Encapsulation and Dynamic Modularity in the Pi-Calculus

Daniel Hirschkoff, Aurélien Pardon, Tom Hirschowitz, Samuel Hym, Damien Pous

Encapsulation and Dynamic Modularity in the π-calculus

Daniel Hirschkoff, Aurélien Pardon

ENS Lyon, LIP, Université de Lyon, CNRS, INRIA

Tom Hirschowitz

LAMA, Université de Savoie, CNRS

Samuel Hym

LIFL, Université de Lille 1, INRIA, CNRS

Damien Pous

SARDES, LIG, Grenoble, CNRS, INRIA

Abstract

We describe a process calculus featuring high level constructs for component-oriented programming in a distributed setting. We propose an extension of the higher-order π-calculus intended to capture several important mechanisms related to component-based programming, such as dynamic update, reconfiguration and code migration. In this paper, we are primarily concerned with the possibility to build a distributed implementation of our calculus. Accordingly, we define a low-level calculus, that describes how the high-level constructs are implemented, as well as details of the data structures manipulated at runtime. We also discuss current and future directions of research in relation to our analysis of component-based programming.

Keywords: Distributed computing, component-based programming, process algebra, higher-order calculi, abstract machine.

1 A Core Calculus for Dynamic Modularity

1.1 Motivations of this Work

This paper describes work on component-oriented programming and the π-calculus. Our long term goal is the design and implementation of a prototype programming language meeting the following requirements.

⋆ This work has been supported by the French project ANR Arassia *"Modularité Dynamique Fiable"*.

- It should be suitable for *concurrent, distributed programming*. For instance, usual distributed, parallel algorithms should be easily implementable, as well as lower-level communication infrastructure for networks. Furthermore, it should enjoy a well-understood and tractable behavioural theory.
- It should provide constructs for *modularity*, in the standard, informal sense that programs should be built as an assembly of relatively independent computation units (or *modules*) interacting at explicit interfaces. Moreover, modularity should come with *encapsulation* features, e.g., it should be possible to exchange two modules implementing the same interface without affecting the rest of the code.
- The modular structure of programs should be available at execution time, so as to ease standard dynamic operations such as migration, dynamic update, or passivation of modules. We call this requirement *dynamic modularity*. The notion of dynamic modularity gathers the most challenging features of component based programming we are interested in modelling and analysing.
- Finally, we are seeking a reasonably implementable language, at least permitting rapid prototyping of distributed applications.

In this paper, we describe our proposal for a core process algebra to represent and analyse dynamic modularity (this section). In designing this formal model, we have put a stress on the possibility to deploy and execute processes in a distributed fashion. In Sect. 2, we describe our prototype implementation by defining an abstract machine for the distributed execution of processes. This description is given via the definition of a new process calculus, where low-level aspects related to the implementation (notably, communication protocols), as well as the data structures at work in the machine, are made explicit.

1.2 Syntax and Semantics of $k\pi$

In order to provide a formal treatment of the questions described above, we study an extension of the higher-order π-calculus, called $k\pi$. We choose the π-calculus for two main reasons: first, message-based concurrency seems an appropriate choice to define a model for concurrent programming at a reasonable level of abstraction. Second, working in the setting of a process algebra like the π-calculus makes it possible to define a core formalism in which we can analyse the main questions, both theoretical and pragmatic, related to the implementation of primitives for dynamic modularity. Third, we might hope for this to benefit from the considerable amount of research that has been made on π-calculus based formalisms.

Our calculus inherits ideas from numerous previous studies, among which [8,7,3], and in particular the Kell calculus [2,14]. The grammar for processes is given in Fig. 1 – we suppose two infinite sets of names (a, b, m, n, k, x) and process variables (X, Y). In addition to the usual π-calculus constructs, we have *modules*, $n[P]$, which can be seen as located processes (note that modules can be nested). $M.P$ is a process willing to emit (first- or higher-order) message M and then proceed as P. $R \triangleright P$ stands for a process willing to acquire a resource: this can mean either receiving a first- or higher-order message (cases $a(x)$ and $a(X)$, respectively), or

$$
\begin{array}{rcll}
P, Q & ::= & P \mid P \mid (\boldsymbol{\nu} n)P \mid \mathbf{0} \mid n[P] \mid n[X] \mid M.P \mid R \triangleright P & \text{(processes)} \\
M & ::= & \overline{a}\langle n \rangle \mid \overline{a}\langle P \rangle \mid \overline{a}\langle X \rangle & \text{(output prefixes)} \\
R & ::= & a(x) \mid a(X) \mid n[X] & \text{(input prefixes)} \\
\mathbb{E} & ::= & [] \mid \mathbb{E} \mid P \mid (\boldsymbol{\nu} n)\mathbb{E} \mid n[\mathbb{E}] & \text{(evaluation contexts)}
\end{array}
$$

