]]$$

This rule essentially mandates a rendez-vous between ambient n and ambient m. Thus, a distributed implementation of this rule, i.e. one where ambient n and ambient m are located on different physical sites, would require a distributed synchronization between the two sites. The inherent complexity of the required distributed synchronization has been made clear in the Distributed Join calculus implementation of Mobile Ambients reported in [10].

Part of the difficulty in implementing Mobile Ambients is related to the presence of "grave interferences", as explained in [12]. However, even with variants of Mobile Ambients with co-capabilities and a type system ensuring ambient single-threadedness (i.e. ensuring that at any one time there is at most one process inside an ambient that carries a capability), realizing ambient migration as authorized e.g. by the Safe Ambients `in` primitive

$$n[\texttt{in}\, m.P_1 \mid P_2] \mid m[\overline{\texttt{in}}.Q_1 \mid Q_2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2]$$

still requires a rendez-vous between ambients.

This is illustrated by the Safe Ambients abstract machine, called PAN, described in [16], which requires a 2-phase protocol involving ambients n and m above, together with their parent ambient to implement the `in` and `out` moves.

Interestingly, the PAN abstract machine is further simplified by adopting an unconventional interpretation: ambients are considered to represent only logical loci of computation, and not physical locations. Each ambient is mapped to a physical location but the `in` and `out` primitives do not modify the physical location of ambients. Instead, the `open` primitive, as a side-effect, modifies the physical location of processes running inside the ambient to be dissolved. With this interpretation, of course, ambients cease to be meaningful abstractions for the control of the physical distribution of computations.

The problem with the Mobile Ambient primitives is not so much that they are difficult to implement in a distributed setting, but that they provide the only means for communication between remote ambients. This, in turn, means that a simple message exchange between remote ambients must bear the cost of a distributed synchronization. This is clearly not acceptable: there are many useful applications that require only simple asynchronous point-to-point message exchanges, and relying on Ambient-like primitives for remote communication would result in a heavy performance loss for these applications. As a minimum, therefore, one should look for a programming model where costly migration primitives coexist with simple asynchronous message exchange for remote communications. Boxed Ambients (BA) and their NBA adopt this approach. Communication in BA or NBA is synchronous but one can argue that it is in fact a form of local communication since it only takes place between an ambient and a process located in its parent ambient. Thus, communication between two remote ambients, i.e. siblings located in a 'network ambient', necessarily involves two different communication events (i.e. an emitting event at the ambient that originates the communication and a receiving event at the ambient that receives the communication). The Distributed Join calculus [9] makes the same separation between remote communications and locality migration, which is provided by the `go` primitive and also involves some form of distributed rendez-vous to be faithfully implemented.

Still, the distributed synchronization implied by mobility primitives raises important issues. In a distributed setting, failures are inevitable, be they permanent or transient, network or site failures. Taking into account such failures would require, as a minimum, turning mobility primitives into abortable transactions, thus preserving their atomicity but making explicit their behavior in presence of failure. This, in turn, suggest that it would be useful to split up ambient migration primitives, especially the `in` primitive of Mobile Ambients or the `enter` primitive of NBA, into finer grained primitives whose implementation need not rely on some distributed synchronization. To illustrate, one could think of splitting the Mobile Ambients `in` primitive into a pair of primitives `move` and `enter` whose behavior would be given by the following reduction rules (we use co-capabilities and passwords, as in

the NBA calculus):

$$n[\texttt{move}\langle m, h\rangle.P \mid Q] \mid \overline{\texttt{move}}(x, y).R \rightarrow \texttt{enter}\langle n, m, h, P, Q\rangle \mid R\{m/x, h/y\}$$
$$\texttt{enter}\langle n, m, h, P, Q\rangle \mid m[\overline{\texttt{enter}}(x, h).S \mid T] \rightarrow m[S\{x/n\} \mid T \mid n[P \mid Q]]$$

However, we do not pursue that approach here for several reasons. First, one may envisage further extensions allowing for more sophisticated authentication schemes, or dynamic security checks (e.g. additional parameters allowing for proof-carrying code schemes). Second, several questions remain concerning migration primitives and their combination. For instance, should we go for communications à la Boxed Ambients or should we consider instead to split up the migration primitives such as `to` migration primitive in the $\mathbf{M}^3$ calculus, yielding a form of communication similar to Dπ [11] or Nomadic Pict [22], where communication is a side-effect of process migration ? Should we allow for more objective forms of migration to reflect control that ambients can exercize on their content ?

The possible variants seem endless. This is why we follow instead the lead of higher-order calculi such as $D\lambda\pi$ [23] and the M-calculus [18], where process migration is a side-effect of higher-order communication. Indeed, as demonstrated in the M-calculus, higher-order communication, coupled with programmable localities, provides the means to model different forms of migration protocols, and different forms of locality semantics. The M-calculus avoids embedding predefined choices concerning migration primitives and their interplay. Instead, these choices can be defined, within the calculus itself, by programming the appropriate behavior in locality "membranes" (the control part P of an M-calculus locality $a(P)[Q]$). The M-calculus, however, may appear as rather complex, especially compared to Mobile Ambients. In particular, its operational semantics features several so-called routing rules which it would be interesting to reduce to a few simple cases.

The calculus we introduce in this paper is an attempt to define a calculus with process migration that avoids the need for distributed synchronization, while preserving the simplicity of Mobile Ambients and retaining the basic insights of the M-calculus: migration as higher-order communication, programmable locality "membranes". We call this new calculus the Kell calculus (the word "kell" is a variation on the word "cell", and denotes a locality or locus of computation). This calculus constitutes a direct extension of the asynchronous higher-order π-calculus with hierarchical localities.

This paper is organized as follows. Section 2 informally introduces the main constructs of the Kell calculus, together with several examples. Section 3 gives the syntax and operational semantics of the calculus. Section 4 presents several encodings of known process calculi, thus demonstrating the expressive power of the Kell calculus. Section 5 concludes the paper with a discussion of related work and of directions for further research.

2 Introducing the Kell calculus

The Kell calculus is in fact a family of calculi that share the same constructs and that differ only in the language of message patterns used in triggers (see below). In this paper we present an element of this famly that enjoys a very simple pattern language. In this section, we present informally the different constructs of the calculus.

The core of the calculus is the asynchronous higher-order π-calculus. Among the basic constructs of the calculus we thus find:

- the *null* process, $\mathbf{0}$; process *variables*, x;
- the *restriction*, $\nu a.P$, where a is a name, P is an arbitrary Kell calculus process, and ν is a binding operator;
- the *parallel* composition, $P \mid Q$;
- *messages* of the form, $a\langle \boldsymbol{u} \rangle$, where a is a name, $\boldsymbol{u}$ is a vector of elements u, and where each element u can be either a name or a process.
- *triggers*, or receivers, of the form $\xi \triangleright P$, where ξ is a message pattern (analogous to the join patterns in the Join calculus) and P is an arbitrary kell calculus process.

To this higher-order π-calculus core, we add just one construct, the kell construct, $a[P]$, which is used to localize the execution of a process P at location (we say "kell") a.

In this paper, patterns are given by the following grammar:

$$\xi ::= J \quad | \quad J \mid a[x]$$
$$J ::= a\langle \boldsymbol{w} \rangle \quad | \quad a\langle \boldsymbol{w} \rangle^{\uparrow} \quad | \quad a\langle \boldsymbol{w} \rangle_{\downarrow} \quad | \quad J \mid J$$

where $\boldsymbol{w}$ is a vector of elements w, which can be either a name, a name variable of the form (b), where b is a name, or a process variable x. Name variables and process variables are of course bound in patterns and their scope extend to the process of the right-hand side of the trigger sign $\triangleright$.

In the Kell calculus, computing actions can take four simple forms, illustrated below:

(i) Receipt of a local message, as in the reduction below, where a message, $a\langle Q \rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x \rangle \triangleright P$:

$$a\langle Q \rangle \mid (a\langle x \rangle \triangleright P) \;\rightarrow\; P\{Q/x\}$$

(ii) Receipt of a message originated from the environment of a kell, as in the reduction below, where a message, $a\langle Q \rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x \rangle^{\uparrow} \triangleright P$, located in kell b :

$$a\langle Q \rangle \mid b[a\langle x \rangle^{\uparrow} \triangleright P] \;\rightarrow\; b[P\{Q/x\}]$$

In pattern $a\langle x \rangle^{\uparrow}$, the up arrow $^{\uparrow}$ denotes a message that should come from the outside of the immediately enclosing kell.

(iii) Receipt of a message originated from a sub-kell, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, and coming from sub-kell b, is received by the trigger $a\langle x\rangle_{\downarrow} \triangleright P$, located in the parent kell of kell b:

$$(a\langle x\rangle_{\downarrow} \triangleright P) \mid b[a\langle Q\rangle \mid R] \;\rightarrow\; P\{Q/x\} \mid b[R]$$

(iv) Passivation of a kell, as in the reduction below, where the sub-kell named a is destroyed, and the process Q it contains is sent in a message on port b:

$$a[Q] \mid (a[x] \triangleright b\langle x\rangle) \;\rightarrow\; b\langle Q\rangle$$

Actions of the form (i) above are standard π-calculus actions. Actions of the form (ii) and (iii) are just extensions of the message receipt action of the π-calculus to the case of triggers located inside a kell. They can be compared to the communication actions in the Seal calculus [6] and in the Boxed Ambients calculus.

Actions of the form (iv) are characteristic of the Kell calculus. They allow the environment of a kell to exercize control over the execution of the process located inside a kell. Consider for instance the process P, defined as $P \triangleq \mathtt{stop}\langle(b)\rangle \triangleright (b[x] \triangleright \mathbf{0})$. We have the following reductions:

$$\mathtt{stop}\langle a\rangle \mid P \mid a[Q] \;\rightarrow\; (a[x] \triangleright \mathbf{0}) \mid a[Q] \;\rightarrow\; \mathbf{0}$$

In this example, the environment of kell a collects it, thus destroying it and the process Q that it holds. Other forms of control over process execution are possible. Consider the process P and R defined as:

$$P \triangleq \mathtt{suspend}\langle a\rangle \triangleright (a[x] \triangleright R \mid a\langle x\rangle) \qquad R \triangleq \mathtt{resume}\langle a\rangle \triangleright (a\langle x\rangle \triangleright a[x])$$

We have the following reductions:

$$\begin{aligned}\mathtt{resume}\langle a\rangle \mid \mathtt{suspend}\langle a\rangle \mid P \mid a[Q] &\rightarrow \mathtt{resume}\langle a\rangle \mid (a[x] \triangleright R \mid a\langle x\rangle) \mid a[Q] \\ &\rightarrow \mathtt{resume}\langle a\rangle \mid R \mid a\langle Q\rangle \\ &\rightarrow (a\langle x\rangle \triangleright a[x]) \mid a\langle Q\rangle \\ &\rightarrow a[Q]\end{aligned}$$

In this example, the environment of kell a first suspends its execution (there is no evaluation under a $a\langle.\rangle$ context), and then resumes it (processes can execute under a $a[.]$ context).

The calculus has no primitive for recursion, such as a replication operator $!P$. The reason is that, because of its higher-order character, it is possible to define *receptive triggers*, i.e. triggers that are preserved during a reduction (much like definitions in the Join calculus). Let $t \in \mathsf{N}$, ξ and P be such that t does not occur in ξ or P. In a manner reminiscent of the fixed point operator defined in CHOCS [21], we define $\xi \diamond P$ by:

$$\begin{aligned}\xi \diamond P &\triangleq \nu t.\, Y(P,\xi,t) \mid t\langle Y(P,\xi,t)\rangle \\ Y(P,\xi,t) &\triangleq \xi \mid t\langle y\rangle \;\triangleright\; P \mid y \mid t\langle y\rangle\end{aligned}$$

It is easy to see with the rules of reduction given in Section 3.2 that if $M \mid (\xi \triangleright P) \rightarrow P\theta$, where θ is a substitution, then we have $M \mid (\xi \diamond P) \rightarrow (\xi \diamond P) \mid P\theta$.

The higher-order nature of the calculus, together with the passivation construct, allows the definition of different forms of programmable "membranes" around kells. Here are some simple examples. Assume that all triggers in process K are of the form $a\langle x\rangle \triangleright \ldots$, and that all messages emitted towards the environment of kell a are of the form $\mathtt{m}\langle b, \ldots\rangle$, where b is a target kell. We can define around kell a the following membranes:

- Transparent membrane: Let $M \triangleq (a\langle x\rangle^{\uparrow} \diamond a\langle x\rangle) \mid (\mathtt{m}\langle (b), x\rangle_{\downarrow} \diamond b\langle x\rangle)$. Then $c[M \mid a[K]]$ defines a membrane around kell a that does nothing (it just allows messages destined to, or emitted by, a to be transmitted without any control).
- Intercepting membrane: Let $M \triangleq (a\langle x\rangle^{\uparrow} \diamond P(x)) \mid (\mathtt{m}\langle b, y\rangle_{\downarrow} \diamond Q(b, y))$. Then $c[M \mid a\langle K\rangle]$ defines a membrane around kell a that triggers behaviour $P(x)$ when a message $a\langle x\rangle$ seeks to enter kell a, and behaviour $Q(b, y)$ when a message $\mathtt{m}\langle b, y\rangle$ seeks to leave kell a. Notice how this allows the definition of wrappers with pre and post-handling of messages.
- Migration membrane: Let

$$M \triangleq (\mathtt{enter}\langle a, x\rangle^{\uparrow} \mid a[y] \diamond a[y \mid x]) \mid (\mathtt{go}\langle (b)\rangle_{\downarrow} \mid a[y] \diamond \mathtt{enter}\langle b, a[y]\rangle)$$

  Then $c[M \mid a[K]]$ defines a membrane around kell a that allows new processes to enter kell a via the `enter` operation, and allows kell a to move to a different kell b via the `go` operation. Compare these operations with the migration primitives of Mobile Ambients, and the `go` primitive of the Distributed Join calculus.
- Localities with failures: Let

$$M \triangleq (\mathtt{stop}\langle a\rangle^{\uparrow} \mid a[y] \triangleright S) \mid (\mathtt{ping}\langle a, (r)\rangle^{\uparrow} \diamond \mathtt{m}\langle r, \mathtt{up}\rangle)$$
$$S \triangleq \mathtt{ping}\langle a, (r)\rangle^{\uparrow} \diamond \mathtt{m}\langle r, \mathtt{down}\rangle$$

  Then, $c[M \mid a[K]]$ defines a membrane around kell a that allows to `stop` the execution of locality a (simulating a failure in a fail-stop model), and that implements a simple failure detector via the `ping` operation. Compare these operations with the π_{1l}-calculus [1] or the Distributed Join calculus models of failures.
- Localities with fail-stop failures and recovery: Let

$$M \triangleq (\mathtt{stop}\langle a\rangle \mid a[y] \mid c \diamond b\langle y\rangle \mid f) \mid (\mathtt{ping}\langle a, (r)\rangle \mid c \diamond c \mid \mathtt{m}\langle r, \mathtt{up}\rangle)$$
$$S \triangleq (\mathtt{ping}\langle a, (r)\rangle \mid f \diamond f \mid \mathtt{m}\langle r, \mathtt{down}\rangle) \mid (\mathtt{recover}\langle a\rangle \mid b\langle y\rangle \mid f \diamond c \mid a[y])$$

  Then, $e[\nu c\, f.\, c \mid M \mid S \mid a[K]]$ defines a membrane around kell a that models fail-stop failures of kell a together with a simple failure detector and the possibility of recovery.

These different examples can be coded very similarly in the M-calculus, illustrating the fact that the Kell calculus retains the ability of the M-calculus to define localities with different semantics. The first two examples, when involving only first-order communication, can be coded analogously in variants of Boxed Ambients and in the Seal calculus. The third example can be coded in the Seal calculus and in Ambients calculi (in the latter, by coding the objective move into a protocol

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid a\langle \boldsymbol{u} \rangle \mid P \mid P \mid a[P] \\
\xi & ::= & J \mid J \mid a[x] \\
J & ::= & \epsilon \mid a\langle \boldsymbol{w} \rangle \mid a\langle \boldsymbol{w} \rangle^{\uparrow} \mid a\langle \boldsymbol{w} \rangle_{\downarrow} \mid J \mid J \\
u & ::= & a \mid P \\
w & ::= & a \mid (a) \mid x \\
a & \in & \mathsf{N} \\
x & \in & \mathsf{V}
\end{array}
$$

Fig. 1. Syntax of the Kell Calculus

activating subjective migration). The fourth example can be coded in the Seal calculus. The fifth example can be only partially simulated in the Seal calculus since one can isolate, duplicate or destroy a seal, but one cannot freeze its execution.

Actions in the Kell calculus obey a locality principle that states that any computing action should involve only one locality at a time (and its environment, when considering crossing locality boundaries). In particular, notice that there are no reductions in the calculus that, similar to the Mobile Ambients `in` move, would involve two adjacent kells. In particular, we *do not* have reductions of the following forms:

$$a[\mathtt{in}\langle Q \rangle] \mid b[\mathtt{in}\langle x \rangle \triangleright x] \;\rightarrow\; a[\mathbf{0}] \mid b[Q] \qquad a[Q] \mid b[a[x] \triangleright a[x]] \;\rightarrow\; b[a[Q]]$$

3 The Kell calculus: syntax and semantics

3.1 Syntax

The syntax of the Kell calculus is given in Figure 1. We assume an infinite set N of *names*, and an infinite set V of *process variables*, such that $\mathsf{N} \cap \mathsf{V} = \emptyset$. We let a, b, n, m and their decorated variants range over N; and p, q, x, y range over V. The set L of *identifiers* is defined as $\mathsf{L} = \mathsf{N} \cup \mathsf{V}$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q and their decorated variants range over processes. We call *kell* a process of the form $a[u]$. The name a in a kell $a[u]$ is called the name of the kell. In a kell of the form $a[\ldots \mid a_j[u_j] \mid \ldots \mid Q_k \mid \ldots]$ we call *subkells* the processes $a_j[u_j]$. We call *message* a process of the form $a\langle \boldsymbol{u} \rangle$. We let M, N and their decorated variants range over messages and parallel composition of messages. We call *trigger* a process of the form $\xi \triangleright P$, where ξ is a *receipt pattern* (or *pattern*, for short). A pattern can be a *join pattern* J, or a *control pattern* of the form $J \mid a[x]$, in which the join pattern J may be empty (i.e. $J = \epsilon$ – we set $J \mid \epsilon = \epsilon \mid J = J$).

In a term $\nu a.P$, the scope extends as far to the right as possible. We use $\boldsymbol{u}$ to denote finite vectors $(u_1, \ldots, u_q)$ (vectors can be empty; the empty vector is noted $\langle\rangle$). Abusing the notation, we equate $\boldsymbol{u} = (u_1, \ldots, u_n)$ with the word $u_1 \ldots u_n$ and the set $\{u_1, \ldots, u_n\}$. We note $|\boldsymbol{u}|$ the length n of a vector $\boldsymbol{u} = (u_1, \ldots, u_n)$. We

$$\mathbf{C} ::= \cdot \mid \xi \triangleright \mathbf{C} \mid \nu a.\mathbf{C} \mid (P \mid \mathbf{C}) \mid a[\mathbf{C}] \mid a\langle \mathbf{C} \rangle$$
$$\mathbf{E} ::= \cdot \mid \nu a.\mathbf{E} \mid a[\mathbf{E}] \mid P \mid \mathbf{E}$$

Fig. 2. Syntax of Contexts

use standard abbreviations from the the π-calculus: $\nu a_1 \ldots a_q.P$ for $\nu a_1.\ldots.\nu a_q.P$, or $\nu \boldsymbol{a}.P$ if $\boldsymbol{a} = (a_1, \ldots, a_q)$. By convention, if the name vector $\boldsymbol{a}$ is empty, then $\nu \boldsymbol{a}.P \triangleq P$. We abbreviate a a message of the form $a\langle \mathbf{0} \rangle$. We also note $\prod_{j \in J} P_j$, $J = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $J = \emptyset$, then $\prod_{j \in J} P_j \triangleq \mathbf{0}$.

A Kell calculus context is a term $\mathbf{C}$ built according to the grammar given in Figure 2. Filling the hole $\cdot$ in $\mathbf{C}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{C}\{Q\}$. We let $\mathbf{C}$ and its decorated variants range over Kell calculus contexts. We make use of a specific form of contexts, called evaluation contexts (noted $\mathbf{E}$), which are used to specify the operational semantics of the calculus.

A pattern ξ acts as a binder in the calculus. A pattern ξ can bind *name variables*, of the form (a), where $a \in \mathsf{N}$, and *process variables*. All name and process variables that occur in a pattern ξ are bound in ξ. Name variables can only match names. Process variables can only match processes. Process variables can only occur linearly in a pattern ξ, i.e. if x occurs in ξ, then there is only one occurrence of x in ξ. The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Notions of free names ($\mathtt{fn}$) and free (process) variables ($\mathtt{fv}$) are defined in Figure 3. We note $\mathtt{fn}(\boldsymbol{u})$ to mean $\mathtt{fn}(u_1) \cup \mathtt{fn}(u_n)$, and likewise for $\mathtt{fv}(\boldsymbol{u})$, $\mathtt{bn}(\boldsymbol{u})$, $\mathtt{bv}(\boldsymbol{u})$. We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

We call *substitution* a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{V} \to \mathsf{K}$, from names to names and process variables to processes, that is the identity except on a finite set of names and variables. We write $P\theta$ the image under the substitution θ of process P. We note Θ the set of substitutions, and $\mathtt{supp}$ the support of a substitution (i.e. $\mathtt{supp}(\theta) = \{i \in \mathsf{L} \mid \theta(i) \neq i\}$).

Let J be a join pattern, and θ be a substitution such that $\mathtt{supp}(\theta) = \mathtt{bn}(J) \cup \mathtt{bv}(J)$. We define the image $J\theta$ of J under substitution θ as $\mathtt{cj}(J)\theta$, where $\mathtt{cj}$ is the function defined inductively as:

$$\mathtt{cj}(J \mid J') = \mathtt{cj}(J) \mid \mathtt{cj}(J') \qquad \mathtt{cj}(\epsilon) = \mathbf{0}$$
$$\mathtt{cj}(a\langle \boldsymbol{w} \rangle) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle \qquad \mathtt{cj}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle \qquad \mathtt{cj}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle$$
$$\mathtt{cj}(a) = a \qquad \mathtt{cj}((a)) = a \qquad \mathtt{cj}(x) = x$$

3.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [2], via a structural congruence and a reduction relation. The structural congruence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 4. The rules

$$\begin{array}{ll}
\mathtt{fn}(\mathbf{0}) = \emptyset & \mathtt{fv}(\mathbf{0}) = \emptyset \\
\mathtt{fn}(a) = \{a\} & \mathtt{fv}(a) = \emptyset \\
\mathtt{fn}(x) = \emptyset & \mathtt{fv}(x) = \{x\} \\
\mathtt{fn}(\nu a.P) = \mathtt{fn}(P) \setminus \{a\} & \mathtt{fv}(\nu a.P) = \mathtt{fv}(P) \\
\mathtt{fn}(a[P]) = \mathtt{fn}(P) \cup \{a\} & \mathtt{fv}(a[P]) = \mathtt{fv}(P) \\
\mathtt{fn}(a\langle \boldsymbol{u} \rangle) = \mathtt{fn}(\boldsymbol{u}) \cup \{a\} & \mathtt{fv}(a\langle \boldsymbol{u} \rangle) = \mathtt{fv}(\boldsymbol{u}) \\
\mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q) & \mathtt{fv}(P \mid Q) = \mathtt{fv}(P) \cup \mathtt{fv}(Q) \\
\mathtt{fn}(\xi \triangleright P) = \mathtt{fn}(\xi) \cup (\mathtt{fn}(P) \setminus \mathtt{bn}(\xi)) & \mathtt{fv}(\xi \triangleright P) = \mathtt{fv}(P) \setminus \mathtt{bv}(\xi) \\
\mathtt{fn}(J \mid a[x]) = \mathtt{fn}(J) \cup \{a\} & \mathtt{bn}(J \mid a[x]) = \mathtt{bn}(J) \\
\mathtt{fn}(J \mid J') = \mathtt{fn}(J) \cup \mathtt{fn}(J') & \mathtt{bn}(J \mid J') = \mathtt{bn}(J) \cup \mathtt{bn}(J') \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{fn}(a\langle \boldsymbol{w} \rangle) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{bn}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{fn}(a\langle \boldsymbol{w} \rangle) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{bn}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle) = \{a\} \cup \mathtt{fn}(w) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle) = \mathtt{bn}(\boldsymbol{w}) \\
\mathtt{fn}((a)) = \emptyset & \mathtt{bn}((a)) = \{a\} \\
\mathtt{bn}(a) = \emptyset & \mathtt{bn}(x) = \emptyset \\
\mathtt{bv}(J \mid a[x]) = \mathtt{bv}(J) \cup \{x\} & \mathtt{bv}(J \mid J') = \mathtt{bv}(J) \cup \mathtt{bv}(J') \\
\mathtt{bv}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{bv}(a\langle \boldsymbol{w} \rangle) & \mathtt{bv}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{bv}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{bv}(a\langle \boldsymbol{w} \rangle) = \mathtt{bv}(\boldsymbol{w}) & \mathtt{bv}(a) = \emptyset \\
\mathtt{bv}((a)) = \emptyset & \mathtt{bv}(x) = \{x\}
\end{array}$$

Fig. 3. Free names and free variables

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \ [\text{S.PAR.ASSOC}] \qquad P \mid Q \equiv Q \mid P \ [\text{S.PAR.COMM}]$$

$$P \mid \mathbf{0} \equiv P \ [\text{S.PAR.NIL}] \qquad \nu a.\mathbf{0} \equiv \mathbf{0} \ [\text{S.NU.NIL}]$$

$$\nu a.\nu b.P \equiv \nu b.\nu a.P \ [\text{S.NU.COMM}] \qquad \frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \ [\text{S.NU.PAR}]$$

$$\frac{P =_{\alpha} Q}{P \equiv Q} \ [\text{S.}\alpha] \qquad \frac{P \equiv Q}{\mathbf{C}\{P\} \equiv \mathbf{C}\{Q\}} \ [\text{S.CONTEXT}]$$

Fig. 4. Structural congruence

S.PAR.ASSOC, S.PAR.COMM, S.PAR.NIL state that the parallel operator $\mid$ is associative, commutative, and has $\mathbf{0}$ as a neutral element.

Notice that we do not have structural congruence rules that deal with scope extrusion outside a locality, as in Mobile Ambients. This is because, in presence of the possibility of passivating executing processes, the Ambient scope extrusion rule $a[\nu b.P] \equiv \nu b.a[P] \ \ (b \neq a)$ would give rise to behaviour which would violate the idea that structurally equivalent processes should behave similarly in the same

$$\frac{J = J_1 \mid J_2 \qquad J_1 = \prod_{i\in I_1} a_i\langle \boldsymbol{w_i}\rangle^{\uparrow} \neq \epsilon \qquad J_2 = \prod_{i\in I_2} a_i\langle \boldsymbol{w_i}\rangle \qquad \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) = \emptyset}{J_1\theta \mid b[\nu \boldsymbol{c}.R \mid J_2\theta \mid (J \triangleright P)] \to b[\nu \boldsymbol{c}.R \mid P\theta]}\ [\text{R.IN}]$$

$$\frac{J = J_1 \mid J_2 \mid a[x] \qquad J_1 = \prod_{i\in I_1} a_i\langle \boldsymbol{w_i}\rangle^{\uparrow} \qquad J_2 = \prod_{i\in I_2} a_i\langle \boldsymbol{w_i}\rangle \qquad \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) = \emptyset}{J_1\theta \mid b[\nu \boldsymbol{c}.R \mid J_2\theta \mid a[Q] \mid (J \triangleright P)] \to b[\nu \boldsymbol{c}.R \mid P\theta\{Q/x\}]}\ [\text{R.IN.PASS}]$$

$$\frac{J = J_1 \mid J_2 \qquad J_1 = \prod_{i\in I_1} a_i\langle \boldsymbol{w_i}\rangle_{\downarrow} \neq \epsilon \qquad J_2 = \prod_{i\in I_2} a_i\langle \boldsymbol{w_i}\rangle \qquad \boldsymbol{d} = \boldsymbol{c} \setminus \boldsymbol{e} \qquad \boldsymbol{e} = \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) \qquad \boldsymbol{e} \cap \mathtt{fn}(J \triangleright P, J_2\theta, b) = \emptyset}{J_2\theta \mid (J \triangleright P) \mid b[\nu \boldsymbol{c}.R \mid J_1\theta] \to \nu \boldsymbol{e}.P\theta \mid b[\nu \boldsymbol{d}.R]}\ [\text{R.OUT}]$$

$$\frac{J = J_1 \mid J_2 \mid a[x] \qquad J_1 = \prod_{i\in I_1} a_i\langle \boldsymbol{w_i}\rangle_{\downarrow} \qquad J_2 = \prod_{i\in I_2} a_i\langle \boldsymbol{w_i}\rangle \qquad \boldsymbol{d} = \boldsymbol{c} \setminus \boldsymbol{e} \qquad \boldsymbol{e} = \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) \qquad \boldsymbol{e} \cap \mathtt{fn}(a, b, Q, J \triangleright P, J_2\theta) = \emptyset}{J_2\theta \mid a[Q] \mid (J \triangleright P) \mid b[\nu \boldsymbol{c}.R \mid J_1\theta] \to \nu \boldsymbol{e}.P\theta\{Q/x\} \mid b[\nu \boldsymbol{d}.R]}\ [\text{R.OUT.PASS}]$$

$$\frac{J = \prod_{i\in I} a_i\langle \boldsymbol{w_i}\rangle \neq \epsilon}{J\theta \mid (J \triangleright P) \to P\theta}\ [\text{R.LOCAL}] \qquad \frac{J = J_1 \mid a[x] \qquad J_1 = \prod_{i\in I} a_i\langle \boldsymbol{w_i}\rangle}{J_1\theta \mid a[Q] \mid (J \triangleright P) \to P\theta\{Q/x\}}\ [\text{R.LOCAL.PASS}]$$

$$\frac{P \to Q}{E[P] \to E[Q]}\ [\text{R.CONTEXT}] \qquad \frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}\ [\text{R-STRUCT}]$$

Fig. 5. Reduction Relation

evaluation context, as illustrated by the following reductions:

$$(a[x] \triangleright x \mid x) \mid a[\nu b.P] \;\to\; (\nu b.P) \mid (\nu b.P) \qquad (a[x] \triangleright x \mid x) \mid \nu b.a[P] \;\to\; \nu b.P \mid P$$

The reduction relation $\to$ is the smallest binary relation on K^2 that satisfies the rules given in Figure 5, where we assume that $\mathtt{supp}(\theta) = \mathtt{bn}(J) \cup \mathtt{bv}(J)$.

Some comments are in order. Rules R.IN an R.OUT take into account the presence of restrictions inside kells, since restricted names cannot be automatically extruded out of kells through the structural congruence. Rule R.OUT explicitly extrudes restricted names that are communicated outside a kell boundary. Note that names that are not communicated are not extruded. Rules R.IN and R.OUT govern the crossing of kell boundaries. Note that only messages may cross a kell boundary. In rule R.IN, a trigger receives messages from the local environment as well as from the outside of the enclosing kell. In rule R.OUT, a trigger receives messages from the local environment as well as from a subkell. Rules R.IN.PASS, R.OUT.PASS and R.LOCAL.PASS allow the passivation of a subkell, possibly upon receipt of messages. In rules R.IN and R.OUT, note that the join pattern J_2 may be empty. Likewise, in

rules R.IN.PASS, R.OUT.PASS and R.LOCAL.PASS, the join patterns J_1 and J_2 may be empty.

Rules R.IN, R.OUT, R.IN.PASS, R.OUT.PASS look rather involved, but only because they allow synchronizing the receipt of messages crossing a kell boundary with the receipt of local messages and the passivation of a subkell.

4 Encodings

We present in this section several encodings of process calculi to illustrate the versatility of the Kell calculus. In particular, we present faithful encodings of calculi with localities. By *faithful encoding* of a process calculus with a locality construct $a[P]$, we mean a function $[\![]\!]$ which is such that $[\![a[P]]\!] = a[M(a,c) \mid c[[\![P]\!] \mid \texttt{Aux}]] \mid \texttt{Env}$ or $[\![a[P]]\!] = c[M(a,c) \mid a[\,[\![P]\!] \mid \texttt{Aux}]] \mid \texttt{Env}$, or even $[\![a[P]\,]\!] = a[M(a) \mid [\![P]\!]] \mid \texttt{Env}$, where Env and Aux are stateless processes. In other terms, a faithful encoding translates a locality of name a into a kell of name a, possibly embedded in a controlling kell. The semantics of the locality is then captured by the membrane process $M(a,c)$ or $M(a)$.

To simplify the encodings, we use receptive triggers $\xi \diamond P$. We also use the abbreviations abstraction $(x)P$ and application PQ. The resulting extended calculus is defined by induction thus (notice the implicit typing to ensure well-formed processes):

$$[\![P]\!] \triangleq P \quad \text{if } P \in \mathsf{K}$$
$$[\![(x)P]\!]_f \triangleq f\langle x\rangle \triangleright [\![P]\!]$$
$$[\![PQ]\!] \triangleq \nu f.[\![P]\!]_f \mid f\langle [\![Q]\!]\rangle$$

4.1 Encoding the synchronous π-calculus

The asynchronous π-calculus is a direct subcalculus of the Kell calculus. Because of its higher-order character, the Kell calculus can also encode directly the synchronous π-calculus. An encoding of the synchronous (polyadic) π-calculus with input guarded sums and name matching (cf [17] for a definition) is given below, where we assume that the names $1, \ldots, n, \ldots$, and k do not appear free in P, P_j, Q, and where $\boldsymbol{b} = b_1 \ldots b_n$, $\boldsymbol{b^j} = b^j_1 \ldots b^j_{n_j}$, $j \in J$.

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{0}\\
[\![a\boldsymbol{b}.P]\!] &= a\langle [\![P]\!], b_1, \ldots, b_n\rangle\\
[\![\tau.P]\!] &= \nu \texttt{k}.\texttt{k} \mid (\texttt{k} \triangleright [\![P]\!])\\
[\![[a=b]P]\!] &= \nu l.(l\langle a\rangle \triangleright [\![P]\!]) \mid l\langle b\rangle\\
[\![\nu a.P]\!] &= \nu a.[\![P]\!]\\
[\![P \mid Q]\!] &= [\![P]\!] \mid [\![Q]\!]\\
[\![!P]\!] &= \nu \texttt{k}.\texttt{k} \mid (\texttt{k} \diamond [\![P]\!] \mid \texttt{k})\\
[\![\sum_{j\in J} a_j(\boldsymbol{b}^j).P_j]\!] &= \nu \texttt{k}.\texttt{k} \mid \prod_{j\in J} \texttt{k} \mid a_j\langle x_j, (b^j_1), \ldots, (b^j_{n_j})\rangle \triangleright [\![P_j]\!] \mid x_j
\end{aligned}$$

If we adopt the slightly unconventional semantics for the π-calculus that replaces the usual structural congruence rules matching and replication by the following reduction rules:

$$!P \rightarrow_\pi !P \mid P \qquad [a=a]P \rightarrow_\pi P$$

then we obtain

Proposition 4.1 *If $P \rightarrow_\pi Q$, then $[\![P]\!] \rightarrow^* \equiv [\![Q]\!] \mid R$, where R is a parallel composition of inert processes of the form $\nu a.a \mid \xi \triangleright P$ or $\nu a.a \mid \xi \diamond P$. Conversely, if $[\![P]\!] \rightarrow^* P'$, then there exists Q, R such that $P \rightarrow_\pi Q$, and $P' \rightarrow^* \equiv [\![Q]\!] \mid R$.*

If we assume the existence of a function $f : \mathsf{N} \rightarrow \{0,1\}^*$, that, given a name a, returns a binary encoding $f(a)$ of a, we can strengthen the result into an encoding of the synchronous polyadic π-calculus with input guarded sums, name matching *and* name unmatching. The new encoding $[\![]\!]^+$ is defined as follows:

$$[\![P]\!]^+ = [\![P]\!]_s \mid \mathtt{GC}$$
$$GC = \mathtt{collect}\langle (c), x\rangle_\downarrow \diamond (c[z] \triangleright x)$$

where $[\![]\!]_s$ is just like function $[\![]\!]$ above, except that the definitions for name matching and guarded input have been replaced respectively by:

$$[\![[a=b]P, Q]\!]_s = \nu c r.\ c[\mathtt{check}(a,r) \mid \lceil b \rceil \mid \mathtt{ThenElse}(P,Q,r,c)]$$
$$[\![\sum_{j\in J} a_j(\boldsymbol{b}^j).P_j]\!]_s = \nu \mathtt{k}\, c.$$
$$c[\mathtt{k} \mid \prod_{j\in J} \mathtt{k} \mid a_j\langle x_j, (b_1^j), \ldots, (b_{n_j}^j)\rangle \triangleright \mathtt{collect}\langle c, [\![P_j]\!]_s \mid x_j\rangle]$$

The auxiliary processes $\mathtt{check}$, $\lceil b \rceil$, and $\mathtt{ThenElse}$ are defined as follows (we assume that $0,1 \in \mathsf{N}$, that $f(a) = v_1 \ldots v_n$ and $f(b) = w_1 \ldots w_m$, with $v_i, w_j \in \{0,1\}$, and we set $\overline{0} = 1, \overline{1} = 0$):

$$\mathtt{check}(a,r) = \prod_{i=1}^{n-1} \mathtt{lcheck}(v_i) \mid \mathtt{fcheck}(v_n) \mid \mathtt{return}(r)$$
$$\mathtt{lcheck}(v) = (\mathtt{l}\langle v, x\rangle \triangleright x) \mid (\mathtt{l}\langle \overline{v}, x\rangle \triangleright \mathtt{nok})$$
$$\mathtt{fcheck}(v) = (\mathtt{l}\langle v, x\rangle \triangleright \mathtt{ok} \mid x) \mid (\mathtt{l}\langle \overline{v}, x\rangle \triangleright \mathtt{nok})$$
$$\mathtt{return}(r) = (\mathtt{ok} \mid \mathtt{end} \triangleright r\langle \mathtt{yes}\rangle) \mid (\mathtt{ok} \mid \mathtt{l}\langle (v), x\rangle \triangleright r\langle \mathtt{no}\rangle) \mid (\mathtt{nok} \triangleright r\langle \mathtt{no}\rangle)$$
$$\lceil b \rceil = \mathtt{l}\langle w_1, \mathtt{l}\langle w_2, \ldots \mathtt{l}\langle w_m, \mathtt{end}\rangle \ldots\rangle\rangle$$
$$\mathtt{ThenElse}(P,Q,r,c) = (r\langle \mathtt{no}\rangle \triangleright \mathtt{collect}\langle c, [\![Q]\!]_s\rangle) \mid (r\langle \mathtt{yes}\rangle \triangleright \mathtt{collect}\langle c, [\![P]\!]_s\rangle)$$

Intuitively, the process $\mathtt{GC}$ is a garbage collector that collects auxiliary kells which have been created by the translation and that can be safely discarded after they have served their purpose. The garbage collector is a stateless process that spawns a specific collector for kell c upon receipt of a message $\mathtt{collect}\langle c, P\rangle$. The c collector removes kell c and releases process P as a continuation. We will find variant of the garbage collector idea in other encodings. Process $\mathtt{check}(a,r)$ realizes a straightforward bitwise comparison against the binary encoding of a and returns the result

on port r. The process $\lceil b \rceil$ encodes the bit string corresponding to the binary representation of b as a list. Finally, process `ThenElse` just triggers the collection of the auxiliary kell used during the match, passing the expected continuation as an argument of the `collect` message. It is important to realize that enclosing the matching process inside an auxiliary kell c has two purposes: the first one is to isolate the computation carried out during the matching process from the rest of the computation, to avoid interferences; the second one is to serve as a cell for future garbage collection of useless processes. With this encoding we get:

Proposition 4.2 *If $P \rightarrow_\pi Q$, then $[\![P]\!]^+ \rightarrow^* \equiv [\![Q]\!]^+$. Conversely, if $[\![P]\!]^+ \rightarrow^* P'$, then there exists Q such that $P \rightarrow_\pi Q$, and $P' \rightarrow^* \equiv [\![Q]\!]^+$.*

4.2 Encoding Klaim

We consider now an encoding of a Klaim-like language. Specifically, we consider an (untyped) version of the Klaim language defined in [15]. For simplicity, we do not consider Klaim process definitions, we consider empty Klaim environments (i.e. names of nodes have global significance in this version of Klaim), and that tuples have only a single element. The encoding is defined as follows (an evaluated tuple element u can be either a node name a or a Klaim process P; a tuple element w can be either a node name pattern $!a$, a process pattern $!x$, a node name a, or a Klaim process P; and we define $\overline{(a)} \triangleq a$, $\overline{a} = a$, $\overline{x} = x$, $\overline{P} = P$).

$$
\begin{aligned}
[\![\mathbf{0}]\!]_b &= \mathbf{0} \qquad [\![a]\!]_b = a \\
[\![!a]\!]_b &= (a) \qquad [\![x]\!]_b = x \\
[\![!x]\!]_b &= x \\
[\![P \mid Q]\!]_b &= [\![P]\!]_b \mid [\![Q]\!]_b \\
[\![\mathtt{out}(u)]\!]_b &= \mathtt{out}\langle [\![u]\!]_b \rangle \\
[\![\mathtt{out}(w)@a.P]\!]_b &= \nu c.\, \mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \mid \mathtt{m}\langle b, c\langle \mathtt{ok} \rangle \rangle \rangle \mid (c\langle \mathtt{ok} \rangle \triangleright [\![P]\!]_b) \\
[\![\mathtt{int}(w)@a.P]\!]_b &= \nu c.\, \mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \triangleright \mathtt{m}\langle b, c\langle [\![P]\!]_b \rangle \rangle \rangle \mid (c\langle x \rangle \triangleright x) \\
[\![\mathtt{read}(w)@a.P]\!]_b &= \nu c.\, \mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \triangleright \mathtt{m}\langle b, c\langle [\![P]\!]_b \rangle \rangle \mid \mathtt{out}\langle \overline{[\![w]\!]_a} \rangle \rangle \mid (c\langle x \rangle \triangleright x) \\
[\![\mathtt{eval}(P)@a.Q]\!]_b &= \nu c.\, \mathtt{m}\langle a, [\![P]\!]_a \mid \mathtt{m}\langle b, c\langle \mathtt{ok} \rangle \rangle \rangle \mid (c\langle \mathtt{ok} \rangle \triangleright [\![Q]\!]_b) \\
[\![\mathtt{newloc}(a).P]\!]_b &= \nu c.\, \mathtt{newNode}\langle c \rangle \triangleright c\langle (a) \rangle^\uparrow \triangleright [\![P]\!]_b \\
[\![a :: P]\!] &= a[\mathtt{K}(a) \mid [\![P]\!]_a] \mid \mathtt{KEnv} \\
[\![N_1 \parallel N_2]\!] &= [\![N_1]\!] \mid [\![N_2]\!] \\
\mathtt{K}(a) &= a\langle x \rangle^\uparrow \diamond x \\
\mathtt{KEnv} &= (\mathtt{newNode}\langle (c) \rangle_\downarrow \diamond \nu a.a[\mathtt{K}(a)] \mid c\langle a \rangle) \mid (\mathtt{m}\langle (a), x \rangle_\downarrow \diamond a\langle x \rangle)
\end{aligned}
$$

The idea of the encoding is simple: each Klaim node $a :: P$ is modelled by a kell $a[\mathtt{K}(a) \mid [\![P]\!]]$, where $\mathtt{K}(a)$ constitutes the program of the membrane associated with a Klaim domain. In this case, we do not separate the content P of a Klaim domain from its membrane. Klaim instructions are then encoded as messages bearing some code that will be executed upon arrival at the target node. The environment `Kenv` plays the role of a simple router and of a Klaim node factory. Note that this

encoding does not use the passivation construct of the Kell calculus: Klaim does not support strong mobility (i.e. migrating executing process). For this simplified variant of Klaim, we obtain

Proposition 4.3 *If* $N \rightarrow_K N'$, *then* $[\![N]\!] \rightarrow^* \equiv [\![N']\!]$. *Conversely, if* $[\![N]\!] \rightarrow^* P$, *then there exists* N' *such that* $N \rightarrow_K N'$, *and* $P \rightarrow^* \equiv [\![N']\!]$.

4.3 Encoding Mobile Ambients

For simplicity, we present in this section an encoding of Mobile Ambients without local anonymous communication. The encoding we define below could be easily amended to account for it. The encoding is faithful and deadlock-free, but it relies on a simple locking scheme that reduces the parallelism inherent in ambient reductions. The encoding is not divergence-free (because of the definition of process $\mathtt{F}(a, \mathtt{t})$ below). An encoding that does not suffer from these limitations is certainly possible (e.g. one could mimick the protocol employed in the Join calculus implementation of ambients described in [10]) but it would be more complex. The encoding demonstrates that the passivation construct provides the basis for implementing the subjective moves of Mobile Ambients, as well as its objective `open` primitive.

The encoding of Mobile Ambients in the Kell calculus is given below, where we assume that the names `t`, `to`, `up`, `upup`, `in`, `out`, `open`, `amb`, `make`, `collect`, and `k` do not appear free in P, Q.

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![\mathtt{in}\, a.P]\!] = \mathtt{in}\langle a, [\![P]\!]\rangle$$

$$[\![\nu n.P]\!] = \nu n.[\![P]\!] \qquad [\![\mathtt{out}\, a.P]\!] = \mathtt{out}\langle a, [\![P]\!]\rangle$$

$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \qquad [\![\mathtt{open}\, a.P]\!] = \mathtt{open}\langle a, [\![P]\!]\rangle$$

$$[\![!P]\!] = \nu \mathtt{k}.\, \mathtt{k} \mid (\mathtt{k} \diamond [\![P]\!] \mid \mathtt{k})$$

$$[\![a[P]]\!] = \nu c.\, c[\,\mathtt{A}(a, c) \mid a[\,[\![P]\!] \mid \mathtt{AmbEnv}]] \mid \mathtt{AmbEnv}$$

The `AmbEnv` process is defined below. It consists of three processes: an ambient factory `C`, a garbage collector `GC`, and a process `Aux`, explained later on.

$$\mathtt{AmbEnv} = \nu \mathtt{t}.\, (\mathtt{C}\ \mathtt{t}\ \mathtt{C}) \mid \mathtt{GC} \mid \mathtt{Aux}$$

The ambient factory is defined using abstraction and application abbreviations introduced at the beginning of Section 4. Its role is to create new ambients, upon receipt of a `make` message. The definition of this factory is made complicated by the fact that it must create a copy of itself to be placed alongside the newly created ambient. To this end, we resort to a construction that is very similar to that of the `Y` fixpoint constructor we used in the definition of receptive triggers.

$$\begin{aligned}
\mathtt{C} &= (\mathtt{t}\, x)\, \mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle x\rangle \\
\mathtt{Factory}(\mathtt{t}) &= (\mathtt{t}\langle x\rangle \mid \mathtt{make}\langle (n), p, (k)\rangle_{\downarrow} \diamond \mathtt{MK}(\mathtt{t}, x, n, p, k)) \\
&\quad \mid (\mathtt{t}\langle x\rangle \mid \mathtt{make}\langle (n), p, (k)\rangle \diamond \mathtt{MK}(\mathtt{t}, x, n, p, k)) \\
\mathtt{MK}(\mathtt{t}, x, n, p, k) &= \mathtt{t}\langle x\rangle \mid k\langle \mathtt{AE}(x) \mid \nu c.\, c[\mathtt{A}(n, c) \mid n[p \mid \mathtt{AE}(x)]]\rangle \\
\mathtt{AE}(x) &= \nu a.(x\ a\ x) \mid \mathtt{GC} \mid \mathtt{Aux}
\end{aligned}$$

The garbage collector process GC corresponds to the garbage collector we defined in Section 4.1 for the encoding of the π-calculus with unmatching.

The "membrane process" A is defined below. It consists in the parallel composition of three processes S, T, F, together with a private lock, t, which is used to avoid conflicts between concurrent moves.

$$\mathtt{A}(a,c) = \nu \mathtt{t}.\, \mathtt{t} \mid \mathtt{S}(a,c,\mathtt{t}) \mid \mathtt{T}(a,c,\mathtt{t}) \mid \mathtt{F}(a,c,\mathtt{t})$$

Process S is responsible for dealing with the in and out primitives (which are translated as higher-order messages on ports in and out, respectively). In each case, the behavior is simple: passivate the ambient a, put it in a message (to, in the case of in; up, in the case of out), and asks the garbage collector to collect the now useless enclosing kell. The message bearing the passivated ambient will be released after the garbage collection has taken place.

$$\begin{aligned}\mathtt{S}(a,c,\mathtt{t}) = {} & (\mathtt{t} \mid \mathtt{in}\langle (m), p\rangle_{\downarrow} \triangleright \mathtt{a}[z] \triangleright \mathtt{collect}\langle c, \mathtt{to}\langle a, m, p, z\rangle\rangle) \\ & \mid (\mathtt{t} \mid \mathtt{out}\langle (m), p\rangle_{\downarrow} \triangleright \mathtt{a}[z] \triangleright \mathtt{collect}\langle c, \mathtt{up}\langle a, m, p, z\rangle\rangle)\end{aligned}$$

Process T is responsible for dealing with the open primitive, which is translated as a message on port open, and with the to and up messages generated by the S process of some other ambients (siblings or subambients). For open, the behavior is very simple: passivate the kell and ask the garbage collector to collect the enclosing, passing the content of the passivated kell as a continuation in the message to the garbage collector. For to, the behavior is barely more complex: when receiving the message, passivate the local kell, ask the outside factory to create a new ambient with the required characteristics, and reactivate the kell, inserting the new kell the factory has returned. In the case of up, the behavior is a bit more complex, since it requires the cooperation of the environment: forward the up message to the environment via an upup message; the upup message will be captured by the Aux process in the controlled part of the parent kell, that re-creates the ambient that has initiated the out and installs it as a sibling of the ambient that sent the upup message; upon receipt of the notification from the environment, unlock the ambient.

$$\begin{aligned}\mathtt{T}(a,c,\mathtt{t}) = {} & (\mathtt{t} \mid a[z] \mid \mathtt{open}\langle a, p\rangle^{\uparrow} \triangleright \mathtt{collect}\langle c, (z \mid p)\rangle) \\ & \mid (\mathtt{t} \mid \mathtt{to}\langle (n), a, p, x\rangle^{\uparrow} \mid a[z] \diamond \\ & \qquad (\nu k.\, \mathtt{make}\langle n, p \mid x, k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y]))) \\ & \mid (\mathtt{t} \mid \mathtt{up}\langle (n), a, p, x\rangle_{\downarrow} \diamond \\ & \qquad (\nu k.\, \mathtt{upup}\langle n, p \mid x, k\rangle \mid (k^{\uparrow} \triangleright \mathtt{t}))) \\ \mathtt{Aux} = {} & \mathtt{upup}\langle (n), p, (k)\rangle_{\downarrow} \diamond (\nu h.\mathtt{make}\langle n, p, h\rangle \mid (h\langle y\rangle \triangleright k \mid y))\end{aligned}$$

With the protocol put in place above, we have captured the effects of the in and out primitives, by means of an asynchronous protocol. However, this protocol may fail because the target ambient is not present (in the case of in), or because the enclosing ambient is not of the right name (in the case of out). The process F below handles these two failure cases. It intercepts the to and up command messages and recreates the originating ambient in the exact state it was just prior to the beginning of the migration protocol.

$$\begin{aligned}\mathtt{F}(a,c,\mathtt{t}) = (\mathtt{t} \mid\ & \mathtt{to}\langle(n),(m),p,x\rangle_{\downarrow} \diamond \\ & \mathtt{a}[z] \triangleright \\ & (\nu k.\, \mathtt{make}\langle n, \mathtt{in}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y])) \\ \mid (\mathtt{t} \mid\ & \mathtt{up}\langle(n),(m),p,x\rangle_{\downarrow} \diamond \\ & \mathtt{a}[z] \triangleright \\ & (\nu k.\, \mathtt{make}\langle n, \mathtt{out}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y]))\end{aligned}$$

A few comments on this encoding are in order. The encoding of the ambient construct, $a[P]$, is typical of encoding of calculi with explicit locations. The process $\mathtt{A}(a)$ in the encoding can be understood as implementing the interaction protocol that is characteristics of Mobile Ambients. Encoding of other forms of ambient calculi would involve defining different variants of this process. Process `AmbEnv` is a helper process that characterizes the environment required by Mobile Ambients, and that provides garbage collection and factory facilities.

If we adopt the slightly unconventional semantics for Mobile Ambients that replaces the usual structural congruence rules for replication by the following reduction rule:

$$!P \rightarrow_{MA} !P \mid P$$

then we obtain the following (where $\equiv^c$ is the structural congruence of the Kell calculus augmented with the rule $\mathtt{AmbEnv} \mid \mathtt{AmbEnv} \equiv^c \mathtt{AmbEnv}$):

Proposition 4.4 *If $P \rightarrow_{MA} Q$, then we have $[\![P]\!] \rightarrow^* \equiv^c [\![Q']\!]$, with $Q' \equiv_{MA} Q$. Conversely, if $[\![P]\!] \rightarrow^* P'$, then there exists Q such that $P \rightarrow_{MA} Q$, and $P' \rightarrow^* \equiv^c [\![Q]\!]$.*

4.4 Encoding the DJoin calculus

An encoding of the Distributed Join (DJoin) calculus can be obtained as follows. For simplicity, we consider only the DJoin calculus without failures. An encoding of the Djoin with fail-stop failures can be obtained by refining the encoding below with failure constructs similar to those introduced in section 2. We also slightly extend the language of patterns : this provides for a straightforward encoding. One can avoid this extension by associating with each membrane in the translation the list of names of sub-locations, and by updating this list after each move. The router process `IR` below can be modified to check for the presence or absence of a particular name in the list (e.g. resorting to the encoding of names in Section 4.1 to implement the check), instead of relying on extended patterns. Extending the language of patterns makes for a simpler encoding and points at useful possible variants of the calculus.

The extension of the calculus is defined as follows. With a message pattern $a\langle \boldsymbol{w}\rangle^{\uparrow}$, $a\langle \boldsymbol{w}\rangle_{\downarrow}$, or $a\langle \boldsymbol{w}\rangle$, one can associate a predicate π of the following two forms $a \in \mathtt{C}$ and $a \notin \mathtt{C}$. Intuitively, the predicate $a \in \mathtt{C}$ indicates that an active kell of name a exists somewhere within the tree of subkells routed at the current kell. The join patterns are now defined by:

$$J ::= \epsilon \mid \mu \mid \mu :: \pi \mid J \mid J$$
$$\mu ::= a\langle \boldsymbol{w} \rangle \mid a\langle \boldsymbol{w} \rangle^{\uparrow} \mid a\langle \boldsymbol{w} \rangle$$

By convention, a pattern of the form μ is equivalent to a pattern of the form $\mu :: \top$, where $\top$ corresponds to the boolean `true`. The name a appearing in predicates $\pi = a \in \mathtt{C}$ or $\pi = a \notin \mathtt{C}$ is bound in $\mu :: \pi$, if there is an occurrence of (a) in μ, otherwise it is free. The definition of the application of a substitution θ to a join pattern J is modified by adding the clause $\mathtt{cj}(\mu :: \pi) = \mathtt{cj}(\mu)$ in the definition of function `cj` in Section 3.1. The reduction rules in Figure 5 are modified by introducing the condition $\mathtt{Cond}(J, R, \theta)$ in the premises of rules R.IN, R.OUT, IN.PASS, and OUT.PASS. Assume $J = \prod_{i \in I} \mu_i :: \pi_i$, then $\mathtt{Cond}(J, R)$ is defined as:

$$\mathtt{Cond}(J, R) \triangleq \bigwedge_{i \in I} \mathtt{H}(\pi_i\theta, R) \qquad \mathtt{H}(\top, R) \triangleq \top$$
$$\mathtt{H}(a \in \mathtt{C}, R) \triangleq (R \equiv \mathbf{E}\{a[P]\}) \qquad \mathtt{H}(a \notin \mathtt{C}, R) \triangleq \neg \mathtt{H}(a \in \mathtt{C}, R)$$

The encoding can now be defined as follows. For any DJoin definition D, we note $\mathtt{df}(D)$ the set of names (channels and locations) it defines. The DJoin encoding is a function of a name that keeps track of the current DJoin location. It is defined by induction as follows, where we assume that `m`, `mm`, `collect`, `query`, `make`, `va`, `enter` do not occur free in P, D:

$$[\![a]\!]_b = a \qquad [\![\mathbf{0}]\!]_b = \mathbf{0}$$
$$[\![\top]\!]_b = \mathbf{0} \qquad [\![P \mid Q]\!]_b = [\![P]\!]_b \mid [\![Q]\!]_b$$
$$[\![\mathtt{go}\, a; P]\!]_b = \mathtt{va}\langle a, [\![P]\!]_a \rangle \qquad [\![D, D']\!]_b = [\![D]\!]_b \mid [\![D']\!]_b$$
$$[\![a\langle n_1, \ldots, n_q \rangle]\!]_b = \mathtt{m}\langle b, a, n_1, \ldots, n_q \rangle \qquad [\![D \,\mathtt{in}\, P]\!]_b = \nu \boldsymbol{n}.\, [\![D]\!]_b \mid [\![P]\!]_b \qquad \boldsymbol{n} = \mathtt{df}(D)$$

$$[\![n_1 \boldsymbol{m_1} \mid \ldots \mid n_q \boldsymbol{m_q} \triangleright P]\!]_b = \mathtt{m}\langle b, n_1, (\boldsymbol{m_1}) \rangle \mid \ldots \mid \mathtt{m}\langle b, n_q, (\boldsymbol{m_q}) \rangle \diamond [\![P]\!]_b$$
$$[\![a[D : P]]\!]_b = \nu c.\, c[\mathtt{DJ}(a, c) \mid a[[\![D]\!]_a \mid [\![P]\!]_a]] \mid \mathtt{DJEnv}$$

together with the following auxiliary definitions:

$$\begin{aligned}
\mathtt{DJ}(a, c) &= \nu \mathtt{t}.\, \mathtt{t} \mid \mathtt{IR}(a) \mid \mathtt{Go}(a, c, \mathtt{t}) \mid \mathtt{Enter}(a, \mathtt{t}) \\
\mathtt{IR}(a) &= \mathtt{m}\langle (b), x \rangle^{\uparrow} :: b \in \mathtt{C} \mid a[p] \diamond a[p \mid \mathtt{m}\langle b, x \rangle] \\
&\quad \mid \mathtt{m}\langle (b), x \rangle_{\downarrow} :: b \notin \mathtt{C} \diamond \mathtt{mm}\langle b, x \rangle \\
\mathtt{Go}(a, c, \mathtt{t}) &= \mathtt{t} \mid \mathtt{va}\langle (b), p \rangle_{\downarrow} :: b \notin \mathtt{C} \triangleright \\
&\quad (a[q] \triangleright \mathtt{collect}\langle c, \mathtt{enter}\langle b, a, (q \mid p) \rangle \rangle) \\
\mathtt{Enter}(a, \mathtt{t}) &= \mathtt{t} \mid \mathtt{enter}\langle a, (b), x \rangle^{\uparrow} \mid a[p] \diamond \\
&\quad \nu k.\mathtt{make}\langle b, x, k \rangle \triangleright (k\langle y \rangle^{\uparrow} \triangleright \mathtt{t} \mid a[p \mid y]) \\
\mathtt{DJEnv} &= \nu \mathtt{t}.(\mathtt{C}\, \mathtt{t}\, \mathtt{C}) \mid \mathtt{ER} \mid \mathtt{GC} \\
\mathtt{C} &= (\mathtt{t}\, x)\, \mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle x \rangle \\
\mathtt{Factory}(\mathtt{t}) &= \mathtt{t}\langle x \rangle \mid \mathtt{make}\langle (n), p, (k) \rangle_{\downarrow} \diamond \mathtt{MKJ}(x, n, p, k) \\
\mathtt{MKJ}(x, n, p, k) &= \mathtt{t}\langle x \rangle \mid k\langle \mathtt{DJE}(x) \mid c[\mathtt{DJ}(n, c) \mid n[p]] \rangle \\
\mathtt{DJE}(x) &= \nu a\, c.(x\, a\, x) \mid \mathtt{ER} \mid \mathtt{GC} \\
\mathtt{ER} &= \mathtt{mm}\langle x \rangle_{\downarrow} \diamond \mathtt{m}\langle x \rangle
\end{aligned}$$

Some comments are in order. Note that the encoding of a DJoin locality takes the same general form as that of a Mobile Ambient: a locality a has a controlling process $\mathtt{DJ}(a)$, that implements the basic interaction protocol that governs a DJoin locality. The latter includes: routing messages on the basis of the target locality, implementing locality migration, by means of the $\mathtt{Go}(a)$ and $\mathtt{Enter}(a)$ processes. Note that the encoding given above is faithful to the DJoin semantics, since migration is only allowed if the target locality does not appear as a sublocality of the current locality [1] . We obtain the following (where $\equiv^c$ is the structural congruence of the Kell calculus augmented with the rule $\mathtt{DJEnv} \mid \mathtt{DJEnv} \equiv^c \mathtt{DJEnv}$):

Proposition 4.5 *If $P \rightarrow_{DJ} Q$, then $[\![P]\!] \rightarrow^* \equiv^c [\![Q]\!]$. Conversely, if $[\![P]\!] \rightarrow^* P'$, then there exists Q such that $P \rightarrow_{DJ} Q$ and $P' \rightarrow^*_{DJ} \equiv^c Q$.*

5 Conclusion

We have introduced the Kell calculus, a new process calculus with hierarchical localities, strictly local actions, higher-order communication and locality passivation. Like the M-calculus, the Kell calculus allows an encoding of different forms of locality membranes, including localities with different forms of failures. The Kell calculus, however, appears simpler than the M-calculus, and does not rely on complex routing rules in contrast to the M-calculus.