Fig. 1. $k\pi$: Syntax for processes, evaluation contexts

$$
\text{K-Comm}\ \frac{a, b \notin \mathrm{capt}(\mathbb{E}_1) \cup \mathrm{capt}(\mathbb{E}_2)}{\mathbb{E}[\mathbb{E}_1[\overline{a}\langle b \rangle.P] \mid \mathbb{E}_2[a(x) \triangleright Q]] \rightarrow \mathbb{E}[\mathbb{E}_1[P] \mid \mathbb{E}_2[Q\{b/x\}]]}
$$

$$
\text{K-CommHO}\ \frac{\mathrm{fn}(P') \cap (\mathrm{capt}(\mathbb{E}_1) \cup \mathrm{capt}(\mathbb{E}_2)) = \emptyset}{\mathbb{E}[\mathbb{E}_1[\overline{a}\langle P' \rangle.P] \mid \mathbb{E}_2[a(X) \triangleright Q]] \rightarrow \mathbb{E}[\mathbb{E}_1[P] \mid \mathbb{E}_2[Q\{P'/X\}]]}
$$

$$
\text{K-Pass}\ \frac{}{\mathbb{E}[n[P] \mid n[X] \triangleright Q] \rightarrow \mathbb{E}[Q\{P/X\}]}
$$

$$
\text{K-Congr}\ \frac{P' \equiv P \qquad P \rightarrow Q \qquad Q \equiv Q'}{P' \rightarrow Q'}
$$

Fig. 2. Operational Semantics of $k\pi$

passivating a module. The input prefixes and restriction are binding constructs, and we write $\mathrm{fn}(P)$ for the free names of process P. $\{b/x\}$ (resp. $\{P/X\}$) denotes the capture avoiding substitution of name x (resp. process variable X) with name b (resp. process P).

Notice that a process variable (X) does not form a process by itself: it has to be enclosed in a module ($n[X]$) or a message ($\overline{a}\langle X \rangle.P$). This restriction mainly ensures that the content of several modules cannot be merged into a single module; this helps simplifying the implementation of the calculus.

Fig. 2 presents the rules defining the reduction relation, $\rightarrow$. It makes use of evaluation contexts, which are processes with a hole that does not occur under a prefix (Fig. 1). Given an evaluation context $\mathbb{E}$, we define the set of *captured names* in $\mathbb{E}$, written $\mathrm{capt}(\mathbb{E})$, as the set of names that are bound at the occurrence of the hole in $\mathbb{E}$. The definition of $\rightarrow$ makes use of structural congruence, $\equiv$, the least congruence satisfying the following axioms:

$$
P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R
$$

$$
(\boldsymbol{\nu} c)(\boldsymbol{\nu} d)P \equiv (\boldsymbol{\nu} d)(\boldsymbol{\nu} c)P \qquad (\boldsymbol{\nu} c)(P|Q) \equiv P \mid (\boldsymbol{\nu} c)Q \text{ if } c \notin \mathrm{fn}(P) \qquad (\boldsymbol{\nu} c)\mathbf{0} \equiv \mathbf{0}
$$

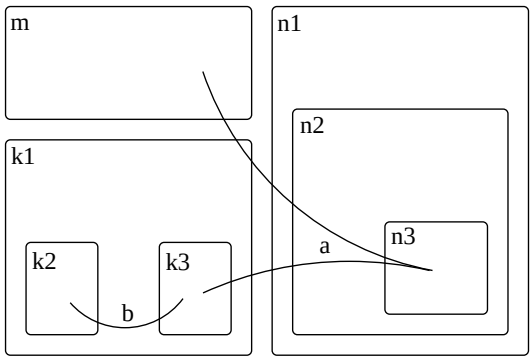

Fig. 3. A configuration.

An example $k\pi$ process.

A $k\pi$ term represents a configuration consisting of a hierarchy of modules, in which processes are executed. To illustrate the mechanisms at work in $k\pi$, we briefly discuss an example configuration. Given some $k\pi$ processes P, P'_i, Q_i Fig. 3 depicts a process of the form

$$(\boldsymbol{\nu} a)(m[P] \mid n_1[P'_1 \mid n_2[P'_2 \mid n_3[P'_3]]] \mid k_1[(\boldsymbol{\nu} b)(Q_1 \mid k_2[Q_2] \mid k_3[Q_3])])$$

(note that the localisation of the restrictions on a and b does not appear on the picture). There are basically two forms of interaction in $k\pi$: *communication* and *passivation*. Communication involves the transmission of a name or a process; it is *distant*, in the sense that a process $\overline{a}\langle b\rangle.P$ can synchronise with a receiver $a(x) \triangleright Q$ sitting in a different location, provided they share the name a. On the picture, a process running in k_3 can exchange messages with another one in n_3 (using channel a), as well as with a third process running in k_2 (using channel b). On the contrary, passivation is local: only a process running at n_1 is able to passivate module n_2. This is described by the third axiom of Fig. 2: up to structural congruence, the module being passivated and the process that takes control over it must be in parallel. Passivation is a central construct in our formalism, and can be used to implement very different kinds of manipulations related to dynamic modularity. For instance, taking $Q = n'[X]$ in the above reduction leads to a simple operation of module renaming; with $Q = \overline{c}\langle X\rangle$, the module n will be 'frozen' and marshalled into a message to be sent on channel c; finally, taking $Q = n[X] \,|\, n'[X]$ makes it possible to duplicate a computation. When considering the actual implementation of the behaviour of $k\pi$ processes, it appears that the last two examples clearly involve much more costly operations than the first one.

1.3 On the Design Choices in the Definition of $k\pi$

1.3.1 Restricted names as localised resources

An important commitment that we make in the design of $k\pi$ is that, contrarily to several existing proposals, we do *not* allow channel names to be extruded across module boundaries: neither do we include an axiom of the form $n[(\boldsymbol{\nu} b)\, P] \equiv (\boldsymbol{\nu} b)\, n[P]$ in structural congruence, nor do we implement name extrusion across

modules along reduction steps that would require it.

This design choice is related to the notion of module that we put forward in $k\pi$: if we were to allow name extrusion, some processes would admit two legitimate but very different reduction paths. Take indeed

$$P \quad \stackrel{\text{def}}{=} \quad m[(\boldsymbol{\nu} a)\overline{b}\langle a\rangle.P] \mid b(x).Q \mid m[X] \triangleright (m_1[X] \mid m_2[X]) \ .$$

We would have the following sequences of reductions:

$$\begin{aligned} P &\to (\boldsymbol{\nu} a)(m[P] \mid Q\{a/x\} \mid m[X] \triangleright (m_1[X] \mid m_2[X])) \\ &\to (\boldsymbol{\nu} a)(Q\{a/x\} \mid m_1[P] \mid m_2[P]) \\ &\text{and} \\ P &\to b(x).Q \mid m_1[(\boldsymbol{\nu} a)\overline{b}\langle a\rangle.P] \mid m_2[(\boldsymbol{\nu} a)\overline{b}\langle a\rangle.P] \\ &\to (\boldsymbol{\nu} a)(Q\{a/x\} \mid m_1[P]) \mid m_2[(\boldsymbol{\nu} a)\overline{b}\langle a\rangle.P] \end{aligned}$$

We claim that none of these paths is fully satisfactory, and that such a situation should be avoided – the choice between these two behaviours being left to the user, through explicit programming. Therefore, we interpret the names declared inside a module as private resources, that should remain local to that module. Passivating module n hence means getting hold of the local computations, as well as of the resources allocated in n. Typically, names allocated in module n can be viewed either as temporary resources allocated for the computations taking place at n, or as methods provided for sub-modules of n, for which n acts as a library.

As a consequence, the user is made aware of the localisation of resources; this choice also helps considerably in the implementation of $k\pi$, essentially because we always know how to route messages to channels (see below; a similar idea is present in existing implementations of π-calculus-related process algebras, such as [4]). At the same time, this hinders the expressiveness of message passing: a process willing to send a name n outside the module where the restriction on n is hosted is stuck. Consequently, for two distant agents to share a common name, this name should be allocated at a place that is visible for both, i.e., above them in the hierarchy of modules. In other words, extrusion is not transparent to the user, and has to be programmed when necessary. Of course, there are situations where one would like to allocate a new name outside the current module. It turns out that a corresponding primitive for remote allocation, $\boldsymbol{\nu} n@m$, can be added at small cost to our implementation (Sect. 2).

Experiments with examples written in $k\pi$ show that the idioms we would like to be able to program are compatible with the discipline we enforce in our formalism. Further investigations need to be made, in particular with larger examples, in order to understand the possibilities offered by programming in $k\pi$.

1.3.2 Modularity vs. physical deployment

It should be noticed that the hierarchy of modules described by a $k\pi$ process does not necessarily correspond to a given mapping from modules to physical sites. We

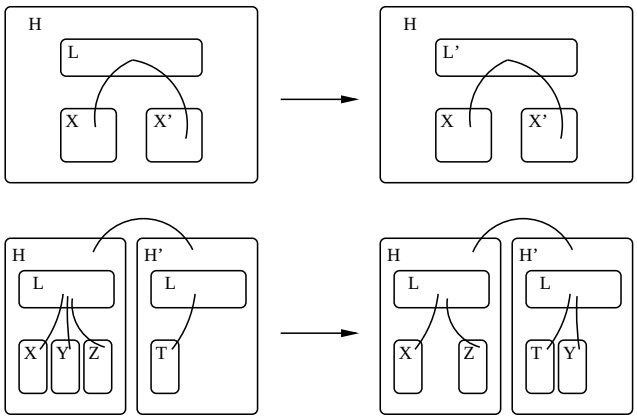

Fig. 4. Some $k\pi$ configurations

show on Fig. 4 some examples illustrating how we intend to use $k\pi$ processes to express typical situations in distributed programming. In the top left diagram, a host module H contains a library module L, as well as two other sub-modules X and X'. X and X' are connected with module L via some names residing at H. Physically, this can well correspond to a situation where H and L are run on the same site (the site that provides the library), while X and X' are on (possibly distinct) distant machines, and rely on remote communication to interact with L. On the top right diagram, a process in H has passivated L (H and L are co-located, physically) to replace it with an updated version of the library (L').

The bottom left diagram depicts a different scenario, based on the same "modularity pattern": here, several sites provide the services of library L, and welcome the users of L. In this configuration, all sub-modules of a given module are executed on the same machine. It might be the case, for load balancing purposes, that one host passivates a local client to send it over the network to another location: this is illustrated on the bottom right diagram.

2 A Distributed Implementation

In this section, we describe a distributed abstract machine that implements $k\pi$. This machine abstracts from issues such as data representation, to focus on the implementation of distributed communication in the presence of passivation. The design of this machine has been tested on a prototypical distributed implementation, so as to make sure that our implementation choices are reasonable (see [9]).

2.1 Implementation Choices

Before moving to the presentation of the formal definition of our abstract machine, we give a high-level view of how it works. This should help in following the more technical explanations we give in Sec. 2.2.

Computation units.

The first (standard) feature of our machine is that it flattens the hierarchy of computation units: for example, each of the seven modules composing the program of Fig. 3 is executed in its own asynchronous *location* by the machine. In order to retain the tree structure, each location stores and maintains the list of its children locations. As expected, when a module creates a sub-module, the latter is spawned in a fresh location. In order to make sure that locations can be implemented in an asynchronous way, we let them interact only by means of (asynchronous) messages.

Communications.