We have shown by means of encodings of (a simplified form of) Klaim, of Mobile Ambients and of the Distributed Join calculus that the Kell calculus has considerable expressive power. The report [19] shows how to encode the M-calculus in the Kell calculus used in section 4. All these encodings are locality-preserving, in the sense that they translate a locality $a[P]$ in one calculus into a kell of the form $a[M(a) \mid [\![P]\!]] \mid \mathtt{Env}$ or $c[M(a,c) \mid c[[\![P]\!]]] \mid \mathtt{Env}$, where $\mathtt{Env}$ is a stateless process. We believe such locality-preserving encodings can be derived for most process calculi with localities which have been proposed in the litterature, including the numerous variants of Mobile Ambients, Nomadic Pict, Dπ [11], Seal [6], and DiTyCo [13]. Obtaining such encodings would give strong evidence that the Kell calculus embodies very fundamental constructs for distributed programming.

The Kell calculus shares the local character of its actions with the Seal calculus [6]. Indeed, as in the Seal calculus, primitive actions in our calculus include local communications and communications across a single locality boundary. In contrast to Seal, however, our communications are higher-order, whereas Seal distinguishes between first-order communications on the one hand and migrating and replicating localities on the other hand. The choice in Seal to eschew higher-order communication was made primarily with a view to simplify its underlying theory. However, as the results in [6] reveal, the higher-order character of the migrate and replicate

[1] This is not the case of the encoding of the DJoin calculus in the M-calculus defined in [18], which does not test for the presence of the target locality as a sublocality of the locality to be migrated. It is possible to faithfully encode the Djoin calculus in the M-calculus but at the cost of a more complex translation than the one reported in [18].

primitives in Seal already poses some problems (e.g. with respect to a complete characterization of contextual equivalence). With the Kell calculus higher-order pirmitives, we gain the ability to handle directly passivated process states. This allows for instance a direct modelling of such failure behaviors as fail-stop with recovery, a behaviour which would be less straightforward to model in Seal (seals can be replicated and destroyed but they cannot be passivated and reactivated; it is possible to place Seals in opaque membranes to simulate passivation but this is not entirely satisfactory since one can allow observation of passivated states – e.g. in the form of checkpoints). Another perceived advantage of the higher-order character of the Kell calculus over Seal is the potential to extend the calculus with multi-stage programming along e.g. the lines of MetaKlaim [8].

To the best of our knowledge, the dual use which is made in the Kell calculus of the locality construct $a[P]$, both as a locus for computation and as a handle for controlling the execution of located process, is new. The encodings provided in this paper show that a single (higher-order) objective control construct is sufficient to capture the variety of subjective migration primitives which have been proposed recently, in ambient calculi and other distributed process calculi. At the same time, this construct is powerful enough to model fail-stop failures, an important requirement for practical distributed programming.

Much work remains to be done, however, to assess the foundational character of the calculus with respect to distributed programming. Apart from the derivation of locality-preserving encodings mentioned above, the following issues are worth considering:

- Developing a bisimulation theory for the Kell calculus. Apart from the difficulties inherent with the higher-order character of the calculus, it would be interesting to obtain a theory parametric in the pattern language used.
- Developing type systems for the Kell calculus. Numerous type systems have been developed for mobile Ambients and their variants. It would be interesting to transfer these results (in particular the ones dealing with resource and security constraints) to the Kell calculus. Of particular interest would be the transfer of the type system developed for the M-calculus that guarantees the unicity of locality names, since this corresponds to a practical constraint in today's networks.

References

[1] R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science, vol. 96*, 1992.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.

[4] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST-TCS '02)*, volume LNCS 2556. Springer, 2002.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[6] G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 2556 in LNCS. Springer, 2002.

[7] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. $\mathbf{M}^3$: Mobility types for mobile processes in mobile ambients. In *CATS 2003)*, volume 78 of *ENTCS*, 2003.

[8] G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A Type-Safe Multi-Stage Language for Global Computing. *to appear in Mathematical Structures in Computer Science*, 2003.

[9] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.

[10] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.

[11] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical report, Technical Report 2/98 – School of Cognitive and Computer Sciences, University of Sussex, UK, 1998.

[12] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.

[13] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Mobile Object Systems Workshop (MOS'99)*, 1999.

[14] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.

[15] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.

[16] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2001.

[17] D. Sangiorgi and S. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[18] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[19] J.B. Stefani. A calculus of higher-order distributed components. Technical report, INRIA RR-4692, 2003.

[20] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *to appear in Proceedings CONCUR 02*, 2002.

[21] B. Thomsen. A Theory of Higher Order Communicating Systems. *Information and Computation, Vol. 116, No 1*, 1995.

[22] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[23] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

Electronic Notes in Theoretical Computer Science 85 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume85.html 15 pages

Towards a Tree of Channels

Xudong Guan [1,2]

INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France

Abstract

This paper presents a generalization of distributed π-calculi to support a hierarchy of locations. We add nested locations on top of a π-calculus core. By unifying channels and locations, we arrive at a computation model which uses mobile agents to pass addresses among immobile and nested locations. We choose a static binding semantics of addresses to enable easy navigation of mobile agents in the location tree. We support dynamic creation of new locations and garbage-collection of empty ones. A sample typed calculus is presented to demonstrate the formalization of type systems and related proof techniques in this calculus.

1 Introduction

The starting points of our work are distributed π-calculi [1,7,12,17] that model distributed mobile computation by extending π-calculus [9,13] with location constructs. By separating processes into named locations, these calculi can model process migration, resource control, failure, etc. characterizing distributed and mobile computation. However, unlike other mobile computation models [3,15,16,6] using nested location structures, existing distributed π-calculi tend to have a flat layout of locations. A natural question is whether we could have a distributed π-calculus equipped with a hierarchy of locations to model hierarchical administrative domains, hierarchical resource access and security control policy, and hierarchical failure semantics.

In this paper, we study a distributed π-calculus called Tπ having a hierarchy of localities. Roughly speaking, Tπ adds a nestable location construct $\mathsf{a}[P]$ on top of an asynchronous π-calculus core. We generalize the subjects and objects of communication from single names to name strings called *addresses*, pointing to remote locations. Channeled communication in π-calculus is replaced by step-by-step migration along the address followed by local anonymous communication in the destination.

[1] Supported by EU project 'MIKADO: Mobile Calculi based on Domains', FET-GC IST-2001-32222.

[2] Email: Xudong.Guan@sophia.inria.fr

As a basic example, we compare below how a piece of secret can be passed in π, Dπ [7], mobile ambients [3], and Tπ. In the latter three distributed calculi, we try to model as far as possible that the secret is a reference to a fresh sub-location of the sender.

In π, a fresh channel $\mathtt{a}$ can be passed through a shared channel $\mathtt{n}$ and serves as a secret [3]. No physical distribution is modeled.

$$\mathtt{n}(x)p \mid (\nu\,\mathtt{a})(\mathtt{n}\langle\mathtt{a}\rangle \mid q) \longrightarrow (\nu\,\mathtt{a})(p\{x:=\mathtt{a}\} \mid q)$$

In a 2-layered distributed π-calculus like Dπ, a fresh location $\mathtt{a}$ belonging to $\mathtt{l}$ can be passed through a shared channel $\mathtt{n}$ to a remote location $\mathtt{k}$. The sub-location relation is modeled logically.

$$\mathtt{l}[\mathtt{n}(x)p] \mid (\nu\,\mathtt{a})(\mathtt{k}[\mathbf{go}\ \mathtt{l}.\mathtt{n}\langle\mathtt{a}\rangle] \mid \mathtt{a}[q]) \longrightarrow^* (\nu\,\mathtt{a})(\mathtt{l}[p\{x:=\mathtt{a}\}] \mid \mathtt{a}[q])$$

In mobile ambients where migrating ambients form a hierarchy, a route giving access to a fresh sub-ambient $\mathtt{a}$ can be passed as a secret. Please note that the route is hard-coded for the specific receiver $\mathtt{l}$.

$$\begin{aligned}&\mathtt{l}[\mathbf{open}\ \mathtt{n}.(x)p] \mid \mathtt{k}[(\nu\,\mathtt{a})(\mathtt{n}[\mathbf{out}\ \mathtt{k}.\mathbf{in}\ \mathtt{l}.\langle\mathbf{out}\ \mathtt{l}.\mathbf{in}\ \mathtt{k}.\mathbf{in}\ \mathtt{a}\rangle] \mid \mathtt{a}[q])]\\ &\longrightarrow^* (\nu\,\mathtt{a})(\mathtt{l}[p\{x:=\mathbf{out}\ \mathtt{l}.\mathbf{in}\ \mathtt{k}.\mathbf{in}\ \mathtt{a}\}] \mid \mathtt{k}[\mathtt{a}[q]])\end{aligned}$$

In Tπ, we have location hierarchy like ambients. We can send the address of the secret location to some other locations using mobile agents.

$$(1)\qquad \mathtt{l}[(x)p] \mid \mathtt{k}[(\nu\,\mathtt{a})(\uparrow.\mathtt{l}\ \langle\mathtt{a}\rangle \mid \mathtt{a}[q])] \longrightarrow^* (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[p\{x:=\uparrow.\mathtt{k}.\mathtt{a}\}] \mid k[\mathtt{a}[q]])$$

In the above process, an agent $\uparrow.\mathtt{l}\ \langle\mathtt{a}\rangle$ carrying the secret address $\mathtt{a}$ first go up one level ($\uparrow$), then go down into location $\mathtt{l}$, according to the address $\uparrow.\mathtt{l}$. The secret address is translated, when arriving $\mathtt{l}$, to $\uparrow.\mathtt{k}.\mathtt{a}$, the exact address needed by $\mathtt{l}$ to access the remote sub-location. This means that we choose a static semantics for the bindings of addresses. Translation is needed to maintain their bindings during migration. Unlike ambients, immobile location together with static binding semantics of addresses in Tπ bring easy navigation of agents in the location tree. The receiver can either use the received address directly for migration, store it in a mobile agent, or send it elsewhere, with the knowledge that the address will always be maintained during migration and the same destination will always be reached whenever the address is followed.

We study the formalization of such an *address-passing* model among immobile and nested locations. After formalizing the syntax and a few basic definitions in Section 2, we give the address translation function and the reduction semantics in Section 3. Although the calculus is quite reasonable in itself, a small typed version is presented in Section 4 to demonstrate induction proof techniques in the presence of address translation. We study in Section 5 the problem of migrating locations and arbitrary processes. Finally, Section 6 concludes.

[3] Strictly speaking, we need side conditions to avoid name capturing in the examples. As for introduction, we simply assume no name capturing here.

Strings:			Processes:		
s, t	$::= \varepsilon$	empty	P, Q	$::= \mathbf{0}$	empty
	$\mid \mathtt{a}.s$	concatenation		$\mid P \mid P'$	parallel composition
Addresses:				$\mid\ !\,P$	replication
g, h	$::= s$	string		$\mid \mathtt{a}[P]$	location
	$\mid \uparrow.g$	up one level		$\mid (\nu\, s)P$	restriction, $s \neq \varepsilon$
Values:				$\mid u\,\chi$	mobile agents
u, v	$::= g$	address	Anonymous communication:		
	$\mid x$	variable	χ	$::= \langle \tilde{u} \rangle$	polyadic output
Threads:				$\mid (\tilde{x})p$	polyadic input
p, q	processes without locations				

Fig. 1. Syntax of Tπ

2 Syntax

This section formalizes the syntax of Tπ, together with a few preliminary definitions like α-conversion and structural equivalence.

Given two disjoint countable sets: *labels* ranged over by $\mathtt{a}, \mathtt{b}, ...$ and *variables* ranged over by $x, y, ...$, we define Tπ processes, ranged by $P, Q, ...$, in Fig. 1. Keeping their usual meanings, $\mathbf{0}$ is the null process; $P \mid P'$ denotes the parallel composition of two processes; and $!\,P$ means the composition of infinitely many P's. Location construct $\mathtt{a}[P]$ models nested locations with label $\mathtt{a}$ and process P running inside. *Strings* of labels, ranged over by $s, t, ...$, are used to identify locations, with the empty string ε denoting the current location. Labels distinguish sibling locations only and remote locations with the same label have no relation. As a result, a string is used in the restriction construct, $(\nu\, s)P$, meaning that the location specified by s is not known out of process P. Careful readers may have already noticed this in the introductory example (1), where the binder $(\nu\, \mathtt{a})$ is changed to $(\nu\, \mathtt{k.a})$ after scope extrusion. We require $s \neq \varepsilon$ in the restriction construct since it is unrealistic for a process to restrict its current location.

Strings can only refer to locations within the current node. *Addresses*, ranged over by $g, h, ...$, are used to refer to arbitrary locations in the hierarchy. They are strings with zero or more prefixing up-arrows ($\uparrow$). Like names in π-calculus, addresses are the first class citizen in Tπ. They are the subjects and objects of communication. We call addresses the *structured names* of locations.

Unlike π-calculus, input and output processes in Tπ are mobile. They are written $u\,\chi$ and are called *mobile agents*. A mobile agent migrates according to its subject address u until it reaches the destination. After this, local anonymous communication may happen. Agents can only carry processes without locations called *threads*, ranged over by $p, q, ...$ This will not have any impact on the expressiveness of the language. We will show in Section 5 how arbitrary processes can be transformed into threads for migration.

$$g \bowtie s \triangleq g.s \qquad \uparrow^k \bowtie h \triangleq \uparrow^k.h \qquad g.\mathtt{a} \bowtie \uparrow.h \triangleq g \bowtie h$$

$$A/_g \triangleq \{g.h \mid g.h \in A\} \qquad fa(\mathtt{a}[P]) \triangleq \mathtt{a} \bowtie fa(P)$$

$$A/_{\overline{g}} \triangleq \{h \mid h \in A \wedge h \notin A/_g\} \qquad fa((\nu\, s)P) \triangleq fa(P)/_{\overline{s}}$$

Fig. 2. Free addresses of processes

For simplicity and expressiveness, we choose asynchronous polyadic communication in our syntax: $\langle \tilde{u} \rangle$ stands for a vector of values $u_1, ..., u_k$ waiting to be read, and $(\tilde{x})p$ the receiver that reads a vector of values and substitutes them for the parameters x_i in thread p. Naturally, parameters x_i are assumed to be pair-wise distinct. Unlike Dπ [7] or ${}^e\pi$ [2], values received in Tπ can only be used alone: they can not be used as components to form other addresses.

Conventions: We often omit the trailing ε in strings and addresses, unless ε appears in its own. In writing addresses, $\uparrow^k$ stands for the concatenations of k copies of $\uparrow$. By abuse of notation, we often write $s_1.s_2$, $\uparrow^k.g$, and $g.s$ for strings and addresses when no confusion arises (i.e. any extra ε introduced by these notations does not exist). We also use $g.h$ if the concatenation does not have any $\uparrow$'s appearing after labels. We abbreviate $\varepsilon\, \chi$ as χ. We often omit $\mathbf{0}$ in $(\tilde{x})\mathbf{0}$ and $\mathtt{a}[\mathbf{0}]$. Among the process constructs, composition has the least precedence, i.e. $(\nu\, s)P \mid Q$ stands for $((\nu\, s)P) \mid Q$.

To name sub-processes precisely, we use the notion of *s-indexed process*, inductively defined as follows. A process Q is an *ε-indexed* process, or a *top-level* process, of P, if Q does not appear inside any locations in P. Q is an *$\mathtt{a.s}$-indexed* process of P, if there is an ε-index process $\mathtt{a}[R]$ of P, and Q is an s-indexed process of R. We define s-indexed values similarly. In process $P = (\nu\, s)(P_1 \mid \mathtt{a}[\mathtt{b}[\,!\, P_2 \mid u\, \chi]])$, for example, P_1, P_2, and u are top-level process, $\mathtt{a.b}$-indexed process, and $\mathtt{a.b}$-indexed value, respectively, of P.

To concatenate arbitrary addresses, we define the *chain* of two addresses, $g \bowtie h$, to be the address obtained by removing all the label-dot-up-arrow pairs in their concatenation. We let $g \bowtie A \triangleq \{g \bowtie h \mid h \in A\}$.

We say address $g.h$ *starts with* address g. For a set of addresses A, we use $A/_g$ to denote the subset of A starting with g, and $A/_{\overline{g}}$ the subset not starting with g. Specifically, we let $A/_\varepsilon = A$ and $A/_{\overline{\varepsilon}} = \emptyset$.

Restriction $(\nu\, s)P$ binds all addresses pointing to s in P. The set of *free addresses* of process, $fa(P)$, is defined in Fig. 2 for constructs $\mathtt{a}[P]$ and $(\nu\, s)P$, and has its usual definition for the rests. For example, in process $(\nu\, \mathtt{a})(\mathtt{a}[\langle\uparrow.\mathtt{a}\rangle] \mid \uparrow.\mathtt{b.a}\langle\uparrow.\mathtt{a}\rangle)$, the $\mathtt{a}$-indexed addresses ε and $\uparrow.\mathtt{a}$ are bound (note that $\langle\uparrow.\mathtt{a}\rangle$ is an abbreviation for $\varepsilon\,\langle\uparrow.\mathtt{a}\rangle$), but the top-level addresses $\uparrow.\mathtt{b.a}$ and $\uparrow.\mathtt{a}$ are free, even though they happen to contain the same label $\mathtt{a}$. As in π-calculus, $u\,(\tilde{x})P$ binds $\tilde{x}$ in P and the set of free variables is defined accordingly. We will not state explicitly the corresponding definitions for threads, since they are just special cases of those of processes.

$$\begin{array}{rcll}
\mathbf{rfl}_t(\mathtt{c}[P], s, \mathtt{a}, \mathtt{b}) & \triangleq & \mathtt{b}[\mathbf{rfl}_{t.\mathtt{c}}(P, s, \mathtt{a}, \mathtt{b})] & \text{if } t = s \text{ and } \mathtt{a} = \mathtt{c} \\
& & \mathtt{c}[\mathbf{rfl}_{t.\mathtt{c}}(P, s, \mathtt{a}, \mathtt{b})] & \text{otherwise} \\
\mathbf{rfl}_t((\nu\, s')P, s, \mathtt{a}, \mathtt{b}) & \triangleq & (\nu\, s'_1.\mathtt{b}.s'_2)\mathbf{rfl}_t(P, s, \mathtt{a}, \mathtt{b}) & \text{if } s' = s'_1.\mathtt{a}.s'_2 \text{ and } s \bowtie s'_1 = t \\
& & (\nu\, s')\mathbf{rfl}_t(P, s, \mathtt{a}, \mathtt{b}) & \text{otherwise} \\
\mathbf{rfl}_t(g, s, \mathtt{a}, \mathtt{b}) & \triangleq & g'.\mathtt{b}.s' & \text{if } g = g'.\mathtt{a}.s' \text{ and } t \bowtie g' = s \\
& & g & \text{otherwise}
\end{array}$$

Fig. 3. Renaming of free locations

$$\begin{array}{llll}
\textsc{Symm} & P \equiv P & \textsc{Par} & P \equiv Q \Longrightarrow P \mid R \equiv Q \mid R \\
\textsc{Refl} & P \equiv Q \Longrightarrow Q \equiv P & \textsc{Res} & P \equiv Q \Longrightarrow (\nu\, s)P \equiv (\nu\, s)Q \\
\textsc{Tran} & P \equiv Q, Q \equiv R \Longrightarrow P \equiv R & \textsc{Loc} & P \equiv Q \Longrightarrow \mathtt{a}[P] \equiv \mathtt{a}[Q] \\
\textsc{Split} & \mathtt{a}[P \mid Q] \equiv \mathtt{a}[P] \mid \mathtt{a}[Q] & \textsc{Par-comm} & P \mid Q \equiv Q \mid P \\
\textsc{Garb} & \mathtt{a}[\mathbf{0}] \equiv \mathbf{0} & \textsc{Par-assoc} & (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
\textsc{Alpha} & P \equiv_\alpha Q \Longrightarrow P \equiv Q & \textsc{Rep-par} & !\, P \equiv P \mid !\, P
\end{array}$$

$$\begin{array}{ll}
\textsc{Res-par} & P \mid (\nu\, s)Q \equiv (\nu\, s)(P \mid Q), \quad \text{if } fa(P)/_s = \emptyset \\
\textsc{Res-loc} & \mathtt{a}[(\nu\, s)P] \equiv (\nu\, \mathtt{a}.s)\mathtt{a}[P], \quad \text{if } fa(P)/_{\uparrow.\mathtt{a}.s} = \emptyset
\end{array}$$

Fig. 4. Structural equivalence of processes

We identify processes up to renaming of bound variables and locations. Renaming of bound variables has its normal definition. Let's consider now renaming of bound locations using example $(\nu\, \mathtt{a})(\nu\, \mathtt{a.c})\mathtt{a}[\mathtt{c}[\langle\uparrow.\uparrow.\mathtt{a}\rangle]]$. To rename location $\mathtt{a}$ to $\mathtt{b}$, we need to change the binder itself, the binder $(\nu\, \mathtt{a.c})$, the label $\mathtt{a}$, and the address $\uparrow.\uparrow.\mathtt{a}$, resulting process $(\nu\, \mathtt{b})(\nu\, \mathtt{b.c})\mathtt{b}[\mathtt{c}[\langle\uparrow.\uparrow.\mathtt{b}\rangle]]$. In Fig. 3, we define *renaming of free locations* for sub-processes and values within the scope of the binder. In $\mathbf{rfl}_t(X, s, \mathtt{a}, \mathtt{b})$, X is a t-indexed process or value, the free location to be renamed is $s.\mathtt{a}$, and $\mathtt{b}$ is the fresh label. As usual, we omit trivial and analogous ones including those for threads.

Now, we define α-conversion as (where $\mathtt{b}$ is chosen fresh):

$$(\nu\, s.\mathtt{a})P \equiv_\alpha (\nu\, s.\mathtt{b})\mathbf{rfl}_\varepsilon(P, s, \mathtt{a}, \mathtt{b})$$

Structural equivalence: We define structural equivalence between processes in Fig. 4, most of which are quite standard. Among those rules, two interesting ones are RES-PAR and RES-LOC. In rule RES-PAR, condition $fa(P)/s = \emptyset$ avoids capture of free addresses in P by $(\nu\, s)$. The condition can always be satisfied by renaming of location s in Q. Rule RES-LOC says that a restriction string, when extruding out of a location, needs to be prefixed with the label of that location. Moreover, condition $fa(P)/_{\uparrow.\mathtt{a}.s} = \emptyset$ prevents free addresses like $\uparrow.\mathtt{a}.s$ in P from becoming bound after scope extrusion. Consider for example process $P = \mathtt{a}[\mathtt{b}[\,] \mid (\nu\, \mathtt{b})\uparrow.\mathtt{a.b}\langle\mathtt{b}\rangle]$. By definition in Fig. 2, $fa(P) = \{\mathtt{a.b}\}$. Address $\uparrow.\mathtt{a.b}$ is free and refers to the free location $\mathtt{a.b}$ in P. Without the side condition, we would have $P \equiv (\nu\, \mathtt{a.b})\mathtt{a}[\mathtt{b}[\,] \mid \uparrow.\mathtt{a.b}\langle\mathtt{b}\rangle]$ and the free address would become captured. Rules SPLIT is commonly adopted in

$$\mathtt{a} \oplus g \triangleq \mathtt{a} \bowtie g \qquad {}^{s}h \oplus_A (\nu\, s')p \triangleq (\nu\, s')\, {}^{s}h \oplus_{A\cup\{s'\}} p$$

$${}^{\mathtt{a}}\uparrow \oplus g \triangleq \begin{cases} t & \text{if } g = \mathtt{a}.t \\ \uparrow \bowtie g & \text{otherwise} \end{cases} \qquad {}^{s}h \oplus_A u \triangleq \begin{cases} {}^{s}h \oplus g & \text{if } u = g \notin A \\ u & \text{otherwise} \end{cases}$$

$${}^{s}h \oplus g \triangleq {}^{\mathtt{a_k}}\uparrow \oplus (...{}^{\mathtt{a_1}}\uparrow \oplus (\mathtt{b_1} \oplus ... \oplus \mathtt{b_i} \oplus g)...), \text{ if } s = s_1.\mathtt{a_1}...\mathtt{a_k} \wedge h = \uparrow^k.\mathtt{b_1}...\mathtt{b_i}$$

Fig. 5. Address translation

distributed π-calculi for a concise reduction semantics.

For structural equivalence, we have:

Lemma 2.1 *If $P \equiv Q$, then $fa(P) = fa(Q)$.*

Proof. The proof is easy by the definitions of $fa(P)$ and $\equiv$. □

In this section, we present the syntax of Tπ, and define free addresses, α-conversion, and structural equivalence. We are now ready to present the operational semantics of Tπ.

3 Operational Semantics

We give the operational semantics of Tπ in this section using a simple binary reduction relation. We first formalize the address translation function during migration. We then present the reduction rules and illustrate with our introductory example (1).

To maintain the static binding semantics of addresses in Tπ, we need to update all the free addresses in migrating agents. The definition of address translation is summarized in Fig. 5. Address g becomes $\mathtt{a} \oplus g$ after it comes out of location $\mathtt{a}$, it becomes ${}^{\mathtt{a}}\uparrow \oplus g$ after it goes down to location $\mathtt{a}$. They are defined such that an address always points to the same location during migration. Address bindings should recover to their original forms after agents migrate back to their original locations. This is indeed the case since it is easy to check that $\mathtt{a} \oplus ({}^{\mathtt{a}}\uparrow \oplus g) = {}^{\mathtt{a}}\uparrow \oplus (\mathtt{a} \oplus g) = g$. We also generalize them and define *batch translation*, ${}^{s}h \oplus g$, that translates addresses after a series of migrations denoted by ${}^{s}h$.

When translating mobile agents, we recursively update all the free addresses in them. We keep restricted addresses untouched so that mobile agents can create fresh locations dynamically. Let X stands for p or χ, the result of the translation, ${}^{s}h \oplus_A X$, is defined in Fig. 5. The subscript A is the set of restricted addresses remembered during the translation. We only include here definitions for translating values and restrictions, the other constructs are analogous. By definition, we have ${}^{s}h \oplus_\emptyset g = {}^{s}h \oplus g$ and ${}^{s}\mathtt{a_1}...\mathtt{a_k} \oplus g = {}^{\varepsilon}\mathtt{a_1}...\mathtt{a_k} \oplus g = \mathtt{a_1} \oplus ... \oplus \mathtt{a_k} \oplus g$. For simplicity, we often abbreviate ${}^{s}h \oplus_\emptyset X$ to ${}^{s}h \oplus X$, and ${}^{s}t \oplus X$ to $t \oplus X$.

The reduction rules of Tπ are summarized in Fig. 6. The two migration rules, UP and DN, are defined using address translation. They enable one step

$$\begin{array}{llll}
\textsc{Up} & \mathtt{a}[\uparrow.g\,\chi] \longrightarrow g\,(\mathtt{a} \oplus \chi) & \textsc{R-ctx} & P \longrightarrow Q \implies P \mid R \longrightarrow Q \mid R \\
\textsc{Dn} & \mathtt{a}.g\,\chi \longrightarrow \mathtt{a}[g\ ({}^{\mathtt{a}}\uparrow \oplus \chi)] & & P \longrightarrow Q \implies (\nu\, s)P \longrightarrow (\nu\, s)Q \\
\textsc{Comm} & (\tilde{x})p \mid \langle \tilde{u} \rangle \longrightarrow p\{\tilde{x} := \tilde{u}\} & & P \longrightarrow Q \implies \mathtt{a}[P] \longrightarrow \mathtt{a}[Q]
\end{array}$$

$$\textsc{R-struct} \qquad P \equiv P', P' \longrightarrow P'', P'' \equiv P''' \implies P \longrightarrow P'''$$

Fig. 6. Reduction rules of Tπ

migrations of agents within the hierarchy. Rule Comm enables anonymous communication of values between co-located processes. Since no explicit locations are presented in threads, *substitution* of free variables in threads with values, written $p\{\tilde{x} := \tilde{u}\}$, is defined as usual [4], with implicit use of α-conversion to avoid capture. We have four other standard rules that enable reductions to happen inside unguarded contexts, and enable structural re-arrangements before and after reductions.

We illustrate the use of address translation and reduction rules with our introductory example (1), restated below for easy cross referencing.

$$\mathtt{l}[(x)p] \mid \mathtt{k}[(\nu\, \mathtt{a})(\uparrow.\mathtt{l}\,\langle \mathtt{a} \rangle \mid \mathtt{a}[q])]$$

In order for the mobile agent $\uparrow.\mathtt{l}\,\langle \mathtt{a} \rangle$ inside $\mathtt{k}$ to move up, we need first to extrude the binder $(\nu\, \mathtt{a})$ out of $\mathtt{k}$ with Res-loc. Suppose $fa(\mathtt{a}[q])/_{\uparrow.\mathtt{k.a}} = \emptyset$ (otherwise rename bound location $\mathtt{a}$), we have $fa(\uparrow.\mathtt{l}\,\langle \mathtt{a} \rangle \mid \mathtt{a}[q])/_{\uparrow.\mathtt{k.a}} = (\{\uparrow.\mathtt{l},\, \mathtt{a}\} \cup fa(\mathtt{a}[q]))/_{\uparrow.\mathtt{k.a}} = \emptyset$. So we obtain:

$$\equiv \mathtt{l}[(x)p] \mid (\nu\, \mathtt{k.a})\mathtt{k}[\uparrow.\mathtt{l}\,\langle \mathtt{a} \rangle \mid \mathtt{a}[q]]$$

By separating $\mathtt{k}$ using Split, we have:

$$\equiv \mathtt{l}[(x)p] \mid (\nu\, \mathtt{k.a})(\mathtt{k}[\uparrow.\mathtt{l}\,\langle \mathtt{a} \rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Now we can have an upward movement using rule Up and R-ctx. The address $\mathtt{a}$ becomes $\mathtt{k} \oplus \mathtt{a} = \mathtt{k.a}$ after address translation.

$$\longrightarrow \mathtt{l}[(x)p] \mid (\nu\, \mathtt{k.a})(\mathtt{l}\,\langle \mathtt{k.a} \rangle \mid \mathtt{k}[\mathtt{a}[q]])$$

A downward movement into location $\mathtt{l}$ can follow immediately, turning $\mathtt{k.a}$ to ${}^{l}\uparrow \oplus \mathtt{k.a} = \uparrow.\mathtt{k.a}$ this time.

$$\longrightarrow \mathtt{l}[(x)p] \mid (\nu\, \mathtt{k.a})(\mathtt{l}[\langle \uparrow.\mathtt{k.a} \rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Suppose that $fa(\mathtt{a}[(x)p])/_{\mathtt{k.a}} = \emptyset$ (otherwise rename bound location $\mathtt{k.a}$), we can use the scope extrusion rule Res-par:

$$\equiv (\nu\, \mathtt{k.a})(\mathtt{l}[(x)p] \mid \mathtt{l}[\langle \uparrow.\mathtt{k.a} \rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

We use rule Split to join the two processes at location $\mathtt{l}$:

$$\equiv (\nu\, \mathtt{k.a})(\mathtt{l}[(x)p \mid \langle \uparrow.\mathtt{k.a} \rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Finally, local communication happens, producing the following process that we have already used in (1).

$$\longrightarrow (\nu\, \mathtt{k.a})(\mathtt{l}[p\{x := \uparrow.\mathtt{k.a}\}] \mid \mathtt{k}[\mathtt{a}[q]])$$

[4] Substituting free variables in arbitrary processes will be a little complex, requiring a rule like $\mathtt{a}[P]\{x := \alpha\} \stackrel{\triangle}{=} \mathtt{a}[P\{x := {}^{\mathtt{a}}\uparrow \oplus \alpha\}]$ for the location construct.

We now state an important property of our reduction semantics, i.e., reduction will never introduce new free addresses.

Lemma 3.1 *If $P \longrightarrow Q$, then $fa(Q) \subseteq fa(P)$.*

Proof (Sketch) For rule Comm, it is easy to check since substitution will not introduce free addresses. For the migration rules, we can check the result by the definition of address translation. The other rules can be proved by induction, with the help of Lemma 2.1 for the rule R-struct. □

To this point, we have arrived at a reasonable formalization of our address-passing model with nested locations. We have a tree of channels whose addresses can be passed around. Agents migrate according to addresses, and communication can happen at any level in the tree. In the following sections, we study a simple type version eliminating communication errors, and we discuss the problem of migrating arbitrary processes.

4 Typed Calculus

In this section, we study a typed version of Tπ assigning Milner's sorts [8] to locations to eliminate communication errors for well-typed processes. The type system is standard and simple. We focus here on the presentation of type systems and the related proof methods in the presence of structured names and address relocation.

Value types are ranged over by $\mathrm{U}, \mathrm{V}, ...$, and the typed version is obtained in the standard way by adding type annotations for location and variable binders in the syntax. All values in Tπ have address type of the form $\mathtt{loc}(\tilde{\mathrm{V}})$, where $\tilde{\mathrm{V}}$ is the vector of value types exchanged in the location.

A type environment Γ is a collection of assignments of values to types of the form $u{:}\mathrm{U}$. A *well-formed* environment Γ doesn't contain multiple entries for a same value. We identify environments up to reordering of assignments. We extend address translation "${}^{s}h \oplus_{A} \cdot$" to type environments. Please note that well-formedness is not preserved by address translation. Consider for example $\Gamma = \mathtt{b} : \mathrm{U}_1,\ \uparrow.\mathtt{a.b} : \mathrm{U}_2$, the environment $\mathtt{a} \oplus \Gamma$ will have multiple entries for address $\mathtt{a.b}$. In case $\mathrm{U}_1 = \mathrm{U}_2$, we assume the duplicate removed. Otherwise, $\mathtt{a} \oplus \Gamma$ is not well-formed.

The typing rules in Fig. 7 are quite standard. The most interesting one due to structured names is T-loc. To get the right type environment for process $\mathtt{a}[P]$ from $\Gamma \vdash P$, we need to update the free addresses in Γ by prefixing them with label $\mathtt{a}$. We require $\mathtt{a} \oplus \Gamma$ to be also well-formed. In rule T-res, the side condition ensures that the types of all the sub-locations of the restricted location have already been declared. Consider for example process $(\nu\, \mathtt{a}{:}\mathrm{U})\mathtt{a}[\mathtt{b}[\langle\tilde{v}\rangle]]$. This process is untypable since the type of location $\mathtt{a.b}$ is not known. One should write instead $(\nu\, \mathtt{a} : \mathrm{U})(\nu\, \mathtt{a.b} : \mathrm{V})\mathtt{a}[\mathtt{b}[\langle\tilde{v}\rangle]]$ for some proper type V of location $\mathtt{a.b}$.

Well-formed environments:

$$\textsc{T-empty}\ \emptyset \vdash \diamond \qquad \textsc{T-intro}\ \frac{\Gamma \vdash \diamond \quad u \notin dom(\Gamma)}{\Gamma, u{:}\mathrm{U} \vdash \diamond} \qquad \textsc{T-val}\ \Gamma, u{:}\mathrm{U} \vdash u{:}\mathrm{U}$$

Well-typed processes:

$$\textsc{T-nil}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \textsc{T-par}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \textsc{T-rep}\ \frac{\Gamma \vdash P}{\Gamma \vdash\, !\, P}$$

$$\textsc{T-res}\ \frac{\Gamma, s{:}\mathrm{U} \vdash P \quad fa(P)/_s \subseteq \{s\}}{\Gamma \vdash (\nu\, s{:}\mathrm{U})P} \qquad \textsc{T-loc}\ \frac{\Gamma \vdash P \quad \mathtt{a} \oplus \Gamma \vdash \diamond}{\mathtt{a} \oplus \Gamma \vdash \mathtt{a}[P]}$$

$$\textsc{T-in}\ \frac{\Gamma \vdash u{:}\mathtt{loc}(\tilde{\mathrm{U}}) \quad \Gamma, \tilde{x}{:}\tilde{\mathrm{U}} \vdash p}{\Gamma \vdash u\,(\tilde{x}{:}\tilde{\mathrm{U}})p} \qquad \textsc{T-out}\ \frac{\Gamma \vdash u{:}\mathtt{loc}(\tilde{\mathrm{U}}), \tilde{v}{:}\tilde{\mathrm{U}}}{\Gamma \vdash u\,\langle\tilde{v}\rangle}$$

Fig. 7. Typing rules

$$\begin{array}{rcll} {}^{\mathtt{a}}\!\uparrow \bullet \emptyset & \triangleq & \emptyset & \\ {}^{\mathtt{a}}\!\uparrow \bullet (\Gamma \uplus \{u : \mathrm{U}\}) & \triangleq & ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) \uplus \{s{:}\mathrm{U},\ \uparrow.\mathtt{a}.s{:}\mathrm{U}\} & u = \mathtt{a}.s \\ & & ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) \uplus \{({}^{\mathtt{a}}\!\uparrow \oplus u){:}\mathrm{U}\} & \text{otherwise} \end{array}$$

Fig. 8. Back-tracing an environment

The main interest in this section is the proof for the subject reduction theorem in the presence of address translation during reductions. First, to use inductive proof, we need to find the right environment for P from $\Gamma \vdash \mathtt{a}[P]$. It is the *back-trace* of Γ, ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$, defined in Fig. 8. We use $\uplus$ for the union of (disjoint) type environments. It is easy to check that ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$ is always valid if Γ valid.

Lemma 4.1 *The followings hold:*

(1) $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \oplus \Gamma) = \Gamma$*;*

(2) $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) = \Gamma$*;*

(3) If $\Gamma \vdash \diamond$*, then* ${}^{\mathtt{a}}\!\uparrow \oplus \Gamma \vdash \diamond$*;*

(4) If $\Gamma \vdash \diamond$*, then* ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash \diamond$*.*

Proof. (1)(2) are easily checked by definition. For (3), we may check that the function " ${}^{\mathtt{a}}\!\uparrow \oplus \cdot$" on addresses is injective, so the result is well-formed if the source is well-formed. For (4), although one assignment in Γ may become two in ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$, there will still be no duplicates by definition if the original environment is well-formed. □

The following result justifies our definition of back-trace, and enables inductive reasoning with $\Gamma \vdash P$.

Lemma 4.2 *If* $\Gamma \vdash \mathtt{a}[P]$*, then* ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash P$*.*

Proof. The result $\Gamma \vdash \mathtt{a}[P]$ can only be obtained by T-LOC with $\Gamma' \vdash P$ and $\Gamma = \mathtt{a} \oplus \Gamma'$. We can show that ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$ always maps a value to the same type that Γ' maps to, that is, ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash P$ is a weaken result of $\Gamma' \vdash P$. □

The following one is the stand point for proving subject reduction of migrations.

Lemma 4.3 *If* $\Gamma \vdash p$ *and* ${}^s g \oplus_A \Gamma \vdash \diamond$*, then* ${}^s g \oplus_A \Gamma \vdash {}^s g \oplus_A p$.

Proof. Each value u in Γ used to prove $\Gamma \vdash p$ can be used in the same way but as ${}^s g \oplus_A u$ in the proof of ${}^s g \oplus_A \Gamma \vdash {}^s g \oplus_A p$. □

Now we may prove the subject reduction theorem, which guarantees that communication errors never happen in well-typed processes.

Theorem 4.4 (Subject reduction) *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$.

Proof. By induction on the derivation of $\Gamma \vdash P$. We only show the case of DN:

Case DN): In this case we have $\Gamma \vdash \mathtt{a}[\uparrow.g\,\chi]$. By Lemma 4.2 we have ${}^{\mathtt{a}}\uparrow \bullet \Gamma \vdash \uparrow.g\,\chi$. By Lemma 4.1.(2), we know $\Gamma = \mathtt{a} \oplus ({}^{\mathtt{a}}\uparrow \bullet \Gamma)$. By Lemma 4.3 we have $\Gamma \vdash \mathtt{a} \oplus (\uparrow.g\,\chi)$. That is, $\Gamma \vdash g\,\mathtt{a} \oplus \chi$. □

We end this section with an example. Let U be some type, we write below a simplified case of example (1) (where $p = !\langle x\rangle$ and $q = \mathbf{0}$) using type annotations.

$$\mathtt{l}[(x{:}\mathrm{U})\,!\langle x\rangle] \mid \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle]$$

We show how to derive the following type judgment using our type system:

$$\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{l}[(x{:}\mathrm{U})\,!\langle x\rangle] \mid \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle] \tag{2}$$

We first detail the derivation of $\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{l}[(x{:}\mathrm{U})\,!\langle x\rangle]$ below:

(a)	$\emptyset \vdash \diamond$	T-EMPTY
(b)	$x{:}\mathrm{U} \vdash \diamond$	(a)+T-INTRO
(c)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash \diamond$	(b)+T-INTRO
(d)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash x{:}\mathrm{U}$	(c)+T-VAL
(e)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash \varepsilon{:}\mathtt{loc}(\mathrm{U})$	(c)+T-VAL
(f)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash \langle x\rangle$	(d)(e)+T-OUT
(g)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash !\langle x\rangle$	(f)+T-REP
(h)	$\varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash \diamond$	(a)+T-INTRO
(i)	$\varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash \varepsilon{:}\mathtt{loc}(\mathrm{U})$	(h)+T-VAL
(j)	$\varepsilon{:}\mathtt{loc}(\mathrm{U}) \vdash (x{:}\mathrm{U})\,!\langle x\rangle$	(g)(i)+T-IN
(k)	$\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{l}[(x{:}\mathrm{U})\,!\langle x\rangle]$	(j)+T-LOC

The last step using T-LOC is the most interesting. Notice how the environment $\varepsilon : \mathtt{loc}(\mathrm{U})$ becomes $\mathtt{l} : \mathtt{loc}(\mathrm{U})$ by address translation on environments. The derivation of $\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle]$ is similar. Note that we need to check $fa(\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle)/_{\mathtt{a}} \subseteq \{\mathtt{a}\}$ when using rule T-RES. Judgment (2) is the combination of these two results using T-PAR.

This section presents an example of how to develop type system in the presence of structured names. Although the type system is not sought for resolving any particular problem of the language, rules like T-LOC and T-RES require a few careful thought. Moreover, results like Lemma 4.2 and 4.3

demonstrate how to enable induction on judgment and reasoning with address translation. More powerful type systems expressing e.g. I/O sub-typing [10] can be developed further with similar proof techniques.

5 Migrating Arbitrary Processes

In this section, we give examples illustrating the expressiveness of Tπ, especially the migration of arbitrary processes, which seems to be limited by the syntax. We first rewrite the well-known reference cell example in Tπ. We show how a cell can be accessed uniformly by remote nodes at different locations. As a cell uses a private location to keep the value, it can not be migrated directly as threads. We provide a way of converting processes to threads, and we show how to enable mobile agents creating cells dynamically in any remote locations.

All examples in this section are written in the untyped version, since annotating types in processes will make them not as readable.

Defining a cell: We rewrite the cell example from [7] in Tπ. We define a cell process that can keep one piece of value. A user may invoke on the two methods provided, `put` and `get`, to access the cell.

$$\begin{array}{l} Cell(u) \stackrel{\triangle}{=} (\nu\, \mathtt{a})\ (\mathtt{a}[\langle u\rangle] \\ \qquad\qquad | \,!\,\mathtt{put}\,(y,z)\ \mathtt{a}\,(x)(a\,\langle y\rangle \mid z\,\langle\rangle) \\ \qquad\qquad | \,!\,\mathtt{get}\quad (z)\ \mathtt{a}\,(x)(\mathtt{a}\,\langle x\rangle \mid z\,\langle x\rangle)\ ) \end{array}$$

The value is kept inside a private location `a` in the cell. Two public methods are provided as the interface for the cell. Knowing the two methods, we may define a user that stores a value v in the cell, reads that value, and sends it to a printer.

$$\begin{array}{l} User(g,h) \stackrel{\triangle}{=} (\nu\, \mathtt{ack})(\nu\, \mathtt{ret}) \\ \qquad\qquad\qquad (g\,\langle v, \mathtt{ack}\rangle \\ \qquad\qquad\qquad \mid \mathtt{ack}\,()(h\,\langle \mathtt{ret}\rangle \mid \mathtt{ret}\,(x)\ \mathtt{print}\langle x\rangle) \end{array}$$

Now we may have the following interactions.

$$Cell(u) \mid User(\mathtt{put}, \mathtt{get}) \longrightarrow^* Cell(v) \mid \mathtt{print}\langle v\rangle \tag{3}$$

One advantage of Tπ is that programmers need not care much about the location where the code will be deployed. This frees the programmers from writing dedicated routing code in designing applications, thus supports better code reuse. As an example, the user and the cell can reside at any node in the network. They can interact correctly as long as the user knows the right addresses of the two methods. Readers may check the following reductions.

$$\begin{array}{l} \mathtt{svr}[Cell(u)] \mid User(\mathtt{svr.put}, \mathtt{svr.get}) \longrightarrow^* \mathtt{svr}[Cell(v)] \mid \mathtt{print}\langle v\rangle \\ \mathtt{svr}[Cell(u)] \mid \mathtt{usr}[User(\uparrow.\mathtt{svr.put}, \uparrow.\mathtt{svr.get})] \\ \qquad\qquad\qquad\qquad\qquad\qquad \longrightarrow^* \mathtt{svr}[Cell(v)] \mid \mathtt{usr}[\mathtt{print}\langle v\rangle] \end{array}$$

Migrating locations: One might argue that it is a severe limit that agents in Tπ can only carry threads, not arbitrary processes having locations inside. In this example, we will show how to create dynamic locations in Tπ.

First we define the following abbreviation:

$$\mathbf{go}\ g.X \quad \stackrel{\triangle}{=} \quad g\,\langle\rangle \mid g\,()X$$

We can use this abbreviation to migrate arbitrary threads:

$$\mathbf{go}\ \mathtt{a}.p \ \longrightarrow^* \ \mathtt{a}[^{\mathtt{a}}\!\uparrow \oplus p]$$

Now we show how to create dynamic locations by carrying them in mobile agents. For example, instead of writing

$$\langle g\rangle \mid \langle h\rangle \mid (x)\mathbf{go}\ x.(\nu\,\mathtt{a})(\mathtt{a}[\,!\,\langle\uparrow.\mathtt{b}\rangle] \mid \mathtt{c}\,\langle\mathtt{a}\rangle),$$

which is not a valid Tπ process, we can write

$$\langle g\rangle \mid \langle h\rangle \mid (x)\mathbf{go}\ x.(\nu\,\mathtt{a})(\,!\,\mathtt{a}\,\langle\mathtt{b}\rangle \mid \mathtt{c}\,\langle\mathtt{a}\rangle),$$

which is a valid process and the destination of location $\mathtt{a}$ can only be determined at runtime. We may check that the two processes $(\nu\,\mathtt{a})(\mathtt{a}[\,!\,\langle\uparrow.\mathtt{b}\rangle] \mid \mathtt{c}\,\langle\mathtt{a}\rangle)$ and $(\nu\,\mathtt{a})(\,!\,\mathtt{a}\,\langle\mathtt{b}\rangle \mid \mathtt{c}\,\langle\mathtt{a}\rangle)$ have the same behavior. The former has an explicit location construct, while the latter hides the explicit location in its addresses.

Similarly, we can rewrite the cell process as a thread:

$$\begin{aligned} \mathit{Cell}'(u) \stackrel{\triangle}{=} (\nu\,\mathtt{a})\ (\ & \mathtt{a}\,\langle\mathtt{a}\oplus u\rangle \\ & \mid !\,\mathtt{put}\,(y,z)\ \mathtt{a}\,(x)(a\,\langle y\rangle \mid z\,\langle\rangle) \\ & \mid !\,\mathtt{get}\quad (z)\ \mathtt{a}\,(x)(\mathtt{a}\,\langle x\rangle \mid z\,\langle x\rangle)\ ) \end{aligned}$$

Following Dπ, we may now write a cell server able to create cells anywhere according to client requests.

$$\begin{aligned} \mathit{Server} \quad \stackrel{\triangle}{=}\ & \mathtt{req}[\,!\,(\mathit{dest},\mathit{ret},x)\ \mathbf{go}\ \mathit{dest}\ .\ (\nu\,\mathtt{put})(\nu\,\mathtt{get}) \\ & \qquad\qquad (\mathit{ret}\langle\mathtt{put},\mathtt{get}\rangle \mid \mathit{Cell}'(x)\ )] \\ \mathit{Client}(g) \stackrel{\triangle}{=}\ & (\nu\,\mathtt{a})\ (\ g\,\langle\mathtt{a},\varepsilon,u\rangle \mid (x,y)\ \mathit{User}(x,y)) \end{aligned}$$

Given a destination *dest*, a return address *ret*, and an initial value x, the cell server creates dynamically a cell at *dest* with initial value x, and sends the addresses of the two methods to *ret*. A typical configuration like the following works.

$$\begin{aligned} & \mathtt{svr}[\mathit{Server}] \mid \mathtt{usr}[\mathit{Client}(\uparrow.\mathtt{svr}.\mathtt{req})] \\ \longrightarrow\ & \mathtt{svr}[\mathit{Server}] \mid \mathtt{usr}\ [\ (\nu\,\mathtt{a})(\nu\,\mathtt{a}.\mathtt{put})(\nu\,\mathtt{a}.\mathtt{get}) \\ & \qquad\qquad (\ \mathtt{a}[\mathit{Cell}'(^{\mathtt{a}}\!\uparrow \oplus u)] \mid \mathit{User}(\mathtt{a}.\mathtt{put},\mathtt{a}.\mathtt{get})\ )\ ] \end{aligned}$$

The user now has the sole access to the newly created cell at location $\mathtt{a}$.

Migrating arbitrary processes: To go a step further, we now show how to transform arbitrary processes to threads, so as to support migration of arbitrary processes in Tπ. We define in Fig. 9 the transformation of s-indexed processes to threads: $\langle\!\langle P\rangle\!\rangle_s$.

For any process, we can show that P and $\langle\!\langle P\rangle\!\rangle_\varepsilon$ have the same external

$$\begin{array}{rcl@{\qquad}rcl}
\langle\!\langle \mathbf{0} \rangle\!\rangle_s & \triangleq & \mathbf{0} & \langle\!\langle \mathtt{a}[P] \rangle\!\rangle_s & \triangleq & \langle\!\langle P \rangle\!\rangle_{s.\mathtt{a}} \\
\langle\!\langle P \mid Q \rangle\!\rangle_s & \triangleq & \langle\!\langle P \rangle\!\rangle_s \mid \langle\!\langle Q \rangle\!\rangle_s & \langle\!\langle (\nu\, t)P \rangle\!\rangle_s & \triangleq & (\nu\, s.t)\langle\!\langle P \rangle\!\rangle_s \\
\langle\!\langle\, !\, P \rangle\!\rangle_s & \triangleq & !\, \langle\!\langle P \rangle\!\rangle_s & \langle\!\langle u\, \chi \rangle\!\rangle_s & \triangleq & (s \oplus u)\, (s \oplus \chi)
\end{array}$$

Fig. 9. Transforming s-indexed processes to threads

behavior [5], while the latter is a thread without any explicit location. As a result, we can write mobile agents like $u\,(x)P$ as a short hand for $u\,(x)\langle\!\langle P \rangle\!\rangle_\varepsilon$. We can now justify our claim in Section 2, that the limitation in the syntax doesn't have any impact on expressiveness. It only makes a simpler definition for address translation and substitution.

One may wonder on the other hand why we need explicit location construct in the syntax, since every process has a corresponding thread version with the same behavior. We now make the following explanations. As a mobile and distributed computation model, Tπ needs locations for modeling network and computational nodes. Although virtual locations within a same computational node can be eliminated, physical locations representing these nodes can not be. They are essential to express distribution, migration, and security control. Without the location construct, we would go back to a single machine model. The resulting calculus would become to some extent just an asynchronous π-calculus with the set of strings as its channels (not ${}^e\pi$, since addresses must be used as a whole).

Comparing Tπ with π-calculus and $lsd\pi$ [12]: By thinking π-calculus channels as a flat layer of locations, we may take (asynchronous) π-processes as a subset of Tπ threads. The communication rule $\mathtt{a}(x).p \mid \mathtt{a}\langle u \rangle \longrightarrow p\{x := u\}$ in π can be simulated in Tπ by two downward movements, one anonymous communication in location $\mathtt{a}$, and one upward movement. Address translations during these migrations have no accumulated effect, since we have $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \oplus p) = p$. We believe that $lsd\pi$, or at least some trivial variant of it, is properly contained in Tπ also.

6 Conclusion

In this paper, we make some preliminary investigations in the formalization of an address-passing model in a distributed π-calculus with location hierarchy. Our approach of using addresses as the basic semantical entities seems to be quite natural in the presence of location hierarchy. The use of address translation makes the semantics quite simple and consistent with π and $lsd\pi$. With anonymous communication inside locations, we also unify channels and locations, which used to be two different classes in other distributed π-calculi.

The idea in this paper benefits a lot from existing works in distributed π-calculi and ambient calculi. Anonymous communications inside locations are

[5] The formal study of process behavior will be reported in another paper.

borrowed from ambients. However, in ambient calculi, ambients are mobile instead of fixed, and the idea of structured names doesn't make much sense. As early as in [11] (Chapter 14), Priami studied distributed name manager and relative addressing for π-calculus in a distributed setting using parallel composition. A simple form of address translation is first introduced for the two layer distributed calculus $lsd\pi$, carried out by dedicated substitutions of local and remote channels. Our address translation, however, is more general and works consistently within a tree of locations.

Location trees have also been explored in other process calculi close to π. In the local area π-calculus [4], a hierarchy of local areas is modeled and a same channel can have within its scope several disjoint local areas. The hierarchy, however, is predefined, and explicit process mobility is not modeled in the local area π-calculus. Locations in the distributed Join calculus (DJoin) [5] are also organized as a tree. However, DJoin use transparent remote communication and migration based on simple names. Locations and process mobility in DJoin are mainly used for modeling distribution and failure.

Another close work to Tπ is the π-calculus with polyadic synchronization ($^e\pi$) [2]. Our calculus is even closer to $^e\pi$ if we use labels in communications and allow variables to appear as part of addresses. The notion of location and address translation, however, distinguish Tπ from $^e\pi$. These two calculi have different motivations and it is hard to express one in another, e.g. Tπ lacks the powerful name matching ability, while $^e\pi$ processes have to be coordinated in a centralized manner.

The syntax of Tπ studied in this paper is kept to its minimum to focus on the study of the static binding semantics of structured names. Extension could be made to express more features characterizing distributed mobile computation. Inspired by works like [7,4,14], we are investigating an extension of Tπ with *resource* names that are dynamically bound. Apart from anonymous communication, we now have nominal communication on these resource names. Resources are dynamically bound, and are not affected by address translation. We envisage embedding both Dπ and $lsd\pi$ in this version of Tπ.

Acknowledgments: Comments from Gérard Boudol, Ana Matos, and three anonymous referees improved this paper.

References

[1] Amadio, R. M., *On modelling mobility*, Theoretical Computer Science **240** (2000), pp. 147–176.

[2] Carbone, M. and S. Maffeis, *On the expressive power of polyadic synchronisation in pi-calculus*, in: *Proc. of the 9th International Workshop on Expressiveness in Concurrency (EXPRESS'02)*, ENTCS **68.2**, 2002.

[3] Cardelli, L. and A. D. Gordon, *Mobile ambients*, Theoretical Computer Science

240 (2000), pp. 177–213, an extended abstract appeared in *Proceedings of FoSSaCS '98*: 140–155.

[4] Chothia, T. and I. Stark, *A distributed calculus with local areas of communication*, in: *HLCL '00: Proceedings of the 4th International Workshop on High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science **41.2** (2001).

[5] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Proceedings of POPL '96*, ACM, 1996, pp. 372–385.

[6] Godskesen, J. C., T. Hildebrandt and V. Sassone, *A calculus of mobile resources*, in: *Proceedings of CONCUR'02*, LNCS **2421**, 2002, pp. 272–287.

[7] Hennessy, M. and J. Riely, *Resource access control in systems of mobile agents*, Journal of Information and Computation **173** (2002), pp. 82–120, an extended abstract appeared in *Proceedings of HLCL'98*, pages 3–17.

[8] Milner, R., *The polyadic π-calculus: A tutorial*, in: F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Series F **94**, NATO ASI (1993), available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[9] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, part I/II*, Journal of Information and Computation **100** (1992), pp. 1–77.

[10] Pierce, B. C. and D. Sangiorgi, *Typing and subtyping for mobile processes*, Mathematical Structures in Computer Science **6** (1996), pp. 409–454, an extract appeared in *Proceedings of LICS '93*: 376–385.

[11] Priami, C., "Enhanced Operational Semantics for Concurrency," Ph.D. thesis, Dipartimento di Informatica, Università di Pisa-Genova-Udine (1996), available as PhD number TD-08/96.

[12] Ravara, A., A. Matos, V. T. Vasconcelos and L. Lopes, *A lexically scoped distributed pi-calculus*, Di/fcul tr, DIFCUL (2002).

[13] Sangiorgi, D. and D. Walker, "The π-calculus: a Theory of Mobile Processes," Cambridge University Press, 2001.

[14] Schmitt, A., *Safe dynamic binding in the join calculus*, in: *Proceedings of the International IFIP Conference TCS 2002*, 2002, pp. 563–575.

[15] Schmitt, A. and J.-B. Stefani, *The m-calculus: a higher-order distributed process calculus*, in: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2003), pp. 50–61, extended version as Rapport de Recherche RR-4361, INRIA Rhone-Alpes, January 2002.

[16] Vitek, J. and G. Castagna, *Seal: A framework for secure mobile computations*, in: H. Bal, B. Belkhouche and L. Cardelli, editors, *Internet Programming Languages*, LNCS **1686**, 1999, pp. 47–77.

[17] Wojciechowski, P. and P. Sewell, *Nomadic pict: Language and infrastructure design for mobile agents*, IEEE Concurrency **8** (2000), pp. 42–52.

From Ambients To A Routing Calculus

Xudong Guan [1,2]

INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France

Abstract

From a more simplified encoding of π-calculus into pure ambients, we derive a routing calculus which is simple and expressive. While being a direct subset of pure robust ambients, the basic version of this calculus is able to encode π, and simulate Dπ to some extent. By adding a primitive that enables mobile agent acquiring the current location name, the calculus is able to give a reasonable and much simpler encoding of Dπ.

1 Introduction

The π-calculus [MPW92,SW01] is a formal model for concurrent and mobile systems based on name-passing between parallel processes. The calculus of mobile ambients [CG00] is a formal model for mobile and distributed computation, where computation is modeled by ambients moving around and being opened. Recently, results have been shown [Zim00] that name-passing in π-calculus can be simulated by ambient movements and openings. This paper presents our further investigation into this problem.

Our start point is a new encoding from π to pure [3] robust ambients[GYY00] (ROAM). The encoding is simpler, in terms of both size and the computation steps required to simulate one π-reduction, than Zimmer's encoding and our previous one [Gua02]. Like our previous one, this new encoding is a well-behaved one, in that ambients are either single-threaded or immobile. From this encoding, we design a sub-calculus of ROAM called $\mathtt{PR}^-$ that allows only single-threaded or immobile ambients by syntactical means, while still being able to encode π. Meanwhile, $\mathtt{PR}^-$ itself exhibits its own packet-routing model for mobile and distributed computation.

[1] Supported by EU project 'MIKADO: Mobile Calculi based on Domains', FET-GC IST-2001-32222.

[2] Email: Xudong.Guan@sophia.inria.fr

[3] In ambient jargon, *pure* stands for the basic version without communication.

As a step further on the expressiveness of $\mathtt{PR}^-$ we present an encoding of Dπ [HR98], a distributed π-calculus supporting process distribution, migration, and dynamic-binding of channels. In the encoding, we show how to realize both channel name substitution and location name substitution in ambients. As one can expect, the encoding is rather complex, even though it is written in the more readable $\mathtt{PR}^-$ instead of ROAM.

To overcome this problem, we add to $\mathtt{PR}^-$ a primitive called *out-binding* that enables ambients to know the name of their locations. For simplicity, we also anonymize all single-threaded ambients. In this new calculus, the encoding of π and Dπ are simpler. Although this calculus is no longer a proper sub-calculus of ROAM, it is also interesting and worth further exploration in its own.

The rest of this paper is organized as follows. Section 2 gives a review of π and ROAM. Section 3 presents the more concise encoding. Section 4 gives the corresponding encoding in $\mathtt{PR}^-$, and gives a translation of $\mathtt{PR}^-$ processes to ROAM processes. Section 5 studies the encoding of Dπ in $\mathtt{PR}^-$. Section 6 presents the extension and the corresponding new encodings in the extended calculus. Finally, Section 7 concludes the paper. In the appendix, we list the two previous encodings from π to pure ambients, and give some comparisons of them to the one proposed in this paper.

2 Review of the two languages

2.1 π-calculus

As in [Zim00], we use the sum-free synchronous π-calculus (π-calculus for short) as our source language. For the sake of encoding, names are distinguished by two non-overlapping sets, the set of channel names (often ranged over by $n, m, ...$), and the set of variable names (often ranged over by $x, y, ...$), collectively called values (denoted by $u, v, ...$). The set of π-processes is defined by the follow grammar:

$$P, Q \quad ::= \quad \mathbf{0} \quad | \quad P \mid Q \quad | \quad !P \quad | \quad (\nu n)P \quad | \quad u\langle v\rangle.P \quad | \quad u(x).P$$

The main reduction rule of π-calculus is the following communication rule, enabling an *output* process on channel n ($n\langle u\rangle.P$) to pass a name to an input process on the same channel ($n(x).P$).

$$n\langle u\rangle.P \mid n(x).Q \longrightarrow P \mid Q\{x{:=}u\}$$

Reductions can happen through structural re-arrangements, and inside evaluation contexts. The evaluation context of π is defined as:

$$E \quad ::= \quad - \quad | \quad E \mid P \quad | \quad (\nu n)E$$

More detailed explanation could be found in standard π-calculus literature [MPW92,SW01].