The protocol for distant communication is rather standard: a process willing to send a message sends it to the location holding the *queue* that implements the channel; accordingly, a process willing to receive a message sends its location to the queue location and suspends execution, so that it can be awaken when a message is available. Using our interpretation of modules, the natural location to run the channel queue is that of the module that created the name; this is made possible by our choice to prevent the channel name from being extruded out of this module: if the module gets passivated, all of its sub-modules too, so that any process trying to communicate on the channel will get passivated as well.

Passivation.

Passivation cannot be atomic, because the hierarchy of modules has been flattened, as described above (this departs importantly from the machine in [1]). Thus, we implement it in an incremental fashion, from the passivated module down to its sub-modules, transitively. Along the propagation of a passivation session, we must handle two main sources of interferences.

- First, in the case where some sub-module has already started a passivation, our machine gives priority to the inner passivation session – the other option would have been to cancel the latter and let the dominating passivation proceed instead.
- More importantly, we need to clean up running communication sessions in the passivated sub-modules. As explained above, this is not problematic for communications on names belonging to the passivated module. For communications on names that reside *above* the passivated module, we use simple interactions with the modules owning the involved channels: status messages are used to query whether commitment to a communication already occurred, so that the computation can be either completed or aborted.

Distributed Implementation.

We have written an OCaml implementation of this abstract machine [9]. This implementation exploits two libraries: one for high-level communications, where message passing is executed either as memory write-ups or as socket communication, depending on whether it is local or distant; and another one for communication, thunkification and spawning of OCaml threads, together with their sets of defined

$$
\begin{array}{rr}
a, n, x, u, h, g, s \in Names & \text{(Names)} \\
X \in PVars & \text{(Process Variables)} \\
\hline
B ::= (x) \mid (X) \mid [X] \mid \langle a \rangle \mid \langle P \rangle \mid \langle X \rangle & \text{(Binder/Argument)} \\
P ::= \mathbf{0} \mid P \mid P \mid (\boldsymbol{\nu} a) P \mid n[X] \mid aB \triangleright P & \textbf{(Process)} \\
\hline
V ::= u^g \mid K \mid \bullet \mid \neg & \text{(Value)} \\
G ::= \mathbf{0} \mid G \mid G \mid s(h, u, V) \mid s(?) & \text{(Request)} \\
I ::= P \mid I \mid I \mid \omega^s(aB \triangleright P, M) \mid \alpha^s(V) \mid \kappa_i^h(I) & \text{(Local state)} \\
\rho ::= \emptyset \mid \rho, a \mapsto u^g \mid \rho, X \mapsto K & \text{(Table)} \\
K ::= I \; ; \rho \; ; g & \text{(Thunk)} \\
M ::= \mathbf{0} \mid h\langle I \rangle \mid g\langle G \rangle & \text{(Messages)} \\
S ::= M \mid S \mid S \mid (\boldsymbol{\nu} a) S \mid h[K] \mid g[G] & \textbf{(State)} \\
\hline
\mathbb{H}_h^\rho ::= h[[] \mid I \; ; \rho \; ; g] & \text{(Contexts)} \\
\mathbb{G}_g ::= g[[] \mid G] &
\end{array}
$$

Fig. 5. Syntax of the abstract machine.

names (to optimise the process of passivation, the data structure implementing a module comes with a table collecting all names known by the module). Finally, each syntactic construction of the language is compiled into a simple function that uses the previous libraries along the lines of the formal specification of the machine we describe below. In particular, we do not need to manipulate explicit abstract syntax trees at runtime.

2.2 Formal Definition of the Abstract Machine

The abstract machine for the distributed execution of $k\pi$ processes is defined as a process calculus, where details about the data structures we use in the implementation and about the protocols at work are made apparent.

Configurations of the Abstract Machine.

Fig. 5 presents the grammar for the abstract machine. As previously, we rely on two infinite sets of names ($Names$) and process variables ($PVar$); names will be used for $k\pi$ processes as well as for various internal identifiers required by the machine: allocated name identifiers (u), logical locations (g, h), and session identifiers (s). Moreover, we let i range over relative numbers ($\mathbb{Z}$). Although they are presented in a slightly different way, processes (P) of the machine are essentially the same as previously: the only difference is that we allow modules to contain process variables only (the entry $n[P]$ disappeared, only $n[X]$ is left). To obey this constraint, we recursively define a pre-compilation step $\wr . \int$ mapping processes of the calculus (as defined in Fig. 1) to those of the machine (Fig. 5); the only non-trivial

case is:

$$\wr n[P]\wr = (\boldsymbol{\nu} a)(a\langle\wr P\wr\rangle \triangleright 0 \mid a(X) \triangleright n[X])$$

(where a is a fresh name). The final translation from closed source processes to states is the following, where h and g are two arbitrary distinct names:

$$[\![P]\!] = (\boldsymbol{\nu} h, g)(h[\wr P\wr\ ;\ \emptyset\ ;\ g] \mid g[\mathbf{0}])\ .$$

At runtime, this state will evolve to a state where each module (node) of the tree structure of P is represented by a configuration of the form $h[K] \mid g[G]$, where $K = I\ ;\ \rho\ ;\ g$ is a *thunk*, and:

- h is a low-level *logical* location; several such locations can be executed on the same *physical* location;
- I is the local process being executed, together with some state information;
- g is the address of another associated low-level logical location, where names allocated by the module will be handled (with the help of G) – we call g the *top-level communication channel* of K; this channel is recorded in K in order to allocate new names.
- ρ is a *table* (or environment) binding process variables to thunks, and names to pairs of the form $u^{g'}$: u is a unique identifier and g' is the channel where the name is handled.

In the OCaml implementation h and g correspond to two channels (implemented either with shared memory or with sockets, depending on the physical locations of the endpoints). In turn, K and G correspond to two threads, listening to those channels. Messages transmitted on these channels are of the forms I and G, respectively.

We explain the various syntactic entries together with the operational semantics, given below. Tables are seen as partial functions from names and process variables to values; these functions are recursively extended to binders and arguments (entry B in the grammar) as follows:

$$\begin{aligned} \rho(\langle a\rangle) &= \rho(a) \\ \rho(\langle P\rangle) &= (P\ ;\ \rho\ ;\ g) \quad (\text{for some } g \notin \mathrm{fn}(\rho)) \\ \rho(\langle X\rangle) &= \rho(X) \\ \rho((x)) = \rho((X)) &= \bullet \\ \rho([X]) &\ \text{ is undefined} \end{aligned}$$

The following operation will be useful in order to extend tables:

$$\rho + B \mapsto V = \begin{cases} \rho, x \mapsto u^g & \text{if } B = (x) \text{ and } V = u^g\,; \\ \rho, X \mapsto K & \text{if } B = (X) \text{ or } B = [X], \text{ and } V = K\,; \\ \rho & \text{otherwise.} \end{cases}$$

We will also need the following substitution operation on thunks:

$$K\{g\} = I\{g/g'\}\ ;\ \rho\{g/g'\}\ ;\ g \quad \text{where } K = I\ ;\ \rho\ ;\ g' \ .$$

Operational semantics.

We have enough material to explain the operational semantics of the abstract machine, which is presented in Fig. 6. The first three rules are standard; structural congruence is not defined here, it contains no surprise: structural congruence of source $k\pi$ processes is inherited; parallel compositions and $\mathbf{0}$ elements form associative commutative monoids; and alpha-conversion and scope-extrusion are allowed. We explain the other reduction rules below; they make use of contexts ($\mathbb{H}^{\rho}_{h}$, $\mathbb{G}_{g}$ – see Fig. 5) in order to focus on single logical locations (we omit ρ, h or g when they are not relevant for the rule).

- Communication.
  - · In rule I-Req, a prefixed process is executed by sending a request $s(h, u, V)$ (for reception or for emission, according to the shape of B) on the channel that handles the communications taking place on a. A fresh session identifier, s, allows us to uniquely identify this part of the communication protocol. The process waiting to resume computation is stored locally in an element of the shape $\omega^s(aB \triangleright P, M)$; message M will be sent in case a passivation occurs before the communication actually take place – we explain this protocol below. Notice that this rule does not apply if B is of the form $[X]$: in this case, $\rho(B)$ is undefined. This situation correspond to a passivation prefix, which is handled by rule I-StartPass.
  - · Rule I-Compl describes how a successful completion message unleashes the continuation of a prefix action: a message $\alpha^s(V)$ meets an element $\omega^s(\dots)$ with the same session identifier (s): the continuation process (P) is released, and, in case we are on the side of the receiver, a new binding is generated and added to the table of the local process (ρ).
  - · In rule I-Abort, a message of the shape $\alpha^s(\neg)$ tells the frozen process to undo its commitment: the original prefixed process is installed again. As we shall see below, this may occur in case of a passivation.
- Routing.
  - · Rules I-Route and I-Route' are used to transmit messages on both kinds of channels to their actual destination channel.
  - · Rule I-Comm describes how two requests for emission and reception, respectively, occurring on the same name identifier u, and originating from locations h_1 and h_2, meet: downwards acknowledgment messages are generated, with the identifying session names; the content of the message is transmitted to the receiver process.
  - · Rule I-Stat shows how messages for cancelling commitments ($\alpha^s(\neg)$) are generated: this happens when a message requesting communication gets caught up by a status message $s(?)$ (see rule I-PassSess below).
- Distribution.

Contextual closure

$$\frac{S \to S'}{S_0 \mid S \to S_0 \mid S'} \qquad \frac{S \to S'}{(\boldsymbol{\nu} a)S \to (\boldsymbol{\nu} a)S'} \qquad \frac{S \equiv S_0 \qquad S_0 \to S_0' \qquad S_0' \equiv S'}{S \to S'}$$

Communication

$$\text{I-Req}\ \frac{\rho(a) = u^g \qquad \rho(B) = V \qquad s \notin \mathrm{fn}(\mathbb{H}_h^\rho[aB \triangleright P])}{\mathbb{H}_h^\rho[aB \triangleright P] \to (\boldsymbol{\nu} s)\big(g\langle s(h,u,V)\rangle \mid \mathbb{H}_h^\rho[\omega^s(aB \triangleright P, g\langle s(?)\rangle)]\big)}$$

$$\text{I-Compl}\ \frac{V \neq \neg \qquad \rho' = \rho + B \mapsto V}{\mathbb{H}^\rho[\omega^s(aB \triangleright P, \_) \mid \alpha^s(V)] \to \mathbb{H}^{\rho'}[P]} \qquad \text{I-Abort}\ \frac{}{\mathbb{H}[\omega^s(aB \triangleright P, \_) \mid \alpha^s(\neg)] \to \mathbb{H}[aB \triangleright P]}$$