2.2 Robust ambients and type system

ROAM is a variant of safe ambients [LS00] (SA), having slightly different semantics for co-actions. We briefly review a typed version of ROAM in this section. More details could be found in [GYY00,GYY01,Gua02].

Given a set of names range over by $n, m, ...$, the set of ROAM processes is defined as:

$$P ::= \mathbf{0} \quad | \quad P \mid Q \quad | \quad !P \quad | \quad (\nu n : T)P \quad | \quad n[P] \quad | \quad M.P$$

Parallel composition, replication and restriction have their usually meanings. $n[P]$ stands for an *ambient* able to perform different *actions* according to the running process P inside. $M.P$ is a process that can perform action M and then becomes P. There are six different types of actions in ROAM, formulated in three pairs [4].

$$M ::= \texttt{in}\ n \quad | \quad \overline{\texttt{in}}\ n \quad | \quad \texttt{out} \quad | \quad \overline{\texttt{out}}\ n \quad | \quad \texttt{open}\ n \quad | \quad \overline{\texttt{open}}$$

Actions, except $\texttt{open}\ n$, command their surrounding ambients. We introduce the functionalities of these actions briefly: $\texttt{in}\ n$ tells the surrounding ambient to move inside ambient n, while $\overline{\texttt{in}}\ n$ allows an ambient named n to move inside; $\texttt{out}$ tells the surrounding ambient to move outside its current location, while $\overline{\texttt{out}}\ n$ allows an ambient named n to go out; $\texttt{open}\ n$ opens ambient n, while $\overline{\texttt{open}}$ gives permission for others to open the ambient. The evaluation contexts and reduction axioms of ROAM is summarized below. Reduction can not only happen inside compositions and restrictions, but also inside ambients.

$$E \quad ::= \quad - \quad | \quad E \mid P \quad | \quad (\nu n : T)E \quad | \quad n[E]$$

$$\begin{array}{rcl} m[\texttt{in}\ n.P_1 \mid P_2] \mid n[\overline{\texttt{in}}\ m.Q_1 \mid Q_2] & \longrightarrow & n[m[P_1 \mid P_2] \mid Q_1 \mid Q_2] \\ n[m[\texttt{out}.P_1 \mid P_2] \mid \overline{\texttt{out}}\ m.Q_1 \mid Q_2] & \longrightarrow & m[P_1 \mid P_2] \mid n[Q_1 \mid Q_2] \\ \texttt{open}\ n.P \mid n[\overline{\texttt{open}}.Q_1 \mid Q_2] & \longrightarrow & P \mid Q_1 \mid Q_2 \end{array}$$

Every ambient is assigned a type T. It can be a *simple type* Z^Y, having mobility Z ($\curvearrowright$ - mobile, or $\veebar$ - immobile) and threads Y (0 - no thread, 1 - single-threaded, or ω - multi-threaded), and can't be opened. It can also be an *evolving type* $Z^Y[T]$, denoting that the ambient, apart from having mobility Z and threads Y, can be opened and release processes of type [5] T. We are mainly interested in *well-behaved* processes in which ambients are either immobile or single-threaded. The results in [Gua02] shows that well-behaved processes have a better behavior theory than arbitrary processes.

[4] We use a version of ROAM with no parameter for action "$\texttt{out}$".

[5] Ambient type is actually computed from the type of the process inside it. Refer to [GYY01] for more details.

$$
\begin{array}{rcl}
\textbf{fwd}\ u & \triangleq & !\,route[\overline{\texttt{in}}\ route.\texttt{open}\ route.\texttt{out.in}\ u.\texttt{in}\ route.\overline{\texttt{open}}] \\
\langle\!\langle P \rangle\!\rangle^* & \triangleq & \langle\!\langle P \rangle\!\rangle \mid !\,\texttt{open}\ cont \\
\langle\!\langle \mathbf{0} \rangle\!\rangle & \triangleq & \mathbf{0} \\
\langle\!\langle P \mid Q \rangle\!\rangle & \triangleq & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle\, !\, P \rangle\!\rangle & \triangleq & !\,\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle (\nu n) P \rangle\!\rangle & \triangleq & (\nu n : \underline{\vee}^{\omega})\ (\ n[\ !\,\overline{\texttt{in}}\ route \mid !\,\texttt{open}\ route \mid !\,\overline{\texttt{out}}\ comm \\
 & & \qquad \mid\ !\,route[\overline{\texttt{in}}\ route.\texttt{open}\ route.\overline{\texttt{open}}]\ ]\ \mid\ \langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle u\langle v \rangle.P \rangle\!\rangle & \triangleq & route\ [\ \texttt{in}\ u.\texttt{in}\ route.\overline{\texttt{open}} \\
 & & \qquad \mid\ comm\ [\ \texttt{in}\ comm.\overline{\texttt{open}}\ \mid\ \textbf{fwd}\ v \\
 & & \qquad\quad \mid\ cont[\texttt{out}.\overline{\texttt{open}} \mid \langle\!\langle P \rangle\!\rangle]]] \\
\langle\!\langle u(x).P \rangle\!\rangle & \triangleq & (\nu x : \underline{\vee}^{\omega})\ (\ route[\ \texttt{in}\ u.\texttt{in}\ route.\overline{\texttt{open}} \\
 & & \qquad \mid\ comm[\ \overline{\texttt{in}}\ comm.\texttt{open}\ comm.\texttt{out.in}\ x.\overline{\texttt{open}} \\
 & & \qquad\quad \mid\ cont[\texttt{out}.\overline{\texttt{open}} \mid \langle\!\langle P \rangle\!\rangle]]] \\
 & & \quad \mid\ x[\ !\,\overline{\texttt{in}}\ route \mid !\,\overline{\texttt{out}}\ route \\
 & & \qquad \mid\ !\,\overline{\texttt{out}}\ cont \mid !\,\overline{\texttt{in}}\ comm \mid !\,\texttt{open}\ comm])
\end{array}
$$

Fig. 1. Encoding π-calculus in ROAM: a concise one.

3 A more concise encoding

As our start point, we give in this section a new encoding of π-calculus into well-behaved ROAM processes. We write the encoding $\langle\!\langle P \rangle\!\rangle^*$. We use three special ambient names: *route* for routing communication processes, *comm* for interaction between communication processes, and *cont* for wrapping continuation processes. The encoding is defined only for closed processes without free names. For those having free names, we need to supply additional free channel ambients in parallel to the encoding.

The encoding works as follows. For every channel, we construct an immobile ambient with the same name, inside which input and output processes can meet and communicate. An input/output process has the form $route[\texttt{in}\ u.\texttt{in}\ route.\overline{\texttt{open}} \mid P]$. It can route its cargo P, composed by all the sub-ambients of it, to the correct channel specified by u, which is either a channel name itself, or a variable name. The routing service in a channel ambient, $!\,route[\overline{\texttt{in}}\ route.\texttt{open}\ route.\overline{\texttt{open}}]$, tells the incoming packet that this is the right destination. Meanwhile, the routing service $\textbf{fwd}\ v$ in a variable ambient turns the incoming packet to the one having destination v.

Let's now see how read and write process actually interact when they reach the right channel. The cargo of a read process with the variable x and continuation P is of the form

$$comm[\overline{\texttt{in}}\ comm.\texttt{open}\ comm.\texttt{out.in}\ x.\overline{\texttt{open}} \mid cont[\texttt{out}.\overline{\texttt{open}} \mid \langle\!\langle P \rangle\!\rangle]],$$

while the cargo of a write process sending v with continuation Q is:

$$comm[\texttt{in}\ comm.\overline{\texttt{open}} \mid \textbf{fwd}\ v \mid cont[\texttt{out}.\overline{\texttt{open}} \mid \langle\!\langle Q \rangle\!\rangle]]$$

$$
\begin{array}{rcl}
\textbf{fwd}\ u & \triangleq & !\,route(\texttt{get}\ route.\texttt{out.in}\ u.\texttt{put}\ route)[\,] \\
\langle\!\langle P \rangle\!\rangle^* & \triangleq & \langle\!\langle P \rangle\!\rangle \mid cont^- \\
\langle\!\langle \mathbf{0} \rangle\!\rangle & \triangleq & \mathbf{0} \\
\langle\!\langle P \mid Q \rangle\!\rangle & \triangleq & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle\, !\, P \rangle\!\rangle & \triangleq & !\,\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle (\nu n) P \rangle\!\rangle & \triangleq & (\nu n)(\ n\ \langle route^{\mp} \mid comm^{\uparrow} \rangle \\
 & & \qquad [\ !\,route(\texttt{get}\ route.\texttt{dis})[\,]\,]\ \ \mid\ \ \langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle u\langle v \rangle.P \rangle\!\rangle & \triangleq & route\ (\ \texttt{in}\ u.\texttt{put}\ route) \\
 & & \qquad [\ comm\ (\ \texttt{put}\ comm) \\
 & & \qquad\quad [\ \textbf{fwd}\ v \mid\ cont(\texttt{out.dis})[\langle\!\langle P \rangle\!\rangle]\,]\ ] \\
\langle\!\langle u(x).P \rangle\!\rangle & \triangleq & (\nu x)\ (\ route(\ \texttt{in}\ u.\texttt{put}\ route) \\
 & & \qquad [\ comm(\texttt{get}\ comm.\texttt{out.in}\ x.\texttt{dis}) \\
 & & \qquad\quad [\ cont(\texttt{out.dis})[\langle\!\langle P \rangle\!\rangle]\,]\ ] \\
 & & \quad \mid\ x\ \langle route^{\updownarrow} \mid cont^{\uparrow} \mid comm^{\mp} \rangle [\,]\ )
\end{array}
$$

Fig. 2. Encoding π-calculus in ROAM using abbreviations

The result of their interaction is an ambient that first come out of the channel ambient, then go into the variable ambient x with the redirector **fwd** v, together with two continuation processes P and Q, which are then released after communication.

Compared with previous encodings in SA and ROAM, this one works under the same framework, but is simpler in terms of both encoding size and computation steps required to simulate one π-reduction. See Appendix B for detailed comparisons of these encodings. The encoding is also well-behaved, in that we have all the channel ambients and variable ambients being immobile, and the rest being single-threaded (all the three system ambients have type $\curvearrowright^1 [\underline{\vee}^0]$).

4 Rewrite the encoding using abbreviations

In this section, we present a new syntax for a subset of well-behaved ROAM processes. We let $\texttt{get}\ a.P \triangleq \overline{\texttt{in}}\ a.\texttt{open}\ a.P$, $\texttt{put}\ a \triangleq \texttt{in}\ a.\overline{\texttt{open}}.\mathbf{0}$, and $\texttt{dis} \triangleq \overline{\texttt{open}}.\mathbf{0}$. We write $a(M)[P]$ for single-threaded ambients $a[M.\mathbf{0} \mid P]$, where P is the collection of all sub-ambients of a. We use abbreviations like $a^{\downarrow} \triangleq !\,\overline{\texttt{in}}\ a$, $a^{\uparrow} \triangleq !\,\overline{\texttt{out}}\ a$, and $a^{-} \triangleq !\,\texttt{open}\ a$. We write $a^{\ddagger}$ for $a^{\downarrow} \mid a^{\uparrow} \mid a^{-}$, etc. We write $a\langle C \rangle[P]$ for immobile ambients $a[C \mid P]$, where C are those three kinds of replicated actions. We omit the type annotations in Figure 1 and re-write the encoding using these abbreviations in Figure 2.

The routing protocol in Figure 1 can be expressed concretely using abbreviations like *get* and *put*. To go a step further, we define a subset of ROAM processes using these notations so that all processes in the subset are well-

Processes:

P, Q	$::= \mathbf{0}$	empty
	$\mid P \mid Q$	composition
	$\mid\ !P$	replication
	$\mid (\nu n)P$	restriction
	$\mid a(M)[P]$	single threaded ambient
	$\mid w\langle C\rangle[P]$	immobile ambient

Names:

n, m	$::= w$	wall name
	$\mid a$	agent name

Capabilities:

M, N	$::= \texttt{in}\ w.M$	move in
	$\mid\ \texttt{out}.M$	move out
	$\mid\ \texttt{get}\ a.M$	get some cargo
	$\mid\ \texttt{put}\ a$	put cargo
	$\mid\ \texttt{dis}$	dissolve

Controllers:

C, D	$::= \mathbf{0}$	empty
	$\mid\ a^{\downarrow}$	allow move in
	$\mid\ a^{\uparrow}$	allow move out
	$\mid\ a^{-}$	allow dissolving
	$\mid\ C \mid C'$	composition

Evaluation contexts:

$$E \quad ::= \quad - \quad \mid \quad E \mid P \quad \mid \quad (\nu n)E \quad \mid \quad a(M)[E] \quad \mid \quad w\langle C\rangle[E]$$

Reduction axioms:

$$\begin{aligned}
w\langle a^{\downarrow} \mid C\rangle[P] \mid a(\texttt{in}\ w.M)[Q] &\longrightarrow w\langle a^{\downarrow} \mid C\rangle[P \mid a(M)[Q]] \\
w\langle a^{\uparrow} \mid C\rangle[P \mid a(\texttt{out}.M)[Q]] &\longrightarrow w\langle a^{\uparrow} \mid C\rangle[P] \mid a(M)[Q] \\
w\langle a^{-} \mid C\rangle[P \mid a(\texttt{dis})[Q]] &\longrightarrow w\langle a^{-} \mid C\rangle[P \mid Q] \\
a(\texttt{get}\ b.M)[P] \mid b(\texttt{put}\ a)[Q] &\longrightarrow a(M)[P \mid Q]
\end{aligned}$$

Fig. 3. Syntax and reduction semantics of $\texttt{PR}^{-}$

$$\begin{aligned}
\{\!|\texttt{in}\ w.M|\!\} &\triangleq \texttt{in}\ w.\{\!|M|\!\} & \{\!|\mathbf{0}|\!\} &\triangleq \mathbf{0} \\
\{\!|\texttt{out}.M|\!\} &\triangleq \texttt{out}.\{\!|M|\!\} & \{\!|P \mid Q|\!\} &\triangleq \{\!|P|\!\} \mid \{\!|Q|\!\} \\
\{\!|\texttt{get}\ a.M|\!\} &\triangleq \overline{\texttt{in}}\ a.\texttt{open}\ a.\{\!|M|\!\} & \{\!|!\,P|\!\} &\triangleq !\,\{\!|P|\!\} \\
\{\!|\texttt{put}\ a|\!\} &\triangleq \texttt{in}\ a.\overline{\texttt{open}}.\mathbf{0} & \{\!|(\nu a)P|\!\} &\triangleq (\nu a : \curvearrowright^{1} [\veebar^{0}])\{\!|P|\!\} \\
\{\!|\texttt{dis}|\!\} &\triangleq \overline{\texttt{open}}.\mathbf{0} & \{\!|(\nu w)P|\!\} &\triangleq (\nu w : \veebar^{\omega})\{\!|P|\!\} \\
\{\!|a^{\downarrow}|\!\} &\triangleq !\,\overline{\texttt{in}}\ a & \{\!|a(M)[P]|\!\} &\triangleq a[\{\!|M|\!\} \mid \{\!|P|\!\}] \\
\{\!|a^{\uparrow}|\!\} &\triangleq !\,\overline{\texttt{out}}\ a & \{\!|w\langle C\rangle[P]|\!\} &\triangleq w[\{\!|C|\!\} \mid \{\!|P|\!\}] \\
\{\!|a^{-}|\!\} &\triangleq !\,\texttt{open}\ a & & \\
\{\!|C \mid D|\!\} &\triangleq \{\!|C|\!\} \mid \{\!|D|\!\} & &
\end{aligned}$$

Fig. 4. Translating $\texttt{PR}^{-}$ to ROAM

behaved. We separate the set of ambient names to agent names (ranged over by $a, b, ...$) and wall names (ranged over by $w, v, ...$), and define *pure ROAM minus* ($\texttt{PR}^{-}$), a sub-calculus of ROAM in Figure 3. The reduction semantics is defined to be compliant with that of ROAM.

Clearly, every $\texttt{PR}^{-}$ process is also a ROAM process. Translating a $\texttt{PR}^{-}$ process P back to a ROAM process, $\{\!|P|\!\}$, can be defined easily in Figure 4.

We have the following result on the translation.

Networks:

$N, M ::= \mathbf{0}$		empty
	$M \mid N$	composition
	$(\nu l)N$	location res.
	$(\nu_l\, a)N$	channel res.
	$l[\![P]\!]$	location

Processes:

$P, Q ::= \mathtt{stop}$		empty
	$P \mid Q$	composition
	$!P$	replication
	$(\nu l)P$	location res.
	$(\nu a)P$	channel res.
	$\mathtt{go}\ u_{loc}.P$	migration
	$u_{chn}\langle u\rangle.P$	write value
	$u_{chn}(x).P$	read value

$u_{loc} ::= l \mid x_{loc}$ $\quad u ::= u_{loc} \mid u_{chn}$

$u_{chn} ::= a \mid x_{chn}$ $\quad x ::= x_{loc} \mid x_{chn}$

Evaluation contexts:

$$E ::= - \mid E \mid N \mid (\nu l)N \mid (\nu_l\, a)E$$

Reduction axioms:

$$\begin{aligned}
l[\![P \mid Q]\!] &\longrightarrow l[\![P]\!] \mid l[\![Q]\!] \\
l[\![\mathtt{go}\ k.P]\!] &\longrightarrow k[\![P]\!] \\
l[\![a\langle u_{loc}\rangle.P]\!] \mid l[\![a(x_{loc}).Q]\!] &\longrightarrow l[\![P]\!] \mid l[\![Q\{x_{loc} := u_{loc}\}]\!] \\
l[\![a\langle u_{chn}\rangle.P]\!] \mid l[\![a(x_{chn}).Q]\!] &\longrightarrow l[\![P]\!] \mid l[\![Q\{x_{chn} := u_{chn}\}]\!]
\end{aligned}$$

Fig. 5. Syntax and reduction semantics of Dπ

Theorem 4.1 *For any* $\mathtt{PR}^-$ *process* P, $\{\!|P|\!\}$ *is a well-behaved ROAM process.*

Proof. This can be proved easily according to definition of $\{\!|P|\!\}$. Every agent ambient can be assigned with type $\curvearrowright^1 [\underline{\vee}^0]$. Every wall ambient can be assigned with type $\underline{\vee}^\omega$. □

Albeit a sub-calculus of ROAM, $\mathtt{PR}^-$ also exhibits a packet-routing model at an higher abstraction level than ROAM. It makes ROAM processes more readable and makes it easier to write complex processes. In the next section, we show an encoding of Dπ in ambients written in $\mathtt{PR}^-$.

5 The encoding of Dπ

The calculus of Dπ [HR98] adds distributed location on top of the π-calculus. Processes are located and written $l[\![P]\!]$. A new primitive **go** $l.P$ can migrate process P to location l. Communication can only happen if both the read process and the write process reside at the same location. During communication, either channel names or location names can be communicated.

For the sake of encoding, we distinguish channels from locations. We let l, k range over location names, a, b range over channel names, x_{loc}, y_{loc} range over location variables, and x_{chn}, y_{chn} range over channel variables. The syntax and reduction semantics of Dπ is summarized in Figure 5.

To simulate Dπ, we use two layers of walls: a top layer for Dπ locations, and a second layer for Dπ channels. To realize substitution, we need to build, as what we have done in the encoding of π, redirection ambients $x[\mathbf{fwd}\ u]$ that can route communication ambients *comm* to the right channel. In Dπ,

both channels and locations can be communicated. We identify 4 kinds of redirection ambients as follows:

(i) $x_{loc}[\mathbf{fwd}(l)]$: location variable x_{loc} instantiated as location name l;

(ii) $x_{loc}[\mathbf{fwd}(y_{loc})]$: location variable x_{loc} instantiated as another location variable y_{loc};

(iii) $x_{chn}[\mathbf{fwd}(a)]$: channel variable x_{chn} instantiated as channel name a;

(iv) $x_{chn}[\mathbf{fwd}(y_{chn})]$: channel variable x_{chn} instantiated as another channel variable y_{chn}.

We need to decide which layer should those redirection ambients be located. Those redirection ambients for location variables should be parallel with locations, so that it makes little different between processes migrating to locations and processes migrating to redirection ambients. As for redirection ambients for channel variables, we put them also in parallel with locations, since otherwise, if we put them in parallel with channels, we need to create one such redirecting ambients in every location, even in newly created ones.

One interesting property in Dπ is the ability to dynamically bind channel names. For example, in process $b(x).\mathtt{go}\ x.a(y).P$, the location of channel a is not known in advance. The actual channel that a will interact with depends on the value it receives from b. To encode this process using redirection ambients, the process $a(y).P$ must find its way back to where it from after interaction with the redirection ambient y outside of its location. We need some tricks in the encoding to enable processes come back to where they were. For this reason, we add two primitives in $\mathtt{PR}^-$: "$\mathtt{wait}\ a$" and "$\mathtt{sig}\ a$". They enable an agent to "wait" for some activities of its cargo before it fires the dissolve action. Their definitions and operational semantics are summarized below.

$$M ::= ... \quad | \quad \mathtt{wait}\ a.M \quad | \quad \mathtt{sig}\ a.M$$

$$a(\mathtt{wait}\ s.M)[P \mid b(\mathtt{sig}\ s.N)[Q]] \longrightarrow a(M)[P \mid b(N)[Q]]$$

Their corresponding implementation in ROAM are defined as:

$$\{\!|\mathtt{wait}\ s.M|\!\} \stackrel{\triangle}{=} \overline{\mathtt{out}}\ s.\mathtt{open}\ s.\{\!|M|\!\}$$

$$\{\!|\mathtt{sig}\ s.M|\!\} \stackrel{\triangle}{=} s[\mathtt{out}.\overline{\mathtt{open}}] \mid \{\!|M|\!\}$$

It is easy to verify that Theorem 4.1 in the previous section still holds with these extensions, and $\mathtt{PR}^-$ with these addition is still a sub-calculus of ROAM.

Now we can write the encoding from Dπ network to $\mathtt{PR}^-$ processes: $\{\!\{N\}\!\}^*$.

$$\begin{aligned}
\{\!\{N\}\!\}^* &\stackrel{\triangle}{=} root\langle tonet^- \rangle[\{\!\{N\}\!\}] \\
\{\!\{\mathbf{0}\}\!\} &\stackrel{\triangle}{=} \mathbf{0} \\
\{\!\{N \mid M\}\!\} &\stackrel{\triangle}{=} \{\!\{N\}\!\} \mid \{\!\{M\}\!\} \\
\{\!\{(\nu l)N\}\!\} &\stackrel{\triangle}{=} (\nu l)(l\langle \mathbf{C_l} \rangle[\mathbf{B}(l)] \mid \{\!\{N\}\!\}) \\
\{\!\{(\nu_l a)N\}\!\} &\stackrel{\triangle}{=} (\nu a)(toloc(\mathtt{in}\ l.\mathtt{dis})[a\langle \mathbf{C_n} \rangle[\,]] \mid \{\!\{N\}\!\}) \\
\{\!\{l[\![P]\!]\}\!\} &\stackrel{\triangle}{=} toloc(\mathtt{in}\ l.\mathtt{dis})[\{\!\{P\}\!\}]
\end{aligned}$$

$$
\begin{aligned}
\{\!|\texttt{stop}|\!\} &\triangleq \mathbf{0}\\
\{\!|P \mid Q|\!\} &\triangleq \{\!|P|\!\} \mid \{\!|Q|\!\}\\
\{\!|\,!\,P|\!\} &\triangleq !\,\{\!|P|\!\}\\
\{\!|(\nu l)P|\!\} &\triangleq (\nu l)(\mathit{tonet}(\texttt{out.dis})[l\langle\mathbf{C_l}\rangle[\mathbf{B}(l)]] \mid \{\!|P|\!\})\\
\{\!|(\nu a)P|\!\} &\triangleq (\nu a)(a\langle\mathbf{C_n}\rangle[\,] \mid \{\!|P|\!\})\\
\{\!|\texttt{go}\ l.P|\!\} &\triangleq \mathit{toloc}(\texttt{out.in}\ l.\texttt{dis})[\{\!|P|\!\}]\\
\{\!|\texttt{go}\ x_{loc}.P|\!\} &\triangleq \mathit{toloc}(\texttt{out.in}\ x_{loc}.\texttt{put}\ \mathit{toloc})[\{\!|P|\!\}]\\
\{\!|a\langle u\rangle.P|\!\} &\triangleq \mathit{tochn}(\texttt{in}\ a.\texttt{dis})[\mathbf{O}(u,P)]\\
\{\!|x_{chn}\langle u\rangle.P|\!\} &\triangleq \mathit{tovar}(\texttt{get}\ \mathit{back}.\texttt{out.in}\ x_{chn}.\texttt{put}\ \mathit{tovar})\\
&\qquad [\,\mathit{toloc}(\texttt{put}\ \mathit{toloc})[\,\mathit{tochn}(\texttt{put}\ \mathit{tochn})[\mathbf{O}(u,P)]\,]\,]\\
\{\!|a(x).P|\!\} &\triangleq (\nu x)\ (\ \mathit{tonet}(\texttt{out.dis})[\mathbf{Var}(x)]\\
&\qquad \mid \mathit{tochn}(\texttt{in}\ a.\texttt{dis})[\mathbf{I}(x,P)]\ )\\
\{\!|x_{chn}(x).P|\!\} &\triangleq (\nu x)\ (\ \mathit{tonet}(\texttt{out.dis})[\mathbf{Var}(x)]\\
&\qquad \mid \mathit{tovar}(\texttt{get}\ \mathit{back}.\texttt{out.in}\ x_{chn}.\texttt{put}\ \mathit{tovar})\\
&\qquad\qquad [\mathit{toloc}(\texttt{put}\ \mathit{toloc})[\,\mathit{tochn}(\texttt{put}\ \mathit{tochn})[\mathbf{I}(x,P)]]]\,)\\
\mathbf{O}(u,P) &\triangleq \mathit{comm}(\texttt{put}\ \mathit{comm})\\
&\qquad [\,\mathit{tovar}(\texttt{put}\ \mathit{tovar})[\mathbf{fwd}(u)] \mid \mathit{cont}(\texttt{dis})[\{\!|P|\!\}]]\\
\mathbf{I}(x,P) &\triangleq \mathit{comm}(\texttt{get}\ \mathit{comm}.\texttt{out.wait}\ s.\texttt{dis})\\
&\qquad [\,\mathit{tovar}(\texttt{get}\ \mathit{tovar}.\texttt{sig}\ s.\texttt{out.in}\ x.\texttt{dis})[\,]\\
&\qquad \mid\ \mathit{cont}(\texttt{dis})[\{\!|P|\!\}]\,]\\
\mathbf{fwd}(a) &\triangleq !\,\mathit{tovar}(\texttt{get}\ \mathit{tovar}.\texttt{wait}\ s.\texttt{dis})\\
&\qquad [\mathit{toloc}(\texttt{put}\ \mathit{toloc})[\mathit{tochn}(\texttt{get}\ \mathit{tochn}.\texttt{sig}\ s.\texttt{in}\ a.\texttt{dis})[\,]]]\\
\mathbf{fwd}(x_{chn}) &\triangleq !\,\mathit{tovar}(\texttt{get}\ \mathit{tovar}.\texttt{out.in}\ x_{chn}.\texttt{put}\ \mathit{tovar})[\,]\\
\mathbf{fwd}(l) &\triangleq !\,\mathit{toloc}(\texttt{get}\ \mathit{toloc}.\texttt{out.in}\ l.\texttt{dis})[\,]\\
\mathbf{fwd}(x_{loc}) &\triangleq !\,\mathit{toloc}(\texttt{get}\ \mathit{toloc}.\texttt{out.in}\ x_{loc}.\texttt{put}\ \mathit{toloc})[\,]\\
\mathbf{Var}(x_{chn}) &\triangleq x_{chn}\langle\mathbf{C_{vn}}\rangle[\,]\\
\mathbf{Var}(x_{loc}) &\triangleq x_{loc}\langle\mathbf{C_{vl}}\rangle[\,]\\
\mathbf{B}(l) &\triangleq !\,\mathit{back}(\texttt{put}\ \mathit{tochn})[\mathit{toloc}(\texttt{get}\ \mathit{toloc}.\texttt{get}\ \mathit{toloc}.\\
&\qquad\qquad \texttt{sig}\ s.\texttt{out.in}\ l.\texttt{wait}\ s.\texttt{dis})[\,]]\\
\mathbf{C_l} &\triangleq \mathit{toloc}^{\ddagger} \mid \mathit{tonet}^{\uparrow} \mid \mathit{tovar}^{\uparrow} \mid \mathit{comm}^{-} \mid \mathit{cont}^{-}\\
\mathbf{C_n} &\triangleq \mathit{tochn}^{\ddagger} \mid \mathit{comm}^{\uparrow}\\
\mathbf{C_{vl}} &\triangleq \mathit{toloc}^{\updownarrow} \mid \mathit{tovar}^{\ddagger}\\
\mathbf{C_{vn}} &\triangleq \mathit{toloc}^{\uparrow} \mid \mathit{tovar}^{\ddagger}
\end{aligned}
$$

We introduced more system agent names in the encoding. Apart from *comm* and *cont*, which are similar to their counterpart in the encoding of π, we use four different routing agents: *tonet*, *toloc*, *tochn*, and *tovar*. They route cargos respectively to the network level, inside locations, inside channels, or

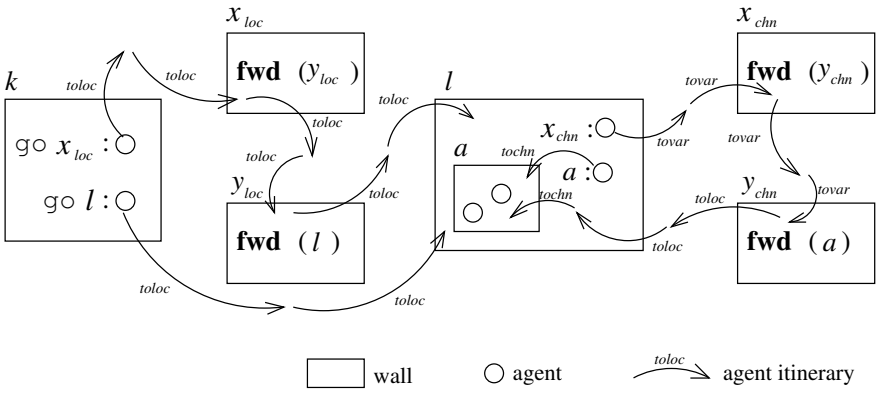


Fig. 6. Wall layouts and agent itineraries in the encoding of Dπ.

inside redirection variables. We always use name s for the signal agent name. The encoding could be better understood with Figure 6 illustrating how the encoding works.

In Figure 6, two locations k and l, together with four redirection ambients are located in the first layer. A channel a is located inside location l. Processes are indicated by circles, and their migration trials are illustrated with arcs. Agent names during migration are shown next to these arcs.

The process $\mathtt{go}\ x_{loc}.P$ inside location k reaches location l after it visits redirection ambient $x_{loc}\langle\mathbf{C_{vl}}\rangle[\mathbf{fwd}\ y_{loc}]$ and $y_{loc}\langle\mathbf{C_{vl}}\rangle[\mathbf{fwd}\ l]$, while process $\mathtt{go}\ l.P$ can go to location l directly. In location l, a communication process on channel x_{chn} first goes out of l and visit redirection ambient $x_{chn}\langle\mathbf{C_{vn}}\rangle[\mathbf{fwd}\ y_{chn}]$ and $y_{chn}\langle\mathbf{C_{vn}}\rangle[\mathbf{fwd}\ a]$, and then goes back to location l and enters channel ambient a, while the communication process on channel a can move directly into channel ambient a.

The encoding is rather complex due to different kinds of redirection ambients. The use of replicated agents *back* inside each location is essential to bring communication processes back after they visit redirection ambients. The use of "`wait`" and "`sig`" prevents possible interference between different communication processes inside redirection ambients. Moreover, the encoding is not very satisfying if we consider the distributed nature of Dπ locations. The top-level redirection ambients make up a central service point that every location depends on for its computation. This should be avoided since a distributed network like Dπ should have as few centralized control as possible. This problem will be alleviated in the next section with some modification to $\mathtt{PR}^{-}$.

6 Towards a routing calculus

In the Dπ encoding, it is crucial for an agent to remember where it was when visiting redirection ambients. This section extends $\mathtt{PR}^{-}$ and include a primitive

Capabilities:			Processes:	
$M, N ::=$	$\texttt{in}\ u.M$	move in	$P, Q ::= \mathbf{0}$	empty
$\mid$	$\texttt{out}(x).M$	move out	$\mid \quad P \mid Q$	composition
$\mid$	$\texttt{get}.M$	get some cargo	$\mid \quad !\,P$	replication
$\mid$	$\texttt{put}$	put cargo	$\mid \quad (\nu n)P$	restriction
$\mid$	$\texttt{dis}$	dissolve	$\mid \quad M\langle P\rangle$	anony. agent
Values: $u ::= n \mid x$			$\mid \quad n[P]$	wall

Evaluation context:

$$E \quad ::= \quad - \quad \mid \quad E \mid P \quad \mid \quad (\nu n)E \quad \mid \quad n[E]$$

Reduction axioms:

$$\begin{array}{rcl} n[P] \mid \texttt{in}\ n.M\langle Q\rangle & \longrightarrow & n[P \mid M\langle Q\rangle] \\ n[P \mid \texttt{out}(x).M\langle Q\rangle] & \longrightarrow & n[P] \mid M\{x{:=}n\}\langle Q\{x{:=}n\}\rangle \\ \texttt{dis}\langle P\rangle & \longrightarrow & P \\ \texttt{get}.M\langle P\rangle \mid \texttt{put}\langle Q\rangle & \longrightarrow & M\langle P \mid Q\rangle \end{array}$$

Fig. 7. The syntax and reduction semantics of the wagon calculus

called *out-binding* that enables agents to come back easily, so as to simplify the previous sophisticated routing protocols. With out-binding, the primitives `wait` and `sig` added in the encoding of Dπ are not needed anymore.

A second alternation to the $\texttt{PR}^-$ is the use of anonymous agents. We write $M\langle P\rangle$ instead of $a(M)[P]$, we drop controllers in walls, and we forbid reduction inside cargos. We call the simplified version the *wagon calculus*. It's syntax and reduction semantics are defined in Figure 7.

In process $\texttt{out}(x).M\langle P\rangle$, variable x in M and P are bound. Substitution is made to both M and P after the agent moves out. We sometime abbreviate $\texttt{out}(x)$ as $\texttt{out}$ if the bind variable x doesn't occur in M and P.

Obviously, wagon is no longer a sub-calculus of ROAM.

Since wagon has also the name substitution primitive. It is easy to write the encoding of π to wagon.

$$\begin{array}{rcl@{\qquad}rcl} \langle\!\langle \mathbf{0} \rangle\!\rangle & \triangleq & \mathbf{0} & \langle\!\langle P \mid Q \rangle\!\rangle & \triangleq & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\ \langle\!\langle\, !\,P \rangle\!\rangle & \triangleq & !\,\langle\!\langle P \rangle\!\rangle & \langle\!\langle (\nu n)P \rangle\!\rangle & \triangleq & (\nu n)(n[\,] \mid \langle\!\langle P \rangle\!\rangle) \\ \langle\!\langle u\langle v\rangle.P \rangle\!\rangle & \triangleq & \multicolumn{4}{l}{\texttt{in}\ u.\texttt{get}.\texttt{out}.\texttt{in}\ v.\texttt{dis}\langle \texttt{out}.\texttt{dis}.\langle\langle\!\langle P \rangle\!\rangle\rangle\rangle} \\ \langle\!\langle u(x).P \rangle\!\rangle & \triangleq & \multicolumn{4}{l}{\texttt{in}\ u.\texttt{put}\langle \texttt{out}(x).\texttt{dis}\langle\langle\!\langle P \rangle\!\rangle\rangle\rangle} \end{array}$$

In the above encoding, an output process can route an input process into the channel corresponding to the output value, the input process then read the channel name with out-binding, thus substitutes all the occurrence of the variable with the desired value.

The major advantage of using wagon is demonstrated in the encoding of Dπ. Now we don't need redirection ambients and sophisticated routing protocols to get agents back. Passing channel names as well as location names

can both be simulated easily using out-binding.

$$\begin{array}{rcl@{\qquad}rcl}
\{\!\{\mathbf{0}\}\!\} & \triangleq & \mathbf{0} & \{\!\{\mathtt{stop}\}\!\} & \triangleq & \mathbf{0} \\
\{\!\{N \mid M\}\!\} & \triangleq & \{\!\{N\}\!\} \mid \{\!\{M\}\!\} & \{\!\{P \mid Q\}\!\} & \triangleq & \{\!\{P\}\!\} \mid \{\!\{Q\}\!\} \\
\{\!\{(\nu l)N\}\!\} & \triangleq & (\nu l)(l[\,] \mid \{\!\{N\}\!\}) & \{\!\{!P\}\!\} & \triangleq & !\{\!\{P\}\!\} \\
\{\!\{(\nu_l a)N\}\!\} & \triangleq & (\nu a)(\mathtt{in}\ l.\mathtt{dis}\langle a[\,]\rangle \mid \{\!\{N\}\!\}) & \{\!\{(\nu l)P\}\!\} & \triangleq & (\nu l)(\mathtt{out.dis}\langle l[\,]\rangle \mid \{\!\{P\}\!\}) \\
\{\!\{l[\![P]\!]\}\!\} & \triangleq & \mathtt{in}\ l.\mathtt{dis}\langle\{\!\{P\}\!\}\rangle & \{\!\{(\nu a)P\}\!\} & \triangleq & (\nu a)(a[\,] \mid \{\!\{P\}\!\})
\end{array}$$

$$\begin{array}{rcl}
\{\!\{\mathtt{go}\ u_{loc}.P\}\!\} & \triangleq & \mathtt{out.in}\ u_{loc}.\mathtt{dis}\langle\{\!\{P\}\!\}\rangle \\
\{\!\{u_{chn}\langle v_{loc}\rangle.P\}\!\} & \triangleq & \mathtt{out}(y).\mathtt{in}\ y.\mathtt{in}\ u_{chn}.\mathtt{get.out.out.in}\ v_{loc}.\mathtt{dis} \\
 & & \quad \langle \mathtt{out.in}\ y.\mathtt{dis}\langle\{\!\{P\}\!\}\rangle\rangle \quad y \text{ fresh} \\
\{\!\{u_{chn}\langle v_{chn}\rangle.P\}\!\} & \triangleq & \mathtt{in}\ u_{chn}.\mathtt{get.out.in}\ v_{chn}.\mathtt{dis}\langle\mathtt{out.dis}\langle\{\!\{P\}\!\}\rangle\rangle \\
\{\!\{u_{chn}(x_{loc}).P\}\!\} & \triangleq & \mathtt{out}(y).\mathtt{in}\ y.\mathtt{in}\ u_{chn}.\mathtt{put} \\
 & & \quad \langle \mathtt{out}(x_{loc}).\mathtt{in}\ y.\mathtt{dis}\langle\{\!\{P\}\!\}\rangle\rangle \quad y \text{ fresh} \\
\{\!\{u_{chn}(x_{chn}).P\}\!\} & \triangleq & \mathtt{in}\ u_{chn}.\mathtt{put}\langle\mathtt{out}(x_{chn}).\mathtt{dis}\langle\{\!\{P\}\!\}\rangle\rangle
\end{array}$$

The output processes either route the input processes to locations, when they want to send location values, or to channels inside locations, when they want to send channel values. Note that for exchanging a location, both the input and output process need to return to their original location after communication, so they need to remember where they were by reading the location name with $\mathtt{out}(y).\mathtt{in}\ y$ at the beginning.

Compared with the encoding into $\mathtt{PR}^-$ in the previous section, this encoding is clearly simpler and doesn't suffer the centralized control problem of the former.

7 Conclusion

The first result of this paper is a simplified encoding of π-calculus into pure ambients. We then study an interesting sub-calculus of ROAM called $\mathtt{PR}^-$. $\mathtt{PR}^-$ syntactically defines a subset of well-behaved ROAM processes, and holds most of its expressiveness. We even managed to give an encoding of Dπ in it. Although reference [CDGS03] gives an encoding of Dπ into a version of ambient calculus with anonymous communication, we are not aware of any previous encoding of Dπ into pure ambients.

To simplify the encoding of π and Dπ in $\mathtt{PR}^-$, we extend $\mathtt{PR}^-$ to wagon, where agents are anonymous and can know their current location names. By such modifications, the wagon calculus exhibits a pure packet routing model of computation, which is simple but expressive. We conjecture that most of the expressive power of ambients lies in this computation model. We plan to study the behavior theory of the wagon calculus in the near future.

The encodings in this paper are not formally proved. Those encodings in $\mathtt{PR}^-$ should be easy, since $\mathtt{PR}^-$ processes are simply well-behaved ROAM

processes. Proofs for those encodings in wagon would also be easy, after the development of its equational theory similar to that of ROAM.

References

[CDGS03] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *CATS 2003*, volume 78 of *ENTCS*, 2003.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*: 140–155.

[Gua02] Xudong Guan. *Type system and algebrais theory of robust ambients*. PhD thesis, Department of Computer Science and Engineering, Shanghai Jiao Tong Univ., 2002. in Chinese. An extended abstract in English is avaible at: http://www-sop.inria.fr/mimosa/personnel/Xudong.Guan.

[GYY00] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proc. ICS2000*, pages 377–384, 2000.

[GYY01] X. Guan, Y. Yang, and J. You. Typing evolving ambients. *Information Processing Letters*, 80(5):265–270, 2001.

[HR98] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version as CogSci Report 2/98, University of Sussex, Brighton.

[LS00] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00*, pages 352–364. ACM, January 2000.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001. To appear.

[Zim00] Pascal Zimmer. On the expressiveness of pure mobile ambients. In Luca Aceto and Björn Victor, editors, *Preliminary Proceedings of EXPRESS '00*, volume NS-00-2 of *BRICS Notes Series*, pages 81–104, 2000. The Final Proceedings will be published as volume 39 of ENTCS, Elsevier Science Publishers.

A Previous encodings

As an appendix, we list here two previous encodings of π-calculus in pure ambients.

A.1 Zimmer's encoding in SA [Zim00]

$$
\begin{array}{rcl}
\textbf{server}\ n.P & \triangleq & !\,enter[\mathtt{in}\ n.\overline{\mathtt{open}}\ enter.P] \\
\textbf{request}\ rw\ n & \triangleq & \mathtt{in}\ n.\overline{\mathtt{in}}\ rw.\mathtt{open}\ enter \\
\textbf{request}\ rw\ x & \triangleq & \mathtt{in}\ x.\overline{\mathtt{in}}\ rw.\mathtt{open}\ enter.\mathtt{out}\ x \\
\textbf{fwd}\ \ u & \triangleq & \textbf{server}\ write.\textbf{request}\ write\ u \mid \textbf{server}\ read.\textbf{request}\ read\ u \\
n\ \textbf{be}\ m.P & \triangleq & m[\mathtt{out}\ n.\overline{\mathtt{in}}\ m.(\mathtt{open}\ n \mid P)] \mid \overline{\mathtt{out}}\ n.\mathtt{in}\ m.\overline{\mathtt{open}}\ n \\
\textbf{allowIO}\ n & \triangleq & !\,\overline{\mathtt{in}}\ n \mid !\,\overline{\mathtt{out}}\ n \\
\{\![\mathbf{0}]\!\} & \triangleq & \mathbf{0} \\
\{\![P \mid Q]\!\} & \triangleq & \{\![P]\!\} \mid \{\![Q]\!\} \\
\{\![\,!\,P]\!\} & \triangleq & !\,\{\![P]\!\} \\
\{\![(\nu n)P]\!\} & \triangleq & (\nu n)\ (\ n[\ \textbf{allowIO}\ n \\
 & & \qquad \mid\ \textbf{server}\ read.(\nu p) \\
 & & \qquad\quad (\overline{\mathtt{out}}\ read.read\ \textbf{be}\ p.\overline{\mathtt{in}}\ p.\mathtt{out}\ n.p\ \textbf{be}\ read \\
 & & \qquad\quad \mid\ enter[\mathtt{out}\ read.\mathtt{in}\ write.\overline{\mathtt{open}}\ enter. \\
 & & \qquad\qquad\quad \mathtt{in}\ p.\overline{\mathtt{open}}\ write])] \\
 & & \quad \mid \{\![P]\!\}) \\
\{\![u\langle v\rangle.P]\!\} & \triangleq & (\nu p)\ (\ write\ [\ \textbf{request}\ write\ u \\
 & & \qquad \mid \textbf{fwd}\ \ v \\
 & & \qquad \mid p[\mathtt{out}\ read.\overline{\mathtt{open}}\ p.\{\![P]\!\}]] \\
 & & \quad \mid \mathtt{open}\ p) \\
\{\![u(x).P]\!\} & \triangleq & (\nu p)\ (\ \ read\ [\ \textbf{request}\ read\ u \\
 & & \qquad \mid \mathtt{open}\ write.\overline{\mathtt{out}}\ read.(\nu x)\ read\ \textbf{be}\ x. \\
 & & \qquad\quad (\ \overline{\mathtt{out}}\ x.\textbf{allowIO}\ x \\
 & & \qquad\quad \mid p[\mathtt{out}\ x.\overline{\mathtt{open}}\ p.\{\![P]\!\}])] \\
 & & \quad \mid \mathtt{open}\ p)
\end{array}
$$

A.2 Our previous encoding in ROAM [Gua02]

$$
\begin{array}{rcl}
\textbf{server}\ rw.P & \triangleq & !\,enter[\ \mathtt{in}\ rw.\overline{\mathtt{open}}.P] \\
\textbf{request}\ n & \triangleq & \mathtt{in}\ n.\overline{\mathtt{in}}\ enter.\mathtt{open}\ enter \\
\textbf{request}\ x & \triangleq & \mathtt{in}\ x.\overline{\mathtt{in}}\ enter.\mathtt{out}\ x.\mathtt{open}\ enter \\
\textbf{fwd}\ \ M & \triangleq & \textbf{server}\ r.\textbf{request}\ M \mid \textbf{server}\ w.\textbf{request}\ M \\
\textbf{allowIO} & \triangleq & !\,\overline{\mathtt{in}}\ r \mid !\,\overline{\mathtt{out}}\ r \mid !\,\overline{\mathtt{in}}\ w \mid !\,\overline{\mathtt{out}}\ w \\
\textbf{wHead} & \triangleq & \mathtt{in}\ r.\overline{\mathtt{open}} \\
\textbf{wTail}(M,\ P) & \triangleq & \quad c\ \ [\,\mathtt{out}\ r.\overline{\mathtt{open}}.\langle\!\langle P\rangle\!\rangle] \\
 & & \mid w_1\,[\,\mathtt{in}\ r_1.\overline{\mathtt{open}}.(\textbf{fwd}\ \ M \mid \textbf{allowIO})\,] \\
\textbf{rHead}(n) & \triangleq & \overline{\mathtt{in}}\ w.\mathtt{out}\ n.\overline{\mathtt{out}}\ c.\mathtt{open}\ w.\overline{\mathtt{out}}\ c.\mathtt{open}\ r_2.\overline{\mathtt{open}} \\
\textbf{rTail}(x,\ P) & \triangleq & (\nu x)
\end{array}
$$

$$
\begin{array}{rcl}
 & & (\ \ c\ [\,\mathtt{out}\ r.\overline{\mathtt{open}}.\langle\!\langle P\rangle\!\rangle\,] \\
 & & \mid\ r_1\,[\,\overline{\mathtt{in}}\ w_1.\mathtt{in}\ x.\overline{\mathtt{open}}\,] \\
 & & \mid\ x\ \ [\,\overline{\mathtt{in}}\ r_1.\overline{\mathtt{out}}\ r_2.\mathtt{open}\ r_1.\mathtt{open}\ w_1 \\
 & & \quad\ \mid r_2[\,\mathtt{out}\ x.\overline{\mathtt{open}}\,]\,]\,) \\
\mathbf{chn}(n) & \stackrel{\triangle}{=} & \mathbf{allowIO} \mid \mathbf{server}\ w.\mathbf{wHead} \mid \mathbf{server}\ r.\mathbf{rHead}(n) \\
\langle\!\langle \mathbf{0} \rangle\!\rangle & \stackrel{\triangle}{=} & \mathbf{0} \\
\langle\!\langle P \mid Q \rangle\!\rangle & \stackrel{\triangle}{=} & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle\, !\, P \rangle\!\rangle & \stackrel{\triangle}{=} & !\,\langle\!\langle P \rangle\!\rangle \\
\langle\!\langle (\nu n)P \rangle\!\rangle & \stackrel{\triangle}{=} & (\nu n)(n[\mathbf{chn}(n)] \mid \langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle M\langle M'\rangle.P \rangle\!\rangle & \stackrel{\triangle}{=} & w[\mathbf{request}\ M \mid \mathbf{wTail}(M',\ P)] \\
\langle\!\langle M(x).P \rangle\!\rangle & \stackrel{\triangle}{=} & r[\mathbf{request}\ M \mid \mathbf{rTail}(x,\ P)] \mid \mathtt{open}\ c \mid \mathtt{open}\ c \mid \mathtt{open}\ r
\end{array}
$$

B Comparing the Three Encodings

This appendix gives some comparisons of the three encodings, namely Zimmer00, Guan02, and the one in this paper (Guan03). We use two metrics in our comparison: *size* and *complexity*.

For easy approximation, we define the size of an encoding to be the total number of actions (including macros) used in the right side of "$\stackrel{\triangle}{=}$". This is surely not very accurate. It just serves as a fast approximation.

We define complexity to be the number of reduction steps in the target language needed to simulate one π-reduction. Since a π-reduction in ambient encoding can be divided roughly into the following three stages, we will further list the steps required in each of these stages.

Redirection : I/O processes interacting with redirection ambients (zero or more redirections may be needed to enter the next stage);

Pre-communication : I/O processes entering the right communication channel and getting ready before they select each other;

Post-communication : I/O processes selecting each other and making the new redirection ambients.

As a means to understand the mechanisms of each encoding, we list below the detailed reduction steps in each of these stages. Each reduction step is represented with the subject, the action, and the object (we use the surrounding ambient as the subject of "`open`"). Those reduction steps that can be executed in parallel are listed in parallel. We distinguish actions taken by input and output processes in the first two stages.

Redirection :

Zimmer00 (8 steps):

$$\begin{cases} write\ \texttt{in}\ x \ \rightarrow\ enter\ \texttt{in}\ write \ \rightarrow\ write\ \texttt{open}\ enter \ \rightarrow\ write\ \texttt{out}\ x \\ read\ \texttt{in}\ x \ \rightarrow\ enter\ \texttt{in}\ read \ \rightarrow\ read\ \texttt{open}\ enter \ \rightarrow\ read\ \texttt{out}\ x \end{cases}$$

Guan02 (8 steps):

$$\begin{cases} w\ \texttt{in}\ x \ \rightarrow\ enter\ \texttt{in}\ w \ \rightarrow\ w\ \texttt{out}\ x \ \rightarrow\ w\ \texttt{open}\ enter \\ r\ \texttt{in}\ x \ \rightarrow\ enter\ \texttt{in}\ r \ \rightarrow\ r\ \texttt{out}\ x \ \rightarrow\ r\ \texttt{open}\ enter \end{cases}$$

Guan03 (8 steps):

$$\begin{cases} route\ \texttt{in}\ x \ \rightarrow\ route\ \texttt{in}\ route \ \rightarrow\ route\ \texttt{open}\ route \ \rightarrow\ route\ \texttt{out}\ x \\ route\ \texttt{in}\ x \ \rightarrow\ route\ \texttt{in}\ route \ \rightarrow\ route\ \texttt{open}\ route \ \rightarrow\ route\ \texttt{out}\ x \end{cases}$$

Pre-communication :

Zimmer00 (8 steps):

$$\begin{cases} write\ \texttt{in}\ n \\ read\ \texttt{in}\ n \ \rightarrow\ enter\ \texttt{in}\ read \ \rightarrow\ read\ \texttt{open}\ enter \ \rightarrow\ (1) \end{cases}$$

$$(1) \ \rightarrow\ enter\ \texttt{out}\ read \ \rightarrow\ p\ \texttt{out}\ read \ \rightarrow\ read\ \texttt{in}\ p \ \rightarrow\ p\ \texttt{open}\ read$$

Guan02 (6 steps):

$$\begin{cases} w\ \texttt{in}\ n \ \rightarrow\ enter\ \texttt{in}\ w \ \rightarrow\ w\ \texttt{open}\ enter \\ r\ \texttt{in}\ n \ \rightarrow\ enter\ \texttt{in}\ r \ \rightarrow\ r\ \texttt{open}\ enter \end{cases}$$

Guan03 (8 steps):

$$\begin{cases} route\ \texttt{in}\ n \ \rightarrow\ route\ \texttt{in}\ route \ \rightarrow\ route\ \texttt{open}\ route \ \rightarrow\ n\ \texttt{open}\ route \\ route\ \texttt{in}\ n \ \rightarrow\ route\ \texttt{in}\ route \ \rightarrow\ route\ \texttt{open}\ route \ \rightarrow\ n\ \texttt{open}\ route \end{cases}$$

Post-communication :

Zimmer00 (15 steps):

$$enter\ \texttt{in}\ write \ \rightarrow\ write\ \texttt{open}\ enter \ \rightarrow\ write\ \texttt{in}\ p \ \rightarrow\ p\ \texttt{out}\ n \ \rightarrow\ (1)$$

$$(1) \ \rightarrow\ read\ \texttt{out}\ p \ \rightarrow\ p\ \texttt{in}\ read \ \rightarrow\ read\ \texttt{open}\ p \ \rightarrow\ (2)$$

$$(2) \ \rightarrow\ read\ \texttt{open}\ write \ \rightarrow\ p\ \texttt{out}\ read \ \rightarrow\ \begin{cases} root\ \texttt{open}\ p \\ x\ \texttt{out}\ read \ \rightarrow\ (3) \end{cases}$$

$$(3) \ \rightarrow\ read\ \texttt{in}\ x \ \rightarrow\ x\ \texttt{open}\ read \ \rightarrow\ p\ \texttt{out}\ x \ \rightarrow\ root\ \texttt{open}\ p$$

Guan02 (14 steps):

$$w\ \texttt{in}\ r \ \rightarrow\ r\ \texttt{out}\ n \ \rightarrow\ c\ \texttt{out}\ r \ \rightarrow\ (1)$$

$$(1) \ \rightarrow\ \begin{cases} root\ \texttt{open}\ c \\ r\ \texttt{open}\ w \ \rightarrow\ \begin{cases} c\ \texttt{out}\ r \ \rightarrow\ root\ \texttt{open}\ c \\ w_1\ \texttt{in}\ r_1 \ \rightarrow\ r_1\ \texttt{in}\ x \ \rightarrow\ r_2\ \texttt{out}\ x \ \rightarrow\ (2) \end{cases} \end{cases}$$

$$(2) \rightarrow \begin{cases} r \texttt{ open } r_2 \rightarrow root \texttt{ open } r \\ x \texttt{ open } r_1 \rightarrow x \texttt{ open } w_1 \end{cases}$$

Guan03 (9 steps):

$$comm \texttt{ in } comm \rightarrow comm \texttt{ open } comm \rightarrow comm \texttt{ out } n \rightarrow (1)$$

$$(1) \rightarrow comm \texttt{ in } x \rightarrow x \texttt{ open } comm \rightarrow \begin{cases} cont \texttt{ out } x \rightarrow root \texttt{ open } cont \\ cont \texttt{ out } x \rightarrow root \texttt{ open } cont \end{cases}$$

We summarized our result in the following table. From this table, we can conclude that the encoding in this paper is simpler than both Zimmer00 and Guan02.

	Zimmer00	Guan02	Guan03
size	47	56	36
complexity*	$8 \times n + 8 + 15$	$8 \times n + 6 + 14$	$8 \times n + 8 + 9$

Note *: We write complexity as $r * n + pre + post$, where r, pre, and $post$ are the reduction steps in the three stages respectively, and n means the average number of redirection steps a process need to perform before reaching its destination channel, which should be a constant in our comparison.

A Calculus of Bounded Capacities*

F. Barbanera[1], M. Bugliesi[2], M. Dezani-Ciancaglini[3], and V. Sassone[4]

[1] Università di Catania, Viale A.Doria 6, 95125 Catania (Italy)
barba@dmi.unict.it
[2] Università "Cà Foscari", Via Torino 155, 30170 Venezia (Italy)
michele@dsi.unive.it
[3] Università di Torino, Corso Svizzera 185, 10149 Torino (Italy)
dezani@di.unito.it
[4] University of Sussex, Falmer, Brighton BN1 9RH UK
vs@susx.ac.uk

Abstract. Resource control has attracted increasing interest in foundational research on distributed systems. This paper focuses on space control and develops an analysis of space usage in the context of an ambient-like calculus with bounded capacities and weighed processes, where migration and activation require space. A type system complements the dynamics of the calculus by providing static guarantees that the intended capacity bounds are preserved throughout the computation.

Introduction

Emerging computing paradigms, such as Global Computing and Ambient Intelligence, envision scenarios where mobile devices travel across domains and networks boundaries. Current examples include smart cards, embedded devices (e.g. in cars), mobile phones, PDAs, and the list keeps growing. The notion of third-party resource usage will raise to a central role, as roaming entities will need to borrow resources from host networks and, in turn, provide guarantees of bounded resource usage. This is the context of the present paper, which focuses on *space consumption* and *capacity bound* awareness.

Resource control, in diverse incarnations, has recently been the focus of foundational research. Topics considered include the ability to read from and to write to a channel [22], the control of the location of channel names [29], the guarantee that distributed agents will access resources only when allowed to do so [12, 21, 2, 10, 11]. Specific work on the certification of bounds on resource consumption include [13], which introduces a notion of resource type representing an abstract unit of space, and uses a linear type system to guarantee linear space consumption; [7] where quantitative bounds on time usage are enforced using a typed assembly language; and [15], which puts forward a general formulation of resource usage analysis.

* F. Barbanera is partially supported by MIUR project NAPOLI, M. Bugliesi by EU-FET project 'MyThS' IST-2001-32617, and by MIUR project MEFISTO, M. Dezani-Ciancaglini by EU-FET project DART IST-2001-33477, and by MIUR Projects COMETA and McTati, V. Sassone by EU-FET project 'MIKADO' IST-2001-32222. The funding bodies are not responsible for any use that might be made of the results presented here.

We elect to formulate our analysis of space control in an ambient-like calculus, BoCa, because the notion of ambient mobility is a natural vehicle to address the intended application domain. Relevant references to related work in this context include [8], which presents a calculus in which resources may be moved across locations provided suitable space is available at the target location; [27], which uses typing systems to control resource usage and consumption; and [6], which uses static techniques to analyse the behaviour of finite control processes, i.e., those with bounded capabilities for ambient allocation and output creation.

Overview. BoCa relies on a physical, yet abstract, notion of "resource unit" defined in terms of a new process constructor, noted ▬ (read "*slot*"), which evolves out of the homonym notion of [8]. A slot may be interpreted as a unit of computation space to be allocated to running processes and migrating ambients. To exemplify, the configuration

$$P \mid \underbrace{▬ \mid \ldots \mid ▬}_{\mathsf{k}\ \text{times}}$$

represents a system which is running process P and which has k resource units available for P to spawn – i.e. activate – new subprocesses and to accept migrating agents willing to enter. In both cases, the activation of the new components is predicated to the presence of suitable resources: only processes and agents requiring cumulatively no more than k units may be activated on the system. As a consequence, process activation and agent migration involve a protocol to "negotiate" the use of resources with the enclosing, resp. receiving, context (possibly competing with other processes).

For migrating agents this is accounted for by associating each agent with a tag representing the space required for activation at the target context, as in $a^{\mathsf{k}}[\,P\,]$. A notion of well-formedness will ensure that k provides a safe estimate of the space needed by $a[\,P\,]$; namely, the number of resource units allocated to P. Correspondingly, the negotiation protocol for mobility is represented formally by the following reductions (where $▬^{\mathsf{k}}$ is short for $▬ \mid \ldots \mid ▬$, k times):

$$a^{\mathsf{k}}[\,\mathbf{in}\ b.P \mid Q\,] \mid b[\,▬^{\mathsf{k}} \mid R\,] \quad \searrow \quad ▬^{\mathsf{k}} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,] \mid R\,]$$
$$▬^{\mathsf{k}} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\ b.Q \mid R\,]\,] \quad \searrow \quad a^{\mathsf{k}}[\,Q \mid R\,] \mid b[\,P \mid ▬^{\mathsf{k}}\,]$$

In both cases, the migrating agent releases the space required for its computation at the source site and gets corresponding space at the target context. Notice that the reductions construe ▬ both as a representation of the physical space available at the locations of the system, and as a particular new kind of co-capability.

Making the weight of an ambient depend explicitly on its contents allows a clean and simple treatment of the open capability: opening does not require resources, as those needed to allocate the contents are exactly those taken by enclosing ambient.

$$\mathbf{open}\ a.P \mid a[\,\overline{\mathbf{open}}.Q \mid R\,] \quad \searrow \quad P \mid Q \mid R$$

Notice that in order for these reductions to provide the intended semantics of resource negotiation, it is crucial that the redexes are well-formed (in the sense discussed above). Accordingly, the dynamics of ambient mobility is inherently dependent on the

assumption that all migrating agents are well-formed. As we shall discuss, this assumption is central to the definition of behavioural equivalence as well.