Routing

$$\text{I-Route}\ \frac{}{\mathbb{G}_g[\mathbf{0}] \mid g\langle G\rangle \to \mathbb{G}_g[G]} \qquad \text{I-Route'}\ \frac{}{\mathbb{H}_h[\mathbf{0}] \mid h\langle I\rangle \to \mathbb{H}_h[I]}$$

$$\text{I-Comm}\ \frac{}{\mathbb{G}[s_1(h_1,u,V) \mid s_2(h_2,u,\bullet)] \to \mathbb{G}[\mathbf{0}] \mid h_1\langle \alpha^{s_1}(\bullet)\rangle \mid h_2\langle \alpha^{s_2}(V)\rangle}$$

$$\text{I-Stat}\ \frac{}{\mathbb{G}[s(h,u,V) \mid s(?)] \to \mathbb{G}[\mathbf{0}] \mid h\langle \alpha^s(\neg)\rangle}$$

Distribution

$$\text{I-Fresh}\ \frac{a, u \notin \mathrm{fn}(h[I \mid (\boldsymbol{\nu} a)P \;;\; \rho \;;\; g])}{h[I \mid (\boldsymbol{\nu} a)P \;;\; \rho \;;\; g] \to (\boldsymbol{\nu} u)h[I \mid P \;;\; \rho, a \mapsto u^g \;;\; g]}$$

$$\text{I-Spawn}\ \frac{\rho(X) = K \qquad h, g \notin \mathrm{fn}(\mathbb{H}_{h'}^\rho[n[X]])}{\mathbb{H}_{h'}^\rho[n[X]] \to (\boldsymbol{\nu} hg)\big(h[K\{g\}] \mid g[\mathbf{0}] \mid \mathbb{H}_{h'}^\rho[\omega^h(n[X] \triangleright n[X],\ h\langle \kappa_0^{h'}(\mathbf{0})\rangle)]\big)}$$

Passivation

$$\text{I-StartPass}\ \frac{M \neq \mathbf{0}}{\mathbb{H}[n[X] \triangleright P \mid \omega^h(n[X] \triangleright n[X],\ M)] \to \mathbb{H}[\omega^h(n[X] \triangleright P, \mathbf{0})] \mid M}$$

$$\text{I-PassSess}\ \frac{M \neq \mathbf{0}}{\mathbb{H}[\kappa_i^h(I) \mid \omega^s(aB \triangleright P, M)] \to \mathbb{H}[\kappa_{i+1}^h(I \mid \omega^s(aB \triangleright P, \mathbf{0}))] \mid M}$$

$$\text{I-Decr}\ \frac{}{\mathbb{H}[\kappa_i^h(I) \mid \alpha(V)] \to \mathbb{H}[\kappa_{i-1}^h(I \mid \alpha(V))]} \qquad \text{I-Pack}\ \frac{}{h'[P \mid \kappa_0^h(I) \;;\; \rho \;;\; g] \to h\langle \alpha^{h'}(P|I \;;\; \rho \;;\; g)\rangle}$$

Fig. 6. Operational semantics for the abstract machine.

- Rule I-Fresh shows how $k\pi$ names are allocated: a new identifier u is generated, and we store in the local table the information that communications on name a are handled with identifier u by the top-level communication channel g, which is associated to the current location (h).
- Rule I-Spawn is the most complicated rule: it spawns a new location, in order to execute a sub-module of the current module. As explained above, this involves generation of two channels, one (h) for executing the code of the sub-module, and another (g) to handle communications on channels allocated by that sub-module (g is the top-level communication channel).

A sub-module waiting to be executed is of the form $n[X]$. The local table (ρ) tells which thunkified computation is associated to it (K). This thunk has been previously linked to a top-level communication channel, that was used to

handle communication channels (K might be a process that has been running and allocating names for some time, before being passivated). Since this channel cannot be used anymore (because the thunk may have moved physically, or been replicated) a new channel (g) and a new thread ($g[\mathbf{0}]$) have to be created, and we need to replace the old address with g in all tables that can be found in K, whence the update $K\{g\}$. Note that in the actual implementation, thanks to the use of tables, applying this substitution does involve inspecting the code at runtime: we only perform an operation on tables, which are runtime data structures to supervise the execution of the code.

The tricky part of the rule is the element $\omega^h(n[X] \triangleright n[X],\ h\langle\kappa_0^{h'}(\mathbf{0})\rangle)$ which is left in the parent location after the sub-module has been spawned. This element allows us to recall that h is a child of h' in the original tree hierarchy; it will be used in two ways, in case the parent location wants to passivate this child, or gets passivated. These two cases respectively correspond to the rules (I-StartPass,I-PassSess) presented below.

- Passivation.
  - Rule I-StartPass initiates a passivation protocol: the current module wants to passivate a sub-module named n, and we know that there is such a sub-module thanks to the element of the form $\omega(n[X] \triangleright n[X],\ M)$. It suffices to release message M: according to rule I-Spawn, M is of the form $h\langle\kappa_0^{h'}(\mathbf{0})\rangle$, where h is the location of the child, and we shall see in the remaining rules that upon reception of that message, the sub-module will passivate itself and send its thunkified version back to h', the parent location. Moreover, we update the ω-element so that it will yield the expected result upon reception of the passivated child, through rule I-Compl. Notice that it is important to check that $M \neq \mathbf{0}$: otherwise, in case P is of the form $n[X]$, we could incorrectly apply the rule twice.
  - Passivation gets propagated to the whole sub-tree by rule I-PassSess: the κ-element acts as a buffer that records the current state of the passivation, and accepts any element of the form $\omega(\_, M)$ by sending message M. This may correspond to two situations:
    - the ω-element denotes a sub-module, in which case the message tells the sub-module to passivate itself, recursively;
    - the ω-element corresponds to a pending request for communication. In this case, due to rule I-Req, M is of the form $g\langle s(?)\rangle$: by sending this *status* message, we force g to answer the request (if the completion message was just sent, nothing happens, the status message gets lost; otherwise, rule I-Stat applies, yielding an abortion message $h\langle s(\neg)\rangle$).