Resource management and consumption do not concern exclusively mobility, as *all* processes need and use space. In particular, since "spawning" processes requires resources, process replication must be controlled so as to guard against processes that may consume an *infinite* amount of resources. The action of spawning a new process is made explicit in BoCa by introducing a new process construct, the prefix $\mathsf{k}\triangleright$, whose semantics is defined by the following reduction:

$$\mathsf{k}\triangleright P \mid \blacksquare^{\mathsf{k}} \quad \searrow \quad P$$

Here $\mathsf{k}\triangleright P$ is a "spawner" which launches P provided that the local context is ready to allocate enough fresh resources for the activation. The tag k represents the "activation cost" for process P, viz. its weight, while $\mathsf{k}\triangleright P$, the "frozen code" of P, weighs 0: here, again, the hypothesis of well-formedness of terms is critical to make sense of the spawning protocol. The adoption of an explicit spawning operator allows us to delegate to the "spawner" the responsibility of resource control in the mechanism for process replication. In particular, we restrict the replication primitive "$\mathbf{!}$" to 0-weight processes only. We can then rely on the usual congruence rule that identifies $\mathbf{!}P$ with $\mathbf{!}P \mid P$, and use $\mathbf{!}(\mathsf{k}\triangleright P)$ to realise a resource-aware version of replication. This results in a system which separates process *duplication* from process *activation*, and so allows a fine analysis of resource consumption in computation.

The definition of BoCa is completed by two constructs that provide for the dynamic allocation of resources. In our approach resources are not "created" from the void, but rather acquired dynamically – in fact, transferred – from the context, again as a result of a negotiation.

$$a^{\mathsf{k}+1}[\,\mathbf{put}.P \mid \blacksquare \mid Q\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\,a.R \mid S\,] \quad \searrow \quad a^{\mathsf{k}}[\,P \mid Q\,] \mid b^{\mathsf{h}+1}[\,R \mid \blacksquare \mid S\,]$$

$$\mathbf{put}^{\downarrow}.P \mid \blacksquare \mid a^{\mathsf{k}}[\,\mathbf{get}^{\uparrow}.Q \mid R\,] \quad \searrow \quad P \mid a^{\mathsf{k}+1}[\,\blacksquare \mid Q \mid R\,]$$

Resource transfer is realised as a two-way synchronisation in which a context offers some of its resource units to any enclosed or sibling ambient that makes a corresponding request. The effect of the transfer is reflected in the tags that describe the resources allocated to the receiving ambients. We formalise slot transfers only between siblings and from father to child. As we shall see, transfers across siblings make it possible to encode a notion of private resource, while transfer from child to parent can easily be encoded in terms of the existing constructs.

Contributions and main results. Resource allocation in BoCa is controlled by a system of capacity types that guarantees capacity bounds on computational ambients. In this system, ambient types allow resource control policies to be specified by imposing capacity bounds that control the movement and the spawning of processes inside ambients. Then, the typing system enables us to certify statically the absence of under-/over-flows, potentially arising from an uncontrolled use of the capabilities for dynamic space allocation.

Additional, and finer, control on the dynamics of space allocation/deallocation is provided by means of a naming mechanism for slots. We introduce it with a refined version of the calculus in which the notion of private resource is made primitive, rather than

encoded. As we will show, the new mechanisms provide the calculus with a rich and tractable algebraic theory. The semantics theory of the refined calculus is supported by a labelled transition system, yielding a bisimulation congruence adequate with respect to barbed congruence. Besides enabling powerful co-inductive characterizations of process equivalences, the labelled transition system yields an effective tool for contextual reasoning on process behavior. More specifically, it enables a formal representation of *open systems*, in which processes may acquire resources and space from their enclosing context.

Our approach in the definition of BoCa is typical of a way to couple language design with type analysis. This coupling is critical in frameworks like Global Computing, where it is ultimately unrealistic to assume acquaintance with all the entities which may in the future interact with us, as it is usually done for standard type systems. The openness of the network and its very dynamic nature deny us any substantial form of global knowledge. Therefore, syntactic constructs must be introduced to support the static analysis, as e.g., our "negotiation" protocols. In our system, the possibility of dynamically checking particular space constraints is a consequence of the explicit presence of the primitive $\blacksquare$.

Structure of the paper. In §1 we give the formal description of BoCa and illustrate it with a few examples. In §2 we introduce the system of capacity types, and prove it sound. In §3 we introduce the refined calculus and study its its operational and behavioural semantics (based on barbed congruence). We conclude with final remarks in §4. A separate appendix describes the labelled transition system for the calculus.

A preliminary version of this paper appeared in [1].

1 The calculus BoCa

The calculus is a conservative extension of the Ambient Calculus. We presuppose two mutually disjoint sets: $\mathcal{N}$ of names, and $\mathcal{V}$ of variables. The set $\mathcal{V}$ is ranged over by letters at the end of the alphabet, typically x,y,z, while a,b,c,d,n,m range over $\mathcal{N}$. Finally, h,k and other letters in the same font denote integers. The syntax of the (monadic) calculus is defined below, with W an exchange type as introduced in §2.

Definition 1 (Preterms and Terms). The set of process *preterms* is defined by the following productions (where we assume $\mathsf{k} \geq 0$):

$$
\begin{array}{lll}
\textit{Processes} & P & ::= \blacksquare \mid \mathbf{0} \mid \pi.P \mid P \mid P \mid M^{\mathsf{k}}[\,P\,] \mid !\pi.P \mid (\boldsymbol{\nu} a : W)P \\
\textit{Capabilities} & C & ::= \mathbf{in}\, M \mid \mathbf{out}\, M \mid \mathbf{open}\, M \mid \overline{\mathbf{open}} \mid \mathbf{get}\, M \mid \mathbf{get}^{\uparrow} \mid \mathbf{put} \mid \mathbf{put}^{\downarrow} \\
\textit{Messages} & M & ::= a \in \mathcal{N} \mid x \in \mathcal{V} \mid C \mid M.M \\
\textit{Prefixes} & \pi & ::= M \mid (x : W) \mid \langle M \rangle \mid \mathsf{k} \triangleright
\end{array}
$$

A (well-formed) *term* P is a preterm such that $w(P) \neq \bot$, where $w : \mathit{Processes} \rightharpoonup \omega$ is the partial *weight* function defined as follows:

$$\begin{aligned} w(\mathbf{0}) = 0 \qquad w(\_) = 1 \qquad w(P \mid Q) &= w(P) + w(Q) \\ w(M.P) = w((x:W)P) = w(\langle M\rangle P) &= w((\boldsymbol{\nu} a : W)P) = w(P) \\ w(a^{\mathsf{k}}[\,P\,]) &= \text{if } w(P) \text{ is } \mathsf{k} \text{ then } \mathsf{k} \text{ else } \bot \\ w(\mathsf{k} \triangleright P) &= \text{if } w(P) \text{ is } \mathsf{k} \text{ then } 0 \text{ else } \bot \\ w(!P) &= \text{if } w(P) \text{ is } 0 \text{ then } 0 \text{ else } \bot \end{aligned}$$

We use the standard notational conventions for ambient calculi. In particular, we write $(x:W)P$ and $\langle M\rangle P$ for $(x:W).P$ and $\langle M\rangle.P$, respectively, and similarly $\mathsf{k} \triangleright P$ to denote $\mathsf{k} \triangleright .P$. We omit types when not relevant; we write $a[\,P\,]$ instead of $a^{\mathsf{k}}[\,P\,]$ when the value of k does not matter. In this regard, note that in $a^{\mathsf{k}}[\,P\,]$ the weight tag k refers to the ambient process $a[\,P\,]$ and *not* to the name a. We use $\_^{\mathsf{k}}$ as a shorthand for $\_ \mid \ldots \mid \_$ (k times) and C^{k} as a shorthand for $C.\ldots.C$ (again k times). Following a well-known approach, we restrict replication to prefixed processes, as this allows for a simplified treatment in the labelled transition system.

1.1 Reduction

The dynamics of the calculus is defined as usual in terms of structural congruence and reduction (cf. Figure 1). Unlike other calculi, however, in BoCa both relations are only defined for proper terms, a fact we will leave implicit in the rest of the presentation.

The reduction relation $\searrow$ formalizes the intuitions discussed in the introduction; we denote with $\searrow_*$ the reflexive and transitive closure of $\searrow$. Structural congruence is essentially standard. The assumption of well-formedness is central to both relations. In particular, the congruence $!P \equiv P \mid !P$ only holds with P a proper term of weight 0. Thus, to duplicate arbitrary processes we need to first "freeze" them under $\mathsf{k}\triangleright$, i.e. we decompose arbitrary duplication into "template replication" and "process activation." Notice that replication only applies to prefixed processes, a restriction that is technically convenient and that does not involve any significant loss of expressive power.

A few remarks are in order on the form of the transfer capabilities. The **put** capability (among siblings) does not name the target ambient, as is the case for the dual capability **get**. We select this particular combination because it is the most liberal one for which our results hold. Of course, more stringent notions are possible, as e.g. when both partners in a synchronisation use each other's names. Adopting any of these would not change the nature of the calculus and preserve, mutatis mutandis, the validity of our results. In particular, the current choice makes it easy and natural to express interesting programming examples (cf. the memory management in §1.2), and protocols: e.g., it enables us to provide simple encoding of named (and private) resources allocated for spawning (cf. §3). Secondly, a new protocol is easily derived for transferring resources "upwards" from children to parents using the following pair of dual put and get.

$$\mathbf{get}^{\downarrow} a.P \triangleq (\boldsymbol{\nu} m)(\mathbf{open}\, m.P \mid m^0[\,\mathbf{get}\, a.\overline{\mathbf{open}}\,]), \quad \text{and} \quad \mathbf{put}^{\uparrow} \triangleq \mathbf{put}$$

Fig. 1 Structural Congruence and Reduction

Structural Congruence: $(|, \mathbf{0})$ is a commutative monoid.

$$
\begin{array}{lcllcl}
(\boldsymbol{\nu} a)(P \mid Q) & \equiv & (\boldsymbol{\nu} a)P \mid Q \;\; (a \notin \mathrm{fn}(Q)) & \qquad (\boldsymbol{\nu} a)a^0[\,\mathbf{0}\,] & \equiv & \mathbf{0} \\
(\boldsymbol{\nu} a)\mathbf{0} & \equiv & \mathbf{0} & (\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P & \equiv & (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P \\
!P & \equiv & P \mid !P & a[\,(\boldsymbol{\nu} b)P\,] & \equiv & (\boldsymbol{\nu} b)a[\,P\,] \;\; (a \neq b)
\end{array}
$$

Reduction: $E ::= \{\cdot\} \mid E \mid P \mid (\boldsymbol{\nu} m)E \mid m^{\mathsf{k}}[\,E\,]$ is an evaluation context

$$
\begin{array}{lrcl}
(\textsc{enter}) & a^{\mathsf{k}}[\,\mathbf{in}\, b.P \mid Q\,] \mid b[\,\blacksquare^{\mathsf{k}} \mid R\,] & \searrow & \blacksquare^{\mathsf{k}} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,] \mid R\,] \\
(\textsc{exit}) & \blacksquare^{\mathsf{k}} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\, b.Q \mid R\,]\,] & \searrow & a^{\mathsf{k}}[\,Q \mid R\,] \mid b[\,P \mid \blacksquare^{\mathsf{k}}\,] \\
(\textsc{open}) & \mathbf{open}\, a.P \mid a[\,\overline{\mathbf{open}}.Q \mid R\,] & \searrow & P \mid Q \mid R \\
(\textsc{getS}) & a^{\mathsf{k}+1}[\,\mathbf{put}.P \mid \blacksquare \mid Q\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\, a.R \mid S\,] & \searrow & a^{\mathsf{k}}[\,P \mid Q\,] \mid b^{\mathsf{h}+1}[\,R \mid \blacksquare \mid S\,] \\
(\textsc{getD}) & \mathbf{put}^{\downarrow}.P \mid \blacksquare \mid a^{\mathsf{k}}[\,\mathbf{get}^{\uparrow}.Q \mid R\,] & \searrow & P \mid a^{\mathsf{k}+1}[\,\blacksquare \mid Q \mid R\,] \\
(\textsc{spawn}) & \mathsf{k} \triangleright P \mid \blacksquare^{\mathsf{k}} & \searrow & P \\
(\textsc{exchange}) & (x : W)P \mid \langle M \rangle Q & \searrow & P\{x := M\} \mid Q \\
(\textsc{struct}) & P \equiv P' \quad P' \searrow Q' \quad Q' \equiv Q & \Longrightarrow & P \searrow Q \\
(\textsc{context}) & P \searrow Q & \Longrightarrow & E\{P\} \searrow E\{Q\}
\end{array}
$$

Transfers affect the amount of resources allocated at different nesting levels in a system. We delegate to the type system of §2 to control that no nesting level suffers from resource over- or under-flows. The reduction semantics itself guarantees that the global amount of resources is preserved, as it can be proved by an inspection of the reduction rules.

Proposition 1 (Resource preservation). *If $w(P) \neq \bot$, and $P \searrow_* Q$, then $w(Q) = w(P)$.*

Two remarks about the above proposition are worthwhile. First, resource preservation is a distinctive property of *closed* systems; in open systems, instead, a process may acquire new resources from the environment, or transfer resources to the environment, by exercising the **put** and **get** capabilities. Secondly, the fact that the global weight of a process is invariant through reduction does not imply that the amount of resources available for computation also is invariant.

Indeed, our notion of slot is an economical way to convey the three different concepts of a resource being *free*, *allocated*, or *wasted*, according to the context in which $\blacksquare$ occurs during the computation. Unguarded slots, as in $a[\,\blacksquare \mid P\,]$, represent free resources available for spawning or movement at a given nesting level; guarded slots, like $M.\blacksquare$, represent allocated resources, which may be released and become free; and unreachable slots, like $(\boldsymbol{\nu} a)\mathbf{in}\; a.\blacksquare^{\mathsf{k}}$ or $(\boldsymbol{\nu} a)a^{\mathsf{k}}[\,\blacksquare^{\mathsf{k}}\,]$, represent wasted resources that will never be released.

Computation changes the state of resources in the expected ways: allocated resources may be freed, as in $\mathbf{open}\ a.\blacksquare \mid a[\,P\,] \searrow_{\blacksquare} \mid P$; free resources may be allocated, as in $\blacksquare \mid 1 \triangleright M.\blacksquare \searrow M.\blacksquare$, or wasted as in $\mathbf{put}^{\downarrow} \mid \blacksquare \mid (\boldsymbol{\nu} a)a[\,\mathbf{get}^{\uparrow}\,] \searrow (\boldsymbol{\nu} a)a[\,\blacksquare\,]$. No further transition is possible for a wasted resource: in particular, it may never become free, and re-allocated. Accordingly, while the global amount of resources is invariant through reduction, as stated in Proposition 1, the computation of a process does in general consume resources and leaves a possibly decreased, certainly non-increased amount of free and allocated resources. We will return on a more precise analysis resource usage based on the characterization we just outlined in §3.

1.2 Examples

We illustrate the calculus with examples and encodings of systems that require usage and control of space.

Recovering Mobile Ambients. The Ambient Calculus [5] is straightforwardly embedded in (an untyped version of) BoCa: it suffices to insert a process $!\overline{\mathbf{open}}$ in all ambients. The relevant clauses of the embedding are as follows:

$$[\![a[\,P\,]]\!] \triangleq a^0[\,!\overline{\mathbf{open}} \mid [\![P]\!]\,], \quad [\![(\boldsymbol{\nu} a)P]\!] \triangleq (\boldsymbol{\nu} a)[\![P]\!]$$

and the remaining ones are derived similarly; clearly all resulting processes weigh 0.

Parent-child swap. Given that ambients may have non null weights, in BoCa this swap is possible only in case the father and child nodes have the same weight. We present it for example in the case of weight 1. Notice the use of the primitives for child to father slot transfer we defined in §1.

$$b^1[\,\mathbf{get}^{\downarrow} a\,.\,\mathbf{put}\,.\,\mathbf{in}\ a\,.\,\mathbf{get}^{\uparrow} \mid a^1[\,\mathbf{put}^{\uparrow}\,.\,\mathbf{out}\ b\,.\,\mathbf{get}\ b\,.\,\mathbf{put}^{\downarrow} \mid \blacksquare\,]\,] \searrow_* a^1[\,b^1[\,\blacksquare\,]\,]$$

Ambient renaming. We can represent in BoCa a form of ambient self-renaming capability. First, define $\mathbf{spawn}^{\mathsf{k}}\ P\ \mathbf{outside}\ a \triangleq exp^0[\,\mathbf{out}\ a\,.\,\overline{\mathbf{open}}\,.\,\mathsf{k} \triangleright P\,]$ and then use it to define

$$a\ \mathbf{be}^{\mathsf{k}} b\,.\,P \triangleq \mathbf{spawn}^{\mathsf{k}}\ (b^{\mathsf{k}}[\,\blacksquare^{\mathsf{k}} \mid \mathbf{open}\ a\,])\ \mathbf{outside}\ a \mid \mathbf{in}\ b\,.\,\overline{\mathbf{open}}\,.\,P$$

Since $\mathbf{open}\ exp \mid \blacksquare^{\mathsf{k}} \mid a^{\mathsf{h}}[\,\mathbf{spawn}^{\mathsf{k}}\ (b^{\mathsf{k}}[\,P\,] \mid Q)\ \mathbf{outside}\ a\,] \searrow_* b^{\mathsf{k}}[\,P\,] \mid a^{\mathsf{h}}[\,Q\,]$ where k, h are the weights of P and Q, respectively, we get

$$a^{\mathsf{k}}[\,a\ \mathbf{be}^{\mathsf{k}} b\,.\,P \mid R\,] \mid \blacksquare^{\mathsf{k}} \mid \mathbf{open}\ exp \searrow_* b^{\mathsf{k}}[\,P \mid R\,] \mid \blacksquare^{\mathsf{k}}$$

So, an ambient needs to *borrow* space from its parent in order to rename itself. We conjecture that renaming cannot be obtained otherwise.

A memory module. A user can take slots from a memory module MEM_MOD using MALLOC and release them back to MEM_MOD after their use.

$$\begin{aligned}
\text{MEM_MOD} &\triangleq mem[\,\blacksquare^{256MB} \mid \overbrace{\mathbf{open}\ m \mid \ldots \mid \mathbf{open}\ m}^{256MB}\,] \\
\text{MALLOC} &\triangleq m[\,\mathbf{out}\ u\,.\,\mathbf{in}\ mem\,.\,\overline{\mathbf{open}}\,.\,\mathbf{put}\,.\,\mathbf{get}\ u\,.\,\mathbf{open}\ m\,] \\
\text{USER} &\triangleq u[\,\ldots.\text{MALLOC} \mid \ldots.\mathbf{get}\ mem \ldots \mathbf{put} \mid \ldots\,]
\end{aligned}$$

A cab trip. As a further example, we give a new version of the the cab trip protocol from [27], formulated in our calculus. A customer sends a request for a cab, which then arrives and takes the customer to his destination. The use of slots here enables us to model very naturally the constraint that only one passenger (or actually any fixed number of them) may occupy a cab.

$$\begin{array}{ll}
CALL(from,client) & \triangleq \\
\quad call^1[\,\mathbf{out}\ client\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{open}}\,.\,\mathbf{in}\ from\,.(load^0[\,\mathbf{out}\ cab\,.\,\mathbf{in}\ client\,.\,\overline{\mathbf{open}}\,]\mid \blacksquare)\,] & \\
TRIP(from,to,client) & \triangleq trip^0[\,\mathbf{out}\ client\,.\,\overline{\mathbf{open}}\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ to\,.\,done^0[\,\mathbf{in}\ client\,.\,\overline{\mathbf{open}}\,]\,] \\
CLIENT(from,to) & \triangleq (\boldsymbol{\nu} c)c^1[CALL(from,c)\mid \mathbf{open}\ load\,.\,\mathbf{in}\ cab\,.\,TRIP(from,to,c) \\
 & \qquad\qquad \mid \mathbf{open}\ done\,.\,\mathbf{out}\ cab\,.\,bye^0[\,\mathbf{out}\ c\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{open}}\,.\,\mathbf{out}\ to\,]] \\
CAB & \triangleq cab^1[\,\blacksquare\mid !(\mathbf{open}\ call\,.\,\mathbf{open}\ trip\,.\,\mathbf{open}\ bye)\,] \\
SITE(i) & \triangleq site_i[\,CLIENT(site_i,site_j)\mid CLIENT(site_i,site_l)\mid\cdots\mid\blacksquare\mid\blacksquare\mid\cdots] \\
CITY & \triangleq city[\,CAB\mid CAB\mid\cdots\mid\cdots\mid SITE(1)\mid\cdots\mid SITE(n)\mid\blacksquare\mid\blacksquare\mid\cdots\,]
\end{array}$$

The fact that only one slot is available in *cab* together with the weight 1 of both *call* and *client* prevents the cab to carry more than one call and/or more than one client. Moreover this encoding limits also the space in each site and in the whole city.

Comparing with [27], we notice that we can deal with the cab's space satisfactorily with no need for 3-way synchronisations. Unfortunately, as already observed in [27], this encoding may lead to unwanted behaviours, since there is no way of preventing a client to enter a cab different from that called and/or the ambient *bye* to enter a cab different from that the client has left. We discuss these and related issues in further detail below.

1.3 Discussion

In its present definition, the calculus provides a simple, yet effective, framework for expressing resource usage and consumption. On the other hand, as observed above, it is less effective to express and enforce *policies* for resource *allocation*, and their *distribution* to distinct, possibly, competing components.

Policies for resource allocation should provide safeguards against denial-of-service threats, based on the ability of misbehaved agents to attack a host by repeated space transfer requests that could overfill the target host or leave it with no space to spawn its local processes. Similarly, policies for resource distribution should be able to express protocols in which a given resource unit is selectively allocated to a specific agent, and protected against unintended use. To illustrate, consider the following term (and assume it well-formed):

$$a^1[\,\mathbf{in}\ b.P\,]\mid b[\,1\triangleright Q\mid\blacksquare\mid d[\,c^1[\,\mathbf{out}\ d.R\,]\,]\,]$$

Three agents are competing for the resource unit in ambient b: ambients a and c, which would use it for their move, and the local spawner inside ambient b. While the race between a and c may be acceptable – the resource unit may be allocated by b to any

migrating agent – it would also be desirable for b to reserve resources for internal use, i.e. for spawning new processes.

In the remainder of the paper we attack both problems. The system of capacity types in §2 provides static guarantees that the resources available at a given site remain within the intended bounds. The mechanism for slot naming, in §3, yields new and effective primitives for the selective distribution of resources, and for protecting processes against the presence of races for resource acquisition.

2 Bounding Resources – by Typing

As outlined above, the type system provides static guarantees for a simple behavioural property, namely the absence of space under- and over-flows arising as a result of transfers during the computation. To deal with this satisfactorily, we need to take into account that transfer capabilities can be acquired by way of exchanges. The type of a capability will hence have to express how it affects the space of the ambient in which it can be performed.

2.1 Capacity Types

We use $\mathcal{Z}$ to denote the set of integers, and note $\mathcal{Z}^+$ and $\mathcal{Z}^-$ the sets of non-negative and non-positive integers respectively. We define the following domains:

$$\begin{array}{rl} \textit{Intervals} & \iota \in \mathfrak{I} \triangleq \{[\mathsf{n},\mathsf{N}] \mid \mathsf{n},\mathsf{N} \in \mathcal{Z}^+,\ \mathsf{n} \leq \mathsf{N}\} \\ \textit{Effects} & \varepsilon \in \mathcal{E} \triangleq \{(\mathsf{d},\mathsf{i}) \mid \mathsf{d} \in \mathcal{Z}^-,\ \mathsf{i} \in \mathcal{Z}^+\} \\ \textit{Thread Effects} & \phi \in \Phi \triangleq \mathcal{E} \to \mathcal{E} \end{array}$$

Intervals and effects are ordered in the expected way, namely: $[\mathsf{n},\mathsf{N}] \leq [\mathsf{n}',\mathsf{N}']$ when $\mathsf{n}' \leq \mathsf{n}$ and $\mathsf{N} \leq \mathsf{N}'$ and $(\mathsf{d},\mathsf{i}) \leq (\mathsf{d}',\mathsf{i}')$ when $\mathsf{d}' \leq \mathsf{d}$ and $\mathsf{i} \leq \mathsf{i}'$. It is also convenient to define the component-wise sum operator for effects: $(\mathsf{d},\mathsf{i}) + (\mathsf{d}',\mathsf{i}') = (\mathsf{d}+\mathsf{d}',\mathsf{i}+\mathsf{i}')$, and lift it to Φ pointwise: $\phi_1 + \phi_2 = \lambda\varepsilon.\phi_1(\varepsilon) + \phi_2(\varepsilon)$.

The syntax of types is defined by the following productions:

$$\begin{array}{rl} \textit{Message Types} & W ::= \mathit{Amb}\langle\iota,\varepsilon,\chi\rangle \mid \mathit{Cap}\langle\phi,\chi\rangle \\ \textit{Exchange Types} & \chi ::= \mathit{Shh} \mid W \\ \textit{Process Types} & \Pi ::= \mathit{Proc}\langle\varepsilon,\chi\rangle \end{array}$$

Type $\mathit{Proc}\langle\varepsilon,\chi\rangle$ is the type of processes with ε effects and χ exchanges. Specifically, for a process P of type $\mathit{Proc}\langle(\mathsf{d},\mathsf{i}),\chi\rangle$, the effect (d,i) bounds the number of slots delivered ($|\mathsf{d}|$) and acquired (i) by P as the cumulative result of exercising P's transfer capabilities.

Type $\mathit{Amb}\langle\iota,\varepsilon,\chi\rangle$ is the type of ambients with weight ranging in ι, and enclosing processes with ε effects and χ exchanges. As in companion type systems, values that can be exchanged include ambients and (paths of) capabilities, while the type *Shh* indicates no exchange. As for capability types, $\mathit{Cap}\langle\phi,\chi\rangle$ is the type of (paths of) capabilities

which, when exercised, unleash processes with χ exchanges, and compose the effect of the unleashed process with the thread effect ϕ. The functional domain of thread effects helps compute the composition of effects. In brief, thread effects accumulate the results from **get**'s and **put**'s, and compose these with the effects unleashed by occurrences of **open**.

We introduce the following combinators (functions in Φ) to define the thread effects of the **put**, **get** and **open** capabilities.

$$\begin{aligned} \mathsf{Put} &= \lambda(\mathsf{d},\mathsf{i}).(\mathsf{d}-1,\mathsf{max}(0,\mathsf{i}-1)) \\ \mathsf{Get} &= \lambda(\mathsf{d},\mathsf{i}).(\mathsf{min}(0,\mathsf{d}+1),\mathsf{i}+1) \\ \mathsf{Open}(\varepsilon) &= \lambda(\mathsf{d},\mathsf{i}).(\varepsilon+(\mathsf{d},\mathsf{i})) \end{aligned}$$

The intuition is as follows. A **put** that prefixes a process P with cumulative effect (d,i), contributes to a "shift" in that effect of one unit. The effect of a **get** capability is dual. To illustrate, the thread effect ε associated with $P = \textbf{put.}\,\textbf{put.}\,\textbf{get}\,a$ is computed as follows, where we use function composition in standard order (i.e. $f \circ g(x) = f(g(x))$):

$$\varepsilon \;=\; (\mathsf{Put} \circ \mathsf{Put} \circ \mathsf{Get})((0,0)) \;=\; (-2,0).$$

The intuition about an open capability is similar, but subtler, as the effect of opening an ambient is, essentially, the effect of the process unleashed by the open: in **open** $n.P$, the process unleashed by **open** n runs in parallel with P. Consequently, open has an additive import in the computation of the effect. To motivate, assume that $n : Amb\langle\iota,\varepsilon,\chi\rangle$. Opening n unleashes the enclosed process in parallel to the process P. To compute the resulting effect we may rely on the effect ε declared by n to bound the effect of the unleashed process: that effect is then added to the effect of the continuation P. Specifically, if P has effect ε', the composite effect of **open** $n.P$ is computed as $\mathsf{Open}(\varepsilon)(\varepsilon') = \varepsilon+\varepsilon'$.

2.2 The typing rules

The typing rules are collected in Figures 2 and 3, where we denote with id_Φ the identity element in the domain Φ.

The rules in Figure 2 derive judgements $\Gamma \vdash M : W$ for well-typed messages. The environment Γ is a set of assumptions either of the shape $a : Amb\langle\iota,\varepsilon,\chi\rangle$ with $a \in \mathcal{N}$ or of the shape $x : W$ with $x \in \mathcal{V}$, where all names and variables are distinct. The rules draw on the intuitions we gave earlier. Notice, in particular, that the capabilities **in**, **out** and the cocapability $\overline{\textbf{open}}$ have no effect, as reflected by the use id_Φ in their type. The same is true also of the the co-capability $\textbf{put}^{\downarrow}$. In fact, by means of the superscript k in $a^{\mathsf{k}}[\,P\,]$ we can record the actual weight of the ambient (cf. reduction rule (GETD)). This implies that the weight of an ambient in which $\textbf{put}^{\downarrow}$ is executed does not change: the ambient loses a slot, but the weight of one of its sub-ambients increases.

The rules in Figure 3 derive judgements $\Gamma \vdash P : Proc\langle\varepsilon,\chi\rangle$ for well-typed processes. An inspection of the typing rules shows that any well-typed process is also well-formed (in the sense of Definition 1). We let $0_{\mathcal{E}}$ denote the null effect $(0,0)$: thus, rules (**0**) and (▬) simply state that the inhert process and the slot form have no effects. Rule (*prefix*)

Fig. 2 Good Messages

$$\frac{}{\Gamma, M : W \vdash M : W}\ (axiom) \qquad \frac{\Gamma \vdash M : Cap\langle \phi, \chi\rangle \quad \Gamma \vdash M' : Cap\langle \phi', \chi\rangle}{\Gamma \vdash M.M' : Cap\langle \phi \circ \phi', \chi\rangle}\ (path)$$

$$\frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{get}\ M : Cap\langle \mathsf{Get}, \chi\rangle}\ (\mathbf{get}\ M) \qquad \frac{}{\Gamma \vdash \mathbf{put} : Cap\langle \mathsf{Put}, \chi\rangle}\ (\mathbf{put})$$

$$\frac{}{\Gamma \vdash \mathbf{get}^{\uparrow} : Cap\langle \mathsf{Get}, \chi\rangle}\ (\mathbf{get}^{\uparrow}) \qquad \frac{}{\Gamma \vdash \mathbf{put}^{\downarrow} : Cap\langle \mathsf{id}_{\Phi}, \chi\rangle}\ (\mathbf{put}^{\downarrow})$$

$$\frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{in}\ M : Cap\langle \mathsf{id}_{\Phi}, \chi\rangle}\ (\mathbf{in}\ M) \qquad \frac{\Gamma \vdash M : Amb\langle -,-,-\rangle}{\Gamma \vdash \mathbf{out}\ M : Cap\langle \mathsf{id}_{\Phi}, \chi\rangle}\ (\mathbf{out}\ M)$$

$$\frac{\Gamma \vdash M : Amb\langle -, \varepsilon, \chi\rangle}{\Gamma \vdash \mathbf{open}\ M : Cap\langle \mathsf{Open}(\varepsilon), \chi\rangle}\ (\mathbf{open}\ M) \qquad \frac{}{\Gamma \vdash \overline{\mathbf{open}} : Cap\langle \mathsf{id}_{\Phi}, \chi\rangle}\ (\overline{\mathbf{open}})$$

computes the effects of prefixes, by applying the thread effect of the capability to the effect of the process. Rule (*par*) adds up the effects of two parallel threads, while the constructs for input, output and restriction do not have any effect.

Rule (*amb*) governs the formation of ambient processes. The declared weight k of the ambient must reflect the weight of the enclosed process. Two further conditions ensure (i) that k modified by the effect (d,i) of the enclosed process lies within the interval $[\mathsf{n},\mathsf{N}]$ declared by the ambient type, and (ii) that effect ε declared by the ambient type is a sound approximation for the effects released by opening the ambient itself. The condition that ensures (i) is simply $[\mathsf{max}(\mathsf{k}+\mathsf{d},0),\mathsf{k}+\mathsf{i}] \leq [\mathsf{n},\mathsf{N}]$, where the use of $\mathsf{max}(\mathsf{k}+\mathsf{d},0)$ is justified by observing that the weight of an ambient may never grow negative as a result of the enclosed process exercising **put** capabilities. To motivate the condition that ensures (ii), first observe that opening an ambient which encloses a process with effect (d,i) may only release effects $\varepsilon \leq (\mathsf{d}-\mathsf{i},\mathsf{i}-\mathsf{d})$. The lower bound arises in a situation in which the ambient is opened right after the enclosed process has completed its i **get** 's and is thus left with $|\mathsf{d}-\mathsf{i}|$ **put**'s unleashed in the opening context. Dually, the upper bound arises when the ambient is opened right after the enclosed process has completed its $|\mathsf{d}|$ **put**'s, and is left with $\mathsf{i}-\mathsf{d}$ **get** 's. On the other hand, we also know that the maximum increasing effect released by opening ambients with weight ranging in $[\mathsf{n},\mathsf{N}]$ is $\mathsf{N}-\mathsf{n}$. Collectively, these two observations justify the condition $(\mathsf{d}-\mathsf{i},\mathsf{min}(\mathsf{N}-\mathsf{n},\mathsf{i}-\mathsf{d})) \leq \varepsilon$ in rule (*amb*).

In rule (*spawn*), the effect of $\mathsf{k} \triangleright P$ is the same as that of the reduct P. Finally, to prevent the effects of duplicated processes to add up beyond control, with unpredictable consequences, rule (*bang*) enforces duplicated processes to have null effects.

Fig. 3 Good Processes

$$\frac{}{\Gamma \vdash \_ : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\_) \qquad \frac{}{\Gamma \vdash \mathbf{0} : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (\mathbf{0})$$

$$\frac{\Gamma \vdash M : Cap\langle \phi, \chi\rangle \quad \Gamma \vdash P : Proc\langle \varepsilon, \chi\rangle}{\Gamma \vdash M.P : Proc\langle \phi(\varepsilon), \chi\rangle}\ (prefix)$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon, \chi\rangle \quad \Gamma \vdash Q : Proc\langle \varepsilon', \chi\rangle}{\Gamma \vdash P \mid Q : Proc\langle \varepsilon + \varepsilon', \chi\rangle}\ (par) \qquad \frac{\Gamma, x : W \vdash P : Proc\langle \varepsilon, W\rangle}{\Gamma \vdash (x : W)P : Proc\langle \varepsilon, W\rangle}\ (input)$$

$$\frac{\Gamma \vdash M : W \quad \Gamma \vdash P : Proc\langle \varepsilon, W\rangle}{\Gamma \vdash \langle M\rangle P : Proc\langle \varepsilon, W\rangle}\ (output) \qquad \frac{\Gamma, a : Amb\langle \iota, \varepsilon, \chi\rangle \vdash P : Proc\langle \varepsilon', \chi'\rangle}{\Gamma \vdash (\boldsymbol{\nu} a : Amb\langle \iota, \varepsilon, \chi\rangle)P : Proc\langle \varepsilon', \chi'\rangle}\ (new)$$

$$\frac{\begin{array}{ll}\Gamma \vdash M : Amb\langle [\mathsf{n}, \mathsf{N}], \varepsilon, \chi'\rangle & [\mathsf{max}(\mathsf{k}+\mathsf{d}, 0), \mathsf{k}+\mathsf{i}] \leq [\mathsf{n}, \mathsf{N}] \\ \Gamma \vdash P : Proc\langle (\mathsf{d}, \mathsf{i}), \chi'\rangle \qquad w(P) = \mathsf{k} & (\mathsf{d}-\mathsf{i}, \mathsf{min}(\mathsf{N}-\mathsf{n}, \mathsf{i}-\mathsf{d})) \leq \varepsilon\end{array}}{\Gamma \vdash M^{\mathsf{k}}[P] : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (amb)$$

$$\frac{\Gamma \vdash P : Proc\langle 0_{\mathcal{E}}, \chi\rangle \quad w(P) = \mathsf{k}}{\Gamma \vdash \mathsf{k} \triangleright P : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (spawn) \qquad \frac{\Gamma \vdash \pi.P : Proc\langle 0_{\mathcal{E}}, \chi\rangle \quad w(\pi.P) = 0}{\Gamma \vdash !\pi.P : Proc\langle 0_{\mathcal{E}}, \chi\rangle}\ (bang)$$

A first property of the type system is that all typable preterms are proper, hence well-formed, terms. In addition, the following result complements Proposition 1 and shows that capacity bounds on ambients are preserved during computations, while the processes' ability to shrink or expand reduces.

Theorem 1 (Subject Reduction). *Assume* $\Gamma \vdash P : Proc\langle \varepsilon, \chi\rangle$ *and* $P \searrow_* Q$. *Then* $\Gamma \vdash Q : Proc\langle \varepsilon', \chi\rangle$ *for some* $\varepsilon' \leq \varepsilon$.

Proof. The proof is by induction on the length of the reduction from P to Q. In the inductive case, the reasoning is by a cases analysis of the reduction in question. The interesting cases are those for rules (GETS) and (GETD). Below, we give the case of (GETS), the treatment of (GETD) being similar. Also, we give a proof disregarding the exchange components of our capacity types, as they are irrelevant for our argument.

Let $\mathcal{D}$ be a derivation for the redex of a (GETS) reduction:

$$\Gamma \vdash a^{\mathsf{k}+1}[\,\mathbf{put}\,.\,P \mid \_ \mid Q\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\,a\,.\,R \mid S\,] : Proc\langle 0_{\mathcal{E}}\rangle$$

An inspection of the typing rules – (*amb*) and (*par*) – shows that the type of the redex must indeed be of the form given. Inside $\mathcal{D}$ there must be judgements of the shape:

$$\Gamma \vdash a : Amb\langle \iota, \varepsilon\rangle, \ \ \Gamma \vdash P : Proc\langle (\mathsf{d}, \mathsf{i})\rangle, \ \ \Gamma \vdash Q : Proc\langle (\mathsf{d}', \mathsf{i}')\rangle$$

for some $\iota, \varepsilon, i, i', d, d'$. By rules (**put**) and (*prefix*), $\Gamma \vdash P : Proc\langle (d, i) \rangle$ implies $\Gamma \vdash$ **put**$.P : Proc\langle (d-1, \max(i-1,0)) \rangle$. Then the application of rule (*amb*) for deriving

$$\Gamma \vdash a^{k+1}[\,\textbf{put}.P \mid \_ \mid Q\,] : Proc\langle 0_{\mathcal{E}} \rangle$$

has the following conditions:

$$[\max(k+1+d-1+d', 0), k+1+\max(i-1,0)+i'] \leq \iota$$
$$(d-1+d'-\max(i-1,0)-i', \min(N-n, \max(i-1,0)+i'-d+1-d') \leq \varepsilon,$$

where $\iota = [n, N]$. The conditions required to apply rule (*amb*) for the derivation of the judgement $\Gamma \vdash a^k[\,P \mid Q\,] : Proc\langle 0_{\mathcal{E}} \rangle$ are

$$[\max(k+d+d', 0), k+i+i'] \leq \iota$$
$$(d+d'-i-i', \min(N-n, i+i'-d-d') \leq \varepsilon.$$

These conditions can be shown to derive from the previous ones by easy algebraic manipulations. With a similar reasoning one checks that the judgement $\Gamma \vdash b^{h+1}[\,\_ \mid R \mid S\,] : Proc\langle 0_{\mathcal{E}} \rangle$ is derivable. From this, we can conclude that the judgement $\Gamma \vdash a^k[\,P \mid Q\,] \mid b^{h+1}[\,\_ \mid R \mid S\,] : Proc\langle 0_{\mathcal{E}} \rangle$ is derivable, as desired.

It follows as a direct corollary that no ambient may be subject to under/over-flows during the computation of a process.

Theorem 2 (Absence of under/over-flow). *Assume* $\Gamma \vdash P : Proc\langle \varepsilon, \chi \rangle$ *and let* $P \searrow_* Q$. *If* $a : Amb\langle [n, N], -, - \rangle \in \Gamma$*, then, for any subterm of* Q *of the form* $a^k[\,R\,]$*, not in the scope of a binder for* a*, we have* $n \leq k \leq N$.

The proviso "not in the scope of a binder for a" is needed since the environment Γ could contain an assumption for the name a which has no connection whatsoever with the type of a bound occurrence of the same name a. In particular these two types could have very different weight ranges.

2.3 Typed Examples

All the examples in §1.2 that do not use both the open and the transfer capabilities are easily seen to typecheck. For instance, in the *cab trip* protocol, one verifies that all the processes typecheck, by taking $W_1 = Amb\langle [1,1], 0_{\mathcal{E}} \rangle$, $W_0 = Amb\langle [0,0], 0_{\mathcal{E}} \rangle$, by declaring the c with W_1 and assuming the following types for the free names $call : W_1$, $cab : W_1$, $trip : W_0$, $load : W_0$, $done : W_0$, $bye : W_0$.

Below we illustrate the typing system at work with a typed version of the memory module of Section 1.2. We start with the *malloc* ambient

$$\text{MALLOC} \triangleq m[\,\textbf{out}\,u.\textbf{in}\,mem.\overline{\textbf{open}}.\textbf{put}.\textbf{get}\,u.\textbf{open}\,m\,]$$

Since there are no exchanges, we give the typing annotation and derivation disregarding the exchange component from the types. Let P_{malloc} denote thread enclosed within the

ambient m. If we let $m : Amb\langle[0,0],(-1,0)\rangle \in \Gamma$, an inspection of the typing rules for capabilities and paths shows that the following typing is derivable for any ambient type assigned to *mem*:

$$\Gamma \vdash \mathbf{out}\ u\,.\,\mathbf{in}\ mem\,.\,\overline{\mathbf{open}}\,.\,\mathbf{put}\,.\,\mathbf{get}\ u\,.\,\mathbf{open}\ m : Cap\langle \mathsf{Put} \circ \mathsf{Get} \circ \lambda\varepsilon.((-1,0)+\varepsilon)\rangle$$

From this one derives $\Gamma \vdash P_{malloc} : Proc\langle(-1,0)\rangle$, which gives $\Gamma \vdash \text{MALLOC} : Proc\langle 0_{\mathcal{E}}\rangle$. As to the memory module itself, it is a routine check to verify that the process

$$\text{MEM\_MOD} \triangleq mem[\,\_^{256MB} \mid \mathbf{open}\ m \mid \ldots \mid \mathbf{open}\ m\,]$$

typechecks with $m : Amb\langle[0,0],(-1,0)\rangle$, $mem : Amb\langle[0,256MB],(-256MB,256MB)\rangle$.

2.4 Inferring Effects

We have shown that the type system gives enough flexibility to typecheck interesting protocols. On the other hand, the structure of the types is somehow complex: ideally, one would like to be able to use ambient types only to specify resource-related policies, without having to worry about the effects. Thus, for instance, one would specify the memory module protocol by simply declaring $m : Amb[0,0]$ and $mem : Amb[0,256MB]$ and leave it to the type system to infer the effect-related part of the types required to ensure that such bounds are indeed complied with.

We look at effect inference below. To ease the presentation, we give an inference system for the combinatorial subset of the calculus and disregard communications (and hence we do not consider the set of variables $\mathcal{V}$).
The ambient types give only the range ι for the weights of processes inside them, i.e. they have the shape: $Amb^{-}\langle\iota\rangle$. For each name a in $\mathcal{N}$ we introduce two integer variables $\mathtt{d}_a$, $\mathtt{i}_a$. So, in a sense, a pair $(\mathtt{d}_a,\mathtt{i}_a)$ can be looked at as an *effect variable* for the ambient a. Our inference system determines (by a set of inequalities constraints) the instances of effect variables which make a process typeable.

In the inference system an effect is no longer a pair of integers, but a pair of expressions (e,e') where $e,e' \in EXP$ and EXP is defined by:

$$EXP ::= \mathtt{d}_a \mid \mathtt{i}_a \mid \mathsf{n} \in \mathcal{Z} \mid EXP + EXP \mid EXP - EXP \mid \mathsf{min}(EXP,EXP) \mid \mathsf{max}(EXP,EXP)$$

with $a \in \mathcal{N}$. Of course this implies that thread effects are now functions on expressions, that is:

$$\textit{Effects} \qquad \varepsilon \in \mathcal{E} \triangleq \{(e,e') \mid e,e' \in EXP\}$$

$$\textit{Thread Effects} \qquad \phi \in \Phi \triangleq \mathcal{E} \to \mathcal{E}$$

The rules for messages in our inference system are those in Figure 2, but for rule (**open** M) which becomes:

$$\frac{\Gamma \vdash a : Amb^{-}\langle - \rangle}{\Gamma \vdash \mathbf{open}\ a : Cap\langle \mathsf{Open}(\mathtt{d}_a,\mathtt{i}_a)\rangle}\ (\widehat{\mathbf{open}\ a})$$

Fig. 4 Effect Inference

$$\frac{}{\Gamma \vdash \_ : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \emptyset}\ \widehat{(\_)} \qquad \frac{}{\Gamma \vdash \mathbf{0} : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \emptyset}\ \widehat{(\mathbf{0})}$$

$$\frac{\Gamma \vdash M : Cap\langle \phi \rangle \quad \Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta}{\Gamma \vdash M.P : Proc\langle \phi(\varepsilon) \rangle \Downarrow \Delta}\ \widehat{(prefix)}$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta \quad \Gamma \vdash Q : Proc\langle \varepsilon' \rangle \Downarrow \Delta'}{\Gamma \vdash P \mid Q : Proc\langle \varepsilon + \varepsilon' \rangle \Downarrow \Delta \cup \Delta'}\ \widehat{(par)}$$

$$\frac{\Gamma, a : Amb^{-}\langle \iota \rangle \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta}{\Gamma \vdash (\boldsymbol{\nu} a : Amb^{-}\langle \iota \rangle) P : Proc\langle \varepsilon \rangle \Downarrow \Delta}\ \widehat{(new)}$$

$$\frac{\Gamma \vdash a : Amb^{-}\langle [\mathsf{n}, \mathsf{N}] \rangle \quad \Gamma \vdash P : Proc\langle (e, e') \rangle \Downarrow \Delta \quad w(P) = \mathsf{k}}{\Gamma \vdash a^{\mathsf{k}}[P] : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta \cup \left\{ \begin{array}{l} \mathsf{n} \leq \max(\mathsf{k} + e, 0),\ \mathsf{k} + e' \leq \mathsf{N}, \\ \mathtt{d}_a \leq e - e',\ \min(\mathsf{N} - \mathsf{n}, e' - e) \leq \mathtt{i}_a \end{array} \right\}}\ \widehat{(amb)}$$

$$\frac{\Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta \quad w(P) = \mathsf{k}}{\Gamma \vdash \mathsf{k} \triangleright P : Proc\langle \varepsilon \rangle \Downarrow \Delta}\ \widehat{(spawn)} \qquad \frac{\Gamma \vdash \pi.P : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta \quad w(\pi.P) = 0}{\Gamma \vdash !\pi.P : Proc\langle 0_{\mathcal{E}} \rangle \Downarrow \Delta}\ \widehat{(bang)}$$

The inference rules are in Figure 4 and the judgements they derive have the shape $\Gamma \vdash P : Proc\langle \varepsilon \rangle \Downarrow \Delta$. This means that in our system, beside providing a process P with a process type $Proc\langle \varepsilon \rangle$, we produce for P also a set of inequality constraints Δ, whose satisfiability implies the typability of the process in our former system. The constraints are of the form $e \leq e'$ where the (integer) unknowns are of the form $\mathtt{d}_a$, $\mathtt{i}_a$.

The only rules that contribute to the formation of the constraint set are $\widehat{(par)}$ and $\widehat{(amb)}$. Rule $\widehat{(par)}$ simply joins the sets of constraints of two parallel processes. It is only by means of rule $\widehat{(amb)}$ that we really produce new constraints. These ones are simply a different way of stating, in our inference setting, the conditions of the (amb) rule of the type system. In our inference system, instead of checking a condition, we simply record it as a constraint whose satisfiability shall be checked at the end of the inference process.

In order to check the satisfiability of a set of constraints, we can transform them in a standard way, by adding two fresh variables with suitable conditions[1] for each $\mathsf{max}()$ or $\mathsf{min}()$ subexpression, obtaining an integer linear programming problem (see for exam-

[1] Let M be the maximum weight in the types of ambients occurring in the process P and h the total number of **put** and **get** capabilities occurring in P. Then $\mathsf{M} + 2\mathsf{h}$ is an upper bound for all integers which can occur in the set of constraints for P. We replace each $\mathsf{min}(e, e')$

ple [20]). Notice that we are only interested in the satisfiability (non emptyness of the solution space) of the resulting linear programming problem, that is we have no particular target function to optimize (but the auxiliary ones introduced when we get rid of the $\mathsf{max}()$ or $\mathsf{min}()$ subexpressions). In fact, given a process P, what matters is whether the *effect variables* can be instantiated enabling the typing of P. An optimization problem arises only in case we need to insert P in a particular context. For instance, if P is in a context where one of its ambients a has to be opened, we could try to minimise the effect of the type of a in order to have more chances the capacity of the opening ambient not be exceeded. On the contrary, if P is in a context where one of its ambients a must contain a process Q, we need to maximize the effect of the type of a in order to allow more freedom in the choice of the process Q.

It is worth noticing that any set of constraints can be extended with the following two, making the set of possible solutions always finite: $-\mathsf{h} \leq \mathtt{d}_a \leq 0$ and $0 \leq \mathtt{i}_a \leq \mathsf{h}$, where h is the total number of **put** and **get** capabilities occurring in P. The soundness of these additional constraints can be checked by a simple inspection of the rules.

The type inference illustrated above can easily be adapted to infer the ranges ι of ambients (instead of using environments): it suffices to consider for each name a other two integer variables $\mathtt{n}_a$, $\mathtt{N}_a$ and replace them respectively to n, N in rule (*amb*). So in this case we obtain a set of constraints where also $\mathtt{n}_a$ and $\mathtt{N}_a$ are unknown: if the constraints can be satisfied then there is also a solution such that $0 \leq \mathtt{n}_a \leq \mathsf{k} \leq \mathtt{N}_a$, where k is the weight of the process P. Furthermore, notice that the judgements $\vdash P : \mathit{Proc}\langle\varepsilon\rangle \Downarrow \Delta$ are built in a compositional way, and then $\mathit{Proc}\langle\varepsilon\rangle \Downarrow \Delta$ can be seen as the *principal typings* of the processes P in the sense of [28].

3 Controlling Resource Races – BoCa Revisited

In this section we extend the calculus with further, term-level, mechanisms to complement the typing system in providing for a richer and stronger control over the dynamics of space allocation and distribution.

As we observed in §1.3, a basic requirement is to provide for the ability by an ambient to reserve resources for internal use, i.e. for spawning new processes. In fact, reserving private space for spawning is possible with the current primitives, by encoding a notion of "named resource". This can be accomplished by defining:

$$\blacksquare_a^{\mathsf{k}} \triangleq a[\,\mathbf{put}^{\mathsf{k}} \mid \blacksquare^{\mathsf{k}}\,], \quad \text{and} \quad \mathsf{k} \triangleright (a,P) \triangleq (\boldsymbol{\nu} n)(n[\,(\mathbf{get}\ a)^{\mathsf{k}}.\mathsf{k} \triangleright \overline{\mathbf{open}}.P\,] \mid \mathbf{open}\ n)$$

Then, assuming $w(P) = \mathsf{k}$, one has $(\boldsymbol{\nu} a)(\blacksquare_a^{\mathsf{k}} \mid \mathsf{k} \triangleright (a,P)) \cong P$, as desired. It is also possible to encode a form of "resource renaming", by defining:

$$\{x/y\}.P \triangleq (\nu n)(n[\,\mathbf{get}\ y.\mathbf{put}.\overline{\mathbf{open}}\,] \mid x[\,\mathbf{get}\ n.\mathbf{put}\,] \mid \mathbf{open}\ n.P)$$

subexpression by a fresh variable t and we add the following conditions:

$$\begin{aligned} 0 &\leq e - t \ \leq (\mathsf{M}+2\mathsf{h})y \\ 0 &\leq e' - t \leq (\mathsf{M}+2\mathsf{h})(1-y) \end{aligned}$$

with $y \in \{0,1\}$ and fresh. We deal with $\mathsf{max}()$ subexpressions similarly.

Then, a y-resource can be turned in to an x-resource: $\{x/y\}.P \mid \__y \searrow_* P \mid \__x$.

Encoding a similar form of named, and reserved, resources for mobility is subtler. On the one hand, it is not difficult to encode a construct for reserving a x-slot for ambients named x. For example, ambients a and b may agree on the following protocol to reserve a private slot for the move of a into b. If we want to use the space in ambient b for moving a we can write the process:

$$(\boldsymbol{\nu} p,q)(p[\,\mathbf{in}\, b\,.\,\mathbf{get}\, q\,.\, 1 \triangleright \overline{\mathbf{open}}\,.\, a^1[\,\_\,]\,] \mid b[\,P \mid q[\,\_ \mid \mathbf{put}\,] \mid \mathbf{open}\, p\,])$$

On the other hand, defining a mechanism to release a named resource to the context from which it has been received is more complex, as it amounts to releasing a resource with the *same* name it was allocated to. This can be simulated loosely with the current primitives, by providing a mechanism whereby a migrating ambient releases an anonymous slot, which is then renamed by the context that is in control of it. The problem is that such a mechanism of releasing and renaming lacks the *atomicity* required to guard against unexpected races for the released resource. Indeed, we believe that such atomic mechanisms for named resources can not be defined in the current calculus.

3.1 Named resources and their semantics

To counter the lack of atomicity discussed above, we enrich the calculus with named resources as primitive notions, and tailor the constructs for mobility, transfer and spawning accordingly. Resource units come now always with a tag, as in $\__\eta$, where $\eta \in \mathcal{N} + \mathcal{V} + \{*\}$ is the unit name. To make the new calculus a conservative extension of the one presented in §1, we make provision for a special tag '*', to be associated with *anonymous* units: any process can be spawned on an anonymous slot, as well any ambient can be moved on it. In addition, we extend the structure of the transfer capabilities, as well as the construct for spawning and ambient as shown in the productions below, which replace the corresponding ones in §1.

Processes	$P ::= \__\eta \mid M[\,P\,]_\eta \mid \ldots$	as in Section 1
Capabilities	$C ::= \mathbf{get} M_\eta \mid \mathbf{get}^{\uparrow}_\eta \mid \ldots$	as in Section 1
Messages	$M ::= \ldots$	as in Section 1
Prefixes	$\pi ::= \mathsf{k} \triangleright_\eta \mid \ldots$	as in Section 1

Again, a (well-formed) *term* is a preterm such that in any subterm of the form $a^{\mathsf{k}}[P]$ or $\mathsf{k} \triangleright_\eta P$, P has weight k. The weight of a process can be computed by rules similar to those of Section 1. The anonymous slots $\__*$ will be often denoted simply as $\_$.

The dynamics of the refined calculus is again defined by means of structural congruence and reduction. Structural congruence is exactly as in Figure 1, the top-level reductions are defined in Figure 5.

Ambients acquire tags, as in $a[\,P\,]_\eta$, as they move. Initially, we may interpret each occurrence of an ambient $a[\,P\,]$ as denoting the tagged occurrence $a[\,P\,]_a$. To complete a move an ambient a must be granted an anonymous resource or an a-resource. The migrating ambient releases a resource under the name that it was assigned upon the

Fig. 5 Top-level reductions with named units

The reductions for ambient opening and exchanges are as in Figure 1, and the rules (ENTER) and (EXIT) have $\rho, \eta \in \{a, \star\}$ as side condition. The omitted subscripts on ambients are meant to remain unchanged by the reductions.

$$\begin{array}{lrcl}
\text{(ENTER)} & a^{\mathsf{k}}[\,\mathbf{in}\,b.P \mid Q\,]_{\rho} \mid b[\,\blacksquare^{\mathsf{k}}_{\eta} \mid R\,] & \searrow & \blacksquare^{\mathsf{k}}_{\rho} \mid b[\,a^{\mathsf{k}}[\,P \mid Q\,]_{\eta} \mid R\,] \\
\text{(EXIT)} & \blacksquare^{\mathsf{k}}_{\eta} \mid b[\,P \mid a^{\mathsf{k}}[\,\mathbf{out}\,b.Q \mid R\,]_{\rho}\,] & \searrow & a^{\mathsf{k}}[\,Q \mid R\,]_{\eta} \mid b[\,P \mid \blacksquare^{\mathsf{k}}_{\rho}\,] \\
\text{(GETS)} & b^{\mathsf{h}+1}[\,\mathbf{put}.P \mid \blacksquare_{\eta} \mid Q\,] \mid a^{\mathsf{k}}[\,\mathbf{get}\,b_{\eta}.R \mid S\,] & \searrow & b^{\mathsf{h}}[\,P \mid Q\,] \mid a^{\mathsf{k}+1}[\,R \mid \blacksquare_{\eta} \mid S\,] \\
\text{(GETU)} & \mathbf{put}^{\downarrow}.P \mid \blacksquare_{\eta} \mid a^{\mathsf{k}+1}[\,\mathbf{get}^{\uparrow}_{\eta}.Q \mid R\,] & \searrow & P \mid a^{\mathsf{k}}[\,\blacksquare_{\eta} \mid Q \mid R\,] \\
\text{(SPAWN)} & \mathsf{k} \triangleright_{\eta} P \mid \blacksquare^{\mathsf{k}}_{\eta} & \searrow & P
\end{array}$$

move (as recorded in the tag associated with the ambient construct): this solves the problem we discussed above. Also note that the dynamics of mobility guarantees the invariant that in $a[\,P\,]_{\eta}$ one has $\eta \in \{a, *\}$.

The reductions for the transfer capabilities are the natural extensions of the original reductions of §1. Here, in addition to naming the target ambient, the **get** capabilities also indicate the name of the unit they request. The choice of the primitives enables natural forms of scope extrusion for the names of resources. Consider the following system:

$$S \triangleq n[\,(\boldsymbol{\nu} a)(\mathbf{put}.P \mid \blacksquare_{a} \mid p[\,\mathbf{out}\,n.\mathbf{in}\,m.\overline{\mathbf{open}}.\mathbf{get}\,n_{a}\,])\,] \mid m[\,\mathbf{open}\,p.Q\,]$$

Here, the private resource enclosed within ambient n is communicated to ambient m, as $S \searrow_{*} (\boldsymbol{\nu} a)(n[\,P\,] \mid m[\,Q \mid \blacksquare_{a}\,])$.

Finally, the new semantics of spawning acts as expected, by associating the process to be spawned with a specific set of resources.

These definitions suggest a natural form of resource renaming (or rebinding), noted $\{\eta/\rho\}_{\mathsf{k}}$ with the following operational semantics.

$$\{\eta/\rho\}_{\mathsf{k}}.P \mid \blacksquare^{\mathsf{k}}_{\rho} \searrow P \mid \blacksquare^{\mathsf{k}}_{\eta}$$

Notice that this is a dangerous capability, since it allows processes to give particular names to anonymous slots, and for instance put in place possible malicious behaviours to make all public resources their own: $!\{y/*\}$. This suggests that in many situations one ought to restrict $\mathsf{k} \triangleright_{\eta}$ to $\eta \in \mathcal{N}$. The inverse behaviour, that is a "communist for y spaces," is also well-formed and it is often useful (even though not commendable by everyone). Notice however that it can be harmful too: $!\{*/y\}$. We have not defined the name rebinding capability as a primitive of our calculus since it can be encoded using the new form of spawning as follows, for a fresh.

$$\{\eta/\rho\}_{\mathsf{k}}.P \triangleq (\boldsymbol{\nu} a)(\mathsf{k} \triangleright_{\rho} (\blacksquare^{\mathsf{k}}_{\eta} \mid a^{0}[\,\overline{\mathbf{open}}\,]) \mid \mathbf{open}\,a.P)$$

Observe that the simpler encoding $\mathsf{k} \rhd_\rho (\blacksquare^{\mathsf{k}}_\eta \mid P)$ is allowed only for processes P of weight 0.

The possibility of encoding the renaming capability justifies also our choice of tags in the (SPAWN) reduction. As a matter of fact it would seem more reasonable to define this reduction as $\mathsf{k} \rhd_\eta P \mid \blacksquare^{\mathsf{k}}_\rho \searrow P$ with the side condition $\eta = \rho$ *or* $\rho = *$. This behaviour, however, can be approximated closely enough by putting a renaming process in parallel with the spawning one, namely $\{\eta/_*\} \mid \mathsf{k} \rhd_\eta P$.

It is easy to check that the type system of Section 2 can be used with no substantial modifications also for the calculus with named slots. The obvious changes required are those which guarantee syntax consistency, as e.g. that the x in $\blacksquare_x$ can only be instantiated by a name. For this calculus the same properties proved in Section 2 hold.

Theorem 3 (Subject Reduction and Under/Over-flow absence). *For the processes and reduction relation of this section, we have:*

(i) $\Gamma \vdash P : Proc\langle \varepsilon, \chi \rangle$ *and* $P \searrow_* Q$ *imply* $\Gamma \vdash Q : Proc\langle \varepsilon', \chi \rangle$ *with* $\varepsilon' \leq \varepsilon$.
(ii) *If* $\Gamma, a : Amb\langle [\mathsf{n}, \mathsf{N}], \chi \rangle \vdash P : Proc\langle \varepsilon, \chi \rangle$, $P \searrow_* C[a^{\mathsf{k}}[\, R \,]]$, *and the showed occurrence of* a *is not in the scope of a binder for* a, *then* $\mathsf{n} \leq \mathsf{k} \leq \mathsf{N}$.