    In both cases, we increment the counter stored in the κ-element: we need to record that we have to wait for an answer (an α-message).
  - Rule I-Decr allows the κ-element to consume those answers, by buffering them, and decrementing its counter.
  - Finally, rule I-Pack can be used after the whole state has been cleaned: only a source process, P, remains in the location; all ω and α-elements have been

integrated to the κ-buffer, and the κ-counter is null. The buffer, the remaining source process, the current table and the address of the current name handler, g, are sent back to the parent location (h), and the thread running at h is killed (we could also send a message to the associated channel (g) in order to kill the corresponding thread – we omit this garbage collection optimisation for the sake of simplicity).

2.3 Related works: abstract machines for distributed computation

Several works on the implementation of process calculi for distributed programming have focused on Ambient-like models (Mobile [5], Safe [6,10], Boxed [13]). We have already mentioned [1], which introduces an abstract machine for the Kell calculus. The most important difference between this work and the previous ones is the presence of the passivation operator in the calculus, which is the source of delicate questions when it comes to actual implementation. $k\pi$ is related to the Kell calculus. The machine we have introduced provides a more fine-grained presentation of distributed interaction; in particular, passivation involves a complex distributed protocol, while it is atomic in [1].

We come back to these related studies in the next section, when discussing the formal validation of our machine.

3 Concluding Remarks

The process of the definition of $k\pi$ and the study of its implementation have raised several questions, that we are currently investigating, or that we want to address in the future.

3.1 Ongoing Work

A type system to prevent illegal name extrusions.

In order for the machine to work correctly, we need to make sure that names are not extruded outside their defining module. A solution would be to inspect the content of each message at runtime, and to block illegal communications. To avoid the inefficiencies induced by this approach, we want to rely on a type system to enforce statically this confinement policy: a well-typed term never attempts to extrude a name out of its scope.

The type system we are studying exploits an analysis of the hierarchy of modules to detect ill-formed communications. An output $\overline{a}\langle n\rangle$ is licit only if the restriction binding n is above the one binding a in the structure of modules (or if n is free in the process). If, instead of a or n, we have names bound to be received (as in, e.g., $c(x) \triangleright \overline{a}\langle x\rangle$), then the type information associated to the transmitting channels gives an approximation of the module where the names being communicated are allocated (intuitively, this information boils down to *"name x is allocated above module named m and under module named k"* – both pieces of information are necessary, because the name instanciating x may then be used either as medium

or as object of communication). The communication of process values follows the same ideas: in $\overline{a}\langle P\rangle$, we impose that all free names of P should be allocated above a. Consequently, in the type of both module names and of channels over which processes are transmitted, we provide a spatial bound of this kind on the free names of the process being executed (resp. communicated).

In addition to the standard property of subject reduction, correctness of our type system is expressed by showing that every typeable process is *well scoped*, which intuitively means that such a process does not attempt to emit a name outside its scope. Since this property is preserved by reduction, we can avoid checking for scope extrusions at run time. Although we need to experiment further with the expressiveness of our type system, preliminary attempts show that the policy enforced by our system is reasonable, in the sense that the examples we have in mind can be typed in a rather natural way. We defer the presentation of the type system, as well as the corresponding correctness proof, to a future presentation of our work.

Correctness proof for the abstract machine.

The abstract machine of Sect. 2 provides a rather low-level description of how $k\pi$ processes should be executed in a distributed setting. Proving its correctness, i.e., that the result of the compilation exhibits the same behaviour as the original source process, is a challenging task. Examples like [5,10] illustrate this — on the contrary, machines like the ones in [2,13], by providing a more high-level account of the implementation, make it possible to build simpler correctness proofs. The main difficulty is that proofs of this kind tend to be a really large piece of mathematics; appropriate techniques are necessary to render them more tractable, in order to be able to complete them. This is the case in our setting, notably because the passivation mechanism brings several technical subtleties.

The reductions of a $k\pi$ process and the execution of a machine state are described by two transition systems. We could hope to establish a *bisimulation* result, providing evidence that the compiled version essentially exhibits the same behaviour as the source process. However, because passivation is not atomic in our setting (contrarily to [1]), this is not possible. Indeed, consider the following process: $m[\,a\langle u\rangle \mid n[b\langle v\rangle]\,] \mid m[X] \triangleright Q$. The actual execution of the passivation of m may go through a state where the emission on b is blocked while the one on a is still active; such a state has no counterpart in the original calculus. Instead, correctness of our machine should be stated as a *coupled bisimulation* result [12]: although this behavioural equivalence is weaker than plain bisimulation, it entails operational equivalence (any $k\pi$ reduction step can be simulated by the machine, and any reduction step of the machine can be completed into a step of the calculus). It can be noted that the correctness proof for the machine of [13], which comes in two parts (soundness and completeness), is conceptually close to a 2-simulation result, which in turn is obtained by dropping a clause from the definition of coupled simulation.

Optimisations of the machine.