3.2 Behavioural Semantics

The semantic theory of BoCa is based on *barbed congruence* [19], a standard equality relation based on reduction and a notion of observability. As usual in ambient calculi, our observation predicate, $P \downarrow_a$, indicates the possibility for process P to interact with the environment via an ambient named a. In Mobile Ambients (MA) this is defined as follows:

$$P \downarrow_a \;\triangleq\; P \equiv (\nu \tilde{m})(a[\, P' \,] \mid Q) \qquad a \notin \tilde{m} \tag{1}$$

Since no authorisation is required to cross a boundary, the presence of an ambient a at top level denotes a potential interaction between the process and the environment via a. In the presence of co-capabilities [16], however, the process $(\boldsymbol{\nu} \tilde{m})(a[\, P \,] \mid Q)$ only represents a potential interaction if P can exercise an appropriate co-capability. The same observation applies to BoCa, as many aspects of its dynamics rely on co-capabilities: notably, mobility, opening, and transfer across ambients. Correspondingly, we have several reasonable choices of observation, among which (for $a, \eta \notin \{\tilde{m}\}$):

$$P \downarrow^{opn}_a \;\triangleq\; P \equiv (\boldsymbol{\nu} \tilde{m})(a[\, \overline{\mathbf{open}}.P' \mid Q \,] \mid R) \tag{2}$$

$$P \downarrow^{slt}_a \;\triangleq\; P \equiv (\boldsymbol{\nu} \tilde{m})(a[\, \blacksquare_\eta \mid Q \,] \mid R) \tag{3}$$

$$P \downarrow^{put}_a \;\triangleq\; P \equiv (\boldsymbol{\nu} \tilde{m})(a[\, \mathbf{put}.P' \mid \blacksquare_\eta \mid Q \,] \mid R) \tag{4}$$

As it turns out, definitions (1)–(4) yield the same barbed congruence relation. Indeed, the presence of 0-weighted ambients makes it possible to rely on the same notion of observation as in MA, that is (1), without consequences on barbed congruences. We discuss this in further detail below.

Our notion of barbed congruence is standard in typed calculi, in that we require closure (only) by well-formed contexts. Say that a relation $\mathcal{R}$ is *reduction closed* if $P\mathcal{R}Q$ and $P \searrow P'$ imply the existence of some Q' such that $Q \searrow_* Q'$ and $P'\mathcal{R}Q'$; it is *barb preserving* if $P\mathcal{R}Q$ and $P\downarrow_a$ imply $Q\Downarrow_a$, i.e. $Q \searrow_*\downarrow_a$.

Definition 2 (Barbed Congruence). Barbed bisimulation, noted $\simeq$, is the largest symmetric relation on closed processes that is reduction closed and barb preserving. Two processes P and Q are *barbed congruent*, written $P \cong Q$, if for all contexts $\mathcal{C}[\cdot]$, preterm $\mathcal{C}[P]$ is a term iff so is $\mathcal{C}[Q]$, and then $\mathcal{C}[P] \simeq \mathcal{C}[Q]$.

Let then $\cong_i$ be the barbed congruence relation resulting from Definition 2 and from choosing the notion of observation as in (i) above (with $i \in [1..4]$).

Proposition 2 (Independence from barbs). $\cong_i = \cong_j$ *for all* $i, j \in [1..4]$.

Since the relations differ only on the choice of barb, Proposition 2 is proved by just showing that all barbs imply each other. This can be accomplished, as usual, by exhibiting a corresponding context. For instance, to see that $\cong_3$ implies $\cong_2$ use the context $\mathcal{C}[\cdot] = [\cdot] \mid \mathbf{open}\ a.b^1[\,\blacksquare\,]$, and note that for all P such that b is fresh in P one has $P \Downarrow_a^{opn}$ if and only if $\mathcal{C}[P] \Downarrow_b^{slt}$. Thanks to this Proposition we can denote barbed congruence for BoCa simply by $\cong$.

The import of the processes' weight in the relation of behavioural equivalence is captured directly by the well-formedness requirement in Definition 2. In particular, processes of different weight are distinguished, irrespective of the their "purely" behavioral properties. To see that, note that any two processes P and Q of weight, say, k and h with $\mathsf{h} \neq \mathsf{k}$, are immediately distinguished by the context $\mathcal{C}[\cdot] = a^{\mathsf{k}}[\,[\cdot]\,]$, as $\mathcal{C}[P]$ is well-formed while $\mathcal{C}[Q]$ is not. Ultimately, weight is a 'behavioural' property, in that it requires system's space allocation.

3.3 Algebraic laws

In Appendix A we give a labelled transition system for the calculus of this section, and sketch a proof of adequacy of the resulting notion of labelled bisimilarity with respect to the relation of barbed congruence given in Definition 2. A number of algebraic laws can be proved based on the resulting co-inductive characterization of barbed congruence. We highlight some of these below.

Garbage. There are many different characterization of wasted resources, and all these are congruent, provided they have the same weights.

$$(A_1) \qquad (\mathbf{v}a)a^{\mathsf{k}}[\,\blacksquare^{\mathsf{k}}_{\eta}\,] \;\cong\; (\mathbf{v}a)\mathbf{cap}\ a.\blacksquare^{\mathsf{k}}_{\eta} \qquad (\mathbf{cap} \in \{\mathbf{in}\,, \mathbf{out}\,, \ldots\})$$

Indeed, all these processes are inhert, hence behaviorally equivalent to the null process. Given that they have non-null weight, however, they are not congruent to the $\mathbf{0}$ process, but rather to what may be construed as a new process construct, noted $\mathbf{0}^{\mathsf{k}}$, that provides an explicit representation of a notion of garbage in the calculus.

Spawning. The spawning of a process cannot be observed as long as the space required is protected from other, unintended uses. This is true of the form of private spawning based on the primitive naming mechanism for slots, as well as of the encoding given earlier in this section. Specifically, we have:

$$(A_2) \qquad (\boldsymbol{\nu} a)(a^{\mathsf{k}}[\,\_^{\mathsf{k}} \mid \mathsf{k} \triangleright \overline{\mathbf{open}}.\,P\,] \mid \mathbf{open}\; a) \;\cong\; P$$

$$(A_3) \qquad (\boldsymbol{\nu} a)(\__a \mid \mathsf{k} \triangleright_a P) \;\cong\; P$$

Transfers. Similar laws relate the exchange of slots between ambients. For example:

$$(A_4) \qquad (\boldsymbol{\nu} a)(a^{\mathsf{k}}[\,\_^{\mathsf{k}} \mid \mathbf{put}^{\mathsf{k}}\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\; a^{\mathsf{k}} \mid P\,]) \;\cong\; b^{\mathsf{k}+\mathsf{h}}[\,\_^{\mathsf{k}} \mid P\,]$$

Ambient opening and movement. Given the presence of a co-capability for **open** , ambient opening satisfies the same laws as the calculus of Safe Ambients of [16]. As for movement, corresponding laws hold only for ambients with non-null weight: such ambients may be characterized by means of a typing system in which one requires that all ambient types have a strictly positive lower bound.

For the calculus of §1, the mobility laws are weaker than the corresponding laws in [16], as our slots act uniformly as co-capabilities for **in** and **out** moves. For the calculus of this section, instead, if we assume the typing restriction discussed above, we have:

$$(A_5) \qquad (\boldsymbol{\nu} b)(b^{\mathsf{k}}[\,\mathbf{in}\; a.\,P\,] \mid a^{\mathsf{k}}[\,\__b^{\mathsf{k}}\,]) \;\cong\; (\boldsymbol{\nu} b)(\__b^{\mathsf{k}} \mid a^{\mathsf{k}}[\,b^{\mathsf{k}}[\,P\,]\,])$$

As a further remark, we note that there are congruences, like (A_4), between typeable and untypeable terms for a fixed environment. In fact assuming $w(P) = \mathsf{h}$, $b \notin \Gamma$, and $\Gamma \vdash P : Proc\langle 0_{\mathcal{E}}, \chi\rangle$ the term $b^{\mathsf{k}+\mathsf{h}}[\,\_^{\mathsf{k}} \mid P\,]$ can be typed in the environment $\Gamma, b : Amb\langle[\mathsf{k}+\mathsf{h}, \mathsf{k}+\mathsf{h}], 0_{\mathcal{E}}, \chi\rangle$, while $(\boldsymbol{\nu} a)(a^{\mathsf{k}}[\,\_^{\mathsf{k}} \mid \mathbf{put}^{\mathsf{k}}\,] \mid b^{\mathsf{h}}[\,\mathbf{get}\; a^{\mathsf{k}} \mid P\,])$ cannot.

3.4 More Examples

The cab trip revisited. Named slots allow us to avoid unwanted behaviours when encoding the cab trip example. The main steps of the protocol may now be described as follows:

- ▷ The *cab* initially contains one slot named *call* to signal that it is vacant.
- ▷ Once the ambient *call* reaches a *cab*, it is opened there to drive *cab* to the client's site. In addition, opening *call* inside *cab* leaves in *cab* a slot with the (private) name *client* of the client. Consequently, only the client whose *call* reached *cab* may eventually enter *cab*.
- ▷ When *client* enters *cab*, it leaves a slot named *client* which is then rebound to an anonymous slot in order for the enclosing site to be able to accept new incoming cabs on that slot.
- ▷ Upon completing the trip to destination, *cab* sets out to complete the protocol: it opens the synchronization ambient *arrived*, and rebinds the slot named *client* to the (again private) name of the acknowledgement ambient *bye*. Opening *arrived* also unleashes the inner occurrence of *done* which in turn enters *client* to signal that it is time for *client* to leave *cab*.

▷ At this stage *client* exits *cab* and the protocol completes with ambient *bye* entering *cab* and being opened there to drive *cab* out of the destination site with a slot named *call*.

Let $W_1 = Amb[1,1]$ $W_0 = Amb[0,0]$. As in Subsection 2.3 the typing environment contains $call : W_1$, $cab : W_1$, $trip : W_0$, $done : W_0$: instead, in the new encoding the ambient *bye* has type W_1 and is private to the client. Furthermore, we add the ambient *arrived* with type W_0, and we do not need the ambient *load*.

$$
\begin{array}{l}
CALL(from,client,bye) \triangleq \\
\quad call^1[\,\mathbf{out}\ client\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{open}}\,. \\
\qquad\qquad\qquad\qquad \mathbf{in}\ from\,.(\blacksquare_{client} \mid \mathbf{open}\ arrived\,.\{^{bye}/_{client}\}\,.\,\mathbf{open}\ bye)] \\
TRIP(from,to,client) \triangleq \\
\quad trip^0[\,\mathbf{out}\ client\,.\,\overline{\mathbf{open}}\,.\,\mathbf{out}\ from\,.\,\mathbf{in}\ to\,.\,arrived^0[\,\overline{\mathbf{open}}\,.\,done^0[\,\mathbf{in}\ client\,.\,\overline{\mathbf{open}}\,]\,]\,] \\
CLIENT(from,to) \triangleq \\
\quad (\mathbf{\nu} client : W_1, bye : W_1)\,(client^1[CALL(from,client,bye) \mid \mathbf{in}\ cab\,.\,TRIP(from,to,client) \\
\qquad \mid \mathbf{open}\ done\,.\,\mathbf{out}\ cab\,.\,1 \triangleright_{call} bye^1[\,\mathbf{out}\ client\,.\,\mathbf{in}\ cab\,.\,\overline{\mathbf{open}}\,.\,\mathbf{out}\ to\,.\blacksquare_{call}\,]] \mid \{^*/_{client}\}) \\
CAB \quad \triangleq cab^1[\,\blacksquare_{call} \mid !(\mathbf{open}\ call\,.\,\mathbf{open}\ trip)\,] \mid \{^*/_{cab}\} \\
SITE(i) \triangleq site_i[\,CLIENT(site_i,site_j) \mid CLIENT(site_i,site_l) \mid \cdots \mid \blacksquare \mid \blacksquare \mid \cdots\,] \\
CITY \quad \triangleq city[\,CAB \mid CAB \mid \cdots \mid SITE(1) \mid \cdots \mid SITE(n) \mid \blacksquare \mid \blacksquare \mid \cdots\,]
\end{array}
$$

Using the labelled transition system given in the Appendix we can prove properties of the CAB system, including the expected ones that clients will not be able to board a taxi different from that called and that the ambient *bye* always enter the cab the client just left.

A Travel Agency. We conclude the presentation with an example that shows the expressiveness of the naming mechanisms for resources in the refined calculus. We wish to model clients buying tickets from a travel agency, paying them one slot (the $\blacksquare_{fortkt}$ inside the client), and then use them to travel by plane. At most two clients may enter the travel agency, and they are served one by one. The three components of the systems are defined below.

▷ THE AGENCY: $ag^5[\,\blacksquare^2_{cl} \mid \blacksquare_{req} \mid \blacksquare_{tkt}$
$desk^1[\,\blacksquare_{req} \mid !(\mathbf{open}\ req\,.\,1 \triangleright_{fortkt} .\,tkt^1[\,\mathbf{out}\ desk\,.\,\mathbf{in}\ cl\,.\,CONT\,] \mid \{^{req}/_{tkt}\})]\,]$
where $CONT = (\overline{\mathbf{open}}\,.\,\mathbf{out}\ ag\,.\,\mathbf{in}\ plane\,.\,rdy^0[\,\mathbf{out}\ cl\,] \mid \blacksquare_{getoff} \mid \mathbf{open}\ getoff)$

▷ THE CLIENT: $cl^1[\,\mathbf{in}\ ag\,.\,req^1[\,\mathbf{out}\ cl\,.\,\mathbf{in}\ desk\,.\,\overline{\mathbf{open}}\,.\blacksquare_{fortkt}\,] \mid \{^{tkt}/_{req}\} \mid \mathbf{open}\ tkt\,]$

▷ THE AIRCRAFT: $plane^4[\,\blacksquare^2_{cl} \mid \mathbf{open}\ rdy\,.\,\mathbf{open}\ rdy\,.\,TRIP\,.(GETOFF \mid GETOFF)\,]$
where $GETOFF = getoff^1[\,\mathbf{in}\ \mathrm{cl}\,.\,\overline{\mathbf{open}}\,.\,\mathbf{out}\ plane \mid \blacksquare\,]$ and *TRIP* is the unspecified path modelling the route of the aircraft.

We assume that there exists only one sort of ticket, but it is easy to extend the example with as many kinds of ticket as possible plane routes. What makes the example

interesting is the possibility of letting two clients into the agency, but serving them non-deterministically in sequence. Notice that the use of the named slots is essential for a correct implementation of the protocol. When the request goes to the desk, a slot named *tkt* is left in the client. This slot allows the ticket to enter the client. In this way we guarantee that no ticket can enter a client before its request has reached the desk.

We assume the aircraft to leave only when full. This constraint is implemented by means of the *rdy* ambient. The ambient *getoff* enables the passengers to get off once at destination; assigning weight 1 to the *getoff* ambients prevents them to get both into the same client.

4 Conclusion and Future Work

We have presented an ambient-like calculus centred around an explicit primitive representing a resource unit: the space "slot" $\blacksquare$. The calculus, dubbed BoCa, features capabilities for resource control, namely pairs **get/put** to transfer spaces between sibling ambients and from parent to child, as well as the capabilities **in** a and **out** a for ambient migration, which represent an abstract mechanism of resource negotiation between travelling agent and its source and destination environments. A fundamental ingredient of the calculus is $\triangleright(\_)$, a primitive which consumes space to activate processes. The combination of such elements makes of BoCa a suitable formalism, if initial, to study the role of resource consumption, and the corresponding safety guarantees, in the dynamics of mobile systems. We have experimented with the all important notion of private resource, which has guided our formulation of a refined version of the calculus featuring named resources.

The presence of the space construct $\blacksquare$ induces a notion of weight on processes, and by exercising their transfer capabilities, processes may exchange resources with their surrounding context, so making it possible to have under- and over-filled ambients. We have introduced a type system which prevents such unwanted effects and guarantees that the contents of each ambient remain within its declared capacity.

As we mentioned in the Introduction, our approach is related to the work on *Controlled Mobile Ambients* (CMA) [27] and on *Finite Control Mobile Ambients* [6]. There are, however, important difference with respect to both approaches.

In CMA the notions of process weight and capacity are entirely characterized at the typing level, and so are the mechanisms for resource control (additional control on ambient behavior is achieved by means of a three-way synchronization for mobility, but that is essentially orthogonal to the mechanisms targeted at resource control). In BoCa, instead, we characterize the notions of space and resources directly in the calculus, by means of an explicit process constructor, and associated capabilities. In particular, the primitives for transferring space, and more generally for the explicit manipulation of space and resources by means of spawning and replication appear to be original to BoCa, and suitable for the development of formal analyses of the fundamental mechanism of the usage and and consumption of resources which do not seem to be possible for CMA.

As to [6], their main goal is to isolate an expressive fragment of Mobile Ambients for which the model checking problem against the ambient logic can be made decidable.

Decidability requires guarantees of finiteness which in turn raise boundedness concerns that are related to those we have investigated here. However, a more thorough comparison between the two approaches deserves to be made and we leave it to our future work.

Plans for future include further work in several directions. A finer typing discipline could be put in place to regulate the behavior of processes in the presence of primitive notions of named slots. Also, the calculus certainly needs behavioral theories and proof techniques adequate for reasoning about resource usage and consumption. Such theories and techniques could be assisted by enhanced typing systems providing static guarantees of a controlled, and bounded, use of resources, along the lines of the work by Hofmann and Jost in [14].

A further direction for future development is to consider a version of weighed ambients whose "external" weight is independent of their "internal" weight, that is the weight of their contents. This approach sees an ambient as a packaging abstraction whose weight may have a different interpretation from that of contents'. For instance, modelling a wallet the weight of its contents could represent the value of the money inside, whereas its external weight could measure the physical space it occupies. A directory's internal weight could be the cumulative size of its files, while the external weight their number.

Last, but not least, we would like to identify logics for BoCa to formulate (quantitative) resource properties and analyses; and to model general resource bounds negotiation and enforcement in the Global Computing scenario.

Acknowledgements We gratefully acknowledge the anonymous referees of ASIAN 2003 for careful reading and useful suggestions.

References

1. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A Calculus of Bounded Capacities. In *ASIAN'03, Eighth Asian Computing Science Conference*, number xxxx in Lecture Notes in Computer Science, pages xx–xx, 2003. To Appear.
2. M. Bugliesi and G. Castagna. Secure safe ambients. In *POPL'01*, pages 222–235, New York, 2001. ACM Press.
3. M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *FSTTCS'02, Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, number 2556 in Lecture Notes in Computer Science, pages 71–84. Springer–Verlag, 2002.
4. M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication and Mobility Control in Boxed Ambients. Submitted for publication. Extended and revised version of [3], April 2003.
5. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. Special Issue on Coordination, D. Le Métayer Editor.
6. W. Charatonik, A. D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In D. Le Métayer, editor, *ESOP'02*, volume 2305 of *LNCS*, pages 295–313, Berlin, 2002. Springer-Verlag.
7. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*, pages 184–198, New York, 2000. ACM Press.

8. J. C. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 272–287, Berlin, 2002. Springer-Verlag.
9. A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 12:1–38, 2002.
10. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed system (extended abstract). In A. D. Gordon, editor, *FOSSACS'03*, volume 2620 of *LNCS*, pages 282–299, Berlin, 2003. Springer-Verlag.
11. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.
12. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
13. M. Hofmann. The strength of non size-increasing computation. In *POPL'02*, pages 260–269, New York, 2002. ACM Press.
14. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, pages 185–197, New York, 2003. ACM Press.
15. A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL'02*, pages 331–342, New York, 2002. ACM Press.
16. F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL'00*, pages 352–364. ACM Press, New York, 2000.
17. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*, pages 352–364, New York, 2000. ACM Press.
18. M. Merro and M. Hennessy. Bisimulation Congruences in Safe Ambients. In *POPL'02*, pages 71–80, New York, 2002. ACM Press.
19. R. Milner and D. Sangiorgi. Barbed bisimulation. In W.Kuich, editor, *ICALP'92*, volume 623 of *LNCS*, pages 685–695, Berlin, 1992. Springer-Verlag.
20. G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization*. Wiley, 1988.
21. R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.
22. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
23. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.
24. D. Sangiorgi. Bisimulation for Higher-Order Process Calculi. *Information and Computation*, 131(2):141–178, 1996.
25. D. Sangiorgi. Extensionality and intensionality of the ambient logic. In *POPL'01*, pages 4–13, New York, 2001. ACM Press.
26. D. Sangiorgi and R. Milner. The problem of "Weak Bisimulation up to". In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.
27. D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 288–303, Berlin, 2002. Springer-Verlag.
28. J. Wells. The essence of principal typings. In P. Widmayer, F. Triguero, R. Morales, M. Hennessy, S. Eidenbez, and R. Conejo, editors, *ICALP'02*, volume 2380 of *LNCS*, pages 913–925, Berlin, 2002. Springer-Verlag.
29. N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order mobile processes (extended abstract). In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99*, volume 1664 of *LNCS*, pages 557–573, Berlin, 1999. Springer-Verlag.

A A Labelled Transition system for BoCa

Table 1 Labels, concretions and outcomes

Prefixes	$\gamma ::= \mathbf{in}\ a \mid \mathbf{out}\ a \mid \mathbf{open}\ a \mid \overline{\mathbf{open}} \mid \mathbf{get}\, a_\eta \mid \mathbf{get}^{\uparrow}{}_\eta \mid \mathbf{put} \mid \mathbf{put}^{\downarrow} \mid \mathsf{k} \triangleright \eta$
Labels	$\alpha ::= \gamma \mid \mathsf{k} \mid \blacksquare^{\mathsf{k}}_{\eta} \mid \mathbf{put}\,\eta \mid \mathbf{put}^{\downarrow}\,\eta \mid \langle M \rangle \mid \langle - \rangle$ $\mid\ *[\,\mathbf{get}\, a_\eta\,] \mid *[\,\mathbf{get}^{\uparrow}{}_\eta\,] \mid \eta^{\mathsf{k}}[\,\mathbf{out}\ a\,] \mid \eta^{\mathsf{k}}[\,\mathbf{in}\ a\,] \mid \eta^{\mathsf{k}}[\,\mathbf{exit}\ a\,]$ $\mid\ a[\,\overline{\mathbf{open}}\,] \mid a[\,\mathbf{put}\,\eta\,] \mid a[\,\blacksquare^{\mathsf{k}}_{\eta}\,]$
Concretions	$K ::= (\boldsymbol{\nu}\tilde{m})\langle P \rangle Q \mid (\boldsymbol{\nu}\tilde{m})\langle M \rangle P$
Outcomes	$O ::= P \mid K$

We give a labelled transition system for the calculus of §3. A corresponding LTS can readily be obtained for the calculus of §1 by simply erasing all the occurrences of η and ρ from the labels and the corresponding transitions. Based on the labelled transitions, we then introduce a labelled bisimilarity which, because of its co-inductive nature, will provide powerful proof techniques for establishing equivalences [23, 26, 24]. As usual for ambient calculi [9, 17, 18, 4], the labelled transitions have the form $P \xrightarrow{\alpha} O$, where P is a well-formed term , and

- ▷ the *label* α encodes the minimal contribution by the environment needed for the process to complete the transition;
- ▷ the *outcome* O can be either a *concretion*, i.e. a partial derivative which needs a contribution from the environment to be completed, or a process.

Table 1 defines labels and concretions. In $(\boldsymbol{\nu}\tilde{p})\langle P \rangle Q$ the process P represents the moving ambient and the process Q represents the remaining system not affected by the movement. In $(\boldsymbol{\nu}\tilde{p})\langle M \rangle P$ the message M represents the information transmitted and the process P represents the remaining system not affected by the output. In both cases $\tilde{p}$ is the set of shared private names.

Tables 2 and 3 give the labelled transition system. In writing the rules we will use the following standard conventions:

- ▷ if O is the concretion $(\boldsymbol{\nu}\tilde{p})\langle P \rangle Q$, then:
  - $(\boldsymbol{\nu} r)O = (\boldsymbol{\nu}\tilde{p})\langle P \rangle (\boldsymbol{\nu} r)Q$, if $r \notin \mathrm{fn}(P)$, and $(\boldsymbol{\nu} r)O = (\boldsymbol{\nu} r, \tilde{p})\langle P \rangle Q$ otherwise.
  - $O \mid R = (\boldsymbol{\nu}\tilde{p})\langle P \rangle (Q \mid R)$, where $\tilde{p}$ are chosen so that $\mathrm{fn}(R) \cap \{\tilde{p}\} = \emptyset$.
- ▷ if O is the concretion $(\boldsymbol{\nu}\tilde{p})\langle M \rangle P$, then:
  - $(\boldsymbol{\nu} r)O$ is $(\boldsymbol{\nu}\tilde{p})\langle M \rangle ((\boldsymbol{\nu} r)P)$, if $r \notin \mathrm{fn}(M)$, and $(\boldsymbol{\nu} r, \tilde{p})\langle M \rangle Q$ otherwise.
  - $O \mid R = (\boldsymbol{\nu}\tilde{p})\langle M \rangle (P \mid R)$, where $\tilde{p}$ are chosen so that $\mathrm{fn}(R) \cap \{\tilde{p}\} = \emptyset$.

Table 2 Commitments: Visible transitions

(CAP)

$$\frac{M \in \{\mathbf{in}\ a, \mathbf{out}\ a, \mathbf{open}\ a, \overline{\mathbf{open}}, \mathbf{put}, \mathbf{put}^{\downarrow}, \mathsf{k} \triangleright \eta\}}{M.P \xrightarrow{M} P}$$

(PATH)

$$\frac{M_1.(M_2.P) \xrightarrow{\alpha} P'}{(M_1.M_2).P \xrightarrow{\alpha} P'}$$

(WEIGHT)

$$\frac{w(P) = \mathsf{k}}{P \xrightarrow{\mathsf{k}} P}$$

(SLOT-0)

$$\frac{}{P \xrightarrow{\blacksquare_{\epsilon}^{0}} P}$$

(SLOT-1)

$$\frac{}{\blacksquare \xrightarrow{\blacksquare_{\eta}^{1}} \mathbf{0}}$$

(SLOT-PAR)

$$\frac{P \xrightarrow{\blacksquare_{\eta}^{\mathsf{k}}} P_1 \quad Q \xrightarrow{\blacksquare_{\eta}^{\mathsf{l}}} Q_1 \quad (\mathsf{k} \geq 1)}{P \mid Q \xrightarrow{\blacksquare_{\eta}^{\mathsf{k}+\mathsf{l}}} P_1 \mid Q_1}$$

(GET)

$$\frac{M \in \{\mathbf{get}\, a_{\eta}, \mathbf{get}^{\uparrow}{}_{\eta}\}}{M.P \xrightarrow{M} \blacksquare_{\eta} \mid P}$$

(PUT)

$$\frac{M \in \{\mathbf{put}, \mathbf{put}^{\downarrow}\}\ P \xrightarrow{M} P'\ Q \xrightarrow{\blacksquare_{\eta}^{\mathsf{l}}} Q'}{P \mid Q \xrightarrow{M\,\eta} P' \mid Q'}$$

(SLOT-INC)

$$\frac{M \in \{\mathbf{get}\, a_{\eta}, \mathbf{get}^{\uparrow}{}_{\eta}\} \quad P \xrightarrow{M} P'}{a^{\mathsf{k}}[\,P\,] \xrightarrow{*[M]} a^{\mathsf{k}+1}[\,P'\,]}$$

(SLOT-DEC)

$$\frac{P \xrightarrow{\mathbf{put}\,\eta} P'}{a^{\mathsf{k}+1}[\,P\,] \xrightarrow{a[\,\mathbf{put}\,\eta\,]} a^{\mathsf{k}}[\,P'\,]}$$

(INPUT)

$$\frac{}{(x:\chi).P \xrightarrow{\overline{\langle M \rangle}} P\{x := M\}}$$

(OUTPUT)

$$\frac{}{\langle M \rangle.P \xrightarrow{\langle - \rangle} (\boldsymbol{\nu})\langle M \rangle P}$$

(IN-OUT)

$$\frac{P \xrightarrow{M} P' \quad M \in \{\mathbf{in}\ a, \mathbf{out}\ a\},\ \eta \in \{b, *\}}{b^{\mathsf{k}}[\,P\,]_{\rho} \xrightarrow{\eta^{\mathsf{k}}[M]} (\boldsymbol{\nu})\langle b^{\mathsf{k}}[\,P'\,]_{\eta}\rangle \blacksquare_{\rho}^{\mathsf{k}}}$$

(CO-IN)

$$\frac{P \xrightarrow{\blacksquare_{\eta}^{\mathsf{k}}} P'}{a^{\mathsf{h}}[\,P\,] \xrightarrow{a[\,\blacksquare_{\eta}^{\mathsf{k}}\,]} (\boldsymbol{\nu})\langle P' \rangle \mathbf{0}}$$

(EXIT)

$$\frac{P \xrightarrow{\eta^{\mathsf{k}}[\,\mathbf{out}\ a\,]} (\boldsymbol{\nu}\tilde{p})\langle P_1 \rangle P_2}{a^{\mathsf{h}}[P] \xrightarrow{\eta^{\mathsf{k}}[\,\mathbf{exit}\ a\,]} (\boldsymbol{\nu}\tilde{p})(P_1 \mid a^{\mathsf{h}}[P_2])}$$

(CO-OPEN)

$$\frac{P \xrightarrow{\overline{\mathbf{open}}} P'}{a[\,P\,] \xrightarrow{a[\,\overline{\mathbf{open}}\,]} P'}$$

Table 3 Commitments: τ transitions and structural transitions

(τ-ENTER)

$$\frac{P \xrightarrow{\eta^{k}[\,\mathbf{in}\ a\,]} (\boldsymbol{\nu}\tilde{p})\langle P_1\rangle P_2 \quad Q \xrightarrow{a[\blacksquare^{k}_{\eta}]} (\boldsymbol{\nu}\tilde{q})\langle Q_1\rangle Q_2 \qquad \begin{array}{r}\mathsf{h} = w(P_1 \mid Q_1)\\ (\mathrm{fn}(P_1)\cup\mathrm{fn}(P_2))\cap\{\tilde{q}\} = \emptyset\\ (\mathrm{fn}(Q_1)\cup\mathrm{fn}(Q_2))\cap\{\tilde{p}\} = \emptyset\end{array}}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{p},\tilde{q})(a^{\mathsf{h}}[Q_1 \mid P_1] \mid P_2 \mid Q_2)}$$

(τ-EXIT)

$$\frac{P \xrightarrow{\eta^{k}[\,\mathbf{exit}\ a\,]} P_1 \quad Q \xrightarrow{\blacksquare^{k}_{\eta}} Q_1}{P \mid Q \xrightarrow{\tau} P_1 \mid Q_1}$$

(τ-OPEN)

$$\frac{P \xrightarrow{\mathbf{open}\ a} P_1 \quad Q \xrightarrow{a[\overline{\mathbf{open}}]} Q_1}{P \mid Q \xrightarrow{\tau} P_1 \mid Q_1}$$

(τ-TRANS)

$$\frac{P \xrightarrow{*[\,\mathbf{get}\,a_{\eta}\,]} P_1 \quad Q \xrightarrow{a[\mathbf{put}\ \eta]} Q_1}{P \mid Q \xrightarrow{\tau} P_1 \mid Q_1}$$

(τ-TRAND)

$$\frac{P \xrightarrow{*[\,\mathbf{get}^{\uparrow}\eta\,]} P_1 \quad Q \xrightarrow{\mathbf{put}^{\downarrow}\ \eta} Q_1}{P \mid Q \xrightarrow{\tau} P_1 \mid Q_1}$$

(τ-EXCHANGE)

$$\frac{P \xrightarrow{(M)} P_1 \quad Q \xrightarrow{\langle - \rangle} (\boldsymbol{\nu}\tilde{q})\langle M\rangle Q_1 \quad \mathrm{fn}(P)\cap\tilde{q} = \emptyset}{P \mid Q \xrightarrow{\tau} (\nu\tilde{q})(P_1 \mid Q_1)}$$

(τ-SPAWN)

$$\frac{P \xrightarrow{k\triangleright\eta} P_1 \quad Q \xrightarrow{\blacksquare^{k}_{\eta}} Q_1}{P \mid Q \xrightarrow{\tau} P_1 \mid Q_1}$$

(PAR)

$$\frac{P \xrightarrow{\alpha} O}{P \mid Q \xrightarrow{\alpha} O \mid Q}$$

(RES)

$$\frac{P \xrightarrow{\alpha} O \quad n \notin \mathrm{fn}(\alpha)}{(\boldsymbol{\nu}n)P \xrightarrow{\alpha} (\boldsymbol{\nu}n)O}$$

(τ-AMB)

$$\frac{P \xrightarrow{\tau} P'}{n[P] \xrightarrow{\tau} n[P']}$$

(REPL)

$$\frac{\pi.P \xrightarrow{\alpha} O}{!\pi.P \xrightarrow{\alpha} !\pi.P \mid O}$$

The transitions are similar to the transitions defined for [17, 4] when we interpret an occurrence of a slot as a co-capability for movement and spawning. The newest rule is (WEIGHT) which allows to distinguish processes only on the basis of their weights. Peculiar to our calculus are the rules dealing with slots: in particular rule (SLOT-0) says that each process becomes itself using no slot, instead rule (SLOT-1) says that one slot can be consumed becoming the null process. Rule (SLOT-0) is useful for allowing the movement and the spawning of processes with weight 0. Peculiar are also rules (SLOT-INC) and (SLOT-DEC) in which the weights of the ambient change.

The labelled transition semantics agrees with the reduction semantics: this can be easily checked from the definitions.

Theorem 4. *If $P \xrightarrow{\tau} Q$ then $P \searrow Q$. Conversely, if $P \searrow Q$ then $P \xrightarrow{\tau} Q'$ for some $Q' \equiv Q$.*

Table 4 Commitments: Higher-Order transitions

(HO OUTPUT)

$$\frac{P \xrightarrow{\langle - \rangle} (\boldsymbol{\nu}\tilde{p})\langle M \rangle P' \quad \mathrm{fv}(Q) \subseteq \{x\},\ \tilde{p} \cap \mathrm{fn}(Q) = \emptyset}{P \xrightarrow{\langle - \rangle Q} (\boldsymbol{\nu}\tilde{p})(P' \mid Q\{x := M\})}$$

(HO IN/CO-IN)

$$\frac{P \xrightarrow{M} (\boldsymbol{\nu}\tilde{p})\langle P_1 \rangle P_2 \quad M \in \{\eta^{\mathsf{k}}[\,\mathbf{in}\,a\,], a[\,\blacksquare^{\mathsf{k}}_{\eta}\,]\} \quad \tilde{p} \cap \mathrm{fn}(Q) = \emptyset, \quad \mathsf{h} = w(P_1 \mid Q)}{P \xrightarrow{MQ} (\boldsymbol{\nu}\tilde{p})(a^{\mathsf{h}}[\,P_1 \mid Q\,] \mid P_2)}$$

(HO OUT)

$$\frac{P \xrightarrow{\eta^{\mathsf{k}}[\,\mathbf{out}\,a\,]} (\boldsymbol{\nu}\tilde{p})\langle P_1 \rangle P_2 \quad \tilde{p} \cap \mathrm{fn}(Q) = \emptyset, \quad \mathsf{h} = w(P_2 \mid Q)}{P \xrightarrow{\eta^{\mathsf{k}}[\,\mathbf{out}\,a\,]Q} (\boldsymbol{\nu}\tilde{p})(P_1 \mid a^{\mathsf{h}}[\,P_2 \mid Q\,])}$$

We can also shows that our definition of barb coincides with one particular action of the labelled transition system: the action $a[\,\blacksquare^0_*\,]$. This follows from the fact that for all Q we get $a[\,Q\,] \xrightarrow{a[\,\blacksquare^0_*\,]} (\boldsymbol{\nu})\langle Q \rangle \mathbf{0}$. Below, we write $\overline{\mathbf{in}}$ to denote $\blacksquare^0_*$.

Proposition 3. *$P \downarrow_a$ if and only if $P \xrightarrow{a[\,\overline{\mathbf{in}}\,]} (\boldsymbol{\nu})\langle Q \rangle R$ for some Q, R.*

Following [18, 4], in order to provide a characterization of barbed congruence in terms of (weak) labelled bisimilarity, we introduce a new, higher-order transition for each of the first-order transitions whose outcome is a concretion, rather than a process.

The new transitions are collected in Table 4. The higher-order labels occurring in these transitions encode the minimal contribution by the environment needed for the process to complete a transition. Thus, in (HO OUTPUT) the process Q represents the context receiving the value M output by P, and the variable x is a placeholder for that value. In rule (HO IN) the environment provides an ambient $a[Q]$ in which P_1 moves, while in the rule (HO CO-IN) the environment provides an ambient Q moving into a. Finally in rule (HO OUT) we can imagine the environment wrapping the process P with an ambient $a[Q]$.

We are now ready to give the relation of labelled bisimilarity. Let Λ be the set of all labels including the first-order labels of Table 1 as well as the higher-order labels determined by the transitions in Table 4. We denote with λ any label in the set Λ. As usual, we focus on weak bisimilarities based on weak transitions, and use the following notation:

i) $\stackrel{\lambda}{\Longrightarrow}$ denotes $\xrightarrow{\tau}{}^{*} \xrightarrow{\lambda} \xrightarrow{\tau}{}^{*}$;
ii) $\stackrel{\hat{\lambda}}{\Longrightarrow}$ denotes $\xrightarrow{\tau}{}^{*}$ (also noted $\Longrightarrow$) if $\lambda = \tau$ and $\stackrel{\lambda}{\Longrightarrow}$ otherwise.

Definition 3 (Bisimilarity). *A symmetric relation $\mathcal{R}$ over closed processes is a* bisimulation *if $P\mathcal{R}Q$ and $P \xrightarrow{\lambda} P'$ imply that there exists Q' such that $Q \stackrel{\hat{\lambda}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$. Two processes P and Q are bisimilar, written $P \approx Q$, if $P\mathcal{R}Q$ for some bisimulation $\mathcal{R}$.*

This definition of bisimilarity is only given for closed processes. We generalize it to arbitrary processes as follows:

Definition 4 (Full bisimilarity). *Two processes P and Q are full bisimilar, $P \approx_c Q$, if $P\sigma \approx Q\sigma$ for every closing substitution σ.*

Note that the definition of bisimilarity only tests transitions from processes to processes. As expected the full bisimilarity is a congruence: this can be proved using the technique of [18, 4]. Moreover the full bisimilarity is sound but not complete w.r.t. the reduction barbed congruence.

Theorem 5 (Soundness of full bisimilarity). *If $P \approx_c Q$ then $P \cong Q$.*

Proof. Notice that rule (WEIGHT) distinguishes processes of different weights. Then it is enough to show that $\approx_c$ is reduction closed and barb preserving, up to $\equiv$. Assume that $P \searrow P'$. By Theorem 4 $P \xrightarrow{\tau} \equiv P'$. Since $P \approx_c Q$, there exits Q' such that $Q \searrow_* Q'$ and $P' \equiv \approx_c \equiv Q'$. Now assume $P \approx_c Q$. If $P\downarrow_a$ then, by Proposition 3, and rule (HO CO-IN), $P \xrightarrow{a[\,\overline{\mathbf{in}}\,]R} S$ for some R, S. Since $P \approx_c Q$ we know that $Q \stackrel{a[\,\overline{\mathbf{in}}\,]R}{\Longrightarrow} S'$ for some $S' \approx_c S$, from which $Q\Downarrow_a$, as desired.

The failure of completeness is due to the fact that contexts are insensible to repeated entering and exiting. This phenomena is called *stuttering* in [25] and it is typical of movements which do not consume co-capabilities, as happen in our calculus. Let us recall the example of [25]. If $P+Q$ denotes the sum operator à la CCS (which can be encoded for example as $(\mathbf{\nu} a)(a[\,P \mid \overline{\mathbf{open}}\,] \mid a[\,Q \mid \overline{\mathbf{open}}\,] \mid \mathbf{open}\ a)$), then no context can distinguish between the processes:

$$\mathbf{in}\ a\,.\,\mathbf{out}\ a\,.\,\mathbf{in}\ a\,.\,\mathbf{0}$$
$$\mathbf{in}\ a\,.\,\mathbf{out}\ a\,.\,\mathbf{in}\ a\,.\,\mathbf{0} + \mathbf{in}\ a\,.\,\mathbf{0}$$

Notice that it is crucial for BoCa that the process after the movement has weight 0: in fact if we consider

$$P \equiv \mathbf{in}\ a\,.\,\mathbf{out}\ a\,.\,\mathbf{in}\ a\,.\,P'$$
$$Q \equiv \mathbf{in}\ a\,.\,\mathbf{out}\ a\,.\,\mathbf{in}\ a\,.\,P' + \mathbf{in}\ a\,.\,P'$$

where the weight of P' is $k+1$ (so P weights $k+1$ and Q weights $2k+2$), then a context which discriminates these two processes is $a^{k+1}[\,[\,]\,]$, since $a^{k+1}[\,P\,]$ is well-formed while $a^{k+1}[\,Q\,]$ is not.

O'KLAIM: a coordination language with mobile mixins*

Lorenzo Bettini[1] Viviana Bono[2] Betti Venneri[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze,
{bettini,venneri}@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Torino, bono@di.unito.it

Abstract. This paper presents O'KLAIM (Object-Oriented KLAIM), a linguistic extension of the higher-order calculus for mobile processes KLAIM with object-oriented features. Processes interact by an asynchronous communication model: they can distribute and retrieve resources, sometimes structured as incomplete classes, i.e., mixins, to and from distributed tuple spaces. This mechanism is coordinated by providing a subtyping relation on classes and mixins, which become polymorphic items during communication. We propose a static typing system for: (*i*) checking locally each process in its own locality; (*ii*) decorating object-oriented code that is sent to remote sites with its type. This way, tuples can be dynamically retrieved only if they match by subtyping with the expected type. If this pattern matching succeeds, the retrieved code can be composed with local code, dynamically and automatically, in a type-safe way. Thus a global safety condition is guaranteed without requiring any additional information on the local reconfiguration of local and foreign code, and, in particular, without any further type checking. Finally, we present main issues concerning the implementation of O'KLAIM.

1 Introduction

Mixins [14, 22, 1] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to standard class inheritance. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding or overriding specific sets of methods. The same mixin can be used to produce a variety of classes with the same functionality and behavior, since they all have the same sets of methods added and/or redefined. The superclass definition is not needed at the time of writing the mixin definition, thus improving modularity. The uniform extension and modification of classes is instead absent from the classical class-based languages.

Due to their dynamic nature, mixin inheritance can be fruitfully used in a *mobile code* setting [28, 17]. In [8], we introduced MOMI (Mobile Mixins), a coordination model for mobile processes that exchange object-oriented code. The underlying idea

* This work has been partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222, DART project IST-2001-33477 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

motivating MOMI is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not appear to scale well to a distributed context with mobility. MOMI's approach consists in structuring mobile object-oriented code by using mixin-based inheritance, and this is shown to fit into the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that must access some files, can be completed with a base class in order to provide access methods that are specific of the local file system. Conversely, critical operations of a mobile agent enclosed in a downloaded class can be redefined by applying a local mixin to it (e.g., in order to restrict the access to sensible resources, as in a *sand-box*). Therefore, MOMI is a combination of a core coordination calculus and an object-oriented mixin-based calculus.

MOMI highly relies on typing. The most important feature of MOMI's typing is the *subtyping* relation that guarantees safe, yet flexible, code communication. We assume that the code that is sent around has been successfully compiled in its own site (independently from the other sites), and it travels together with its static type. When the code is received on a site (whose code has been successfully compiled, too), it is accepted only if its type is compliant with respect to the one expected, where compliance is based on subtyping. Thus, dynamic type checking is performed only at communication time. This is a crucial matter for mobility, since mobile code and in particular mobile agents are expected to be autonomous: once the communication successfully occurred, transmitted code behaves remotely in a type safe way (no run-time errors due to type violations). This makes the code exchange an *atomic* action.

This paper presents the experimental language O'KLAIM that is obtained by applying MOMI's approach [8] to the language KLAIM [18, 4], which is specifically designed to program distributed systems where mobile components interact through multiple distributed tuple spaces and mobile code. A preliminary design that led to O'KLAIM was introduced in [7]. KLAIM offers a much more sophisticated, complete, and effective coordination mechanism of mobile processes than the toy language of MOMI, where the focus was mainly on the subtyping relation on classes and mixins. O'KLAIM integrates the mixin-based object-oriented features into a mobile code calculus with an asynchronous coordination mechanism. To this aim, the notion of "tuple" is extended to include object-oriented code, therefore KLAIM processes can retrieve from and insert into tuple spaces object-oriented components (in particular, classes, mixins and objects) as well as standard KLAIM processes. A type system is designed for checking statically the extended notion of processes, so that compiled processes contain some static type information which is used dynamically. Therefore, the tuples that are added to a tuple space are decorated with their type, and a process willing to retrieve a tuple from a tuple space will employ an extended pattern matching mechanism that uses also this tuple type information. This matching essentially consists in checking subtyping on object-oriented components. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors) without requiring any further type checking of the whole code. Type safety of the communication results from the static type soundness of local and foreign code and a (global) subject reduction property. In particular, we show that the subject reduction property is based on a crucial property of substitutivity by subtyping. The underlying substitution operation requires specific

methods renaming, in order to avoid name collision problems that arise when classes and mixins are used as first-class data in a mobile code setting where matching relies on subtyping. These new metatheoretical results about the precise concept of substitution to be used extend and improve the results presented for MOMI in [8].

Summarizing, O'KLAIM aims at two complementary goals. Firstly, subtyping on classes and mixins (as designed for MOMI) is successfully experimented as a tractable mechanism to coordinate mobile code exchange within a process calculus with a more sophisticated communication mechanism. Secondly, the language KLAIM is enriched with object-oriented code. This casts some light on how the same approach can be fruitfully used for extending other mobile code languages with safe object-oriented code exchange. Finally, the implementation of O'KLAIM is presented. This consists in a Java package, `momi`, providing the run-time systems for classes and mixins that can be dynamically manipulated and composed. The programming language X-KLAIM (that implements the basic concepts of KLAIM) has been extended in order to be compiled into Java code exploiting the `momi` package.

2 O'KLAIM: an object-oriented KLAIM

O'KLAIM is a linguistic integration of KLAIM with object-oriented features, following the design of MOMI [8]. The coordination part and the object-oriented part are orthogonal, so that, in principle, such an integration would work for any extension/restriction of KLAIM (as discussed in [4]) and also for other calculi for mobility and distribution, such as *DJoin* [23]. We first recall the main features of KLAIM and MOMI and then we show how they are integrated in order to build O'KLAIM.

2.1 The basics of KLAIM

KLAIM (*Kernel Language for Agent Interaction and Mobility*) [18, 4] is a coordination language inspired by the Linda model [24], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are sequences of information items (called *fields*). There are two kinds of fields: *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type, but two formals never match, and two actual fields match only if they are identical.

For instance, tuple (`"foo"`, `"bar"`, 300) matches with (`"foo"`, `"bar"`, !*val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *val* (an integer variable) will contain the value 300.

Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. The distinction between logical and physical locality (and thus the concept of "allocation environment"), and the creation of new nodes and process definitions are not

relevant in the O'KLAIM context, thus, for the sake of simplicity, we omit them in the present formal presentation. Notice, however, that their integration, being orthogonal, is completely smooth.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can perform four basic operations over nodes. The $\mathbf{in}(t)@\ell$ operation looks for tuple t' that matches with t in the tuple space located at ℓ. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The $\mathbf{read}(t)@\ell$ operation differs from $\mathbf{in}(t)@\ell$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at ℓ. The $\mathbf{out}(t)@\ell$ operation adds the tuple t to the tuple space located at ℓ. The $\mathbf{eval}(P)@\ell$ operation spawns process P for execution at node ℓ.

KLAIM is higher-order in that processes can be exchanged as primary class data. While $\mathbf{eval}(P)@\ell$ spawns a process for (remote) evaluation at ℓ, processes sent with an **out** must be retrieved explicitly at the destination site. The receiver can then execute the received process locally, as in the following process: $\mathbf{in}(!X)@\texttt{self}.\mathbf{eval}(X)@\texttt{self}$.

2.2 MOMI and O'KLAIM

MOMI was introduced in [8], where mixin inheritance is shown to be more flexible than standard class inheritance to fit into the dynamic nature of a mobile code scenario. The key rôle in MOMI's typing is played by a *subtyping* relation that guarantees safe, yet flexible and scalable, code communication. MOMI's subtyping involves not only object subtyping, but also a form of class subtyping and mixin subtyping: therefore, subtyping hierarchies are provided along with the inheritance hierarchies. It is important to notice that we are not violating the design rule of keeping inheritance and subtyping separated, since mixin and class subtyping plays a pivotal role only during the communication, when classes and mixins become genuine run-time polymorphic values.

In synthesis, MOMI consists of:

1. the definition of an object-oriented "surface calculus" containing essential features that an object-oriented language must have to write mixin-based code;
2. the definition of a new subtyping relation on class and mixin types to be exploited dynamically at communication time;
3. a very primitive coordination language consisting in a synchronous send/receive mechanism, to study the communication of the mixin-based code among different site.

O'KLAIM integrates the object-oriented component of MOMI, in particular the subtyping relation on classes and mixins (both described in the next section), within KLAIM, which offers a much more sophisticated, complete, and effective coordination mechanism than the toy one of MOMI.

2.3 O'KLAIM: object-oriented expressions

In this section we present the object-oriented part of O'KLAIM, which is defined as a class-based object-oriented language supporting mixin-based class hierarchies via

exp ::=	v	(value)
	new exp	(object creation)
	$exp \Leftarrow m$	(method call)
	$exp_1 \diamond exp_2$	(mixin appl.)
v ::=	$\{m_i : \tau_{m_i} = b_i \ ^{i \in I}\}$	(record)
	x	(variable)
	class $[m_i : \tau_{m_i} = b_i \ ^{i \in I}]$ end	(class def)
	mixin expect$[m_i : \tau_{m_i} \ ^{i \in I}]$ redef$[m_k : \tau_{m_k}$ with $b_k \ ^{k \in K}]$ def$[m_j : \tau_{m_j} = b_j \ ^{j \in J}]$ end	(mixin def)

Table 1. Syntax of object-oriented terms.

mixin definition and *mixin application* (see Table 1). It is important to notice that specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax of the kernel object-oriented calculus.

Object-oriented expressions offer object instantiation, method call and mixin application; $\diamond$ denotes the mixin application operator. An object-oriented value, to which an expression reduces, is either an object, which is a (recursive) record $\{m_i : \tau_{m_i} = b_i \ ^{i \in I}\}$, or a class definition, or a mixin definition, where $[m_i : \tau_{m_i} = b_i \ ^{i \in I}]$ denotes a sequence of method definitions, $[m_k : \tau_{m_k}$ with $b_k \ ^{k \in K}]$ denotes a sequence of method re-definitions, and I, J and K are sets of indexes. Method bodies, denoted here with b (possibly with subscripts), are closed terms/programs and we ignore their actual structure. A mixin can be seen as an abstract class that is parameterized over a (super)class. Let us describe informally the mixin use through a tutorial example:

```
M = mixin                                C = class
  expect [n : τ]                           [n = ...
  redef [m2 : τ2 with ... next ...]        m2 = ...]
  def [m1 : τ1 = ...n()...]              end
end                                                    (new (M ◇ C))⇐ m1()
```

Each mixin consists of three parts:

1. methods *defined* in the mixins, like m_1;
2. *expected methods*, like n, that must be provided by the superclass;
3. *redefined methods*, like m_2, where *next* can be used to access the implementation of m_2 in the superclass.

The application $M \diamond C$ constructs a class, which is a subclass of C.

The typing for the object-oriented code refines essentially the typing rules sketched in [8]. The set $\mathcal{T}$ of types is defined as follows.

$$\tau ::= \Sigma \mid \iota \mid \tau_1 \rightarrow \tau_2 \mid \mathsf{class}\langle\Sigma\rangle \mid \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \qquad \Sigma ::= \{m_i : \tau_{m_i} \ ^{i \in I}\}$$

ι is a basic type and $\rightarrow$ is the functional type operator. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i} \ ^{i \in I}\}$. if $m_i : \tau_{m_i} \in \Sigma$ we say that the *subject* m_i

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (proj) \qquad \frac{\Gamma \Vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (rec)$$

$$\frac{\Gamma \vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \mathsf{class}\ [m_i:\tau_{m_i} = b_i\ ^{i\in I}]\ \mathsf{end} : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}\ (class)$$

$$\frac{\begin{array}{c}\Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k} \vdash \{m_j:\tau_{m_j} = b_j\ ^{j\in J}\} : \{m_j:\tau_{m_j}\ ^{j\in J}\} \\ \Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k}, \bigcup_{j\in J} m_j:\tau_{m_j}, next:\tau_{m_r} \Vdash b_r:\tau'_{m_r} \quad \tau'_{m_r} <: \tau_{m_r} \quad \forall r\in K \\ Subj(\Sigma_{new}) \cap Subj(\Sigma_{exp}) = \emptyset \quad Subj(\Sigma_{new}) \cap Subj(\Sigma_{red}) = \emptyset \\ Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{mixin} \\ \quad\ \mathsf{expect}[m_i:\tau_{m_i}\ ^{i\in I}] \\ \quad\ \mathsf{redef}[m_k:\tau_{m_k}\ \mathsf{with}\ b_k\ ^{k\in K}]\ :\ \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \quad\ \mathsf{def}[m_j:\tau_{m_j} = b_j\ ^{j\in J}] \\ \quad\ \mathsf{end}\end{array}}\ (mixin)$$

where $\Sigma_{new} = \{m_j:\tau_{m_j}\ ^{j\in J}\}, \Sigma_{red} = \{m_k:\tau_{m_k}\ ^{k\in K}\}, \Sigma_{exp} = \{m_i:\tau_{m_i}\ ^{i\in I}\}$

Table 2. Typing rules for object-oriented values

$$\frac{\Gamma \vdash exp : \{m_i:\tau_{m_i}\ ^{i\in I}\} \qquad j\in I}{\Gamma \vdash exp \Leftarrow m_j : \tau_{m_j}}\ (lookup) \qquad \frac{\Gamma \vdash exp : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}{\Gamma \vdash \mathsf{new}\ exp : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (new)$$

$$\frac{\begin{array}{l}\Gamma \vdash exp_1 : \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \Gamma \vdash exp_2 : \mathsf{class}\langle\Sigma_b\rangle \\ \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red}) \\ Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset\end{array}}{\Gamma \vdash exp_1 \diamond exp_2 : \mathsf{class}\langle\Sigma_b \cup \Sigma_{new}\rangle}\ (mixin\ app)$$

Table 3. Typing rules for object-oriented expressions.

occurs in Σ (with type τ_{m_i}). $Subj(\Sigma)$ is the set of the *subjects* of Σ and $Meth(\Sigma)$ is the set of all the method names occurring in Σ (e.g., if $\Sigma = \{m:\{n:\tau\}\}$ then $Subj(\Sigma) = \{m\}$ while $Meth(\Sigma) = \{m,n\}$). As we left method bodies unspecified (see above), we must assume that there is an underlying system to type method bodies and records. We will denote this typing with $\Vdash$. Rules for $\Vdash$ are obviously not specified, but $\Vdash$-statements are used as assumptions in other typing rules. The typing rules for values are in Table 2.

Mixin types, in particular, encode the following information:

1. record types Σ_{new} and Σ_{red} contain the types of the mixin methods (new and redefined, respectively);
2. record type Σ_{exp} contains the *expected* types, i.e., the types of the methods expected to be supported by the superclass;
3. well typed mixins are well formed, in the sense that name clashes among the different families of methods are absent (the last three clauses of the (*mixin*) rule).

The typing rules for expressions are in Table 3.

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle}\ (\sqsubseteq \mathit{class})$$

$$\frac{\Sigma'_{new} <: \Sigma_{new} \qquad \Sigma_{exp} <: \Sigma'_{exp} \qquad \Sigma'_{red} = \Sigma_{red}}{\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle}\ (\sqsubseteq \mathit{mixin})$$

Table 4. Subtype on class and mixin types.

Rule (*mixin app*) relies strongly on a subtyping relation $<:$. The subtyping relation rules depend obviously on the nature of the object-oriented language we choose, but an essential constraint is that it must contain the *width subtyping* rule for record types: $\Sigma_2 \subseteq \Sigma_1 \Rightarrow \Sigma_1 <: \Sigma_2$.

We consider $m : \tau_1$ and $m : \tau_2$ ($\tau_1 \not\equiv \tau_2$) as distinct elements, and $\Sigma_1 \cup \Sigma_2$ is the standard record union. Σ_1 and Σ_2 are considered *equivalent*, denoted by $\Sigma_1 = \Sigma_2$, if they differ only for the order of their pairs $m_i : \tau_{m_i}$.

In the rule (*mixin app*), Σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. The premises of the rule (*mixin app*) are as follows:

i) $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$ requires that the superclass provides all the methods that the mixin expects and redefines;
ii) $Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset$ guarantees that name clashes cannot occur during the mixin application.

Notice that the superclass may have more methods than those required by the mixin constraints. Thus, the type of the mixin application expression is a class type containing both the signatures of all the methods supplied by the superclass (Σ_b) and those of the new methods defined by the mixin (Σ_{new}).

The key point is the introduction of a novel subtyping relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtyping relation is used to match dynamically the actual parameter's types against the formal parameter's types during communication. The part of the operational semantics of O'KLAIM, which describes communication formally, is presented in Section 2.6. The subtyping relation $\sqsubseteq$ is defined in Table 4. Rule ($\sqsubseteq$ *class*) is naturally induced by the (width) subtyping on record types, while rule ($\sqsubseteq$ *mixin*): permits the subtype to define more 'new' methods; prohibits to override more methods; and enables a subtype to require less expected methods.

2.4 O'KLAIM: processes and nets

O'KLAIM syntax is defined in Table 5. In order to obtain O'KLAIM, we extend the KLAIM syntax of tuples t to include any object-oriented value v (defined in Table 1). In particular, differently from KLAIM, formal fields are now explicitly typed. Actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ can be used to move object-oriented code (together with the other KLAIM items) from/to a locality ℓ, respectively. Moreover, we add to KLAIM processes the construct $\mathsf{let}\ x = exp\ \mathsf{in}\ P$ in order to pass to the sub-process P the result of computing an object-oriented expression exp (for exp syntax see Table 1). We use the following syntactic convention: x, X and χ are variables representing

$$
\begin{array}{lll}
P ::= \mathbf{nil} & \text{(null process)} & N ::= l :: p \quad \text{(single node)} \\
\quad \mid \; act.P & \text{(action prefixing)} & \quad \mid \; N_1 \parallel N_2 \quad \text{(net composition)} \\
\quad \mid \; P_1 \mid P_2 & \text{(parallel composition)} & p ::= P \mid \langle t \rangle \mid p_1 \mid p_2 \quad \text{(located item)} \\
\quad \mid \; X & \text{(process variable)} & \\
\quad \mid \; \mathsf{let}\ x = exp\ \mathsf{in}\ P & \text{(OO expression)} &
\end{array}
$$

$$
act ::= \mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \qquad \ell ::= l \mid \chi \qquad t ::= f \mid f,t
$$

$$
f ::= arg \mid !id : \sigma \qquad id ::= x \mid X \mid \chi \qquad arg ::= id \mid e \mid P \mid l \mid v
$$

Table 5. O'KLAIM syntax (see Table 1 for the syntax of *exp* and *v*, and Section 2.5 for types σ).

$$
\frac{}{\Gamma, id : \sigma \vdash id : \sigma}\ (proj) \qquad \frac{}{\Gamma \vdash l : \mathsf{loc}}\ (loc) \qquad \frac{}{\Gamma \vdash \mathbf{nil} : \mathsf{proc}}\ (nil)
$$

$$
a \equiv \mathbf{in}, \mathbf{read}, \mathbf{out} \qquad \frac{\begin{array}{l}\Gamma \vdash \ell : \mathsf{loc} \\ \Gamma \vdash f_i : \sigma_i \qquad i = 1, \ldots, n \ \wedge\ f_i \equiv arg \\ \Gamma \cup ftypes(f_1, \ldots, f_n) \vdash P : \mathsf{proc}\end{array}}{\Gamma \vdash a(f_1, \ldots, f_n)@\ell.P : \mathsf{proc}}\ (action)
$$

$$
ftypes(f,t) = \begin{cases} \{id : \sigma\} \cup ftypes(t) & \text{if } f \equiv !id : \sigma \\ ftypes(t) & \text{otherwise} \end{cases}
$$

$$
\frac{\Gamma \vdash Q : \mathsf{proc} \qquad \Gamma \vdash \ell : \mathsf{loc}}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P : \mathsf{proc}}\ (eval)
$$

$$
\frac{\Gamma \vdash P_1 : \mathsf{proc} \qquad \Gamma \vdash P_2 : \mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2) : \mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp : \tau \qquad \Gamma, x : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{let}\ x = exp\ \mathsf{in}\ P : \mathsf{proc}}\ (let)
$$

Table 6. Typing rules for processes

object-oriented values, processes and localities, respectively. A constant locality (e.g., IP:port) is denoted by l. Moreover, e is a basic expression (i.e., not object-oriented).

A *Net* is a finite collection of *nodes*. A node is a pair where the first component is a (constant) locality and the second component is either a process P or a tuple $\langle t \rangle$ or a composition of processes and tuples. Thus, a tuple space is represented by the parallel composition of located tuples. Notice that programmers write only located processes, while located tuples are produced at run-time by evaluating **out** actions (see Table 8).

2.5 Typing for O'KLAIM

In order to type processes and nets, we extend the set of types $\mathcal{T}$ to $\mathcal{T}^* = \mathcal{T} \cup \{\mathsf{proc}, \mathsf{loc}\}$. σ will range over $\mathcal{T}^*$; in particular, loc is used to type localities and proc for well-typed processes. Typing rules for processes are defined in Table 6. O'KLAIM type system is not concerned with access rights and capabilities, as it is instead the type system for KLAIM presented in [19]. In the O'KLAIM setting, types serve the purpose of avoiding the "message-not-understood" error when merging local and foreign object-oriented code in a site. Thus, we are not interested in typing actions inside processes:

from our perspective, an O'KLAIM process is well typed when it has type proc, which only means that the object-oriented code that the process may contain is well typed.

O'KLAIM requires that every process is statically type-checked separately on its site and annotated with its type. The annotation process, not formally presented here, can be performed by the compiler during type checking: namely, every tuple item t_i that takes part in the information exchange (which may be an object-oriented value) must be decorated with its type information, denoted by $t_i^{\sigma_i}$. The types of the tuples are built statically by the compiler, while the types of tuple formal fields must be written explicitly by the programmer. In a process of the form $\mathbf{in}(!id:\sigma)@\ell.P$, the type σ is used to statically type check the continuation P, where *id* is possibly used. More generally, concerning (*action*) rule, in a process performing an operation with a tuple (i.e., **out**, **read** and **in**), the actual fields of the tuple are type checked, and the types of formal fields (collected by the function *ftypes*) are used to type check the continuation.

We observe that the typing rules for object-oriented expressions are syntax-driven and do not contain an explicit subsumption rule. Thus, they define an algorithm to assign a principal type to each expression, in a given environment Γ. Analogously, both subtyping and typing rules for processes are in an algorithmic shape.

2.6 Operational semantics for O'KLAIM

The operational semantics of O'KLAIM involves two sets of rules. The first set of rules describes how object-oriented expressions reduce to values and is denoted by $\twoheadrightarrow$. We omit here most of the rules because they are quite standard; they can be found in [6, 3]. However, we want to discuss the rule concerning mixin application, that produces a new class containing all the methods which are added and redefined by the mixin and those defined by the superclass. The rule (*mixinapp*) is presented in Table 7. The function *override*, defined below and used by rule (*mixinapp*), takes care of introducing in the new class the overridden methods. In particular, in the body of a mixin's overridden method m_i, the reserved variable *next* can be used to denote m_i's implementation in the superclass: this "old" implementation is given a fresh name $m_{i'}$. Dynamic binding is then implemented for redefined methods, and old implementations from the super class are basically hidden in the derived class, since they are given a fresh name.

Definition 1. *Given two method sets, ρ_1 and ρ_2, the result of override(ρ_1,ρ_2) is the method set ρ_3 defined as follows:*

- *for all $m_i:\tau_{m_i}=b_i\in\rho_2$ such that $m_i\neq m_j$ for all $m_j:\tau_{m_j}=b_j\in\rho_1$, then $m_i:\tau_{m_i}=b_i\in\rho_3$;*
- *for all $m_i:\tau_{m_i}=b_i\in\rho_1$ such that $m_i:\tau_{m_i}=b_i'\in\rho_2$, let $m_{i'}$ be a fresh method name: then $m_{i'}:\tau_{m_i}=b_i'\in\rho_3$ and $m_i:\tau_{m_i}=b_i[m_{i'}/next]\in\rho_3$.*

Notice that name clashes among methods during the application will never take place, since they have already been solved during the typing of mixin application.

The second set of rules for O'KLAIM, shown in Table 8, concerns processes and it is an extension of the operational semantics of KLAIM. Notice that the O'KLAIM's operational semantics must be defined on typed *compiled* processes, i.e., processes where each object-oriented value and tuples are decorated with their types, as explained in

$$\frac{exp_1 \twoheadrightarrow \left(\begin{array}{l}\mathsf{mixin}\\ \quad \mathsf{expect}[m_i : \tau_{m_i}{}^{i\in I}]\\ \quad \mathsf{redef}[m_k : \tau_{m_k} \text{ with } b_k{}^{k\in K}]\\ \quad \mathsf{def}[m_j : \tau_{m_j} = b_j{}^{j\in J}]\\ \mathsf{end}\end{array}\right) \qquad exp_2 \twoheadrightarrow \mathsf{class}\,[m_l : \tau_{m_l} = b_l{}^{l\in L}]\,\mathsf{end}}{exp_1 \diamond exp_2 \twoheadrightarrow \left(\begin{array}{l}\mathsf{class}\\ \quad [m_j : \tau_{m_j} = b_j{}^{j\in J}] \cup\\ \quad override([m_k : \tau_{m_k} = b_k{}^{k\in K}], [m_l : \tau_{m_l} = b_l{}^{l\in L}])\\ \mathsf{end}\end{array}\right)}$$

Table 7. The (*mixinapp*) operational rule

$$\frac{}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \rightarrowtail l :: P \parallel l' :: P' \mid \langle t \rangle}\ (\text{OUT})$$

$$\frac{match(t,t')}{l :: \mathbf{in}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \mathbf{nil}}\ (\text{IN})$$

$$\frac{match(t,t')}{l :: \mathbf{read}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \langle t' \rangle}\ (\text{READ})$$

$$\frac{}{l :: \mathbf{eval}(P)@l'.P' \parallel l' :: P'' \rightarrowtail l :: P' \parallel l' :: P' \mid P}\ (\text{EVAL})$$

$$\frac{exp \twoheadrightarrow v}{l :: \mathsf{let}\ x = exp\ \mathsf{in}\ P \rightarrowtail l :: P[v/x]}\ (\text{LET}) \qquad \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2 \qquad N_2 \equiv N'}{N \rightarrowtail N'}\ (\text{NET})$$

Table 8. O'KLAIM operational rules

Section 2.5, because the crucial point is the dynamic matching of types. In fact, an **out** operation adds a tuple decorated with a (static) type to a tuple space, and a process can perform an **in** action by synchronizing with a process which represents a matching typed tuple. The rule for let $x = exp$ in P relies on the reduction relation for object-oriented expressions $\twoheadrightarrow$.

The predicate for tuples, *match*, is presented in Table 9. The matching rules exploit the static type information, delivered together with the tuple items, in order to dynamically check that the received item is correct with respect to the type of the formal field, say τ. Therefore, an item is accepted if and only if it is subtyping-compliant with the

$$match(e,e) \qquad match(l,l) \qquad \frac{match(t_2,t_1)}{match(t_1,t_2)}$$

$$\frac{match(t_1,t_2) \quad match(t_3,t_4)}{match((t_1,t_3),(t_2,t_4))} \qquad \frac{match(\sigma,\sigma_i)}{match(!id : \sigma, t_i{}^{\sigma_i})}$$

$$match(\sigma_1,\sigma_2) = \begin{cases} \sigma_1 \sqsubseteq \sigma_2 & \text{if } \sigma_1 \text{ and } \sigma_2 \text{ are mixin or class types} \\ \sigma_1 <: \sigma_2 & \text{otherwise} \end{cases}$$

Table 9. Matching rules (with $\mathsf{proc} <: \mathsf{proc}$ and $\mathsf{loc} <: \mathsf{loc}$)

expected type of the formal field. Informally speaking, one can accept any class containing more resources than expected, and any mixin with weaker requests about methods expected from the superclass can be accepted. This subtyping checking is analogous to the one we would perform in a sequential language where mixins and classes could be passed as parameters to methods. In a sequential setting, this dynamic checking might look as a burden (for example, in [13], mixins and classes are first-order entities, i.e., they can be passed as parameters in methods, but the matching among formal and actual parameters is by syntactic equality on types and not by subtyping), but in a distributed mobile setting the burden seems well-compensated by the added flexibility in communications.

Finally, the semantics for the distributed part is based on structural congruence and reduction relations. Reduction represents individual computation steps, and is defined in terms of structural congruence. The structural congruence $\equiv$ allows the rearrangement of the syntactic structure of a term so that reduction rules may be applied. It is defined as the least congruence relation closed under the following rules.

$$N_1 \parallel N_2 = N_2 \parallel N_1 \qquad (N_1 \parallel N_2) \parallel N_3 = N_1 \parallel (N_2 \parallel N_3) \qquad l :: P = l :: P \mid \mathbf{nil}$$
$$l :: (P_1 \mid P_2) = l :: P_1 \parallel l :: P_2$$

As a final remark, let us observe that we do not define a matching predicate for actual fields containing object-oriented values and processes since this would require to decide equalities on classes, mixins and objects (e.g., equality on their interfaces) and on processes (e.g., a bisimulation). This issue is out of the scope of the present work, since matching between two actual fields does not involve any substitution and then does not cause problems w.r.t. typing.

3 Typing issues and subject-reduction property

The important point in O'KLAIM semantics is that if a process P (statically well-typed) retrieves a tuple by the subtyping matching mechanism and the retrieved value is merged in the continuation of P, then the evaluation proceeds without any additional type-checking. Thus, in order to obtain the subject-reduction theorem, we need to prove that substitution preserves well-typedness, in particular when classes and mixins are replaced to variables inside object-oriented expressions. In the following, we address this issue and we outline the main technical steps, skipping proofs and details for lack of space.

The crucial case concerns mixin application expressions; namely if class/mixin variables are replaced by classes/mixins having a subtype, accidental overrides can occur because of names of the new methods added by the replacing value (see the definition of $\sqsubseteq$). This matter is related to the "width subtyping versus method addition" problem (well known in the object-based setting, see for instance [21]), that in our case boils down to a careful management of these *dynamic name clashes*. Thus, we must define a suitable capture-avoid-substitution, $[\ ]$, requiring possible renaming of methods with fresh names.

Definition 2 (Substitution by refresh). *If x is a class variable of type $\mathsf{class}\langle\Sigma\rangle$ and C is a class value of type $\mathsf{class}\langle\Sigma'\rangle$ such that $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$, then $[C/x]$ denotes the*

replacement of C' to x, where C' is obtained from C by renaming all methods belonging to $Meth(\Sigma') - Meth(\Sigma)$ with fresh names. For mixins variables and values, the renaming acts on all and only the methods belonging to $Meth(\Sigma'_{new}) - Meth(\Sigma_{new})$.

With our solution, new methods added by a class or a mixin value during substitution are hidden by renaming, for each occurrence of the variable to be replaced (this is very similar to the "privacy via subsumption" of [27]). On the other hand, we only rename methods that do not appear in the type of the variable x. This second constraint ensures a basic property: the refreshed version C' of C has a type τ' such that $\tau' \sqsubseteq \mathsf{class}\langle\Sigma\rangle$. The same holds for refreshed mixins.

Now, using this definition, we can prove that substitution is type safe. For simplicity, in order to deal with $<:$ and $\sqsubseteq$ at the same time, we introduce the meta notation:

$$\tau_1 \preccurlyeq \tau_2 = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ is a mixin or a class type} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

Lemma 1 (Substitution by narrowing). *Let v, exp and P be an object-oriented value, an object-oriented expression and a process, respectively,*

1. *if $x : \tau_1 \vdash exp : \tau$ and $\Gamma \vdash v : \tau_2$ where $\tau_2 \preccurlyeq \tau_1$, then $\Gamma \vdash exp[v/x] : \tau'$ with $\tau' \preccurlyeq \tau$;*
2. *if $x : \tau_1 \vdash P : \mathsf{proc}$ and $\Gamma \vdash v : \tau_2$ where $\tau_2 \preccurlyeq \tau_1$, then $\Gamma \vdash P[v/x] : \mathsf{proc}$.*

Sketch of proof:

1. By induction on typing rules for expressions. The only crucial case is when *exp* is a mixin application and *v* is a class value or a mixin. Notice that, *exp* is well-typed and no method occurring in the type *x* is renamed; then the last condition in (*mixin app*) rule is preserved and ensures that no name clash can occur after substitution.
2. By induction on typing rules for processes using the previous point.

Summarizing, the type safety of the communication results from two main properties: (i) static type soundness of local and foreign code; (ii) the preservation of well-typedness under substitution by subtyping. It is standard to verify that all the other rules concerning $\twoheadrightarrow$ preserve well typedness and so we obtain the *subject-reduction theorem*. Thus, the local evaluation of a process cannot produce errors like "message-not-understood" even if it retrieves data from foreign sites and merges it in the local configuration. In other words, a *well-typed net* (i.e., a net where each process in each site is well-typed) remains well typed during its evolution (*global safety condition*).

We remark that, from the point of view of the implementation, the above treatment of "global" fresh names can be solved with static binding for the mentioned methods. The technique of using the static types of variables and the actual types of substituted mixin and class definitions may recall the approach of [22] of allowing overriding, i.e., dynamic binding, only for methods declared in the mixin's *inheritance interface*.

4 The implementation

We recall that the implementation we present is based on X-KLAIM [10, 11] (extended with the proper object-oriented mixin-based primitives), used both as the "surface"

object-oriented calculus and as the coordination language, with the added bonus of being able to write methods that can perform KLAIM actions, all the same guaranteeing absence of message-not-understood run-time errors, as shown in the previous section.

The implementation of the O'KLAIM object-oriented component in Java consists in a package `momi` presented in [2] and described in details in [3]. This package provides the run-time system, or the virtual machine, for classes, mixins and objects that can be downloaded from the network and dynamically composed (via the mixin application operation). It thus provides functionalities for checking subtyping among classes and among mixins and for building at run-time new subclasses. Since we abstract from the specific communication and mobility features, this package does not provide means for code mobility and network communication, so that `momi` can be smoothly integrated into existing Java mobility frameworks. We would like to stress that this package should be thought of as an "assembly" language that is the target of a compiler for a high-level language (in our case the language is X-KLAIM). If `momi`, as it is, was used for directly writing object-oriented expressions, the programmer would be left with the burden of writing methods containing Java statements dealing with `momi` objects, classes and mixins, and to check manually that they are well typed. Basically these are the same difficulties a programmer has to face when using an assembly language directly, instead of a high-level language. We could say that `momi` enhances the functionalities of the Java virtual machine: while the latter already provides useful mechanisms for dynamically loading new classes into a running application, the former supplies means for dynamically building class hierarchies (based on mixins) and for inserting new subclasses into existing hierarchies (which is not possible in Java).

In order to implement O'KLAIM we extended the KLAIM programming framework that consists in the programming language X-KLAIM [10, 11], which extends KLAIM with high-level programming constructs, and KLAVA [12] a Java package that implements the run-time system for X-KLAIM operations (X-KLAIM programs are compiled into Java programs that use KLAVA). The package KLAVA already provided all the primitives for network communication, through distributed tuple spaces, and, in particular, for code mobility, not supplied by `momi`. Thus the package has been modified in order to be able to exchange code that is based on `momi`, and for performing subtyping on `momi` elements during pattern matching by relying on the `MoMiType` classes and the associated subtyping. On the other hand, the X-KLAIM compiler generates code that uses both the KLAVA package and `momi`. Obviously, before generating code, it also performs type checking according to the type system defined by MOMI. All this software is freely available at `http://music.dsi.unifi.it`.

The programming example shown in this section involves mixin code mobility, and implements "dynamic inheritance" since the received mixin is applied to a local parent class at run-time. We assume that a site provides printing facilities for local and mobile agents. Access to the printer requires a driver that the site itself has to provide to those that want to print, since it highly depends on the system and on the printer. Thus, the agent that wants to print is designed as a mixin, that expects a method for actually printing, `print_doc`, and defines a method `start_agent` through which the site can start its execution. The actual instance of the printing agent is instantiated from a class dynamically generated by applying such mixin to a local superclass that provides the

```
mixin MyPrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str
 begin
  return
   this.print_doc
    (this.preprocess("my document"))
 end;
 def preprocess(doc : str) : str
 begin
  return "preprocessed(" + doc +")"
 end
end

rec SendPrinterAgent[server : loc]
 declare
  var response : str
 begin
  out(MyPrinterAgent)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
mixin PrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str;
end

class LocalPrinter
 print_doc(doc : str) : str
 begin
  # real printing code omitted :−)
  return "printed " + doc
 end;
 init()
 begin
  nil # foo init
 end
end

rec ReceivePrinterAgent[]
 declare
  var rec_mixin : mixin PrinterAgent;
  var result : str
 begin
  in(!rec_mixin)@self;
  result :=
   (new rec_mixin <> LocalPrinter).start_agent();
  out(result)@self
 end
```

Listing 1: The printer agent example.

method `print_doc` acting as a wrapper for the printer driver. However the system is willing to accept any agent that has a compatible interface, i.e., any mixin that is a subtype of the one used for describing the printing agent. Thus any client wishing to print on this site can send a mixin that is subtyping compliant with the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 1, where **rec** is the X-KLAIM keyword for defining a process, presents a possible implementation of the printing client node (on the left) and of the printer server node (on the right). The printer client sends to the server a mixin `MyPrinterAgent` that complies with (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent`. In particular `MyPrinterAgent` mixin will print a document on the printer of the server after preprocessing it (method `preprocess`). On the server, once the mixin is received, it is applied to the local (super)class `LocalPrinter`, and an object (the agent) is instantiated from the resulting class, and started so that it can actually print its document. The result of the printing task is then retrieved and sent back to the client.

We observe that the sender does not actually know the mixin name `PrinterAgent`: it only has to be aware of the mixin type expected by the server. Furthermore, the sent mixin can also define more methods than those specified in the receiving site, thanks to the mixin subtyping relation. This adds a great flexibility to such a system, while hiding these additional methods to the receiving site (since they are not specified in the receiving interface they are actually unknown statically to the compiler).

5 Conclusions and related work

We have presented the kernel language O'KLAIM, which extends the higher-order calculus KLAIM for mobile processes with mixin-based object-oriented code. The novel

contributions of this paper, with respect to [8] (where we firstly presented design motivations for a mixin-based approach in a mobile context), can be summarized as follows:

1. we integrate the basic ideas of [8] into a mobile process calculus with an asynchronous and more sophisticated communication mechanism;
2. we refine the typing for the object-oriented component, we define a new typing system for KLAIM processes, and we study main typing concerns (in particular, a notion of substitution with renaming) in order to demonstrate the soundness of the proposed solution;
3. we present an implementation of O'KLAIM.

Keeping the O'KLAIM object-oriented calculus and the O'KLAIM processes separated may appear a limitation, but in fact this is not true. Our system consists of three components: the "surface" object-oriented component, a mixin/class subtyping relation, and a coordination calculus. If the object-oriented component is chosen to be an object-oriented concurrent/mobile language, the two components (object-oriented and concurrent/mobile) may interleave in a deeper way. A good example is the O'KLAIM implementation presented in Section 4: in there, X-KLAIM (extended with the proper object-oriented mixin-based primitives) is both the "surface" object-oriented calculus and the coordination language, so that method bodies can perform KLAIM actions. The matching mechanism that allows safe interactions during code exchange is based on the subtyping relation that acts as a general glue to glue together the two language components, of whichever nature they are (as long as the object-oriented one implements classes and mixins).

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [16] is a lexically-scoped language providing distributed object-oriented computation. Mobile code maintains network references and provides transparent access to remote resources. In [15], a general model for integrating object-oriented features in calculi of mobile agents is presented where agents are extended with constructs for remote method invocations. Other works, such as, e.g., [20, 26, 25] do not deal explicitly with mobile distributed code. Our approach is more related to papers, as [29], where properties of distributed systems are enforced by a typing system equipped with subtyping. In our case the property we address is a flexible and type-safe coordination for exchanging code among processes, up- and down-loading classes and mixins from different sites.

Further extensions of O'KLAIM are topics for future developments: subtyping can be extended to *depth subtyping*, which offers a more flexible communication pattern (see [9] for a preliminary discussion) and the object-oriented component can be enriched with *incomplete objects*, i.e., objects instantiated from mixins [5].

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a java extension with mixins. *ACM Transaction on Programming Languages and Systems*, 2003. To appear.
2. L. Bettini. A Java package for class and mixin mobility in a distributed setting. In *Proc. of FIDJI'03*, LNCS, 2003. To appear.
3. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.

4. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In *Global Computing – Trento*, LNCS. Springer, 2003. To appear.
5. L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *FOOL 11*, 2004.
6. L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Manuscript.
7. L. Bettini, V. Bono, and B. Venneri. Towards Object-Oriented KLAIM. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
8. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In *Proc. of Coordination Models and Languages*, volume 2315 of *LNCS*, pages 56–71. Springer, 2002.
9. L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *Proc. of FOOL*, 2003.
10. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of WETICE*, pages 110–115. IEEE Computer Society Press, 1998.
11. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
12. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.
13. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer, 1999.
14. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA*, pages 303–311, 1990.
15. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
16. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
17. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of ICSE*, pages 22–33. ACM Press, 1997.
18. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
19. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
20. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *Proc. of CONCUR*, volume 1119 of *LNCS*, pages 655–670, Pisa, Italy, 26–29 Aug. 1996. Springer.
21. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.
22. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
23. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. of CONCUR*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
24. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
25. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Proc. of HLCL*, volume 16.3 of *ENTCS*. Elsevier, 1998.
26. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Proc. of TPPP*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
27. J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, 172:2–28, 2002. 3rd special issue of Theory and Practice of Object-Oriented Systems (TAPOS).
28. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.
29. N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher Order Mobile Processes (extended abstract). In *Proc. of CONCUR'99*, volume 1664 of *LNCS*, pages 557–572. Springer-Verlag, 1999.

A Formal Basis for Reasoning on Programmable QoS*

Rocco De Nicola[1], Gianluigi Ferrari[2], Ugo Montanari[2], Rosario Pugliese[1], and Emilio Tuosto[2]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze, Via C. Lombroso 6/17, 50134 Firenze (Italy) {denicola,pugliese}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa, Via M. Buonarroti 2, 56100 Pisa (Italy) {giangi,ugo,etuosto}@di.unipi.it

Abstract. The explicit management of *Quality of Service* (QoS) of network connectivity, such as, e.g., working cost, transaction support, and security, is a key requirement for the development of the novel wide area network applications. In this paper, we introduce a foundational model for specification of QoS attributes at application level. The model handles QoS attributes as semantic constraints within a graphical calculus for mobility. In our approach QoS attributes are related to the programming abstractions and are exploited to select, configure and dynamically modify the underlying system oriented QoS mechanisms.

1 Introduction

Wide-Area Network (WAN) applications have become one of the most important classes of applications in distributed computing. Currently, Internet and World Wide Web are the primary environments for designing, developing and distributing applications. Network services have evolved into self-contained components which inter-operate easily by exploiting WEB-based access protocols [17]. In addition, network services may adapt themselves to match the particular capabilities of a variety of devices ranging from traditional PCs, to Personal Digital Assistants and Mobile Phones having intermittent connectivity to the network.

In this new scenario both final users and WAN application designers put special emphasis on Quality of Service (QoS) issues. For final users, the perceived QoS of their computations is not only dependent on the performance of WEB

* R. De Nicola has been supported by MIUR project **NAPOLI** and EU-FET project **MIKADO** IST-2001-32222. G. Ferrari has been supported by MIUR project **NAPOLI** and EU-FET project **PROFUNDIS** IST-2001-33100. U. Montanari has been supported by MIUR project **COMETA** and EU-FET project **AGILE** IST-2001-32747. R. Pugliese has been supported by MIUR project **NAPOLI** and EU-FET project **AGILE** IST-2001-32747. E. Tuosto has been supported by MIUR project **NAPOLI** and EU-FET project **PROFUNDIS** IST-2001-33100. All authors have been supported by the MIUR project **SP4** "Architetture Software ad Alta Qualità di Servizio per Global Computing su Cooperative Wide Area Networks".

servers but also on the availability of certain resources. Indeed, in addition to access services, users are allowed to control the QoS they receive. Here, QoS is meant as a measure of the *non functional* properties of services along multiple dimensions. For instance, *network bandwidth* is a QoS measure for multimedia services. *Timely response* and *security* are other examples of (higher level) QoS measures.

In general, QoS attributes are special parameters of network services. *Awareness* of these information is crucial for choosing network services to match user requirements. For instance, final users can react to network congestion by binding their network devices to different sites where the requested services are available. Similarly, QoS awareness is exploited by WAN application designers to control resource usage and resource access in order to guarantee and maintain certain security levels and to provide users with differentiated QoS.

The advances in network technologies and the growth of commercial WEB services have prompted questions about suitable mechanisms for providing QoS guarantees. In the last few years, several models have been proposed to meet the demands of QoS. We mention the *Resource Reservation Protocol* (RSVP) [6], *Differentiated Services* [5], *Constraint-based Routing* [26], and we refer to [28] for a detailed discussion of this topic. This stream of research is basically *system-oriented*: it focuses on the lower layers of the Internet protocol stack.

Another significant line of research has dealt with enhancing existing distributed programming middlewares to support QoS features. QoS-aware middlewares allow clients to express their QoS requirements at a higher level of abstraction. In this way the application has good degree of control over QoS without having to deal with low-level details. Examples of QoS-aware middleware are Agilos [22], Mobiware [1], and Globus [13].

Novel computational models exploit the idea of *network awareness* to manage the dynamic nature of network infrastructures. This has led to the development of models and foundational calculi where *mobility* is the basic notion and applications have control over localities where computation progresses. These computational models do not provide natural and expressive QoS mechanisms. Some preliminary results in this direction can be found in [7]; there a calculus is introduced which incorporates a notion of communication rate (bandwidth) and describe some programming constructs based on this calculus.

We believe that the ability of identifying and managing QoS requirements at the early stages of software development (*programmable QoS*) is a key issue and the *added-value* of the evolutionary paradigms for WAN programming. Indeed, programmable QoS allows one to evaluate the impact of QoS requirements on the overall software architectures without committing to specific low level technological details. Moreover, when implementing a specific application, this information can be used to select and configure the primitives of the underlying QoS-aware middleware.

Our research goal is to contribute at a formal understanding of programmable QoS, as a step toward the development of proof techniques and tools for the automated verification and certification of properties of WAN applications. To

achieve this, we provide an appropriate level of abstraction to describe programmable QoS together with a semantic model which can be used to experiment programmable QoS without relying on the current complex network technologies.

In particular, we have abstracted the basic features of the problem in a calculus with primitives which control explicitly QoS attributes. This calculus, that we call KAOS (KLAIM-*based calculus for Application Oriented QoS*), is a first contribution of the paper. KAOS builds on KLAIM (*Kernel Language for Agent Interaction and Mobility*) [9]. KLAIM is an experimental kernel programming language specifically designed to model and program WAN applications with located services and mobility. KLAIM naturally supports a *peer-to-peer* programming model where interacting peers (network nodes or sites, in KLAIM terminology) cooperate to provide common sets of services. KAOS enriches KLAIM networking constructs [2] with attributes which are used to specify the QoS properties of peer groups. This QoS attributes may be seen, e.g., as a value specifying the *abstract* cost c of using a given connection, or as a pair $\langle c, \pi \rangle$ that additionally specifies the set π of *access rights* (in the sense of [10]), or, more generally, as a vector whose components represent different QoS aspects.

Although KAOS provides semantic idioms to deal with programmable QoS, it does not directly handle how QoS attributes can be effectively enforced and guaranteed. To this purpose, we introduce a formal model that enables us to describe and reason about QoS guarantees. Our model is based on (hyper-)graphs and local graph synchronization and extends the graphical calculus for mobility introduced in [15]. Graphs naturally provide the capabilities to describe internetworking systems: edges represent network components and vertices model the network environments. If some edges share a vertex then the corresponding components may interact by exploiting the underlying network communication infrastructure. Graph synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that properties of components are specified as constraints over their local resources. Hence, local evolutions of components depend on the outcome of a (possibly distributed) constraint satisfaction algorithm. In other words, the actual behaviour is the result of a constraint solving algorithm [25, 29].

As a second contribution of the paper we provide an operational semantics for KAOS in terms of the graphical calculus. The key issue of the approach is that QoS attributes become semantic constraints of the graphical calculus. In other words, the model fosters a declarative approach by identifying the points where satisfaction of QoS requirements has a strong impact on behaviours. Hence, the goal of finding a connection between two KAOS peers with a certain QoS level corresponds to find the optimal path with respect to the QoS constraints. The main result of the paper proves that the graphical semantics provides the desired properties. In particular, we show that whenever a remote operation is performed, the graphical calculus always select the optimal path with respect to the QoS constraints set up by the application.

Throughout the paper we focus on KAOS since the specification of QoS attributes and their enforcement in the graphical semantics are specifically designed for KLAIM. However, similar techniques could be applied to other process languages in which network infrastructures can be specified declaratively.

Structure of the paper. Syntax and semantics of KAOS are given in Sections 2 and 3, respectively. An example of how an application can be modeled with KAOS is presented in Section 4. Section 5 defines hypergraphs and their semantics in terms of hyperedge replacement. A mapping of KAOS into hypergraphs and their semantics is defined in Section 6, while productions for edges used in translations of KAOS nets are detailed in Section 7. Section 8 shows how path reservation can be obtained by simple changes of productions for hypergraphs of KAOS nets. Section 9 applies the translation of Section 6 to the example in Section 4 and shows that the paths reserved by using the productions in Section 8 are the optimal paths computed by the Floyd-Warshall algorithm applied to graphs representing KAOS nets. Finally, some concluding remarks are reported in Section 10.

2 KAOS: a Calculus for Programmable QoS

This section presents the syntax of KAOS (KLAIM-*based calculus for Application Oriented QoS*) as reported in Table 1. We assume a set $\mathcal{N}$ of *names* ranged over by metavariables r, s, t, ... Names provide the abstract counterpart of the set of *communicable* values. Generally speaking, communicable values consist of expressions, processes, tuples (ordered sequences of values), and so on. For the sake of simplicity, here communicable values are simply *localities*. These are the syntactic ingredient used to express the idea of administration domain: computations at a certain locality are under the control of a specific authority. We also assume a set of *process variables* ranged over by $X, Y, \ldots$.

Finally, we assume a set of *costs* which are special values used to measure and manage QoS attributes. Cost values (ranged over by κ) are equipped with two binary operations (an additive and a multiplicative operation) together with a partial order relation $\sqsubseteq$. Formally, the algebraic structure of cost values is a *constraint semiring* [3] (or *c-semiring*, for short). An algebraic structure $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ is a constraint semiring if A is a set ($\mathbf{0}, \mathbf{1} \in A$), and $+$ and $\times$ are binary operations on A that satisfy the following properties:

- $+$ is commutative, associative, idempotent, $\mathbf{0}$ is its unit element and $\mathbf{1}$ is its absorbing element;
- $\times$ is commutative, associative, distributes over $+$, $\mathbf{1}$ is its unit element, and $\mathbf{0}$ is its absorbing element.

The additive operation of a c-semiring induces a partial order on A defined as $a \leq_A b \iff \exists c : a + c = b$. The minimal element is thus $\mathbf{0}$ and the maximal $\mathbf{1}$. Hence, $a \leq_A b$ means that a is more constrained than b.

N	::=	NETS	l	::=	LINKS
	$s ::^{L} P$	*Single node*		$\langle s, \kappa \rangle$	*Incoming link*
	$(\nu\, s)N$	*Node restriction*		$\langle \kappa, s \rangle$	*Outgoing link*
	$N_1 \parallel N_2$	*Net composition*			
			P	::=	PROCESSES
γ	::=	ACTIONS		$\mathbf{0}$	*Null process*
	(s)	*Input*		$\gamma.P$	*Action prefixing*
	$\nu(s \cdot \kappa)$	*Node creation*		$\langle t \rangle$	*Output*
	$\overset{\kappa}{\frown} s$	*Login*		$\varepsilon(P)@s$	*Process spawning*
	$s \overset{\kappa}{\smile}$	*Accept*		$P_1 \mid P_2$	*Process composition*
	$\delta\, l$	*Disconnect*		X	*Process variables*
				rec $X.P$	*Recursion*

Table 1. KAOS Syntax

The actual definitions of the operations of the constraint semiring depend on the notion of costs we intend to capture. For instance, the constraint semiring of truth values (the structure $(\{T, F\}, \vee, \wedge, F, T)$) allows reasoning on the availability of network connections.

The following examples give an intuition of some c-semiring structures that will be exploited in next sections.

Example 1. An example of c-semiring is the structure $\langle N, min, +, +\infty, 0 \rangle$, the c-semiring of natural numbers N where the additive operation is min (which induces the obvious order) and the multiplicative operation is the sum over natural numbers. Notice that in this case the partial order induced by the additive operations (i.e. min) is the inverse of the ordinary total order on natural numbers.

Another example is given by $\langle \wp(\{A\}), \cup, \cap, A, A\} \rangle$ where $\wp(A)$ is the powerset of a set A, and $\cup$ and $\cap$ are the usual set union and intersection operations. Notice that in the latter example also the multiplicative operation is idempotent. When this is the case, additional constraint satisfaction algorithms apply (e.g. local propagation).

KAOS programs, called nets, are the parallel composition of a set of *nodes*. A node is characterized by a unique name, representing its locality, and it is a container of resources (data) and active computational entities (processes). Syntactically, a node is written

$$s ::^{L} P$$

where s is the locality of the node, P is the process running at s, and L is the *network interface* of the node, i.e. a set of *links*. Links (ranged over by l) are pairs either of the form $\langle s, \kappa \rangle$ or of the form $\langle \kappa, s \rangle$, where s is a locality and κ is a cost. Pair $\langle s, \kappa \rangle$ ($\langle \kappa, s \rangle$) represents a link from (to) node s with costs κ. Restriction $(\nu\, s)N$ is a binder for s and is used to express the lexical scope of location s in net N. Intuitively, it is similar to local declarations inside a "block",

namely, s is a variable whose scope is local to N. Differently from declarations of usual programming languages, the scope of s can dynamically change. This is a key feature of name passing process calculi (the well known example is the π-calculus [24]) formally defined as *scope extrusion*.

KAOS processes are built up from the special process $\mathbf{0}$, that does not perform any action, and from a set of basic actions by using action prefixing, parallel composition and recursion. Inter-processes communication is local. The output process $\langle s \rangle$ makes available name s in the local repository. Intuitively, the output operation abstracts the idea of *publishing* a service (a data) into a directory (the repository). The input action (s) withdraws a name from the local repository and uses it to replace the formal name s in the rest of the process; if no name is available, the executing process is blocked. In our analogy, input abstracts the idea of resource *discovery*. The operation for creating a new node $\nu(s \cdot \kappa)$ has the effect of establishing a link with cost attribute κ between the creating node and the fresh node (otherwise, the created node would be unreachable, i.e. completely useless). The only possibly remote operation is $\varepsilon(P)@s$ that provides code mobility. The execution of $\varepsilon(P)@s$ has the effect of sending process P for execution at the node s. Process $P_1 \mid P_2$ is the parallel composition of processes P_1 and P_2, namely P_1 and P_2 are executed concurrently. Process $rec\, X.\, P$ is a recursive process. It is equivalent to execute the process obtained from P once process variable X has been replaced by its definition. Variable X can be renamed without affecting the behaviour of $rec\, X.\, P$. Indeed, if Y does not occur in P, $rec\, Y.\, P[{}^{Y}/_{X}]$ is equivalent to $rec\, X.\, P$ ($[{}^{Y}/_{X}]$ denotes the substitution that replace occurrences of X with Y in P if they are not in the scope of a $rec\,.$ binder).

Actions *login*, *accept* and *disconnect* allow specifying the characteristic of connections among nodes. A login operation sends to the receiver a message detailing the QoS attributes of the required connection. The receiver may accept a login request by performing the accept operation. If a login request is accepted a link with the specified QoS attributes is set up among the two nodes involved. The disconnect operation removes the link with the specified QoS attributes from the network interface of a node.

Names occurring in KAOS processes and nets can be *bound*. More precisely, prefix $(s).P$ binds s in P; this prefix is similar to the λ-abstraction of the λ-calculus. Prefix $\nu(s \cdot \kappa).P$ binds s in P and, similarly, $(\nu\, s)N$ binds s in N. A name that is not bound is called *free*. The sets $\mathrm{bn}(\cdot)$ and $\mathrm{fn}(\cdot)$ (respectively, of bound and free names of a term) are defined in Table 2 (where $\mathrm{fn}(L)$ denotes the names occurring in the set of links L). The set $\mathrm{n}(\cdot)$ of names of a term is the union of its sets of free and bound names. We say that two terms are *α-equivalent*, $\equiv_\alpha$, if one can be obtained from the other by renaming bound names.

Hereafter, we will identify KAOS nets which intuitively represent the same net. We therefore define *structural congruence* $\equiv$, namely an equivalence relation over nets that equates terms denoting the same net and differ only for meaningless syntactic details. Relation $\equiv$ relates nets and relies on a relation $\equiv_p$ which defines structural congruence over processes and is defined as:

γ	fn()	bn()
(s)	$\emptyset$	$\{s\}$
$\nu(s \cdot \kappa)$	$\emptyset$	$\{s\}$
$\overset{\kappa}{\frown} s$	$\{s\}$	$\emptyset$
$s \overset{\kappa}{\smile}$	$\{s\}$	$\emptyset$
$\delta \langle \kappa, s \rangle$	$\{s\}$	$\emptyset$
$\delta \langle s, \kappa \rangle$	$\{s\}$	$\emptyset$

P	fn()	bn()
$\mathbf{0}$	$\emptyset$	$\emptyset$
$\gamma.P$	$\text{fn}(P) \setminus \text{bn}(\gamma) \cup \text{fn}(\gamma)$	$\text{bn}(P) \cup \text{bn}(\gamma)$
$\langle s \rangle$	$\{s\}$	$\emptyset$
$\varepsilon(P)@s$	$\text{fn}(P) \cup \{s\}$	$\text{bn}(P)$
$P_1 \mid P_2$	$\text{fn}(P_1) \cup \text{fn}(P_2)$	$\text{bn}(P_1) \cup \text{bn}(P_2)$
X	$\emptyset$	$\emptyset$
$rec\, X.\, P$	$\text{fn}(P)$	$\text{bn}(P)$

N	fn()	bn()
$s ::^L P$	$\{s\} \cup \text{fn}(L) \cup \text{fn}(P)$	$\text{bn}(P)$
$N_1 \parallel N_2$	$\text{fn}(N_1) \cup \text{fn}(N_2)$	$\text{bn}(N_1) \cup \text{bn}(N_2)$
$(\nu\, s)N$	$\text{fn}(N) \setminus \{s\}$	$\text{bn}(N) \cup \{s\}$

Table 2. Free and bound names

- $P \mid \mathbf{0} \equiv_p P$, for any P;
- $P \mid Q \equiv_p Q \mid P$, for any P and Q;
- $P \mid (Q \mid R) \equiv_p (P \mid Q) \mid R$, for any P, Q and R.

The axioms above state that $(P, \mid, \mathbf{0})$ is a *commutative monoid*.

We can define net structural equivalence as the smallest relation over nets such that

1. $\parallel$ is a commutative and associative;
2. $\dfrac{N \equiv_\alpha N'}{N \equiv N'}$;
3. $\dfrac{P \equiv_p Q}{s ::^L P \equiv s ::^L Q}$.

Notice that $\equiv$ identifies only nets whose equality derives from their syntactical structure and has nothing to do with the semantics of nets (which has still to be introduced and shall rely on structural congruence). With a slight abuse of notation, in the following we write $P \equiv Q$ instead of $P \equiv_p Q$; the context will always clarify whether $\equiv$ is the relation on nets or on processes.

A net $s_1 ::^{L_n} P_1 \parallel ... \parallel s_n ::^{L_n} P_n$ is *well-formed* if, and only if, for any i, j, if $i \neq j$ then $s_i \neq s_j$ and if $\langle \kappa, s_j \rangle \in L_i$ ($\langle s_j, \kappa \rangle \in L_i$) then $\langle s_i, \kappa \rangle \in L_j$ ($\langle \kappa, s_i \rangle \in L_j$). Notice that this definition implies that, in well-formed nets, a connection from s to t costing κ is possible only if two links, $\langle t, \kappa \rangle$ and $\langle \kappa, s \rangle$ are in the network interfaces of nodes s and t, respectively. We shall only consider well-formed nets.

3 An LTS Semantics for KAOS

This section presents the operational semantics of KAOS as a standard labeled transition system semantics. For simplicity, we write $M \cup e$ (resp. $M \setminus e$) instead of $M \cup \{e\}$ (resp. $M \setminus \{e\}$).

The semantics is given in terms of a transition relation that describes possible net evolutions and the corresponding abstract costs (we omit the cost when it is negligible).

Definition 1 ($\xrightarrow[\rho]{\alpha}\!\!\!\!\!\rightarrow$). *The LTS semantics of* KAOS *is the minimal relation* $\longrightarrow \subseteq N \times \langle \alpha, \rho \rangle \times N$ *closed under the inference rules of Tables 3 and 4 and rule*

$$\text{(struct)} \qquad \frac{N_1 \equiv N' \xrightarrow[\rho]{\alpha}\!\!\!\!\!\rightarrow N'' \equiv N_2}{N_1 \xrightarrow[\rho]{\alpha}\!\!\!\!\!\rightarrow N_2}.$$

Labels of transition $\longrightarrow$ *are defined as follows:*

$$\begin{array}{l} \alpha ::= \tau \;\mid\; s \lhd t \;\mid\; s \rhd t \;\mid\; s(\eta,\, P)@t \;\mid\; s \,\varepsilon\, t \;\mid\; X@s \\ \quad \mid\; s \overset{\kappa}{\frown} t \;\mid\; s \overset{\kappa}{\smile} t \;\mid\; \delta(s, \langle t, \kappa \rangle) \;\mid\; \delta(s, \langle \kappa, t \rangle) \end{array}$$

$$\rho ::= \epsilon \;\mid\; s, \kappa$$

In the operational rules we adopt a notational convention borrowed from a similar notation for terms. In particular,

- $\mathrm{bn}(\alpha)$ is equal to η if $\alpha = s(\eta,\, P)@t$, $\emptyset$ otherwise,
- $\mathrm{n}(\alpha)$ ($\mathrm{n}(\rho)$ resp.) denotes the set of all names (free and bound) occurring in α (ρ resp.).

Finally, we will write $\kappa \models T(P)$ to indicate that the behaviour of P is compatible with the cost κ. Intuitively, $T(P)$ represents the *type* or *capabilities* of process P. Like for abstract costs, we intentionally do not specify what $T(P)$ exactly is. For example, it could be determined by using a type system like the one in [10]. From a pragmatic point of view, $T(P)$ is a parameter used to discriminate between processes that can be executed at a site and processes that cannot.

Let us now comment on the semantics. Labels α are used to describe process activities.

- τ describes internal activity.
- $s \lhd t$ ($s \rhd t$) says that a process located at s aims at receiving (sending) a name t.
- $s(\eta,\, P)@t$ says that a process at s intends spawning process P for execution at t. Set η contains the localities occurring in P that must be restricted upon migration (because their scope must also include the target node t).
- $s \,\varepsilon\, t$ says that a process from s is migrated to t.
- $X@s$ says that node s does exist and can accept a process for execution at s; variable X is the placeholder where the migrating process that reaches s will be substituted for.
- $s \overset{\kappa}{\frown} t$ says that a process at s intends to establish a link between s and t with cost κ.
- $s \overset{\kappa}{\smile} t$ says that a process at s accepts the establishment of a link between s and t with cost κ.

(out)	$s ::^L \langle t \rangle \xrightarrow{s \rhd t} s ::^L \mathbf{0}$
(in)	$s ::^L (x).P \xrightarrow{s \lhd t} s ::^L P[^t/_x]$
(leval)	$s ::^{L \cup \langle s,\kappa \rangle} \varepsilon(P)@s \xrightarrow{\tau} s ::^{L \cup \langle s,\kappa \rangle} P, \quad$ if $\kappa \models T(P)$
(eval)	$s ::^L \varepsilon(P)@t \xrightarrow[s,1]{s(\emptyset,\ P)@t} s ::^L \mathbf{0}, \quad$ if $s \neq t$
(new)	$s ::^L (\nu(x \cdot \kappa).P) \mid Q \xrightarrow{\tau} (\nu\, x)(s ::^{L \cup \langle \kappa,x \rangle} P \mid Q \parallel x ::^{\langle s,\kappa \rangle} \mathbf{0})$, if $x \notin \mathrm{n}(L) \cup \{s\} \cup \mathrm{fn}(Q)$
(llogin)	$s ::^{L\ \kappa\frown} s.P \xrightarrow{\tau} s ::^{L \cup \{\langle s,\kappa \rangle, \langle \kappa,s \rangle\}} P$
(login)	$s ::^{L\ \kappa\frown} t.P \xrightarrow{s\ \kappa\frown\ t} s ::^{L \cup \langle \kappa,t \rangle} P, \quad$ if $s \neq t$
(accept)	$s ::^L t\ \overset{\kappa'}{\smile}.P \xrightarrow{t\ \overset{\kappa}{\smile}\ s} s ::^{L \cup \langle t,\kappa \rangle} P, \quad$ if $\kappa \leq \kappa'$
(ldisc)	$s ::^L \delta\langle s,\kappa \rangle.P \xrightarrow{\tau} s ::^{L \setminus \langle s,\kappa \rangle \setminus \langle \kappa.s \rangle} P$
(idisc)	$s ::^L \delta\,\langle t,\kappa \rangle.P \xrightarrow{\delta(s,\langle t,\kappa \rangle)} s ::^{L \setminus \langle t,\kappa \rangle} P, \quad$ if $t \neq s$
(odisc)	$s ::^L \delta\,\langle \kappa,t \rangle.P \xrightarrow{\delta(s,\langle \kappa,t \rangle)} s ::^{L \setminus \langle \kappa,t \rangle} P, \quad$ if $t \neq s$
(node)	$s ::^{L \cup \langle r,\kappa \rangle} P \xrightarrow[r,\kappa]{X@s} s ::^{L \cup \langle r,\kappa \rangle} P \mid X, \quad$ if X fresh
(rec)	$s ::^L rec\ X.P \xrightarrow{\tau} s ::^L P[^{rec\ X.P}/_X]$

Table 3. Axioms of KAOS interactive semantics

– $\delta(s, \langle t, \kappa \rangle)$ ($\delta(s, \langle \kappa, t \rangle)$) says that a process running at s wants to remove the link $\langle t, \kappa \rangle$ ($\langle \kappa, t \rangle$) from the network interface of node s.

Labels ρ can be either the empty label ϵ (that carries no information and will be omitted) or the pair $\langle s, \kappa \rangle$ which point out that s is used as gateway for a remote action at node s and κ is the cost of the path from the source node to s.

Axioms in Table 3 describe process activities. The informal semantics has been explained in the previous section. Here we only add a few comments.

– Axiom (in) gives an "early" flavor to the semantics (in the sense of the π-calculus [24]).
– Axioms (leval) and (eval) deal with process spawning. The first one says that local evaluation is authorized only if the type of the process is compatible

with the cost of performing a local spawning. The second axiom accounts for remote evaluation.
- Axioms (llogin) and (login) deal with the establishment of a new link. A local link can always be established, while a remote one needs the authorization of the target node. In both cases, the node network interface is enriched with an outgoing link.
- Axiom (accept) says that any connection request having a cost κ less than κ' can be accepted. The node network interface is enriched with an incoming link.
- Axioms (ldisc), (idisc) and (odisc) handle requests for removing local, incoming and outgoing links, respectively. Removing local links simply updates the network interface of a node, whereas, removing a link from/to a remote node t requires to signal node t that the corresponding link must be removed from its network interface (see rule (remlink) in Table 4).
- Axiom (node) says that a node s dynamically creates an execution context named (by the process variable) X where a (possibly remote) process can be placed. Other than X, the transition label contains the locality r of a node that can be used as an intermediate node (a sort of *gateway*) to reach s (because there exists a link from r to s).

Rules in Table 4 coordinate the behaviour of processes in a net. Most of the rules (e.g. (par1), (par2) and (res)) are standard. Here, we comment on a few peculiarities.

- Rule (com) says that communication is always local and asynchronous.
- Rule (connect) says that in order to establish a link, a synchronization must occur between the requiring process and the accepting one.
- Rule (remlink) says that to remove a link, a synchronization between the two nodes connected by the link is necessary in order to guarantee that the network interfaces of these two nodes will be updated coherently. Notice that this synchronization occurs provided that the link actually exists.
- Scope extrusion of bound names that are exported by migrating processes is implemented by rules (open) and (close) in the style of the π-calculus [24].
- Rules (route) and (close) check step-by-step the existence of a path of links from the node s (performing the remote spawning of process P) to the target node t. In particular, the first premise of these rules checks the existence of a path from s to an intermediate gateway r'. The second premise checks the existence of a link starting from r'. If both checks succeed, and if the type of P complies with the cost of each link along the path, the path can be extended by including the link from r' (which is used as an intermediate node). Additionally, in rule (route) the obtained path connects s to an $r \neq t$ (see the reduction labels in the conclusion of the rule), hence the free context X of r is not used to execute P and must be removed. This is done by simply plugging process $\mathbf{0}$ inside X and by exploiting the fact that $Q \mid \mathbf{0} \equiv \mathbf{0}$ and rule (struct). On the other hand, rule (close) obtains a path from s to t, hence P is now plugged in the execution context X of t (i.e. P is sent for execution

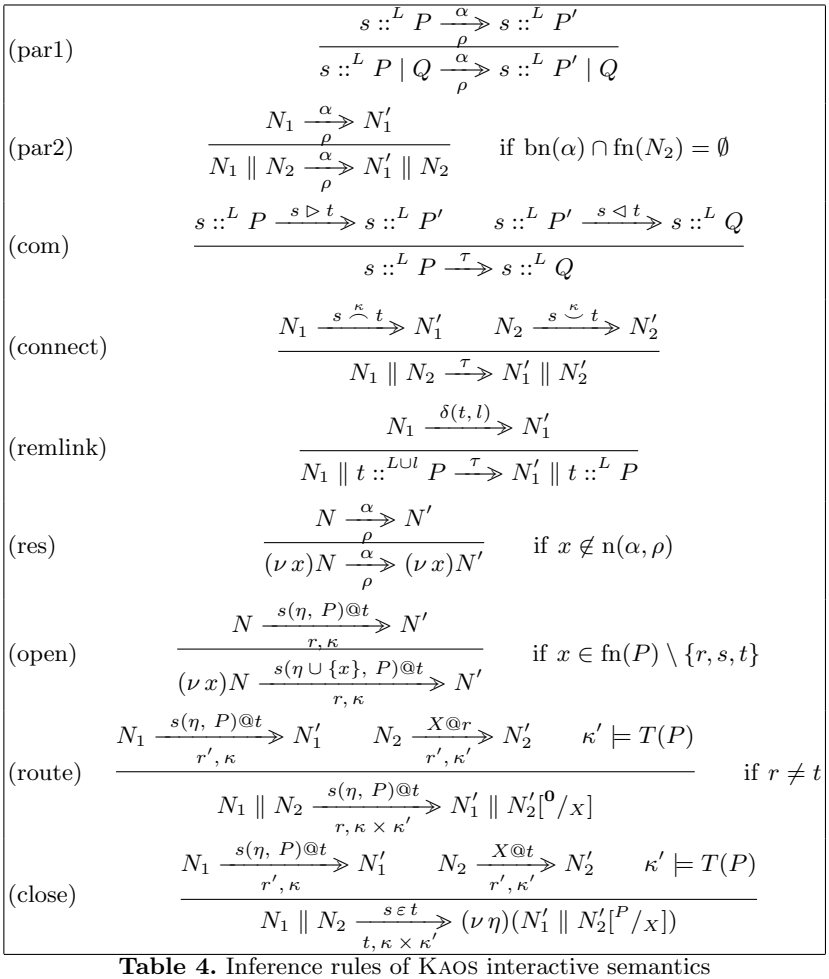


Table 4. Inference rules of KAOS interactive semantics

at t). Moreover, the restrictions over the bound names carried along with P are restored to extend the scope of these names to the resulting net as a whole. An example of use of these two rules is given in Figure 3.

4 A KAOS Application

This section shows how KAOS can be used to model QoS requirements at application level. The scenario is a messaging application. A group of nodes are connected and have to exchange messages (or files). When a message must be sent from node s to node t a notification message is first sent to t. At this point, node t spawns an agent on s that will examine the message and, if all checks are satisfied, the message is effectively moved on t. Intuitively, the agent acts as a "filter" that can be programmed to avoid useless messages to roam the net.

This mechanism also enhances network performances when messages are much greater than notifications and agents. The above message exchange protocol is depicted in Figure 1 where the bold line represents the fact that an agent is spawned from t to s and the dashed line represents the fact that the message is downloaded only if the checking phase is passed.

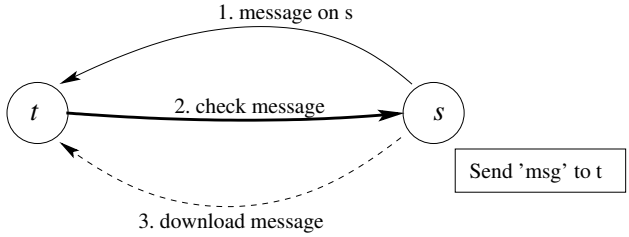


Fig. 1. The filter protocol between s and t

Nodes s and t may not be directly connected but can be connected by means of (a number of) links passing through a number of intermediate nodes that are used for forwarding messages between t and s. We assume that the attributes of a link between r and r' that can affect our messaging application could be

1. the geographical distance between r and r';
2. the capabilities granted to processes executed at r' from r (at r from r');
3. the price of the connection.

Figure 2 depicts a possible way of connecting s and t and the costs associated to the links. Lines which are arrowed at both extremities, see e.g. the line between t and x, represent two links, one from t to x and another from x to t, both links have the same cost (e.g. $\langle 10, \{i, o\}, 2\rangle$). Notice that, in the net of Figure 2, t is connected to s via two paths (the path $t - x - z - s$ and the path $t - y - z - s$), while only one path goes from s to t.

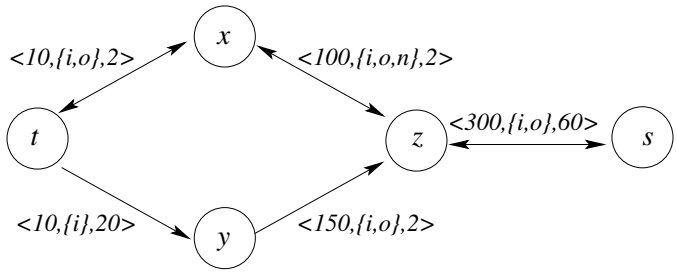


Fig. 2. A net connecting s and t

We now show how KAOS primitives can be exploited both for declaring the network connections and for programing the described messaging application.

Costs First we describe costs as triples $\kappa = \langle d, C, p\rangle$ where

1. d is the geographical distance (in Km);
2. $C \subseteq \{i, o, n\}$ are the capabilities, where i, o and n stand for $i$nput, $o$utput and creation of $n$ew nodes, respectively;
3. p is the price (in euros).

All the components of costs are elements of the following c-semirings, respectively

1. $(N, min, +, +\infty, 0)$,
2. $(\wp(\{i, o, n\}), glb, \cap, \{i, o, n\}, \{i, o, n\})$,
3. $(Q, min, +, +\infty, 0)$.

In [3] it has been proved that the cartesian product of c-semirings is a c-semiring as well. Therefore, we can define operations $\times$ and $+$ of the cost c-semiring as

$$\langle d, C, p\rangle \times \langle d', C', p'\rangle = \langle d + d', C \cap C', p + p'\rangle$$
$$\langle d, C, p\rangle + \langle d', C', p'\rangle = \langle min\{d, d'\}, glb\{C, C'\}, \min\{p, p'\}\rangle.$$

Accordingly, the neutral elements of $\times$ and $+$, respectively are defined as $1 = \langle 0, \{i, o, n\}, 0\rangle$ and $0 = \langle +\infty, \emptyset, +\infty\rangle$.

In the following, we use the convention that κ_{uv} denotes the cost of the link from u to v. In Figure 2, for instance, κ_{xt} is $\langle 10, \{i, o\}, 2\rangle$. We also avoid writing trailing occurrences of $\mathbf{0}$, hence $\gamma.\mathbf{0}$ will be abbreviated as γ.

Connections In order to establish connections among nodes t, x, y, z and s corresponding to Figure 2, we can start with the following KAOS net:

$$t ::^{\emptyset} P_t \parallel x ::^{\emptyset} P_x \parallel y ::^{\emptyset} P_y \parallel z ::^{\emptyset} P_z \parallel s ::^{\emptyset} P_s$$

where

$$P_t \triangleq x \overset{\kappa_{xt}}{\smile} \mid \overset{\kappa_{tx}}{\frown} x \mid \overset{\kappa_{ty}}{\frown} y$$
$$P_x \triangleq t \overset{\kappa_{tx}}{\smile} \mid z \overset{\kappa_{zx}}{\smile} \mid \overset{\kappa_{xt}}{\frown} t \mid \overset{\kappa_{xz}}{\frown} z$$
$$P_y \triangleq t \overset{\kappa_{ty}}{\smile} \mid \overset{\kappa_{yz}}{\frown} z$$
$$P_z \triangleq x \overset{\kappa_{xz}}{\smile} \mid y \overset{\kappa_{yz}}{\smile} \mid s \overset{\kappa_{sz}}{\smile} \mid \overset{\kappa_{zx}}{\frown} x \mid \overset{\kappa_{zs}}{\frown} s$$
$$P_s \triangleq z \overset{\kappa_{zs}}{\smile} \mid \overset{\kappa_{sz}}{\frown} z.$$

According to the semantics of KAOS, after all executions of login and accept actions, the network interfaces of the nodes in the resulting net will correspond

to the graph of Figure 2 and the KAOS net obtained is the following:

$$\begin{aligned}
&t ::^{\{\langle x,\kappa_{xt}\rangle,\langle\kappa_{tx},x\rangle,\langle\kappa_{ty},y\rangle\}}\ \mathbf{0}\ \|\\
&x ::^{\{\langle t,\kappa_{tx}\rangle,\langle z,\kappa_{zx}\rangle,\langle\kappa_{xt},t\rangle,\langle\kappa_{xz},z\rangle\}}\ \mathbf{0}\ \|\\
&y ::^{\{\langle t,\kappa_{ty}\rangle,\langle\kappa_{yz},z\rangle\}}\ \mathbf{0}\ \| \qquad (1)\\
&z ::^{\{\langle x,\kappa_{xz}\rangle,\langle y,\kappa_{yz}\rangle,\langle s,\kappa_{sz}\rangle,\langle\kappa_{zx},x\rangle,\langle\kappa_{zs},s\rangle\}}\ \mathbf{0}\ \|\\
&s ::^{\{\langle z,\kappa_{zs}\rangle,\langle\kappa_{sz},z\rangle\}}\ \mathbf{0}.
\end{aligned}$$

Capabilities In order to consider the intentions of remotely evaluated processes it is necessary to define $T(P)$. Given a process P, we define $T(P)$ as follows

$$T(P) = \begin{cases} \emptyset, & \text{if } P = \mathbf{0} \vee P = \varepsilon(Q)@t \vee P = X \\ \{o\}, & \text{if } P = \langle t\rangle \\ T(P_1) \cup T(P_2), & \text{if } P = P_1 \mid P_2 \\ T(Q), & \text{if } P = rec\ X.Q \\ T(\gamma) \cup T(Q), & \text{if } P = \gamma.Q \end{cases}$$

where

$$T(\gamma) = \begin{cases} \{i\}, & \text{if } \gamma = (x) \\ \{n\}, & \text{if } \gamma = \nu(x \cdot \kappa) \\ \emptyset, & \text{otherwise.} \end{cases}$$

Capabilities of processes that are arguments of spawning actions performed by P are not part of the type of P; instead, they will be looked at when the spawning actions are effectively executed (see rules (route) and (close) in Table 4).

We can now instantiate relation $\models$ to costs and process types defined in this section. Given a process P, we say that a cost $\langle d, C, e\rangle$ *satisfies* $T(P)$ (written $\langle d, C, e\rangle \models T(P)$) if, and only if, $T(P) \subseteq C$. This interpretation states that a remote evaluation of a process P can traverse a link if all the capabilities that P might exercise occur in the cost of the link.

Sender, receiver and filter processes The messaging application can be defined by means of a sender process executed at t, a receiver process executed at s and the agent that filters the file which migrates from t to s and provides the result to t. This three rôles can be formalized in KAOS[1] as follows:

$$\begin{aligned}
S &\triangleq \varepsilon(\langle \text{“}bigfile'', s\rangle)@t \mid \langle \text{“}bigfile'', file\rangle\\
R &\triangleq (f,u).\varepsilon(F_{f,t})@u.(res)...\\
F_{f,t} &\triangleq (f,v).\mathbf{if}\ test(v)\ \mathbf{then}\ \varepsilon(\langle v\rangle)@t\ \mathbf{else}\ \varepsilon(\langle no\rangle)@t
\end{aligned}$$

[1] In this example we use tuples, ground values as string or files, the boolean function *test* and the if-then-else construct. In the formal definition of KAOS we chose not to have many constructs and types in order to give a smoother definition of the calculus. It is, of course, straightforward to extend KAOS to all programming constructs used in our running example.

Process S (allocated at s) notifies to R (allocated at t) that a "big file" is available at s (spawning action of S). When R acquires the notification, the filter F is spawned at u and the result of its check is waited for. Agent F, once in execution, accesses the file, and tests whether it must be sent to t or not. In the first case, the file is effectively sent to t while, in the second case, a conventional signal *no* is sent.