The definition and implementation of the abstract machine plays an essential role in the design of $k\pi$, because it provides practical insight on the main design decisions behind the formal model. In addition to that, the implementation also suggests several improvements or extensions, that we would like to study further. We have already mentioned the primitive for remote name allocation $\boldsymbol{\nu} n@m$, an operation that in principle can be encoded, but comes at a very low cost as a primitive, given the current design of the machine. Another direction worth investigating is how the general behaviour of the machine can be specialised by taking into account information such as, e.g., the fact that a whole module hierarchy runs on a single machine.

Module Interfaces.

As it is, the type system we have sketched above associates rudimentary information to a module name, which is only related to the regions accessed by processes running within this module (properties like *"this module has only access to references situated above module m"*). It would be interesting to define more informative module interfaces, that would in particular describe some aspects of the behaviour associated to the usage of the names hosted by the module (as well as the interfaces associated to sub-modules, recursively). One could think for example of having the type of a channel c describe the arguments that are expected to be sent on c. To go beyond that, the possibility of expressing expectations in terms of resource access and consumption would be helpful. Also, finding meaningful type disciplines to control usages of passivation is an interesting question as well (a simple example is the possibility, for implementation purposes, to be able to guarantee that a passivated process will be used in an affine way – that is, without duplication – by the passivating agent). Existing type systems for π-calculus based formalisms, as well as for object-oriented languages, should be sources of interesting ideas to investigate these issues.

Handling (re)binding.

In its current form, $k\pi$ only makes it possible to implement limited forms of dynamic modularity. When a module is passivated, it can be moved around, duplicated, and computation can be resumed, as long as the confinement constraints associated to the localisation of restrictions are respected. In writing examples in $k\pi$, it appears that it would be helpful, when passivating a module, to be able to somehow disconnect it from some of the local resources it is using. This would make it possible to send the passivated module outside the scope of the names it was using, possibly to another site where computation can be resumed after connecting again to another bunch of resources.

Extending $k\pi$ with mechanisms for dynamic (re)binding while keeping the possibility to assert statically properties of modules about their usage of resources is a

difficult task. The work on Acute [15] may provide interesting inspiration for this. From a more low-level perspective, we believe that the tables that are manipulated at run time in our machine are well-designed to support such an extension.

Behavioural equivalences.

Not only do we want to execute $k\pi$ programs, but we also would like to state and prove their properties. At a foundational level, we would be interested in analysing the notion of behavioural equivalence provided in $k\pi$, and in understanding the role of passivation in this respect. The work of [11] goes in this direction (as well as [8], in the setting of the Homer calculus).

References

[1] Bidinger, P., A. Schmitt and J. Stefani, *An Abstract Machine for the Kell Calculus*, in: *In Proc. FMOODS '05*, LNCS **3535** (2005), pp. 31–46.

[2] Bidinger, P. and J. Stefani, *The Kell Calculus: Operational Semantics and Type System*, in: *In Proc. FMOODS '03*, LNCS **2884** (2003), pp. 109–123.

[3] Fournet, C., "The Join-Calculus: a Calculus for Distributed Mobile Programming," Ph.D. thesis, Ecole Polytechnique (1998).

[4] Fournet, C., F. Le Fessant, L. Maranget and A. Schmitt, *JoCaml: A Language for Concurrent Distributed and Mobile Programming*, in: *In Proc. Advanced Functional Programming 2002*, LNCS **2638** (2002), pp. 129–158.

[5] Fournet, C., J.-J. Lévy and A. Schmitt, *An asynchronous, distributed implementation of mobile ambients.*, in: *In Proc. IFIP TCS '00*, LNCS **1872**, 2000, pp. 348–364.

[6] Giannini, P., D. Sangiorgi and A. Valente, *Safe Ambients: Abstract machine and distributed implementation*, Sci. Comput. Program. **59** (2006), pp. 209–249.

[7] Hennessy, M., "A Distributed π-calculus," Cambridge University Press, 2007.

[8] Hildebrandt, T., J. Godskesen and M. Bundgaard, *Bisimulation Congruences for Homer — a Calculus of Higher Order Mobile Embedded Resources*, Technical Report TR-2004-52, Univ. of Copenhagen (2004).

[9] Hirschkoff, D., T. Hirschowitz, S. Hym, A. Pardon and D. Pous, *Abstract Machine for $k\pi$: Prototype Implementation*, available from `http://perso.ens-lyon.fr/damien.pous/kp/kpam.tgz` (2008).

[10] Hirschkoff, D., D. Pous and D. Sangiorgi, *A Correct Abstract Machine for Safe Ambients*, in: *In Proc. COORDINATION '05*, LNCS **3454** (2005), pp. 17–32.

[11] Lenglet, S., A. Schmitt and J.-B. Stefani, *Normal bisimulations in process calculi with passivation*, Research Report RR-6664, INRIA (2008).

[12] Parrow, J. and P. Sjödin, *Multiway synchronizaton verified with woupled simulation*, in: *In Proc. CONCUR*, 1992, pp. 518–533.

[13] Phillips, A. T., N. Yoshida and S. Eisenbach, *A distributed abstract machine for boxed ambient calculi*, in: *Proc. of ESOP'04*, Lecture Notes in Computer Science **2986** (2004), pp. 155–170.

[14] Schmitt, A. and J. Stefani, *The Kell Calculus: A Family of Higher-Order Distributed Process Calculi*, in: *In Proc. Global Computing*, LNCS **3267** (2005), pp. 146–178.

[15] Sewell, P., J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit and V. Vafeiadis, *Acute: high-level programming language design for distributed computation*, in: *In Proc. ICFP* (2005), pp. 15–26.