If S and R are respectively allocated at nodes s and t of net (1) we can detail how KAOS semantics can deal with the remote operations and costs of connections. In order to avoid cumbersome replica of link costs, in the following we use L_u to denote the network interface of node u. Hence, we consider the initial configuration as given in net (1)

$$t ::^{L_t} R \quad \| \quad x ::^{L_x} \mathbf{0} \quad \| \quad y ::^{L_y} \mathbf{0} \quad \| \quad z ::^{L_z} \mathbf{0} \quad \| \quad s ::^{L_s} S \tag{2}$$

and show how KAOS semantics can determine a path connecting t and s and, for each path, its total cost. For instance, let us consider the spawning action of S. By definition $T = T(\langle \text{“}bigfile'', s\rangle) = \{o\}$.

$$\underbrace{\dfrac{s ::^{L_s} \varepsilon(\langle \text{“}bigfile'', s\rangle)@t \xrightarrow[s,1]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \mathbf{0}}{s ::^{L_s} S \xrightarrow[s,1]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \langle \text{“}bigfile'', file\rangle}}_{eval-s-z}$$

$$\underbrace{\dfrac{eval - s - z \quad z ::^{L_z} \mathbf{0} \xrightarrow[s,\kappa_{sz}]{X@z} z ::^{L_z} X \quad \kappa_{sz} \models T}{s ::^{L_s} S \parallel z ::^{L_z} \mathbf{0} \xrightarrow[z,1\times\kappa_{sz}]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} s ::^{L_s} \langle \text{“}bigfile'', file\rangle \parallel z ::^{L_z} \mathbf{0}}}_{path-s-z}$$

$$\underbrace{\dfrac{path - s - z \qquad x ::^{L_x} \mathbf{0} \xrightarrow[z,\kappa_{zx}]{Y@x} x ::^{L_x} Y \qquad \kappa_{zx} \models T}{N_1 \xrightarrow[x,1\times\kappa_{sz}\times\kappa_{zx}]{s(\emptyset,\ \langle \text{“}bigfile'', s\rangle)@t} N_2}}_{netpath-s-z}$$

$$\dfrac{netpath - s - z \quad t ::^{L_t} R \xrightarrow[x,\kappa_{xt}]{Z@t} t ::^{L_t} R \mid Z \qquad \kappa_{xt} \models T}{N_1 \parallel t ::^{L_t} R \xrightarrow[t,1\times\kappa_{sz}\times\kappa_{zx}\times\kappa_{xt}]{s\,\varepsilon\,t} N_2 \parallel t ::^{L_t} R \mid \langle \text{“}bigfile'', s\rangle}$$

Fig. 3. Proving a net reduction

An example of inference proof that uses the KAOS semantics rules can be found in Figure 3 where, for the sake of readability, we use the following shorthands: N_1 stands for the net $s ::^{L_s} S \parallel z ::^{L_z} \mathbf{0} \parallel x ::^{L_x} \mathbf{0}$ and N_2 for the net

$s ::^{L_s} \langle \text{“}bigfile'', file\rangle \parallel z ::^{L_z} \mathbf{0} \parallel x ::^{L_x} \mathbf{0}$. The first inference is an application of rule (par1) that uses axiom (eval) as premise. The second inference, uses the conclusion of the first one (that we call $eval - s - z$) and axiom (node) for the premises, and then proceeds by first applying (route) and (struct). The resulting net reduction (that we call $path - s - z$ because determines a path from s to z) is then used as the first premise of the third inference proof. This inference also uses axiom (node) as premise and proceeds by applying (route) and (struct). Finally, the resulting net reduction (that we call $netpath - s - z$) and axiom (node) are the premises of the last inference that concludes by applying rule (close).

5 A Calculus of Graphs

Graph-based techniques can be usefully adopted for modeling inter-networking systems. Indeed, *(hyper)edges* (namely, edges that connect more than two vertices) can be used to represent components, while vertices model the network environment of components. Edges sharing a vertex means that the corresponding components may interact by exploiting network communication infrastructure. Structured versions of graphs (typed graphs, term graphs, hierarchical graphs) can precisely model complex inter-network configurations [14, 15] and access control policies [18–20].

Graph synchronization adds to network awareness the ability of dealing with the temporal dimension of computations. Graphs synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that local rewritings depends on the outcome of a (possibly global) constraint satisfaction algorithm.

Graph rewriting based on edge replacement and synchronization was introduced in [8, 11] and related to distributed constraint satisfaction problems in [25]. The version with mobility, which employs a notation based on logical judgments and inference rules, was introduced recently in [14] and extended in [15] to encode the π-calculus. Abstract semantics based on bisimilarity has been discussed in [21].

Next sections introduce syntax and semantics of graphs.

5.1 Syntax of Graphs

We assume that $\mathcal{V}$ is an ordered set of vertices. An *edge*, is an atomic item with a label from a ranked alphabet $\mathcal{L}$. We write $L(x_1, ..., x_n)$ to indicate an edge labeled L connecting vertices x_1,..., x_n. In this case, we say that L has rank n (written as $L : n$) and that $x_1, ..., x_n$ are the *attachment vertices* (or *attachment points*) of L. Figure 5.1 represents edge $L(a, b, c)$ where $L : 3$. Wires connecting vertices a, b and c to L are called tentacles. *Graphs* are built from ranked edges in $\mathcal{L}$ and *vertices* in $\mathcal{V}$. Moreover, we write $L(\boldsymbol{x})$ with the implicit assumption that L has rank $|\boldsymbol{x}|$, namely the length of vector $\boldsymbol{x}$.

We can now define graphs.

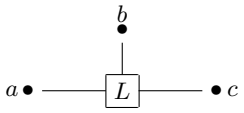


Fig. 4. An edge

Definition 2 (Syntactic judgments). *A* (hyper)graph *is a* syntactic sequent *of the form* $\Gamma \vdash G$*, where*

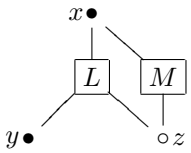
- $\Gamma \subseteq \mathcal{V}$ *is a finite set of vertices called* external vertices*, and*
- G *is one of the terms generated by the following grammar, where* $y \in \mathcal{V}$ *and* $L : |\boldsymbol{x}|$ *is an edge:*

$$G ::= L(\boldsymbol{x}) \mid G \mid G \mid \nu\, y.G \mid nil.$$

We call terms G graph terms*.*

The productions in Definition 2 permits generating single edges ($L(\boldsymbol{x})$), composing terms in parallel ($G \mid G$) and hiding vertices ($\nu\, y.G$). Graph *nil* represents the empty graph. We will use $\mathrm{fv}(G)$ to denote the set of the vertices of G which does not occur in the scope of a ν operator. Hereafter, we omit the curly brackets in judgments and write $x_1, ..., x_n \vdash G$ instead of $\{x_1, ..., x_n\} \vdash G$; moreover, we will often use $\boldsymbol{x} \vdash G$ instead of $x_1, ..., x_n \vdash G$, if $\boldsymbol{x} = (x_1, ..., x_n)$, . We use the notation Γ, x instead of $\Gamma \cup \{x\}$ with the implicit assumption that $x \notin \Gamma$; similarly, we write Γ_1, Γ_2 instead of $\Gamma_1 \cup \Gamma_2$ assuming that $\Gamma_1 \cap \Gamma_2 = \emptyset$.

Example 2. Let us consider the judgment $x, y \vdash \nu\, z.(L(y, z, x) \mid M(x, z))$, where $L : 3$ and $M : 2$; a graphical representation of the judgment is



where filled circles and empty circles are used for representing free and restricted vertices, respectively.

Definition 3 gives the structural congruence rules for graph terms. We take advantage of such congruence to avoid writing cumbersome parenthesis.

Definition 3 (Structural Congruence). *The* structural congruence *is the smallest binary relation* $\equiv$ *over graph terms that obeys axioms in Table 5.*

Axioms (*AG1*), (*AG2*) and (*AG3*) define associativity, commutativity and identity over *nil* for operation $|$, respectively. Axioms (*AG4*) and (*AG5*) state that the vertices of a graph can be restricted in any order and that restriction does not play any rôle on non-free vertices of a graph, respectively. Axiom

$(AG1)$	$(G_1 \mid G_2) \mid G_3 \equiv G_1 \mid (G_2 \mid G_3)$	
$(AG2)$	$G_1 \mid G_2 \equiv G_2 \mid G_1$	
$(AG3)$	$G \mid nil \equiv G$	
$(AG4)$	$\nu\ x.\nu\ y.G \equiv \nu\ y.\nu\ x.G$	
$(AG5)$	$\nu\ x.G \equiv G,$	if $x \notin \mathrm{fv}(G)$
$(AG6)$	$\nu\ x.G \equiv \nu\ y.G\{y/x\},$	if $y \notin \mathrm{fv}(G)$
$(AG7)$	$\nu\ x.(G_1 \mid G_2) \equiv (\nu\ x.G_1) \mid G_2,$	if $x \notin \mathrm{fv}(G_2)$

Table 5. Graphs structural congruence rules

$(AG6)$ deals with alpha conversion of hidden bound vertices, while $(AG7)$ tunes the interplay between hiding and the operator for parallel composition. Occasionally, taking advantage of axiom $(AG4)$, we will write $\nu\ X$, with $X = \bigcup x_i$, to abbreviate $\nu\ x_1.\nu\ x_2 \ldots \nu\ x_n$.

We will work with *well-formed judgments*.

Definition 4 (Well-Formed Judgments). *A judgment is* well-formed *if it is generated by applying the rules in Table 6 up to structural congruence.*

$$\frac{}{x_1, \ldots, x_n \vdash nil}\ (RG1) \qquad \frac{\Gamma \vdash G_1 \qquad \Gamma \vdash G_2}{\Gamma \vdash G_1 \mid G_2}\ (RG3)$$

$$\frac{L : m \qquad y_1, \ldots, y_m \in \{x_1, \ldots, x_n\}}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}\ (RG2) \qquad \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu\ x.G}\ (RG4)$$

Table 6. Well-formed judgments

Rule $(RG1)$ states that graphs with n isolated vertices and no edges are well-formed. Rule $(RG2)$ states that a graph with n vertices and one edge L (having sort m) whose tentacles are all connected to vertices of the graph is well-formed. Notice that $(RG2)$ does not make any assumption on n and m, hence it could be the case that $m \geq n$. This implies that some tentacles of L can be connected to the same vertex. Rule $(RG3)$ allows one to put together (using $|$) two well-formed judgments that share the same set of external vertices. Finally, rule $(RG4)$ permits hiding a vertex from the environment.

The correspondence theorem expressing that well-formed judgments up to structural axioms are isomorphic to graphs up to isomorphism has been proved in [15].

5.2 Graph Rewritings

We propose a new rewriting mechanism for graphs that permits interconnection modification and relies on edge replacement. The idea is that an edge can be rewritten if the *constraints* it imposes on its external vertices accomplish with constraints of all other edges connected to such vertices. What here is meant for "constraint" is a pair consisting of a label and a tuple of vertices. The label in a constraint is an action of a *synchronization algebra*. A synchronization algebra is a structure $(Act,\ S : Act \times Act \rightarrow Act)$ where Act is a set of actions and S is a binary function on Act. If $a = S(a_1, a_2)$ then we say that the a is the action obtained by synchronizing a_1 and a_2.

Observation 1 *The principal and well studied synchronization algebras are* á la *Milner (CCS [23], where* $S(a, \bar{a}) = \tau$*), and* á la *Hoare (CSP [16], where* $S(a, a) = a$*) synchronizations [27]. The former takes two partner to synchronize trough complementary actions, while the latter requires that all participants of a synchronization perform the same action. Hereafter, we consider synchronizations* á la *Milner.*

As will be clarified later, constraints are used to "select" which adjacent edges in a graph must be replaced in a graph rewriting.

Figure 5 aims at giving a graphical intuition of edge replacement. Edge L

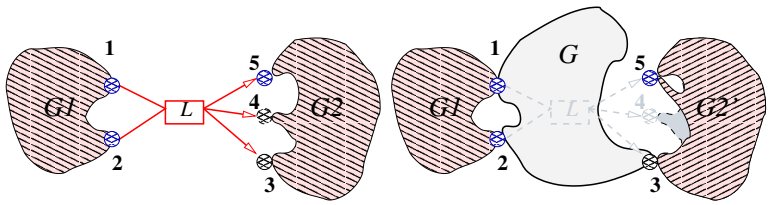


Fig. 5. Hyperedge replacement

in Figure 5(a) is connected to graphs $G1$ and $G2$, indeed external vertices **1** and **2** are attachment points of both L and $G1$, while vertices **3**, **4** and **5** are attachment points of both L and $G2$.

Figure 5(b) represents the graph obtained by replacing edge L with graph G. The dashed gray part of the Figure 5(b) represents the initial situation which disappears after the transition has taken place.

The main things to remark are that (i) $G1$ is not modified after the transition; (ii) all vertices in G different from **1** and **2** are new vertices generated by the transition; (iii) some vertices can be "fused" after the transition as **4** and **5** in Figure 5(b). As will be shown later, this accounts for *mobility* of components, that dynamically can change their connections.

5.3 Productions

A *graph rewriting system*, $\mathcal{G} = \langle \Gamma_0 \vdash G_0, \mathcal{P} \rangle$, consists of an initial graph $\Gamma_0 \vdash G_0$ and a set of *productions*:

Definition 5 (Production). *Let $X \subseteq \mathcal{V}$ be the set $\{x_1, \ldots, x_n\}$ and L be an edge label with rank n. A* production *is a transition of the form*

$$X \vdash L(x_1, \ldots, x_n) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G, \tag{3}$$

where

- *function $\pi : X \to X$ is a* fusion substitution*;*
- *$\Lambda \subseteq X \times Act \times \mathcal{V}^*$ is a* set of constraints*;*
- *$\Gamma = \pi(X) \cup (\mathrm{v}(\Lambda) \setminus X)$;*
- *$\mathrm{fv}(G) \subseteq \Gamma$.*

Production (3) specifies the constraints that the environment must satisfy in order to replace $L(\boldsymbol{x})$ with G. Such constraints are imposed by Λ on the set X of external vertices of L. Once constraints in Λ are satisfied, vertices must be coalesced according to fusion substitution π. Λ and π are detailed below.

Function π is a fusion substitution if

$$\forall x_i, x_j \in X. \pi(x_i) = x_j \Rightarrow \pi(x_j) = x_j,$$

namely π induces an equivalence relation partition $\simeq_\pi$ over X defined as $x \simeq_\pi x' \stackrel{\text{def}}{\iff} \pi(x) = \pi(x')$. Equivalence $\simeq_\pi$ partitions X into equivalence classes where each vertex $x \in X$ is mapped to a *representative element* $\pi(x)$.

Λ associates actions in Act and sequences of vertices to (some of the) external vertices of L. Λ is the graph relation of a partial function with (finite) domain X and codomain in $Act \times \mathcal{V}^*$. Given Λ, we indicate the set of constraints of a vertex x with $\Lambda(x)$. If $(x, a, \boldsymbol{y}) \in \Lambda$ then L can synchronize with edges in the environment that have a tentacle connected to x and satisfy condition a (that will depend on the chosen synchronization algebra). Intuitively, in order to perform a transition, all conditions on external vertices must be in accordance with the synchronization policy. Thus, actions in Act constraint the possible synchronizations among connected edges.

Let us again consider constraint $(x, a, \boldsymbol{y})$[2]; vector $\boldsymbol{y}$ contains the vertices of the constraint; we let $\mathrm{v}(\Lambda)$ denote the union of the vertices of the constraints

[2] We assume that any label $a \in Act$ has an *arity*; we let $|\cdot| : Act \to \omega$ be the arity function on Act. Arities of actions are needed to maintain consistent constraints on vertices. More precisely $|a| = |\boldsymbol{y}|$, for each constraint $(x,\ a,\ \boldsymbol{y})$.

in Λ. Vector $\boldsymbol{y}$ either contains vertices that appear in X or new vertices that will be present in $\Gamma \vdash G$. We impose a further condition on productions, indeed we require that $\mathrm{v}(\Lambda) \cap X \subseteq \pi(X)$; namely, the external vertices used in the synchronization must be representative elements according to $\simeq_\pi$.

Let us now consider the structure of the right hand side of judgment (4). Γ consists of the vertices which are image of $x_1, \ldots, x_n$ through π and the new vertices used in the synchronization, namely those vertices that appear in Λ and are not in X. In general, G may be any graph provided that $\mathrm{fv}(G) \subseteq \Gamma$.

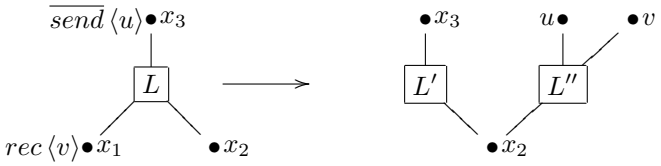
Synchronized edge replacement is obtained by graph rewriting combined with constraint solving. More specifically, we use *context-free* productions labeled with actions useful for coordinating the simultaneous application of two or more productions. Coordinated rewriting allows the propagation of synchronization all over the graph where productions are applied. Determining the productions to synchronize at a given stage corresponds to solving a distributed constraint satisfaction problem [25].

Example 3. Referring to Example 2, let us assume that the following production is given:

$$x_1, x_2, x_3 \vdash L(x_1, x_2, x_3) \xrightarrow[[x_2/x_1]]{\{(x_3,\ \overline{send},\ \langle u\rangle),\ (x_1,\ rec,\ \langle v\rangle)\}} x_2, x_3, u, v \vdash L'(x_3, x_2) \mid L''(u, x_2, v).$$

The above production states that, once constraints on vertices x_1 and x_3 are satisfied by the environment, edge L is replaced by two edges: L' and L''. L' has tentacles to vertices x_2 and x_3, while L'' is connected to x_3 and to two newly generated vertices u and v. Fusion substitution $[x_2/x_1]$ represents the mapping $\begin{cases} x_1 \mapsto x_1 \\ x_2 \mapsto x_1 \\ x_3 \mapsto x_3 \end{cases}$ and determines the partition $\{\{x_1, x_2\}, \{x_3\}\}$, where x_2 is the representative element of $\{x_1, x_2\}$.

The production can be graphically represented as follows:



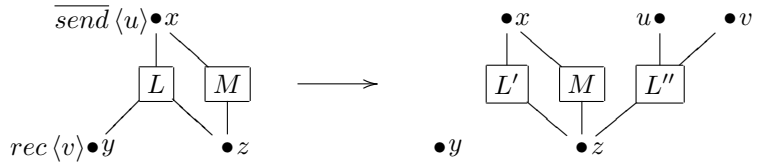
5.4 Edge Replacements

A production rewrites a single edge into an arbitrary graph. Before giving the formal definition of edge replacement we describe an intuitive procedure that can be naively regarded as a procedural way of obtaining edge replacement.

A production $p : L \to R$ can be applied to a graph G yielding H if there exists an occurrence of an edge labeled by L in G. Graph H is obtained from G by

1. removing the occurrence of L,
2. embedding a fresh copy of R in G and
3. coalescing external vertices of R with the corresponding attachment vertices of the occurrence of edge L.

Example 4. If we apply the production defined in Example 3 to the graph of Example 2, i.e. to the judgment $x, y, z \vdash L(y, z, x) \mid M(x, z)$, we obtain the following graph rewriting:



As already stated, it is not mandatory that *all* edges take place in replacements, namely, some components can remain *idle* while others are replaced.

Graphs over edge labels $\mathcal{L}$ and vertices $\mathcal{V}$ obey the usual structural congruence axioms in the same style of Section 2; in particular, given a production

$$\boldsymbol{x} \vdash L(\boldsymbol{x}) \xrightarrow[\pi]{\Lambda} \Gamma \vdash G,$$

renaming can be applied in several ways:

i. external vertices of $\boldsymbol{x}$ can be changed throughout the judgment;
ii. vertices declared in $\mathrm{v}(\Lambda) - \Gamma$ can be α-converted;
iii. the representative vertices chosen by π can be consistently changed.

5.5 Transitions of Graphs

Productions are synchronized via the inference rules in Table 7. Graph semantics is based on productions to specify edge replacement, while inference rules essentially synchronize productions and confer dynamic behaviour to graphs.

A transition is a logical judgment

$$\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda} \Gamma_2 \vdash G_2 \tag{4}$$

where Λ, π, Γ_2 and G_2 obeys the same conditions imposed on productions. Essentially, transitions can be seen as productions having general graphs on their left hand side. Hence transitions describe the dynamic evolutions of graphs.

Transition (4) states that $\Gamma_1 \vdash G_1$ can take part to rewritings that match constraints Λ and determine fusion substitution π. Once such conditions are satisfied, $\Gamma_1 \vdash G_1$ rewrites as $\Gamma_2 \vdash G_2$.

Definition 6 (Graph transitions). *Let $\langle \Gamma_0 \vdash G_0, \mathcal{P}\rangle$ be a graph rewriting system. The set of* transitions *$T(\mathcal{P})$ is the smallest set that contains $\mathcal{P}$ and that is closed under the four inference rules in Table 7.*

$$(merge1)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) = \emptyset \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = [^{\pi(x)}/_{\pi(y)}]}{\Gamma \vdash [^x/_y]G \xrightarrow[(\pi;\rho)_{-y}]{\rho\Lambda}\!\!\gg \mathrm{v}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \vdash \rho G'}$$

$$(merge2)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda \cup \{(x,a,\boldsymbol{v}),(y,\overline{a},\boldsymbol{w})\}}\!\!\gg \Gamma' \vdash G' \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad \rho = mgu\{[^{[^x/_y]\boldsymbol{w}}/_{[^x/_y]\boldsymbol{v}}], [^{\pi(x)}/_{\pi(y)}]\} \qquad \Gamma'' = \mathrm{v}(\rho\Lambda) \cup (\pi;\rho)_{-y}(\Gamma) \qquad U = \rho(\Gamma') \setminus \Gamma''}{\Gamma \vdash [^x/_y]G \xrightarrow[(\pi;\rho)_{-y}]{(\rho\Lambda \cup (x,\tau,\langle\rangle))}\!\!\gg \Gamma'' \vdash \nu\, U.\rho G'}$$

$$(res)\quad \frac{\Gamma, y \vdash G \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma' \vdash G' \qquad \Lambda(y) = \emptyset \vee \Lambda(y) = \{(y,\tau,\langle\rangle)\} \qquad x \simeq_\pi y \Rightarrow y \neq \pi(y) \qquad U = \Gamma' \setminus (\mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma))}{\Gamma \vdash \nu\, y.G \xrightarrow[\pi_{-y}]{\Lambda \setminus (y,\tau,\langle\rangle)}\!\!\gg \mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma) \vdash \nu\, U.G'}$$

$$(par)\quad \frac{\Gamma_1 \vdash G_1 \xrightarrow[\pi]{\Lambda}\!\!\gg \Gamma_2 \vdash G_2 \qquad \Gamma_1' \vdash G_1' \xrightarrow[\pi']{\Lambda'}\!\!\gg \Gamma_2' \vdash G_2' \qquad \Gamma_1 \cap \Gamma_1' = \emptyset}{\Gamma_1 \cup \Gamma_1' \vdash G_1 \mid G_1' \xrightarrow[\pi \cup \pi']{\Lambda \cup \Lambda'}\!\!\gg \Gamma_2 \cup \Gamma_2' \vdash G_2 \mid G_2'}$$

Table 7. Inference rules for graph synchronization

A derivation is obtained by starting from the initial graph and by executing a sequence of transitions, each obtained by synchronizing productions. The synchronization of rewriting rules requires matching of the actions and unification of the third components of the constraints Λ. After productions are applied, the unification function is used to obtain the final graph by merging the corresponding vertices.

In Table 7 we use notation $[^{v_1,\ldots,v_n}/_{u_1,\ldots,u_n}]$ (abbreviated as $[^{\boldsymbol{v}}/_{\boldsymbol{u}}]$) to denote substitutions that are applied both to graphs and sets of constraints. If $\rho = [^{\boldsymbol{v}}/_{\boldsymbol{u}}]$ is a substitution then ρG is the graph obtained by substituting all free occurrences of u_i with v_i in G for each $i = 1, ..., n$, while $\rho\Lambda = \{(x, a, \rho\boldsymbol{y}) : (x, a, \boldsymbol{y}) \in \Lambda\}$ where $\rho\boldsymbol{y}$ is the vector whose components result from applying ρ to the corresponding components of $\boldsymbol{y}$.

Finally, given a function $f : A \to B$ and $y \in A$, $f_{-y} : A \setminus y \to B$ is defined as $f_{-y}(x) = f(x)$, for all $x \in A \setminus y$.

The most important rules in Table 7 are ($merge1$) and ($merge2$). They regulates how vertices can be fused. Rule ($merge1$) fuses two vertices provided that no constraint is required on one of them, whereas rule ($merge2$) handles with vertices upon which complementary actions are required. Rule (res) describes how graph transitions can be performed under vertex restriction. Finally, rule (par) states how transitions on disjoint graphs can be combined together.

Let us comment more on all the rules.

Rule ($merge1$) fuses vertex x and y provided that no constraint is imposed on y (i.e. $\Lambda(y) = \emptyset$) and that x and y are equivalent according to π. Premise $x \simeq_\pi y \Rightarrow \pi(y) \neq y$ imposes that, when y is fused with a different equivalent vertex x, then y must not be the representative element. A transition from $\Gamma, y \vdash G$ may be re-formulated to obtain the transition where y and x are coalesced, provided that fusion of their representative elements, ρ, is reflected on Λ, on π and on continuation $\Gamma' \vdash G'$. Indeed, if y is fused with x, also the other vertices equivalent to them are fused; the fusion substitution in the conclusion of ($merge1$) is $\pi;\rho$ (restricted to Γ), all occurrences of $\pi(y)$ are replaced with $\pi(x)$ in $\mathrm{v}(\Lambda)$ and the final graph is $\rho G'$. It is obtained by merging $\pi(y)$ and $\pi(x)$ in G'.

Rule ($merge2$) synchronizes complementary actions. The rule permits merging x and y in a transition where they offer complementary non-silent actions. As for ($merge1$), x cannot replace the representative element of its equivalence class. Most general unifier ρ takes into account possible equalities due to the transitive closure of substitutions $[^{\boldsymbol{v}}/_{\boldsymbol{u}}]$ after $[^x/_y]$ has been applied. ρ fuses the corresponding vertices of the constraints and propagates previous fusions π. The resulting constraints $\rho\Lambda \cup \{(x, \tau, \langle\rangle)\}$ does not change constraints offered on vertices different from x and y (up to the necessary fusion ρ). Fusion substitution $(\pi;\rho)_{-y}$ acts on Γ by applying ρ. Finally, nodes U are the restricted nodes of $\rho G'$ and are those nodes that neither are in $(\pi;\rho)_{-y}(\Gamma)$ nor are generated by $\rho\Lambda$. This corresponds to the *close* rule of the π-calculus.

Finally, vertices U are the restricted vertices in $\rho G'$ and are those vertices that are neither in $(\pi;\rho)_{-y}(\Gamma)$ nor are generated by $\rho\Lambda$.

Rule (res) deals with vertex restriction. Representative elements cannot be restricted if other vertices are in their equivalence class. Furthermore, only vertices can be restricted where either a synchronization action takes place or no constraint is imposed. If those conditions hold, the (possible) silent action on y is hidden and vertices not in $\Gamma' \setminus (\mathrm{v}(\Lambda) \cup \pi_{-y}(\Gamma))$ are restricted.

Rule (par) simply combines together disjoint judgments. Function $\pi \cup \pi'$ applied to a vertex x is $\pi(x)$ or $\pi'(x)$ depending on $x \in \Gamma'$. Note that $\pi \cup \pi'$ is well defined because $\Gamma \cap \Gamma' = \emptyset$.

6 KAOS Translation

In this section, by exploiting the graphical calculus, we define an alternative semantics for KAOS which takes care of QoS attributes. We first present a trans-

lation scheme from KAOS nets and processes to the graphical calculus, then we present the productions of edges used in the translation.

Our translation relies on two particular edges: *node edges* and *link edges*. A node edge $\mathfrak{S}^s$ models KAOS node s while link edge G_t^κ represents a link to node t with cost κ. Moreover, we use a distinguished vertex $\diamond$ to represent the communication infrastructure used to interact with other node edges. In this work we simply represent the communication infrastructure with a special vertex, however, in general this layer could be arbitrarily complex; for instance it could be an ethernet or an internet connection. The only assumption on $\diamond$ is that it must be able to connect any two node edges, indeed $\diamond$ will be exploited to establish links among KAOS nodes.

It is worth to remark that $\diamond$ does not play any rôle in managing application QoS features (indeed, in our framework, virtual networks are built over an underlying physical net[3]) and QoS attributes are established by applications and are not directly related to the underlying communication infrastructure.

The mapping function $[\![\_]\!]$ associates a graph to a well-formed KAOS net. Function $[\![\_]\!]$ is defined by induction on the syntactical structure of KAOS nets. The most important case is the translation of a KAOS node $s ::^L P$, where $L = \{\langle s_1, \kappa_1\rangle, \ldots, \langle s_m, \kappa_m\rangle, \langle \kappa_1, t_1\rangle, \ldots, \langle \kappa_n, t_n\rangle\}$. Since $s ::^L P$ is part of a well formed net, $L^s = \{l : l = \langle s, \kappa\rangle \in L\}$ and $L_s = \{l : l = \langle \kappa, s\rangle \in L\}$ are in bijective correspondence. We assume fixed a bijective function $\lambda : L_s \to L^s$. We define a set of vertices Γ containing vertex $\diamond$ and a vertex for each link occurrence in L: Let $\Gamma = \{u_1, \ldots, u_m, v_1, \ldots, v_n, \diamond\}$ (hereafter, we write $\boldsymbol{u}$ in place of $u_1, \ldots, u_m$ and $\boldsymbol{v}$ in place of $v_1, \ldots, v_n$). Then $\Gamma \setminus \diamond$ is in bijective correspondence with L and we say that u_i *corresponds* to $\langle s_i, \kappa_i\rangle$ $(i = 1, \ldots, m)$ and that v_j *corresponds* to $\langle \kappa_j, t_j\rangle$ $(j = 1, \ldots, n)$.

$$[\![s ::^L P]\!] \;=\; \pi(\Gamma \vdash (\nu\, \boldsymbol{x}, p)([\![P]\!]_p \mid \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \mid \prod_{j=1}^{n} G^{\kappa_j}_{t_j}(x_j, v_j))) \qquad (5)$$

where $\boldsymbol{x}$ is a vector of n pairwise distinct vertices, one for each outgoing link in L and $\pi : \Gamma \to \Gamma$ is a fusion substitution such that π is the identity for all vertices which do not correspond to links in L_s, whereas for all $v \in \Gamma$ that corresponds to $\langle \kappa, s\rangle \in L_s$, $\pi(v) = u$ iff u corresponds to l' and $\lambda(\langle \kappa, s\rangle) = l'$. In other words, π opportunely connects the outgoing link edges that connects s with itself. Notice that, depending on the chosen λ, π changes, hence the translation depends on λ. However, the obtained graphs are equivalent in the sense that they have the same behaviour up-to renaming of external vertices.

The graph associated to $s ::^L P$ contains an edge $\mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond)$ representing node s. Vertices in $\boldsymbol{x}$ are used to connect link edges $G^{\kappa_j}_{t_j}$ to the node edge. A graphical representation is given in Figure 6. The graph representing process P allocated at s is connected to $\mathfrak{S}^s$ on vertex p which is used for synchronizing $\mathfrak{S}^s$ with local processes. In some sense, edge $\mathfrak{S}^s$ is the coordinator of node s and interfaces incoming links, processes executed at s and links departing from s. The dotted tentacles in Figure 6 aim at remarking that each edge $G^{\kappa'_i}_s$, corresponding

[3] This is a typical *peer-to-peer* fashion of coordinating distributed computations.

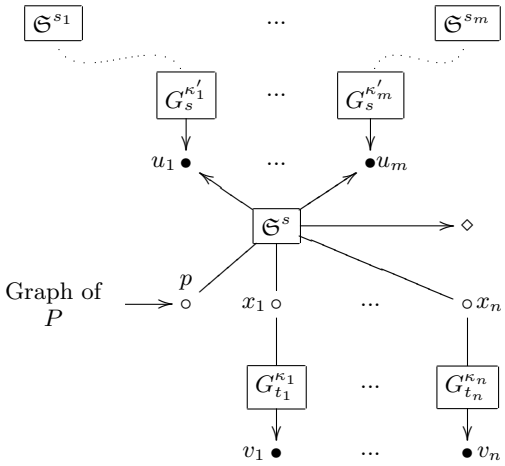


Fig. 6. Graphs for KAOS nodes

to a link from s_i to s, is connected to a restricted node shared with a tentacle of the node edge of s_i.

Net $N_1 \| N_2$ is mapped into a graph obtained by juxtaposing the graphs of the constituent nets, N_1 and N_2 and opportunely connecting their link edges.

$$[\![N_1 \| N_2]\!] = \pi(\Gamma_1, \Gamma_2 \vdash G_1 \mid G_2), \qquad \text{if } [\![N_i]\!] = \Gamma_i \vdash G_i,\ i = 1, 2$$

Function $\pi : \Gamma_1, \Gamma_2 \to \Gamma_1, \Gamma_2$ is fusion substitution that plays that same rôle as in translation of a single done. Indeed, since $N_1 \parallel N_2$ is a well-formed net, there is a bijective correspondence between outgoing links in the network interface of nodes in N_1 and incoming links of nodes in N_2 (and viceversa). Hence, if $v \in \Gamma_1$ ($v \in \Gamma_2$) is a vertex corresponding to a link $\langle \kappa, t \rangle$ in the network interface of node s in N_1 (N_2) and $u \in \Gamma_2$ ($u \in \Gamma_1$) is the vertex corresponding to $\langle s, \kappa \rangle$, then $\pi(v) = u$.

Restriction of nodes is trivially translated according to the following clause:

$$[\![(\nu\, s)N]\!] = \Gamma \setminus \boldsymbol{u} \setminus \boldsymbol{v} \vdash (\nu\, \boldsymbol{u}, \boldsymbol{v}).G, \qquad \text{if } [\![N]\!] = \Gamma \vdash G$$

where, $\boldsymbol{u}$ and $\boldsymbol{v}$ are the vertices where incoming and outgoing link edge of $\mathfrak{S}^s$ are respectively connected, if $\mathfrak{S}^s$ occurs in G and are empty vectors otherwise. The graph of a net with a restricted node name, $(\nu\, s)N$, is computed by first translating N and then restricting all vertices corresponding to node s.

The mapping for processes is described by the equations below:

$$
\begin{aligned}
[\![\ \mathbf{0}\ ]\!]_p &= nil \\
[\![\ \langle t \rangle\ ]\!]_p &= L_{\langle t \rangle}(p) \\
[\![\ \gamma.P\ ]\!]_p &= L_{\gamma.P}(p) \\
[\![\ \varepsilon(P)@s\ ]\!]_p &= (\nu\, u)(\varepsilon_s^{T(P)}(u,p) \mid S_P(u)) \\
[\![\ P_1 \mid P_2\ ]\!]_p &= [\![\ P_1\ ]\!]_p \mid [\![\ P_2\ ]\!]_p \\
[\![\ rec\, X.\, P\ ]\!]_p &= [\![\ P[^{rec\, X.\, P}/_X]\ ]\!]_p.
\end{aligned}
$$

The graph of a process P has an outgoing tentacle toward its execution vertex. The graph relative to the empty process simply is the empty graph; tuple processes and action prefixing are mapped to edges attached to p and labeled with the process. Translation of $\varepsilon(P)@s$ consists of two edges connected through the (hidden) vertex u. On the one hand, edge $\varepsilon_s^T(P)$ is connected to vertex p and handles migration of S_P to its destination node; on the other hand, P cannot be translated as a normal process because it must be executed when the migration has taken place. Hence, edge S_P is used; as will be clear once productions will be specified, S_P remains "idle" until the destination node is reached and at that time, P will be executed on the arrival node. The parallel processes are mapped to the union of the graphs of their parallel components; finally, recursive processes are translated by translating the unfolded process. It is worth to remark that we consider only "guarded" recursion, namely we require that in $rec\, X.\, P$ any process variable is in the scope of a prefix action. This implies that translation always terminates.

The following property holds for the presented translation functions:

Theorem 1. *If N is a well-formed* KAOS *net, then for each link edge $G_s^\kappa(\_, u)$ in $[\![\ N\ ]\!]$ there is a (unique) node edge $\mathfrak{S}_{m,n}^s(\boldsymbol{u}, \_, \_, \diamond)$ in $[\![\ N\ ]\!]$ such that u appears exactly once in $\boldsymbol{u}$.*

7 Productions for KAOS

As anticipated in Section 1, graphs permit path reservation. However, we prefer to introduce first productions which do not consider path reservation, but are more strictly related to KAOS semantics and, later on, we show how a path can be reserved and traversed.

We distinguish between *activity* and *coordination* productions. Indeed, it is necessary to coordinate node, link and process edges in order to detect the best path connecting two vertices. Hence, we separate the presentation of activity and coordination productions in different sections. Section 7.1 describes the activity productions necessary for executing KAOS actions; Section 7.2 and Section 7.3 respectively report coordination productions for node edges and for link edges.

7.1 Activity Productions

Activity productions for KAOS deal with actions for accessing tuple spaces, for managing links, for creating nodes and for spawning processes on remote nodes. Let us consider actions for accessing tuple spaces; the productions for the corresponding edges are:

$$p \vdash L_{(x).P}(p) \xrightarrow{\{(p,\ in\, t,\ \langle\rangle)\}} p \vdash [\![\ P[^{t}/_{x}]\ ]\!]_p$$

$$p \vdash L_{\langle t \rangle}(p) \xrightarrow{\{(p,\ \overline{in\, t},\ \langle\rangle)\}} p \vdash nil$$

The above productions state that edges corresponding to input actions wait on the vertex p for synchronizing with the production of an output. Notice that, in the rhs of the last production the edge corresponding to the output process is removed.

An edge corresponding to $\nu(r \cdot \kappa).P$ synchronizes with its node edge in order to acquire the connection to the net ($\diamond$) and make the node edge of P to add a link to r.

$$p \vdash L_{\nu(r\cdot\kappa).P}(p) \xrightarrow{\{(p,\ \overline{ns},\ \langle y, z\rangle)\}} p, y, z \vdash (\nu\ q, u)([\![\ P\ ]\!]_p \mid \mathfrak{S}^{r}_{1,0}(u, q, z) \mid G^{\kappa}_{r}(y, u)).$$

The above production "reads" into z the attach point to the net; during the synchronization, the node edge also generates a new vertex y, used to connect the outgoing link to vertex u. Production (11) is the complementary of the above production.

Creation of new links requires to synchronize process and node edges.

$$p \vdash L_{\overset{\kappa}{\frown} x.P}(p) \xrightarrow{\{(p,\ \overline{x\ log\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{6}$$

Production (6) sends a signal for creating a new link with cost κ to (the node edge of) x.

The accept action can be similarly handled:

$$p \vdash L_{x\overset{\kappa}{\smile}.P}(p) \xrightarrow{\{(p,\ \overline{x\ acc\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p.$$

According to the above production, the process simply "says" to its node edge that is willing to *acc*ept a request of connection from vertex x with cost κ.

Let x and κ respectively be the node and the cost of a link l; the following productions manage disconnection of l:

$$p \vdash L_{\delta\, l.P}(p) \xrightarrow{\{(p,\ \overline{\kappa\ det\ x},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{7}$$

$$p \vdash L_{\delta\, l.P}(p) \xrightarrow{\{(p,\ \overline{x\ det\ \kappa},\ \langle\rangle)\}} p \vdash [\![\ P\ ]\!]_p \tag{8}$$

Production (7) removes outgoing links, while incoming links are removed by production (8).

Remote process evaluation is managed by the productions for edge $\varepsilon_s^T(u,p)$. Vertex u is used to connect the "quiescent" process that must be spawned

$$u, p \vdash \varepsilon_s^T(u,p) \xrightarrow{\{(r,\ \overline{ev\ T\ s},\ \langle\rangle)\}} u, p \vdash \varepsilon_s'(u,p).$$

The previous production asks for a path to s that can be exploited to move a process with capability T. When the path is found, an $s\,\kappa$ signal is received (where $\kappa \neq 0$):

$$u, p \vdash \varepsilon_s'(u,p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y\rangle),\ (u,\ \overline{run},\ \langle y\rangle)\}} u, p, y \vdash nil; \tag{9}$$

vertex y represents the p-vertex of the remote node. Simultaneously, the quiescent process connected to u is waken by the action $\overline{run}$.

$$u \vdash S_Q(u) \xrightarrow{\{(u,\ run,\ \langle y\rangle)\}} u, y \vdash [\![\ Q\ ]\!]_y. \tag{10}$$

Finally, the quiescent process S_Q starts its execution when it synchronizes with its ε edge on vertex y. This corresponds to move the process Q to the target node.

7.2 Productions for Nodes

Productions for node edges must coordinate the activity of processes. We start with the simplest case. If a process wants to create a new node, then the node edge can immediately synchronize by sending vertex $\diamond$, namely the net connection where the new vertex must be attached. The following production formalizes what informally stated:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{(p,\ ns,\ \langle y, \diamond\rangle)\}} \boldsymbol{x}, y, p, s, \diamond \vdash \mathfrak{S}^s_{m,n+1}(\boldsymbol{u}, y, \boldsymbol{x}, p, \diamond). \tag{11}$$

Notice that production (11) adds a tentacle to the node edge and connects it to the newly generated vertex y which is also returned to the process waiting on p (see productions 9 and 10).

A slightly more complex production is required for handling new link creation:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\Lambda} \boldsymbol{u}, \boldsymbol{x}, p, \diamond, y \vdash \nu\, z.(\mathfrak{S}^s_{m,n+1}(\boldsymbol{u}, z, \boldsymbol{x}, p, \diamond) \mid G^\kappa_y(z,y)),$$

where $\Lambda = \{(p,\ \ t\ log\,\kappa,\ \ \langle\rangle), (\diamond,\ \ t\ acc\,\kappa,\ \ \langle y\rangle)\}$. The intuition is that, when a process asks to its node edge for a new link to node t with attributes κ (action on p), then the node edge synchronizes (over $\diamond$) with the node edge at t that must *acc*ept the connection. The new link edge is connected to vertex z and reaches vertex y. A simpler production is for accepting new links:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\left\{\begin{smallmatrix}(p,\ t\ acc\,\kappa,\ \langle\rangle),\\ (\diamond,\ \overline{t\ acc\,\kappa},\ \langle z\rangle)\end{smallmatrix}\right\}} \boldsymbol{u}, \boldsymbol{x}, z, p, \diamond \vdash \mathfrak{S}^s_{m+1,n}(z, \boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

Indeed, the node edge must simply forward the *acc* signal to the net.

Finally, incoming link disconnection simply requires to forward the *det* signal to the incoming links of node s

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\substack{(p,\ t\ det\ \kappa,\ \langle\rangle),\\ (\boldsymbol{u},\ \overline{t\ det\ \kappa},\ \langle\rangle)}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

The above production synchronizes with an incoming link edge that will disappear.

Removing an outgoing link is more complex because the node edge must first find which is the tentacle to remove. Hence, the node edge forwards the disconnection signal (received on p) to its link edges:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\substack{(p,\ \kappa\ det\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{\kappa\ det\ t},\ \langle\rangle)}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{'s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

Edge $\mathfrak{S}^{'}$ waits for the links to determine whether they must disconnect or not:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{'s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\substack{(x_1,\ nodet,\ \langle\rangle),\\ \{\ldots(x_i,\ det,\ \langle\rangle),\ldots\}\\ (x_n,\ nodet,\ \langle\rangle)}} \boldsymbol{u}, \boldsymbol{z}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n-1}(\boldsymbol{u}, \boldsymbol{z}, p, \diamond)$$

where $\boldsymbol{z} = x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$. The link that replies with the signal (*det*) disappears (see production (15)) while all other edges remain connected to the router.

The last task of node edges is the search of paths for remote process spawning. First, when a process asks for a path to node t such that a process with capabilities T can roam the path (action $(p,\ ev\ T\ t,\ \langle\rangle)$), the signal ev is forwarded to the outgoing links:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\substack{(p,\ ev\ T\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{ev\ T\ t},\ \langle\rangle)}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \overline{F}^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, p, \diamond)$$

where the signal sent over $\boldsymbol{x}$ contains the type of the migrating process T and the target node t. As formally stated in the next section, link edges forward signals $ev\ T\ t$ received from their node edge to the remote node edge they are connected to. Hence, node edges must synchronize with link edges and make the request traverse the net:

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{\substack{(\boldsymbol{u},\ ev\ T\ t,\ \langle\rangle),\\ (\boldsymbol{x},\ \overline{ev\ T\ t},\ \langle\rangle)}\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond).$$

Edge F and $\overline{F}$ have similar productions, the only difference being that $\overline{F}$ forwards search results on vertex p, while F sends them to the incoming links connected to $\boldsymbol{u}$. Therefore, in the following we consider only productions for F. When costs are communicated to F, it starts to forward them to links.

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\ \varepsilon\ t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond) \xrightarrow{\substack{(x_1,\ t\ \kappa_1,\ \langle y_1\rangle), ...,\\ \{\ (x_n,\ t\ \kappa_n,\ \langle y_n\rangle),\ \}\\ (\boldsymbol{u},\ \overline{t\ \kappa_h},\ \langle y_h\rangle)}} \boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{y}, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond)$$

where $\kappa_h = \kappa_1 + ... + \kappa_n$.

Finally, node edges communicate their p-vertex when incoming links require them with an *eval* action (see page 31):

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \xrightarrow{\{(u_i,\ eval,\ \langle p \rangle)\}} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^s_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond).$$

7.3 Productions for Links

Whenever a link $G'^{\kappa}_s(x, v)$ receives a message for searching a path to a vertex t ($t \neq s$) suitable for a process with capabilities T, then it forwards the signal, provided that $\kappa \models T$:

$$x, v \vdash G'^{\kappa}_s(x, v) \xrightarrow{\{(x,\ ev\ T\ t,\ \langle\rangle),\ (v,\ \overline{ev\ T\ t},\ \langle\rangle)\}} x, v \vdash \widehat{G}^{t,\kappa}_s(x, v).$$

$\widehat{G}^{t,\kappa}_s(x, v)$ waits on v for the cost κ' of the path from s to t and sends back to the router edge the new value of the optimal path:

$$x, v \vdash \widehat{G}^{t,\kappa}_s(x, v) \xrightarrow{(v,\ t\ \kappa',\ \langle u \rangle),\ (x,\ \overline{t\ \kappa' \times \kappa},\ \langle u \rangle)} x, v, u \vdash G'^{\kappa}_s(x, v). \tag{12}$$

Otherwise, if $\kappa \not\models T$, the "infinite" cost 0 is backward propagated:

$$x, v \vdash G'^{\kappa}_s(x, v) \xrightarrow{\{(x,\ ev\ T\ t,\ \langle\rangle)\}} x, v \vdash \underline{G}^{t,\kappa}_s(x, v)$$

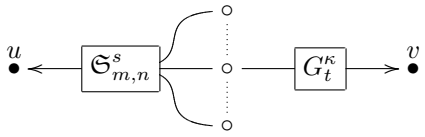
$$x, v \vdash \underline{G}^{t,\kappa}_s(x, v) \xrightarrow{\{(x,\ \overline{t\ 0},\ \langle\rangle)\}} x, v \vdash G'^{\kappa}_s(x, v).$$

Finally, when a link enters the target vertex, then it asks for the p-vertex of the node edge and back-forwards it:

$$x, v \vdash G'^{\kappa}_s(x, v) \xrightarrow{\{(x,\ ev\ T\ s,\ \langle\rangle),\ (v,\ \overline{eval},\ \langle y \rangle)\}} x, v \vdash G'^{\kappa}_s(x, v, y)$$

$$x, v \vdash G'^{\kappa}_s(x, v, y) \xrightarrow{\{(x,\ s\ \kappa,\ \langle y \rangle)\}} x, v \vdash G'^{\kappa}_s(x, v). \tag{13}$$

Given a graph $\Gamma \vdash G$, we say that vertices u and v of G are *link-adjacent* if the graph below is a subgraph of $\Gamma \vdash G$.



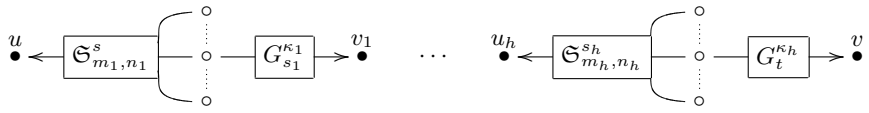
A *link path in* G is a sequence of link-adjacent vertices; we say that (free) vertices of a link path are *link-connected*. The cost of a link path is the sum of the costs associated to each link edge appearing in the path. We can now state an important result on selecting the minimal cost path between two link-connected vertices.

Theorem 2. *Let $\Gamma \vdash G$ be a graph and $u, v \in \Gamma$ and v be the vertex where the edge node is connected. If*

$$\Gamma \vdash G \xrightarrow{\Lambda \cup \{(u,\ t\,\kappa,\ \langle u \rangle)\}} \Gamma' \vdash G' \tag{14}$$

then the following properties hold:

1. *if transition (14) can be derived then u and v are link-connected by a path of cost κ;*
2. *if*



is a link-path between u and v in G, then there is a transition like (14) such that $\kappa \leq \Sigma_{i=1}^{h} \kappa_i$.

Theorem 2 means that the path search triggered by remote actions detects a link-path if it exists in the graph (first part of the theorem), moreover the search always selects the minimal cost path connecting two link-connected vertices (second part of the theorem).

Finally, we must consider the productions for disconnecting links. When link edges receive the logout signal from their router edge, they simply disappears. We model this by transforming the cost of the link in an infinite cost:

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(x,\ \kappa\ det\ s,\ \langle\rangle)\}} x, v \vdash G^{0}_{s}(x, v) \tag{15}$$

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(v,\ s\ det\ \kappa,\ \langle\rangle),\ (x,\ \overline{det},\ \langle\rangle)\}} x, v \vdash G^{0}_{s}(x, v).$$

If the link is not the link selected by the logout signal, the link edge remains connected:

$$x, v \vdash G^{\kappa}_{s}(x, v) \xrightarrow{\{(x,\ t\ det\ \kappa',\ \langle\rangle)\}} x, v \vdash G^{\kappa}_{s}(x, v).$$

8 Path Reservation

This section aims at modifying the productions presented so far in order to permit path reservation and "routing" along reserved link edges. We show how path reservation is essentially obtained by enriching the behaviour of node and link edges with new productions and with slight variations of productions for KAOS actions introduced in Section 7.1.

Let us again consider production:

$$u, p \vdash \varepsilon'_{s}(u, p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y \rangle),\ (u,\ \overline{run},\ \langle y \rangle)\}} u, p, y \vdash nil. \tag{16}$$

In order to reserve paths, we change the behavior of edge ε'_s. Indeed, vertex y should be considered as the "next-hop" vertex instead of being the final vertex. Therefore, we replace production (16) with

$$u, p \vdash \varepsilon'_s(u,p) \xrightarrow{\{(p,\ s\,\kappa,\ \langle y\rangle)\}} u, p, y \vdash \varepsilon''_s(u,y)$$

Edge ε''_s communicates its destination and waits the vertex where to jump to:

$$u, y \vdash \varepsilon''_s(u,y) \xrightarrow{\{(y,\ \overline{dest\ s},\ \langle\rangle)\}} u, y \vdash \widehat{\varepsilon}_s(u,y)$$

$$u, y \vdash \widehat{\varepsilon}_s(u,y) \xrightarrow{\{(y,\ jump,\ \langle z\rangle)\}} u, y, z \vdash \varepsilon''_s(u,z)$$

until a *stop* signal is received. In this case, ε'' triggers the roaming process sending the *run* message:

$$u, y \vdash \varepsilon''_s(u,y) \xrightarrow{\{(y,\ stop,\ \langle p\rangle),\ (u,\ \overline{run},\ \langle p\rangle)\}} u, y, p \vdash nil.$$

As will be clear later, the last link of the route will synchronize with the above production and stop the migration.

It is also necessary to communicate to the link edges whether they are reserved or not. Therefore, the production of F edges must be changed as

$$\boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash F^{s\,\varepsilon\,t}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \boldsymbol{u}, \diamond) \xrightarrow{\Lambda} \boldsymbol{u}, \boldsymbol{x}, p, \diamond \vdash \mathfrak{S}^{s}_{m,n}(\boldsymbol{u}, \boldsymbol{x}, p, \diamond) \mid \widehat{\Delta}^{h}_{n}(\boldsymbol{x})$$

where $\Lambda = (x_1,\ t\,\kappa_1,\ \langle y_1\rangle), ..., (x_n,\ t\,\kappa_n,\ \langle y_n\rangle), (\boldsymbol{u},\ \overline{t\,\kappa_h},\ \langle y_h\rangle)$ and $\kappa_h = \kappa_1 + ... + \kappa_n$. Edge $\widehat{\Delta}^h_n$ informs link edges whether they are reserved or not:

$$\boldsymbol{x} \vdash \widehat{\Delta}^{h}_{n}(\boldsymbol{x}) \xrightarrow{\Lambda} \boldsymbol{x} \vdash nil$$

where $\Lambda = \{(x_i,\ \overline{nores},\ \langle\rangle) : i = 1, ..., n\ \wedge\ i \neq h\}$. This production makes $\widehat{\Delta}^h_n$ to communicate to the h-th link that it is reserved and to the remaining edges that they have not been selected. Of course links must interact with Δ edges in order to accomplish the previous productions. In particular, productions (12) and (13) must be respectively changed with

$$x, v \vdash \widehat{G}^{t,\kappa}_{s}(x,v) \xrightarrow{\{(v,\ t\,\kappa',\ \langle y\rangle), (x,\ \overline{t\,\kappa' \times \kappa},\ \langle x\rangle)\}} x, v, y \vdash Pr^{\kappa}_{s}(x,v,y).$$

and

$$x, v \vdash G'^{\kappa}_{s}(x,v,y) \xrightarrow{\{(x,\ \overline{s\,\kappa},\ \langle x\rangle)\}} x, v, y \vdash Pr^{\kappa}_{s}(x,v,y).$$

The difference lies on the fact that, once the link has backward propagated the cost, it moves to a state Pr^{κ}_{s} where either the *nores* signal is waited or a migrating packet arrives. Edge $Pr^{\kappa}_{s}(x,v,y)$ has an incoming tentacle from x, an outgoing tentacle to v and one to y (where y represents the next-hop vertex).

If a signal *nores* is received, then Pr^{κ}_{s} becomes the link to v as stated in the following production:

$$x, v, y \vdash Pr^{\kappa}_{s}(x,v,y) \xrightarrow{\{(x,\ nores,\ \langle\rangle)\}} x, v, y \vdash G^{\kappa}_{s}(x,v).$$

Otherwise, a packet will be attached to s and Pr_s^κ will take care of its destination. If the destination is s, the packet will terminate its travel:

$$x, v, y \vdash Pr_s^\kappa(x, v, y) \xrightarrow{\{(x,\ dest\, s,\ \langle\rangle)\}} x, v, y \vdash \widehat{Pr}_s^\kappa(x, v, y).$$

$$x, v, y \vdash \widehat{Pr}_s^\kappa(x, v, y) \xrightarrow{\{(x,\ \overline{stop},\ \langle y\rangle)\}} x, v, y \vdash G_s^\kappa(x, v).$$

Once Pr_s^κ receives a signal from an edge ε_s'' that wants to reach s, it replies with a *stop* message where the last hop vertex is communicated. The intention is that y is the p-vertex of the node edge of s.

A *jump* signal is emitted,to let the packet reach vertex t different from s:

$$x, v, y \vdash Pr_s^\kappa(x, v, y) \xrightarrow{\{(x,\ dest\, t,\ \langle\rangle)\}} x, v, y \vdash Pr'^\kappa_s(x, v, y)$$

$$x, v, y \vdash Pr'^\kappa_s(x, v, y) \xrightarrow{\{(x,\ \overline{jump},\ \langle y\rangle)\}} x, v, y \vdash G_s^\kappa(x, v).$$

The productions presented in this section and Theorem 2 in the previous section ensure that whenever a remote operation is performed the graphical calculus always selects the optimal path with respect to the QoS attributes specified by the KAOS networking constructs. This result depends on the outcome of a distributed constraint satisfaction problem, the *rule matching problem* [25]. For the result to hold, QoS attributes must form an ordered c-semiring [3], whose additive and multiplicative operations allow us to compare and compose QoS parameters.

9 Messaging & Graphs

This section shows how the graphical calculus is applied to the messaging application of Section 4. We aim at illustrating how a minimal path between two nodes can be reserved and traversed by a remotely evaluated process. We also describe how the Floyd-Warshall algorithm [12] computes the same paths when applied to the graph.

We consider the configuration reached by net (2) (see page 15) after the notification message has reached t and has been acquired by R. More precisely, we discuss how filter process F is remotely executed at s. In particular, we focus on determining the minimal path from t to s and how F traverses it in the net topology determined in Section 4.

Figure 7 reports the graphical representation of the described configuration; it does not faithfully represent the KAOS net in terms of graphs because we avoid depicting vertex $\diamond$ and all tentacles connecting node edges with it. Moreover, vertices where process edges are attached are graphically represented as $\star$. Both these choices have respectively been adopted because

- no synchronization takes place on vertex $\diamond$ during the routing phase of F, and

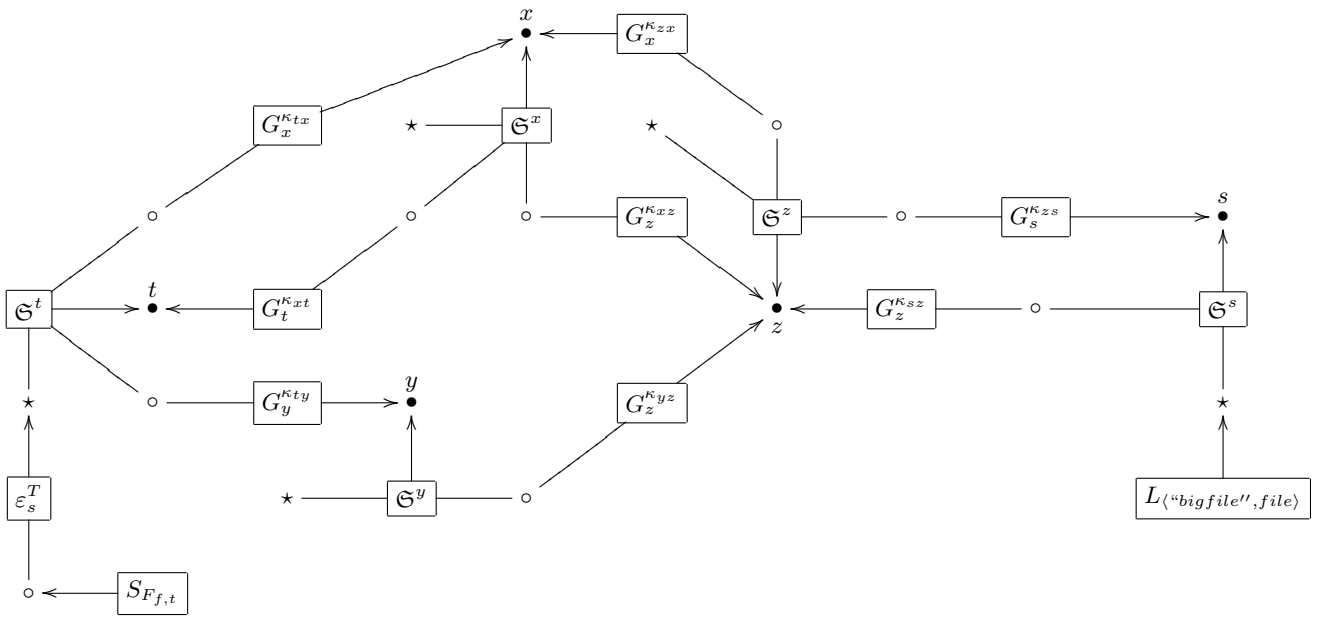


Fig. 7. Graph for messaging net

– Figure 7 becomes more readable.

The graph of Figure 7 is the counterpart of Figure 2 in terms of graphs. Node edges are connected to their outgoing link edges on vertices ∘, while edges for incoming links are attached on vertices •. Initially, edge ε_s^T "wraps" the edge corresponding to the filter process $S_{F_{f,t}}$ and is connected to the ⋆ vertex corresponding to node edge $\mathfrak{S}^t$. This amounts to say that $\varepsilon(F_{f,t})@s$ is allocated at t. Similarly, edge $L_{\langle \text{“}bigfile'', file\rangle}$ is allocated at s.

Instead of detailing the semantical framework for deducing graph transitions, we describe edge behaviour and synchronization in terms of graphical figures (similar to Figure 7), where tentacles are annotated by synchronizing actions. Since tentacles are connected to the vertices where synchronizations take place, we avoid writing those information in the labels.

Figure 8 summarizes the productions used for searching a (minimal) path connecting s and t and represents the graph with the annotated edges. Notice how each node edge $\mathfrak{S}^u$ enriches *ev* labels with name u. For instance, edge $\mathfrak{S}^x$ "receives" the signal *t ev Ts* along tentacle to x and forwards the signal *tx ev Ts* to its links. We remark that node edge $\mathfrak{S}^z$ can non-deterministically synchronize with two different *ev* actions, i.e. the one triggered by the link from x or from y nodes. However, the result does not depend on the chosen synchronization.
Differently from the other link and node edges, $G_s^{\kappa_{zs}}$ and $\mathfrak{S}^s$ synchronize through the *eval* signal because $G_s^{\kappa_{zs}}$ is a link to the target vertex s. This synchronization allows $G_s^{\kappa_{zs}}$ to determine the ⋆ vertex of $\mathfrak{S}^s$ as will be explained in the following.

According to the edge replacement mechanisms, the graph in Figure 8 rewrites as shown in Figure 9 (where also the productions for the next graph transition are listed). Figure 9 depicts the graph after the synchronizations enabled in the graph in Figure 8 have taken place. We let $\kappa = \kappa_{zs} \times \kappa_{xz} \times \kappa_{tx}$ and $\kappa' = \kappa_{zs} \times \kappa_{yz} \times \kappa_{ty}$. Labels appearing on tentacles do not mention the "next-hop" vertex as (formally) required by the corresponding productions because it is graphically represented by the ∘ vertex where link edges are attached at. As stated above, $G_z^{\kappa_{zs}}$ has interacted with $\mathfrak{S}^s$ and has acquired the vertex where (the graph corresponding to) filter F must be connected and executed. This is represented by the dotted tentacle in Figure 9.
In this phase, costs are backwardly propagated by link and node edges. Notice that $F^{t\,\varepsilon\,s}$ edge connected to t forwads the (minimal) cost on its ⋆ edge, whereas the other $F^{t\,\varepsilon\,s}$ edges send the costs on their • vertices. This is due to the fact that the second and third productions reported in Figure 8 distinguish whether the *ev* signal has been received on the ⋆ or the • vertex and consequently determine the "result" vertex r.

The graph resulting from synchronizing productions in Figure 9 is reported in Figure 10. As before, dotted tentacles represents next-hop "address" of the reserved links. Indeed, edge ε' moves on the "tail" vertex of the link connecting t to x. On this vertex, it will synchronize with $Pr_x^{\kappa_{tx}}$ through the *jump* productions and first reach the (reserved) link from x to z, then the link from z to s and, finally, it will receive the signal *stop* together with the p-vertex of $\mathfrak{S}^t$.

$$u, p \vdash \varepsilon_s^T(u,p) \xrightarrow{\{(r,\ \overline{ev\ T\ s},\ \langle\rangle)\}} u, p \vdash \varepsilon'_s(u,p)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(p,\ ev\ T\ t,\ \langle\rangle),\ (\boldsymbol{x},\ \overline{s\ ev\ T\ t},\ \langle\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash F_{n,1}^{s\,\varepsilon\,t}(\boldsymbol{x}, p, s, p, \diamond)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(s,\ \sigma\ ev\ T\ t,\ \langle\rangle),\ (\boldsymbol{x},\ \overline{\sigma s\ ev\ T\ t},\ \langle\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash F_{n,1}^{s\,\varepsilon\,t}(\boldsymbol{x}, p, s, s, \diamond)$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ \sigma\ ev\ T\ t,\ \langle\rangle),\ (s,\ \overline{\sigma\ ev\ T\ t},\ \langle\rangle)\}} x, s \vdash \widehat{G}_s^{t,\kappa}(x,s)$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{(x,\ \sigma\ ev\ T\ t,\ \langle\rangle)} x, s \vdash \underline{G}_s^{t,\kappa}(x,s), \text{ if } s \text{ appears in } \sigma$$

$$x, s \vdash G_s^\kappa(x,s) \xrightarrow{\{(x,\ \sigma\ ev\ T\ s,\ \langle\rangle),\ (s,\ \overline{eval},\ \langle y\rangle)\}} x, s \vdash G'^{\kappa}{}_s(x,s,y)$$

$$\boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond) \xrightarrow{\{(s,\ eval,\ \langle p\rangle)\}} \boldsymbol{x}, p, s, \diamond \vdash \mathfrak{S}_n^s(\boldsymbol{x}, p, s, \diamond)$$

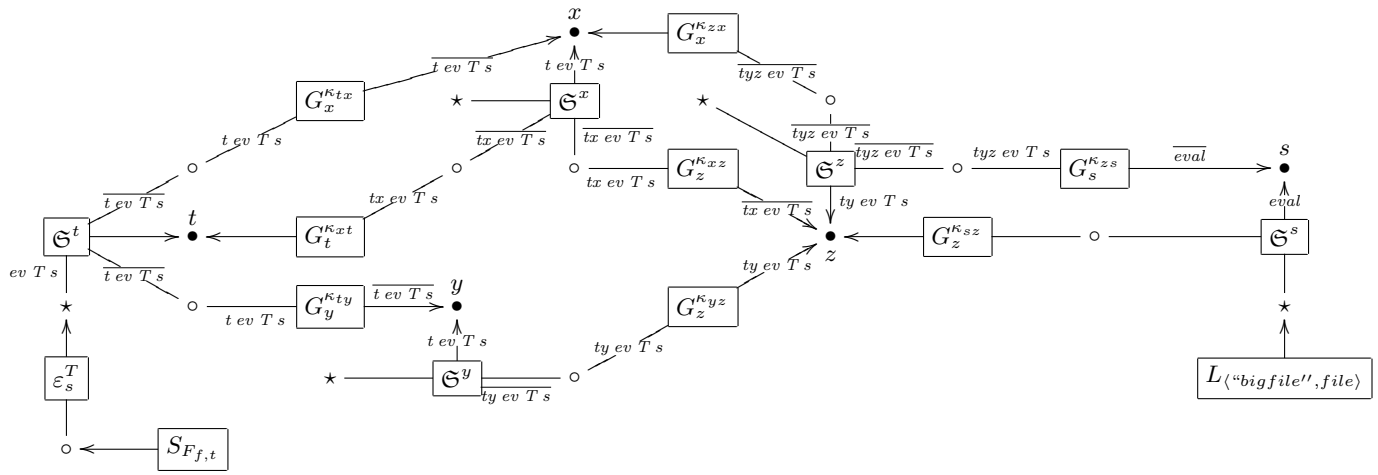


Fig. 8. Asking for paths

$$\boldsymbol{x}, p, s, r, \diamond \vdash F^{s\,\varepsilon\,t}_{n,i}(\boldsymbol{x}, p, s, r, \diamond) \xrightarrow{\substack{(x_1,\ t\,\kappa_1,\ \langle y_1\rangle), ..., \\ (x_n,\ t\,\kappa_n,\ \langle y_n\rangle), \\ (r,\ \overline{t\,\kappa_h},\ \langle y_h\rangle)}} \boldsymbol{x}, p, s, r, \diamond \vdash \mathfrak{S}^s_n(\boldsymbol{x}, p, s, \diamond) \mid \widehat{F}^{t\,\kappa}_{i-1}(r) \mid \widehat{\Delta}^h_n(\boldsymbol{x}),$$
where $i \geq 1$ and $\kappa_h = \kappa_1 + ... + \kappa_n$

$$\boldsymbol{x} \vdash \widehat{\Delta}^h_n(\boldsymbol{x}) \xrightarrow{\Lambda} \boldsymbol{x} \vdash nil, \text{ where } \Lambda = \{(x_i,\ \overline{nores},\ \langle\rangle) : i = 1, ..., n\ \wedge\ i \neq h\} \cup (x_h,\ \overline{res},\ \langle\rangle)$$

$$x, s \vdash \widehat{G}^{t,\kappa}_s(x, s) \xrightarrow{\{(s,\ t\,\kappa',\ \langle y\rangle), (x,\ \overline{t\,\kappa' \times \kappa},\ \langle x\rangle)\}} x, s, y \vdash Pr^{\kappa}_s(x, s, y)$$

$$x, s \vdash G'^{\kappa}_s(x, s, y) \xrightarrow{\{(x,\ \overline{s\,\kappa},\ \langle y\rangle)\}} x, s, y \vdash Pr^{\kappa}_s(x, s, y)$$

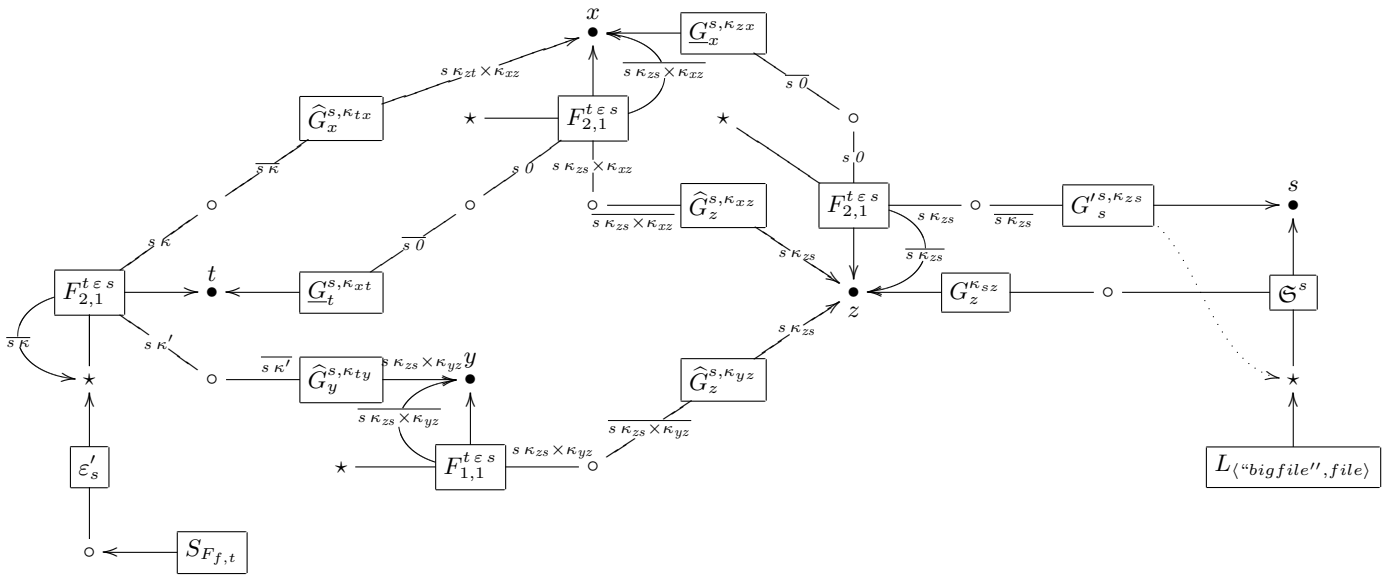


Fig. 9. Computing paths

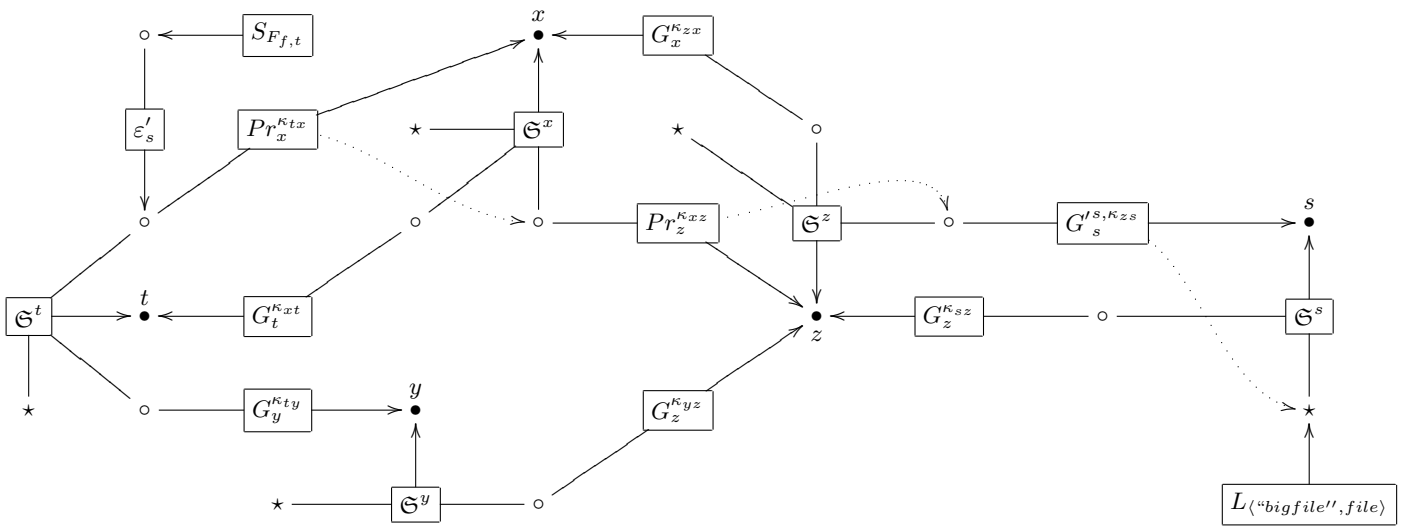


Fig. 10. Minimal path

In this example, in order to give a smooth presentation, we do not consider the productions for path reservation that are determined by synchronizing the first and the second productions in Figure 9. However, such synchronizations are straightforward and simply make link edges become Pr edges or make them return in their initial "state", depending whether they are reserved or not.

Given a graph representing a KAOS net and a process $\varepsilon(Q)@t$ allocated on some node s (as net in Figure 7), we can built a matrix of costs such that the Floyd-Warshall algorithm [12] can be used to compute (one of) the minimal path(s) connecting s and t that can be traversed by Q.

Intuitively, if $\kappa_{uv} \models T(Q)$ then position (u, v) of the matrix contains κ_{uv}, the cost of the the link edge from u to v;if $\kappa_{uv} \not\models T(Q)$ or no link edges connects u and v, then position (u, v) contains 0. Table 8 reports the matrix corresponding to the net in Figure 7.

	t	x	y	z	s
t	0	κ_{tx}	0	0	0
x	κ_{xt}	0	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	0	κ_{zx}	0	0	κ_{zs}
s	0	0	0	κ_{sz}	0

Table 8. The initial matrix

Notice that position (t, y) contains 0 because $\kappa_{ty} \not\models T(F_{f,t})$.

Given a vertex $z \neq t$, let $z{-}\mathbf{1}$ represent the vertex that precedes z in the list $[t, x, y, z, s]$. The Floyd-Warshall algorithm is an iterative algorithm that transforms the matrix of costs according the following relation:

$$\kappa_{uv}^{z} = \kappa_{uv}^{z-\mathbf{1}} + (\kappa_{uz}^{z-\mathbf{1}} \times \kappa_{zv}^{z-\mathbf{1}})$$

where $+$ and $\times$ are the c-semiring operations.

Table 9 reports the matrices computed by the iterations of the Floyd-Warshall algorithm starting from the cost matrix of Table 8. Position (t, s) of the last matrix in Table 9 contains the cost of the minimal path from t to s.

We remark that the Floyd-Warshall algorithm can be applied when costs of edges are totally ordered. In our example this was the case, but in general it is not. For instance, consider two vertices connected by two link edges having costs $\langle 1, T, 2\rangle$ and $\langle 2, T, 1\rangle$, respectively. According to the definition of $+$, we have that $\langle 1, T, 2\rangle + \langle 2, T, 1\rangle = \langle 1, T, 1\rangle$, which does not correspond to any path between the vertices. In order to overcome this problem, we can use the *Hoare powerdomain* of the previous c-semiring of costs. In [4] it has been noticed that if $(A, +, \times, 0, 1)$ is a c-semiring then the Hoare powerdomain $(\wp^H(A), \cup, \times^*, \emptyset, A)$ is also a c-semiring; here

	$\boxed{t}$	x	y	z	s
t	0	κ_{tx}	0	0	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	0	κ_{zx}	0	0	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	$\boxed{x}$	y	z	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	x	$\boxed{y}$	z	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	0
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	0
y	0	0	0	κ_{yz}	0
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	0	0	0	κ_{sz}	0

	t	x	y	$\boxed{z}$	s
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	$\kappa_{tx} \times \kappa_{xz} \times \kappa_{zs}$
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	$\kappa_{xz} \times \kappa_{zs}$
y	$\kappa_{yz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{yz} \times \kappa_{zx}$	0	κ_{yz}	$\kappa_{yz} \times \kappa_{zs}$
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	$\kappa_{sz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{sz} \times \kappa_{zx}$	0	κ_{sz}	$\kappa_{sz} \times \kappa_{zs}$

	t	x	y	z	$\boxed{s}$
t	$\kappa_{tx} \times \kappa_{xt}$	κ_{tx}	0	$\kappa_{tx} \times \kappa_{xz}$	$\kappa_{tx} \times \kappa_{xz} \times \kappa_{zs}$
x	κ_{xt}	$\kappa_{xt} \times \kappa_{tx}$	0	κ_{xz}	$\kappa_{xz} \times \kappa_{zs}$
y	$\kappa_{yz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{yz} \times \kappa_{zx}$	0	κ_{yz}	$\kappa_{yz} \times \kappa_{zs}$
z	$\kappa_{zx} \times \kappa_{xt}$	κ_{zx}	0	$\kappa_{zx} \times \kappa_{xz}$	κ_{zs}
s	$\kappa_{sz} \times \kappa_{zx} \times \kappa_{xt}$	$\kappa_{sz} \times \kappa_{zx}$	0	κ_{sz}	$\kappa_{sz} \times \kappa_{zs}$

Table 9. Iterations of the Floyd-Warshall Algorithm

- $\wp^H(A)$ is the set of all the subsets X of A which are downward closed under the ordering $\leq$ induced on A by the $+$ operation[4], i.e. $\wp^H(A) = \{X \subseteq A : \forall x \in X. \forall y \in A. y \leq x \Rightarrow y \in X\}$;
- $\cup$ is set union and multiplication $\times^*$ is just $\times$ extended to sets, namely $X \times^* Y = \{x \times y : x \in X \wedge y \in Y\}$.

Moreover, if $+$ induces a total order on A then $\wp^H(A)$ is isomorphic to A with an additional bottom element $\emptyset$, hence it does not give any additional information, whereas, if A is not totally ordered by $+$, then application of the Floyd-Warshall algorithm to the Hoare powerdomain computes the costs of all paths. To reduce

[4] Remember that $x \leq y \Leftrightarrow \exists z. x + z = y$.

the computational cost of the algorithm, it is possible to represent each $X \in \wp^H(A)$ with the set of its local maxima which may be small with respect to X. Then the Floyd-Warshall algorithm computes the costs of all the *non-dominated* paths between two nodes, namely all the paths which are maximal and are not comparable according to the order induced by $+$. Once the costs of all the non-dominated paths out of a node s have been computed, to trace an actual path of cost κ to node t it is sufficient to find an edge out of s with cost κ_1 connected to a node r having a path of cost κ_2 to t such that $\kappa = \kappa_1 \times \kappa_2$, and then to proceed similarly from r.

10 Concluding Remarks

We have introduced a formal model that provides mechanisms to specify and reason about application-oriented QoS. We demonstrate the applicability of the approach by providing the formal modeling of KAOS QoS mechanisms.

The novelty of our proposal is given by the combination of the following ingredients:

- adopt a declarative approach to the specification of QoS attributes;
- adopt a graphical calculus to describe system evolution;
- reduce declarative QoS specification to semantic constraints of the graphical calculus.

One may wonder if this approach is too abstract and general and it does not capture the intrinsic limitations of inter-networking computations. We feel that on the one side the generality of the approach can be tamed and adapted to the needs of the various layers of applications, more powerful primitives being made available to upper layers, like *business to business* (B2B) or *computer supported collaborative work* (CSCW). On the other side, some important network technologies actually require the solution of global constraints, like modifying local router tables according to the routing update information sent by the adjacent routers.

As a future work, we plan to investigate the expressive power of the graphical model and to develop proof techniques to analyze QoS properties.

References

1. O. Angin, A. Campbell, M. Kounavis, and R. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine*, August 1998.
2. L. Bettini, M. Loreti, and R. Pugliese. An infrastructure language for open nets. In Proc. of the 2002 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications. ACM Press, 2002.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
4. S. Bistarelli, U. Montanari, and F. Rossi. Soft constraint logic programming and generalized shortest path problems. *Journal of Heuristics*, 8:25–41, 2002.

5. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wand, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, The Internet Engineering Task Force (IETF), 1998.
6. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) - version 1 functional specification.
7. L. Cardelli and R. Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.
8. I. Castellani and U. Montanari. Graph Grammars for Distributed Systems. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag, 1983.
9. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
10. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, June 2000.
11. P. Degano and U. Montanari. A model of distributed systems based of graph rewriting. *Journal of the ACM*, 34:411–449, 1987.
12. R. Floyd. Algorithm97 (shortestpath). *Communication of the ACM*, 5(6):345, 1962.
13. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.
14. D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In A. Porto and G.-C. Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer Verlag, 2000.
15. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *12th International Conference in Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 121–136, Aalborg, Denmark, 2001. Springer Verlag.
16. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.
17. IBM Software Group. Web services conceptual architecture. In *IBM White Papers*, 2000.
18. M. Koch, L. Mancini, and F. Parisi-Presicce. A formal model for role-based access control using graph transformation. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *ESORICS*, volume 1895 of *LNCS*, pages 122–139, 6th European Symposium on Research in Computer Security, 2000. Springer Verlag.
19. M. Koch, L. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the specification of access control policies. In F. Honsell and M. Lenisa, editors, *FoSSaCS*, LNCS, Foundations of Software Science and Computation Structures, 2001. Springer Verlag.
20. M. Koch and F. Parise-Presicce. Describing policies with graph constraints and rules. In A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, editors, *Graph Transformation*, volume 2505 of *LNCS*, pages 223–238, First International Conference on Graph Transformation, Barcelona, Spain, October 2002. Springer Verlag.
21. B. Koenig and U. Montanari. Observational equivalence for synchronized graph rewriting. In *Proc. TACS'01*, LNCS. Springer Verlag, 2001. To appear.
22. B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois, 2000.

23. R. Milner. *Communication and Concurrency*. Printice Hall, 1989.
24. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
25. U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer Verlag, April 1996.
26. J. Sobrinho. Algebra and algorithms for qos path computation and hop-by-hop routing in the internet. *IEEE Transactions on Networking*, 10(4):541–550, August 2002.
27. G. Winskel. Synchronization trees. *Theoretical Computer Science*, May 1985.
28. X. Xiao and L. M. Ni. Internet qos: A big picture. *IEEE Network*, 13(2):8–18, Mar 1999.
29. M. Yokoo and K. Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

ULM (*)

A Core Programming Model for Global Computing

(Extended Abstract)

Gérard Boudol

INRIA Sophia Antipolis

Abstract. We propose a programming model to address the unreliable character of accessing resources in a global computing context, focusing on giving a precise semantics for a small, yet expressive core language. To design the language, we use ideas and programming constructs from the synchronous programming style, that allow us to deal with the suspensive character of some operations, and to program reactive behaviour. We also introduce constructs for programming mobile agents, that move together with their state, which consists of a control stack and a store. This makes the access to references also potentially suspensive.

1. Introduction

According to the *global computing* vision, "*In the future most objects that we work with will be equipped with processors and embedded software* [...] *Many of these objects will be able to communicate with each other and to interact with the environment*" [15]. In a word: processors everywhere. This is gradually becoming a reality: besides computers and laptops, or personal assistants, nowadays telephones and smart cards also are equipped with processors, as well as cars, planes, audio and video devices, household appliances, etc. Then a question is: will we be able to exploit such a highly distributed computing power? How will we compute in such a context? There are clearly many new features to deal with, and many problems to address in order to be able to "compute globally". As pointed out by Cardelli [8, 9], the global computing context introduces new observables, "*so radically different from the current computational norm that they amount to a new model of computation*".

In this paper we address one specific aspect of this global computing vision, namely the fact that "*The availability and responsiveness of resources* [...] *are unpredictable and difficult to control*" [15]. This is indeed something that everybody can already experiment while browsing the web: quite often we observe that an URL appears inaccessible, for various reasons which are not always clearly identified – failure of a node, insufficient bandwidth, congestion, transient disconnection, impossibility to cross a firewall, etc. We would like to have some ability to react and take actions to bypass these obstacles. This is what Cardelli provided for us, with service combinators for scripting the activity of

(*) ULM is the acronym for the french words "*Un Langage pour la Mobilité*". Work partially supported by the MIKADO project of the IST-FET Global Computing Initiative, and the CRISS project of the ACI Sécurité Informatique.

web browsing [10]. Cardelli's combinators are quite specific, however: only the time and rate of transmission can be used as parameters for a reaction. We would like to have programming constructs to deal with more general situations where the availability and responsiveness of resources is not guaranteed. This is not provided by traditional computing models. For instance, in a LAN, the fact that some file is unreachable is considered as an error – unless this is prescribed by some security policy –, and a latency period that exceeds normal response times and communication delays is the sign of a failure. In global computing, by contrast, trying to access something which is, maybe temporarily, absent or unavailable appears to be the rule, rather than the exception. Such a failure should not be regarded as a fatal error. We should rather have means to detect and react to the lack of something.

A typical scenario where these partial failures occur is the one of *mobile code*, which is thought of as a general technique to cope with the new observable features of the global computing context [9, 13]. For instance, a mobile code (which may be code embedded in a mobile device) may fail to link with some specific libraries it might need at a visited site, if the site does not provide them. In this case, a default behaviour should be triggered. Conversely, the mobile code may be unreachable from a site which is supposed to maintain connections with it. In such a case, the user who has delegated the mobile agent, or who tries to contact a remote mobile device, for instance, has to wait and, after a while, take some decisions.

The model we shall use to deal with absence, waiting, and reactions is basically that of *synchronous programming* [2, 14]. This does not mean that, in our approach, something like the web is to be regarded as a synchronous system. We rather take the view that the global computing world is a (wide) GALS, that is, a "Globally Asynchronous, Locally Synchronous" network. Only at a local level does a notion of time make sense, which serves as the basis for taking decisions to react. As regards the way to program reactions, we follow the ESTEREL imperative (or control-oriented) style [3], which seems more appropriate for our purpose than the LUSTRE data flow style [14]. However, in ESTEREL a program is not always "causally correct", and a static analysis is needed to check correctness. This checking is not compositional, and more specifically, the parallel composition of causally correct programs is not always correct. From the point of view of global computing, this is unfortunate, because this makes it impossible to dynamically add new components, like mobile agents, to a system. Therefore, we shall use the *reactive* variant of ESTEREL designed by Boussinot [5, 6], which is slightly less expressive, but is well suited for dynamically evolving concurrent systems. The main ingredients of the synchronous/reactive model, in its control-oriented incarnation – as we see it – are the following:

- *broadcast signals.* Program components react according to the absence or presence of signals, by computing, and emitting signals. These signals are broadcast to all the components of a synchronous area, which is a limited area where a local control over all the components exists.
- *suspension.* Program components may be in a suspended state, either

because of the presence of a signal, or because they are waiting for a signal which is absent at the moment.

• *preemption*. There are means to abort the execution of a program component, depending on the presence or absence of a signal.

• *instants*. These are successive periods of the execution of the program, where the signals are consistently seen as present or absent by all the components.

Suspension is what we need to signify the lack of something, and preemption allows us to program reactive behaviours, aborting suspended parts of a program in order to trigger alternative tasks ([1]). Notice that a notion of time – the instants – is needed to give meaning to actions depending on the absence of a signal: otherwise, when could we decide that we have waited for too long, and that the signal is to be considered as really absent? This last notion of an instant is certainly the main innovation of the synchronous programming model, and also the less easy to grasp. In our formalization, an "instant", or, better, a *time slot* – or simply a slot –, is an interval in the execution of a program, possibly involving many (micro-)steps, the end of which is determined by the fact that all the components are either terminated or suspended, and therefore unable to activate any other component. Moreover, only at the very beginning of an "instant" interactions with the environment may take place. By contrast with ESTEREL, in the reactive programming style of Boussinot, the absence of a signal can only be determined at the end of the "instant", and therefore one cannot suspend a component on the presence of a signal (that is, run it only if the signal is absent), and one can only abort its execution at the end of a time slot. The following example illustrates the use of reactive programming in a global computing perspective:

$$\mathsf{agent}\ \lambda a.\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ \mathsf{thread}\ \lambda t.\mathsf{if}\ \mathsf{present}\ s\ \mathsf{then}\ ()\ \mathsf{else}\ (\mathsf{migrate\_to}\ \ell)a;\ P\ ;\mathsf{emit}\ s$$

This is the code of an agent, named a, that, in some site, tries to executes a program P for one time slot. It spawns a thread (named t) that terminates if executing P succeeds, and otherwise moves the agent elsewhere. To trigger the migration, we use a local signal s which is only present when the task P terminates. One can also easily program a similar migration behaviour triggered by the presence of a failure signal emitted in a site:

$$\mathsf{agent}\ \lambda a.(\mathsf{thread}\ \lambda t.(\mathsf{when}\ \mathit{failure})(\mathsf{migrate\_to}\ \ell)a)\ ;\ P$$

Here the migration instruction is suspended, by means of the when construct, until the *failure* signal is emitted.

The main novelty in this paper is in the way we deal with the *state*, and more generally the context of a program. Apart from the reactive constructs, we adopt a fairly conventional style of programming, say the ML or SCHEME imperative and functional style, where a memory (or heap) is associated with a program, to store mutable values. Then, in the presence of mobile code, a

([1]) This is surely not new from a system or network programming point of view, where one needs to deal with suspended or stuck tasks. However, the relevant mechanisms are usually not integrated in high-level programming languages.

problem arises: when a code migrates, what happens with the portion of the store it may share with other, mobile or immobile components? There are three conceivable solutions:

- *network references.* The owner of a reference (a pointer) keeps it with him, and remote accesses are managed by means of proxies, forwarding the requests through the network.
- *distributed references.* A shared reference is duplicated and each copy is directly accessible, by local operations. Its value is supposed to be the same in all the copies.
- *mobile references.* The owner of a reference keeps it with him, and all non local accesses to the reference fail.

All the three solutions have their own flaws, the last one being the simplest to implement, but also the worse, according to the current computing model: it amounts to systematically introducing dangling pointers upon migration, thus causing run-time errors. Indeed, as far as I can see, this solution has never been used. Regarding the first, the construction and use of chains of forwarders may be costly, but, more importantly, network references rely on the implicit assumption that the network is reliable, in the sense that a distant site is always accessible (otherwise this solution is as bad as the last one). As we have seen, this assumption does not hold in a global computing context, where it is, for instance, perfectly normal for a mobile device to be sometimes disconnected from a given network. Similarly, maintaining the coherence of replicated references in a large network may be very difficult and costly, especially if connections may be temporarily broken. Moreover, with mobile code, the conflicts among several values for a given reference are not easy to solve: if a user possesses such a reference, and launches several agents updating it, which is the right value?

We adopt mobile references in our programming model for global computing, but with an important new twist: trying to access an absent reference is not an error, it is a suspended operation, like waiting for an absent signal in synchronous/reactive programming. Then the reactive part of the model allows us to deal with programs suspended on absent references, exactly as with signals. Although we do not investigate them in this paper, we think that for many other aspects of distributed and global computing, a programming model taking into account suspension as an observable, and offering preemption constructs, is indeed relevant. For instance, we will not consider communications between nodes of a network (apart from the asynchronous migration of code), but it should be clear that, from a local point of view, remote communication operations should have a suspensive character. In particular, this should be the case in an implementation of network references. Similarly, we do not consider the dynamic linking phase that an agent should perform upon migration (we assume that the sites provide the implementation of the constructs of our core model), but clearly an agent using some (supposedly ubiquitous) libraries for instance should involve some backup behaviour in case linking fails. In fact, in a global computing perspective, every operation engaging some code in an interaction with its environment should, from the point of view of this code, be

considered as potentially suspensive. Therefore we think that, although they have been developed for radically different purposes – namely, hard real-time computations, and circuit design – the ideas of synchronous programming that we integrate in our ULM core programming model should prove also useful in the highly interactive and "odd-time" world of global computing.

2. The Computing Model

2.1 Asynchronous Networks of Synchronous Machines

Since the focus in this paper is on local reactive programming, and mobility, we have only minimal requirements on the network model. We assume that computing takes place in a network which is a collection of named nodes (or sites, or localities). Besides the nodes, the network also contains "packets", which merely consist here of frozen programs asynchronously migrating to some destination node. Then, assuming given a set $\mathcal{L}$ of node names, we describe a network using the following syntax:

$$R \ ::= \ \ell[\mathcal{M}] \ \mid \ \ell\langle p\rangle \ \mid \ (R_0 \parallel R_1)$$

where ℓ is any node name. We denote by $\ell[\mathcal{M}]$ a site, named ℓ, containing the configuration $\mathcal{M}$ of a reactive machine, while $\ell\langle p\rangle$ is a packet, en route to the site called ℓ. The syntax of the content of packets, as well as machine configurations, will be given below, and we shall see in a next section how packets are introduced in the network. We denote by $(R_0 \parallel R_1)$ the juxtaposition of the nodes of the two networks R_0 and R_1. We make the assumption that the sets of node names – that is, ℓ in $\ell[\mathcal{M}]$ – are disjoint when joining two networks. This is supposed to be guaranteed by some name service.

As we said in the introduction, our model is the one of a GALS network, a concept that we formalize as follows. The behaviour exhibited by a network is represented by transitions $R \to R'$, and is determined by the behaviour of its nodes. To account for the asynchronous character of network computations, we assume that juxtaposition is associative and commutative. (This also means that all the nodes, and packets, are considered here as neighbours, with no security barrier filtering communication, for instance.) Then, denoting by $\equiv$ the least congruence on networks for which $\parallel$ is associative and commutative, we have:

$$\frac{R_0 \to R_0'}{(R_0 \parallel R_1) \to (R_0' \parallel R_1)} \ (\textsc{Ntwk1}) \qquad \frac{R_0 \to R_0' \quad R_1 \equiv R_0 \quad R_1' \equiv R_0'}{R_1 \to R_1'} \ (\textsc{Ntwk2})$$

Now let us say a few words about the "locally synchronous" aspect of a GALS network. In Section 2.3 we shall define transitions between machine configurations, and we shall see that the behaviour of a synchronous machine, starting from an initial configuration $\mathcal{M}_0$, can be described as a sequence of *time slots*:

$$\begin{array}{lllll} \ell[\mathcal{M}_0] & \to \cdots \to & \ell[\mathcal{M}_0'] & \to \ell[\mathcal{M}_1] & \textit{first slot} \\ \ell[\mathcal{M}_1] & \to \cdots \to & \ell[\mathcal{M}_1'] & \to \ell[\mathcal{M}_2] & \textit{second slot} \\ & & \vdots & & \\ & & & & \textit{and so on.} \end{array}$$

$$
\begin{array}{lcl}
M,\ N\ldots & ::= & V \mid (MN) \mid \mathsf{sig} \\
V,\ W\ldots & ::= & \ell \mid x \mid \lambda x M \mid () \\
& \mid & \mathsf{ref} \mid \mathbf{?} \mid \mathsf{set} \mid (\mathsf{set}\, V) \\
& \mid & \mathsf{emit} \mid \mathsf{when} \mid (\mathsf{when}\, V) \mid \mathsf{watch} \mid (\mathsf{watch}\, V) \\
& \mid & \mathsf{thread} \mid \mathsf{agent} \mid \mathsf{migrate\_to} \mid (\mathsf{migrate\_to}\, V)
\end{array}
$$

Figure 1: Syntax

where the transitions $\ell[\mathcal{M}_i] \xrightarrow{*} \ell[\mathcal{M}'_i]$ represent purely local (micro)-steps to a "stable" configuration, and the last step $\ell[\mathcal{M}'_i] \rightarrow \ell[\mathcal{M}_{i+1}]$ is a special "*tick*" transition. During the local steps of computation, the machine behaves in isolation from its environment. This normally ends up with a state $\mathcal{M}'_i$ where all the concurrent threads running in the machine are either terminated or suspended, waiting for some information, so that no further local computation may occur. Then all the interactions with the environment take place, during the "tick" transition, where the machine by itself does not perform any computation. Therefore one can say that during a time slot, the threads running in the machine have a coherent vision of the outside world. In this paper the interactions between a machine and its environment only consist in the management of migration – incorporating immigrant agents, and sending emigrating agents as packets in the network. However, one could easily imagine other forms of interactions, like receiving input signals or sending calls to remote procedures.

2.2 The Language: Syntax

As we suggested above, a reactive machine $\mathcal{M}$ runs a collection of concurrent threads – among which mobile threads, that we call agents –, organized in a queue which we denote by T. These threads share a common store S and a context (a set) of signals E. Moreover, the configuration of the machine records a set P of emigrating agents. That is, a machine configuration has the form

$$\mathcal{M} = (S, E, T, P)$$

In this section we introduce the core language used to program agents and threads. As in ML or SCHEME, this core language is a call-by-value λ-calculus, enriched with imperative and reactive programming constructs, and with constructs to create and manage threads, including mobile ones. The syntax is given in Figure 1. Let us comment briefly on the primitive constructs of the language.

• Regarding the λ-calculus part of the language, we shall use the standard abbreviations and notations. We denote $\lambda x M$ by $\lambda.M$ when x is not free in M. The capture avoiding substitution of N for x in M is denoted $\{x \mapsto N\}M$.

• The imperative constructs, that is ref, **?** and set are quite standard, except that we denote by **?** the dereferencing operation (which is denoted **!** in ML), to emphasize the fact that this is a potentially suspensive operation. As usual, we also write $M := N$ for $(\mathsf{set}\, M)N$.

• The intuitive semantics of the reactive constructs sig, emit, when and watch is as follows. The constant sig creates, much like the ref construct, a new signal,

that is a name. The created signal is initially absent, and one uses the emit function to make a signal present for the current time slot. The function when is intended to be applied to two arguments, the first one being a signal. It *suspends* the evaluation of its second argument when the first one is absent. The preemption function watch is also a function of a signal and an expression. At the end of the current slot, it *aborts* the execution of its second argument if the watched signal is present.

• The thread function spawns a new thread, as well as the agent function which, in addition, creates an agent name. Agent names are used by the migrate_to function to cause the migration of the designated thread.

As we have suggested in these informal explanations, the evaluation of several constructs of the language, namely ref, sig and agent (and also thread) will create various kinds of names at run-time. These names are values in an extended, run-time language. They are defined as follows. First, we assume given a denumerable set $\mathcal{N}$ of names, disjoint from $\mathcal{L}$. Then, besides node names in $\mathcal{L}$, used in the source language, we shall use three kinds of names:

(i) Simple names, belonging to $\mathcal{N}$ and ranged over by u, v, w .... These are used to name immobile threads, that is, the threads created using the thread construct.
(ii) Compound names of the form $\ell.u$, thus belonging to $\mathcal{L}.\mathcal{N}$. They are used to name signals, immobile references (that is, references created by an immobile thread), and agents. All these names may be imported in a node by a mobile agent, and this is why we use the name ℓ of the creation node as a prefix, to avoid conflicts whith names created elsewhere.
(iii) Compound names of the form $\ell.u.v$, belonging to $\mathcal{L}.\mathcal{N}.\mathcal{N}$. These are the names of references created by agents. As we will see, an agent will carry with it its own portion of the memory upon migration, and it is therefore convenient to distinguish the references created by an agent, by prefixing their name v with the name $\ell.u$ of their creator.

2.3 Operational Semantics

We have already described the semantics of networks, which relies on machine transitions. To introduce the latter, let us first define more precisely the components of a configuration:

• As usual, the *store* S is a mapping from a finite set $\mathsf{dom}(S) \subseteq \mathcal{L}.\mathcal{N} \cup \mathcal{L}.\mathcal{N}.\mathcal{N}$ of memory addresses to values.
• The signal environment $E \subseteq \mathcal{L}.\mathcal{N}$ is the set of names of signals that are *present* during the current time slot.
• The threads queue T is a sequence of named programs M^t, where M is an expression of the run-time language and t is the name of a thread executing M (so that $t \in \mathcal{N} \cup \mathcal{L}.\mathcal{N}$). We denote by $T \cdot T'$ the concatenation of two threads sequences, and by ε the empty sequence.
• The set P of emigrating agents consists in named programs together with a destination, denoted $(\mathsf{to}\,\ell)M^t$.

To run an expression M of the language in a locality ℓ, we start from an initial configuration where M is the only thread in the queue, with an arbitrary (simple)

name, and everything else is empty, that is:

$$\ell[\emptyset, \emptyset, M^u, \emptyset]$$

For the sake of exposition, we distinguish four kinds of machine transitions. These transitions generally concern the head of the thread queue (on the left), that is the code M if $T = M^t \cdot T'$. First, we have purely functional transitions that occur in the active code part M of a configuration, and are essentially independent from the rest, like β-reduction. Then we have the imperative and reactive behaviour, where the code M interacts with the store S or the signal environment E. Next there are transitions having an effect on the thread queue, like thread creation, migration, and scheduling. Last but not least, there are transitions from one time slot to the next. As we said, all interactions with the "outside world" take place at this moment, and in particular the P part of the configuration is only considered at this point. The functional transitions, denoted $M \to_f M'$, are given by the following axioms:

$$(\lambda x M V) \ \to_f \ \{x \mapsto V\}M \qquad (\beta_{\mathrm{v}})$$

$$((\mathsf{when}\, W)V) \ \to_f \ V \qquad (\textsc{When})$$

$$((\mathsf{watch}\, W)V) \ \to_f \ () \qquad (\textsc{Watch})$$

The two transitions regarding when and watch mean that, when the code they control terminates, then the whole expression terminates.

To define the imperative and reactive transitions, acting on tuples (S, E, M^t), we need some auxiliary notions. First, we shall use the standard notion of an *evaluation context*, analogous to a "control stack". Evaluation contexts in our language are defined by the following grammar:

$$\mathbf{E} \ ::= \ [] \ \mid \ (\mathbf{E}N) \ \mid \ (V\mathbf{E})$$

Besides the evaluation contexts, we need the predicate $(S, E, M)\,\dagger$ of *suspension*. In our language, there are two ways in which a code M may be suspended in the context of a store S and a signal environment E: either M tries to access, for reading or writing it, a reference which is not in the domain of the store, or M waits for a signal which is absent. This is easily formalized, see [4].

To abbreviate the notations, in the rules below we will use the predicate $(E, \mathbf{E})\not\downarrow$, meaning that, given the signal environment E, the evaluation context $\mathbf{E}$ is not suspensive, that is, if $\mathbf{E} = \mathbf{E}_0[((\mathsf{when}\, s)\mathbf{E}_1)]$ then $s \in E$. In the rules we also need, in some cases, the knowledge of the name of the locality where they occur. Therefore the imperative and reactive transitions are of the form

$$\vdash_\ell (S, E, M^t) \to_{ir} (S', E', M'^t)$$

For lack of space, we omit here some of the rules of the operational semantics. The complete design has to be found in [4]. For instance, there is a rule by which functional transitions are allowed in any non suspensive context. Then we have rules to create a reference and enlarge the store, and to get or update the value of a reference. These are more or less standard, except that dereferencing and updating are suspensive operations when the reference is not in the domain of

the store (again, see [4] for the details). The creation of a signal is similar to the creation of a reference, that is, a fresh signal name is returned as a value:

$$\frac{(E,\mathbf{E}) \not\downarrow \qquad \ell.u \text{ fresh}}{\vdash_\ell (S,E,\mathbf{E}[\mathsf{sig}]^t) \rightarrow_{ir} (S,E,\mathbf{E}[\ell.u]^t)} \quad (\textsc{Sig})$$

Notice that the signal is still absent, that is $\ell.u \notin E$. The signal is present, for the rest of the current slot, when it has been emitted:

$$\frac{(E,\mathbf{E}) \not\downarrow}{\vdash_\ell (S,E,\mathbf{E}[(\mathsf{emit}\ s)]^t) \rightarrow_{ir} (S,E \cup \{s\},\mathbf{E}[()]^t)} \quad (\textsc{Emit})$$

One can see that, following these transitions, it may happen that the evaluation of M gets suspended at some point. In this case, we have to look for other threads to execute, or to notice that the current slot has ended. This is decribed by the next rules, for transitions of the form

$$\ell[\mathcal{M}] \rightarrow \ell[\mathcal{M}']$$

that we mentioned in Section 2.1. A first rule says that an imperative or reactive transition is also a machine transition, if this transition is performed by the first thread in the queue (see [4]). Then, when the first thread is suspended, it is put at the end of the queue, provided that there is still some computation to perform, that is, there is another thread in the queue which is not terminated, and not suspended. By abuse of notation, we shall denote by $(S,E,T) \not\downarrow$ the fact that there is a thread in the queue T which is active in the context of S and E. The rule for scheduling the queue of threads is as follows:

$$\frac{\neg(S,E,M^t) \not\downarrow \qquad (S,E,T) \not\downarrow}{\ell[S,E,M^t \cdot T,P] \rightarrow \ell[S,E,T \cdot M^t,P]} \quad (\textsc{Sched})$$

We have adopted a round-robin scheduling, placing the current thread at the end of the queue when it is terminated or suspended. We could obviously use any other (fair) strategy, but we think it is important, in a language with effects, to have a deterministic strategy, in order to have some understanding of the semantics. One may notice that if a thread is suspended, waiting for a signal on a $(\mathsf{when}\ s)$ statement, and if this signal is emitted during the current time slot, then the thread will always be activated during this slot.

It remains to see how threads are created, and how migration is managed. The thread function takes a thunk – that is, a frozen expression $\lambda.M$ – as argument, creates a new thread, the body of which is the unfrozen thunk, and terminates. The body of the created thread is subject to the control (by signals, that is by means of when and watch functions) exercized by the context that created it. To formulate this, let us define the control context $\mathbf{E}^w$ extracted from an evaluation context $\mathbf{E}$ as follows:

$$\begin{aligned}
\Box^w &= \Box \\
(\mathbf{E}N)^w &= \mathbf{E}^w \\
(V\mathbf{E})^w &= \begin{cases} (V\mathbf{E}^w) & \text{if } V = (\mathsf{when}\ W) \text{ or } V = (\mathsf{watch}\ W) \\ \mathbf{E}^w & \text{otherwise} \end{cases}
\end{aligned}$$

One can see that $\mathbf{E}^w$ is a sequence of $(\mathsf{when}\, W)$ and $(\mathsf{watch}\, W)$ statements. The rule for thread creation is as follows:

$$\frac{(E, \mathbf{E}) \not\downarrow}{\ell[S, E, \mathbf{E}[(\mathsf{thread}\, V)]^t \cdot T, P] \to \ell[S, E, \mathbf{E}[()]^t \cdot T \cdot \mathbf{E}^w[(V\,())]^u, P]} \quad (\textsc{Thread})$$

Notice that the newly created thread is put at the end of the queue, and therefore it cannot prevent the existing threads to execute, since the thread queue is scanned from left to right by the scheduler.

The agent function also creates threads, but with a different semantics. It is intended to have as argument a function of an agent name, say $\lambda x N$. Then $(\mathsf{agent}\, \lambda x N)$ creates a new agent name, returned as a value, and a new thread, the body of which is N where x is instantiated by the created agent name. Unlike with threads created by thread, the control exercized by the creating context is not transferred on the created agent. The idea is that a thread, created by thread, is intended to cooperate with its creation context in the achievement of some task, while an agent is thought of as detached from a main task, and is intended to move (maybe just to escape from abortion of the main task). The rule is:

$$\frac{(E, \mathbf{E}) \not\downarrow \qquad \ell.u \text{ fresh}}{\ell[S, E, \mathbf{E}[(\mathsf{agent}\, V)]^t \cdot T, P] \to \ell[S, E, \mathbf{E}[\ell.u]^t \cdot T \cdot (V\ell.u)^{\ell.u}, P]} \quad (\textsc{Agent})$$

As one can see, the created name $\ell.u$ is known from both the creating context, and the agent itself. This name could be used to control the behaviour of the agent in various ways. We could for instance have primitive constructs to kill, or resume/suspend the execution of an agent. In this paper, the only construction of this kind we consider is the statement causing the migration of an agent. When we have to execute a $(\mathsf{migrate\_to}\, \ell')t$ instruction, where ℓ' is a node name and t is a thread name (it should be guaranteed by typing that migrate_to only applies to such values), we look in the thread queue for a component named t, and put it, with destination ℓ', in the set of emigrating agents:

$$\frac{(E, \mathbf{E}) \not\downarrow \qquad \mathbf{E}[()]^r \cdot T = T' \cdot N^t \cdot T''}{\ell[S, E, \mathbf{E}[((\mathsf{migrate\_to}\, \ell')t)]^r \cdot T, P] \to \ell[S, E, T' \cdot T'', P \cup \{(\mathsf{to}\, \ell')N^t\}]} \quad (\textsc{Migr})$$

(There is also a rule in the case where the agent is not there, which is omitted.) If $t = r$, we have $N = \mathbf{E}[()]$, and we say that the migration is subjective, since it is performed by the agent itself. Otherwise, that is if the migration instruction is exercized by another thread, we say that the migration is objective. We also observe that (subjective) migration in our model may be qualified as "strong mobility" [13], since the "control stack", that is the evaluation context $\mathbf{E}$ of the migration instruction, is moving (together with the memory of the agent, as we shall see).

The last rule describes what happens at the end of a time slot, which is also the beginning of the next one – we call this the *tick* transition. One can see that no rule applies if the thread queue only consists of terminated or suspended

threads. In this case, that is when $\neg(S, E, T)\,\lightning$, the current slot is terminated, and the computation restarts for a new one, after the following actions have been done:

• The signal environment is reset to the empty set ([2]).

• The preemption actions take place. That is, a sub-expression $((\mathsf{watch}\, s)M)$ (in a non suspensive context) terminates if the signal s was present in the signal environment at the end of the slot. Otherwise – that is, if s is absent –, this expression is resumed for the next slot, as well as the when guards.

• The immigrant agents, that is the contents of network packets $\ell\langle p\rangle$, are incorporated. In a more realistic language, one should have a buffering mechanism, involving a security check, to receive the asynchronously incoming agents. The contents p of a packet is a pair (S, M^t) of a store and a named program. The intuition is that M is the code of a migrating agent, and S is its own, local store. Then the code M is put at the end of the threads queue, and the S store is added to the one of the host machine (thus possibly activating threads suspended on trying to access an absent reference).

• The emigrant agents, in P, are sent in the outside world. More precisely, an element $(\mathsf{to}\,\ell')N^t$ of P is transformed into a packet $\ell'\langle S', N^t\rangle$ where S' is the portion of the store that is owned by the agent N^t. This portion is removed from the store of the locality where the tick transition is performed. Notice that a machine waits until it reaches a stable configuration to send the emigrating agents in the network. This is to maintain a coherent view of the store, as far as the presence or absence of references is concerned, during each time slot (notice also that the other threads may modify the store carried by the agent, even after it has been put in the P part of the configuration).

The precise formulation of the "tick" transition is given in [4].

3. Some Examples

In this section, we examine some examples, to illustrate the expressive power of the constructs of the language. We begin with examples related to the reactive programming primitives. There has been many variations, sometimes just notational, in the choice of primitives in synchronous, imperative programming. To obtain a "minimal" language, suited for formal reasoning, we have chosen to have only two primitives for reactive programming in ULM, one for introducing suspension, and the other for programming preemption. However, the other constructs, functional and imperative, of the language allow us to encode most of the standard constructs of synchronous/reactive programming. Let us see this now.

First we notice that our preemption construct $(\mathsf{watch}\, s)M$ exists in some versions of ESTEREL [2], where it is written (abort M when s), and also in SL [6], where it is written (do M kill s). In synchronous programming, there is a statement, called stop in [6] and pause in [2] (it is not a primitive, but is easily derived in the language of [3]), which lasts just one instant, that is, in our setting,

([2]) One could also imagine that signals could be emitted from the external environment. In this case, they would be incorporated as input signals for the next slot, but only during the "tick" transition.

which is suspended until the end of the current time slot, and terminates at the tick transition. To express this in ULM, it is convenient to first introduce a now construction which, when applied to a thunk, executes it for the current time slot only, so that (now M) always terminates at the end of the slot, at the latest. This is easy to code: we just have to preempt the argument of now on a signal that is immediately emitted:

$$\mathsf{now} =_{\mathrm{def}} \lambda x(\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ (\mathsf{watch}\, s)(\mathsf{emit}\, s\, ;\, x()))$$

Another useful notation, that exists in ESTEREL (see [2]) is

$$\mathsf{await} =_{\mathrm{def}} \lambda s.(\mathsf{when}\, s)()$$

Then the pause construct is easily defined, as waiting for an absent signal, for the current slot only:

$$\mathsf{pause} =_{\mathrm{def}} (\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ (\mathsf{now}\, \lambda.\mathsf{await}\, s))$$

Using a pause statement, a thread suspends itself until the next time slot. We can also define a statement by which a thread just gives its turn, as if it were put at the end of the queue. This is easily done by spawning a new thread (thus put at the end of the queue) that emits a signal on which the "self-suspending" thread waits:

$$\mathsf{yield} =_{\mathrm{def}} (\mathsf{let}\ x = \mathsf{sig}\ \mathsf{in}\ (\mathsf{thread}\ \lambda.\mathsf{emit}\ x)\ ;\ \mathsf{await}\ x)$$

We can also use the now construct to define a present predicate whose value, when applied to a signal, is true (assuming that the language is extended with truth values tt and ff) if the signal is present, and false otherwise. Notice that since we can only determine that a signal is absent at the end of the current time slot, the value false can only be returned at the beginning of the next slot. The code is, using a reference to record the presence or absence of the signal:

$$\begin{array}{lll} \mathsf{present} & =_{\mathrm{def}} & \lambda s\, \mathsf{let}\ r = \mathsf{ref}\, ff \\ & & \quad \mathsf{in}\ (\mathsf{now}\, \lambda.(\mathsf{when}\, s)\, r := tt)\ ;\ ?r \end{array}$$

In a conditional construct if present s then M else N, the code M is started in the current slot if s is present, and otherwise N is executed at the next one.

To finish with the examples, let us briefly discuss the mobile code aspect of the language. Imagine that we want to send, from a main program at site ℓ, an agent to a site ℓ', with the task of collecting some information there and coming back to communicate this to the sender (like we would do with a proxy for a network reference, for instance). We cannot write this as

$$\mathsf{let}\ r = (\mathsf{ref}\, X)\ \mathsf{in}\ \mathsf{migrate\_to}\, \ell'(\mathsf{agent}\, \lambda a(\cdots r := Y\ ;\ (\mathsf{migrate\_to}\, \ell)a))\ ;\ \cdots\ ?r\ \cdots$$

(which is an example of both objective and subjective migration) for two reasons: after some steps, where r is given a value $\ell.u$, that is an address in the store at site ℓ containing the value X, and where a name $\ell.v$ is given for a, the agent runs at ℓ' the code

$$\cdots\ \ell.u := Y\ ;\ (\mathsf{migrate\_to}\, \ell)\ell.v$$

Since the address $\ell.u$ is created by an immobile thread, it will always remain in the store at ℓ, and will never move. Therefore, the agent is blocked in ℓ' waiting

for the reference $\ell.u$ to be there. On the other hand, the "main thread" at ℓ may perform $?\ell.u$, but it will get the value X, which is certainly not what is intended. We have to use a signal to synchronize the operations, like with:

$$\begin{array}{l}\mathsf{let}\ r = (\mathsf{ref}\ X)\ \mathsf{and}\ s = \mathsf{sig}\\ \mathsf{in}\ \mathsf{let}\ \mathit{write}\ =\ \lambda x.r := x\ ;\ \mathsf{emit}\ s\ \mathsf{and}\\ \qquad\quad \mathit{read}\ =\ \mathsf{await}\ s\ ;\ ?r\\ \quad \mathsf{in}\ \cdots\end{array}$$

writing the body of the agent as:

$$\cdots \mathsf{let}\ y = Y\ \mathsf{in}\ (\mathsf{migrate\_to}\ \ell)a\ ;\ \mathit{write}\ y$$

One can see that the main thread cannot read a value from the reference r before the signal s has been set as present, which only occurs (provided that r and s are private to the functions *write* and *read*) when a *write* operation has occurred. This example shows that mobile computing should be "location aware", at least at some low level, as envisioned by Cardelli [9]. This example also shows that an object-oriented style of programming would be useful to complement mobile agent programming, to define synchronization and communication structures, such as the one we just introduced in this example, of a signal with a value, that we may call an *event*. We are currently investigating this.

4. Conclusion

In this paper we have introduced a computing model to deal with some new observables arising in a global computing context, and especially the unreliability of resource accesses resulting from the mobility of code. For this purpose, we have used some ideas of synchronous programming, and most notably the idea that the behaviour of a program can be seen as a succession of "instants", within which a reaction to the suspension of some operations can be elaborated. This is what we need to give a precise meaning to statements like "abort the computation C if it gets suspended for too long, and run C' instead", that we would like to encode as an algorithmic behaviour.

During the last few years of the last century, a lot of proposals have been made to offer support for mobile code (see for instance the surveys [13, 19], and the articles in [20]). As far as I can see none of these proposals addresses the unreliablity issue as we did (and most often they lack a clear and precise semantics). Most of these proposals are assuming, like Cardelli's OBLIQ [7], that a distributed scope abstraction holds, in the sense that the access to a remote resource is transparent, and does not differ, from the programmer's point of view, from a local access. The JoCaml language for instance [12] is based on this assumption. However, we think that this abstraction can only be maintained in a local network. Indeed, Cardelli later argued in [9] that global computing should be location aware, and should also take into account the fact that in a global context, failures become indistinguishable from long delays. The Sumatra proposal [1] acknowledges the need "to be able to react to changes in resource availability", a feature which is called "agility", and is implemented

by means of exception handling mechanisms. However, as far as I can see, no precise semantics is given for what is meant by "react quickly to asynchronous events" for instance. That is, a precise notion of time is missing.

Softwares and languages that are based on LINDA, such as JAVASPACES [18], LIME [16] or KLAIM [11], or on the π-calculus, like the JOCAML language [12] or NOMADICPICT [17], all involve an implicit notion of suspension: waiting for a matching tuple to input in the case of LINDA, waiting for a message on an input channel in the case of π. However, they all lack preemption mechanisms, and a notion of time (indeed, the semantics of these models is usually asynchronous). Summarizing, we can conclude that our proposal appears to be the first (apart from [10]) that addresses, at the programming language level, and with a formal semantics, the issue of reacting to an unreliable computing context. Clearly, a lot of work is to be done to develop this proposal. In the full version of this paper [4], we show how to predict, by a static analysis, that some uses of references may be considered as safe, that is, they do not need testing the presence of the reference. Then, we should extend ULM into a more realistic language, and provide a proof-of-concept implementation. We should also investigate the network communication aspect (network references, RPC for instance), and the dynamic linking of mobile agents, that we did not consider. Last but not least, we should investigate (some of) the (numerous) security issues raised by the mobility of code. (A formal semantics for the mobile code, as the one we designed, is clearly a prerequisite to tackling seriously these issues.) One may have noticed for instance that in our model, a thread may prevent the other components to compute, simply by running forever, without terminating or getting suspended. Clearly, this is not an acceptable behaviour for an incoming agent. We are currently studying means to restrict the agents to always have a cooperative behaviour, in the sense that they only perform, at each time slot, a finite, predictable number of transitions.

References

[1] A. ACHARYA, M. RANGANATHAN, J. SALTZ, *Sumatra: a language for resource-aware mobile programs,* in [20] (1997) 111-130.

[2] A. BENVENISTE, P. CASPI, S. A. EDWARDS, N. HALBWACHS, P. Le GUERNIC, R. de SIMONE, *The synchronous languages twelve years later,* Proc. of the IEEE, Special Issue on the Modeling and Design of Embedded Software, Vol. 91 No. 1 (2003) 64-83.

[3] G. BERRY, G. GONTHIER, *The ESTEREL synchronous programming language: design, semantics, implementation,* Sci. of Comput. Programming Vol. 19 (1992) 87-152.

[4] G. BOUDOL, *ULM, A core programming model for global computing,* available from the author's web page (2003).

[5] F. Boussinot, *La Programmation Réactive – Application aux Systèmes Communicants,* Masson, Coll. Technique et Scientifique des Télécommunications (1996).

[6] F. Boussinot, R. de Simone, *The SL synchronous language,* IEEE Trans. on Software Engineering Vol. 22, No. 4 (1996) 256-266.

[7] L. Cardelli, *A language with distributed scope,* Computing Systems Vol. 8, No. 1 (1995) 27-59.

[8] L. Cardelli, *Global computation,* ACM SIGPLAN Notices Vol. 32 No. 1 (1997) 66-68.

[9] L. Cardelli, *Wide area computation,* ICALP'99, Lecture Notes in Comput. Sci. 1644 (1999) 10-24.

[10] L. Cardelli, R. Dawies, *Service combinators for web computing,* IEEE Trans. on Software Engineering Vol. 25 No. 3 (1999) 303-316.

[11] R. De Nicola, G. Ferrari, R. Pugliese, *KLAIM: a kernel langage for agents interaction and mobility,* IEEE Trans. on Software Engineering Vol. 24, No. 5 (1998) 315-330.

[12] C. Fournet, F. Le Fessant, L. Maranget, A. Schmitt, *JoCaml: a language for concurrent, distributed and mobile programming,* Summer School on Advanced Functional Programming, Lecture Notes in Comput. Sci. 2638 (2003) 129-158.

[13] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding code mobility,* IEEE Trans. on Software Engineering, Vol. 24, No. 5 (1998) 342-361.

[14] N. Halbwachs, *Synchronous Programming of Reactive Systems,* Kluwer (1993).

[15] The IST-FET Global Computing Initiative, `http://www.cordis.lu/ist/fet/gc.htm` (2001).

[16] G.P. Picco, A.L. Murphy, G.-C. Roman, *LIME: Linda meets mobility,* ACM Intern. Conf. on Software Engineering (1999) 368-377.

[17] P. Sewell, P. Wojciechowski, *Nomadic Pict: language and infrastructure design for mobile agents,* IEEE Concurrency Vol. 8 No. 2 (2000) 42-52.

[18] Sun Microsystems, *JavaSpaces Service Specification* `http://www.sun.com/jini/specs` (2000).

[19] T. Thorn, *Programming languages for mobile code,* ACM Computing Surveys Vol. 29, No. 3 (1997) 213-239.

[20] J. Vitek, Ch. Tschudin, Eds, *Mobile Object Systems: Towards the Programmable Internet,* Lecture Notes in Comput. Sci. 1222 (1997).