MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

Languages experiments v2: Programming features and their implementation

MIKADO Deliverable D3.2.2

Editor : R. DE NICOLA (U. OF FLORENCE)

Authors : L. BETTINI, D. FALASSI, R. DE NICOLA, M. LORETI (UNIV. FIRENZE),
M. LACOSTE (FRANCE TÉLÉCOM R&D)
P. BIDINGER, A. SCHMITT, J.B. STEFANI (INRIA, RHONE-ALPES)
V. VASCONCELOS (UNIV. LISBON), L. LOPES, B. TEIXEIRA (UNIV. PORTO)

Classification : Public
Deliverable no. : D3.2.2
Reference : RR/WP3/3
Date : February 2005

© **INRIA, France Telecom R&D, U. of Florence, U. of Sussex, U. of Lisbon**

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

Abstract

In this document, we briefly describe the main contribution to the deliverable on experimenting with the implementation of most of the calculi considered in the project. First, we describe how two well known calculi for mobile processes KLAIM and $D\pi$ have been implemented on the top of IMC. We then describe the implementation of the MiKO programming language, an instance of the parametric calculus introduced in the WP1 with the TyCO calculus as the content of the membrane itself. After this, we outline the description of the implementation of the abstract machine for an instance of the Kell Calculus that dedicates particular attention to the proof of its correctness. Our presentation ends with a discussion of the problem of implementing security membranes on the top of an execution platform.

Introduction

In the final year of the project, we have continued the implementation of IMC but we have also started experimenting with the implementation of most of the languages and calculi that had been considered in the few languages.

IMC is a software framework based on Java that provides the necessary tools for building infrastructures to support the development of applications for systems where mobility and network awareness are key issues. The framework enables platform designers to customize communication protocols and network architectures and guarantees transparency of name management and code mobility in distributed environments. The IMC framework has been exploited by the Firenze unit to re-engineer the implementation of KLAVA and to implement $D\pi$. The latter has been obtained by building on the top of IMC, a small software framework, called $JD\pi$, that provides the runtime environment for executing $D\pi$ programs. The implementation schema is the same as the one adopted for developing KLAIM on the top of KLAVA: $JD\pi$ is the runtime for $D\pi$ and a compiler has then been developed to transform $D\pi$ code into Java code that relies on $JD\pi$.

In parallel with the experiment of language implementation on IMC, we have considered the direct implementation of other languages. The Lisbon unit has worked on the MiKO programming language, an instance of the parametric calculus introduced in the WP1 with the TyCO calculus as the contents of the membrane itself. The Inria unit has instead worked on the implementation of an abstract machine for an instance of the Kell Calculus. The machine is structured to guarantee separation of its proof of correctness from the one of used higher-level communication and migration protocols which can be implemented on top of it.

To pave the way to the enrichment of IMC with security features, the France Telecom unit has started considering the problem of implementing security membranes on the top of an execution platform. Four main design issues have been considered: the choice of a security model; the type of architecture for the execution environment; the layer at which to place security mechanisms; and the assurance level of the platform. In each case, the trade-off between security, flexibility, simplicity, and trustworthiness has been discussed.

Outline of the deliverable

The rest of this deliverable contains four different documents.

The first document [BFDL05] , prepared by the Firenze unit, describes how IMC has been used to implement two well known calculi for mobile processes, namely KLAIM and $D\pi$. It discusses how the run time support for KLAIM has been re-implemented by using all new components of IMC. Moreover, it contains the description of the implementation of two variants of $D\pi$, one very close to the original language, the other handling also node connections and node failures.

The second document has been prepared by Lisbon unit [LTV05]. It describes the MiKO programming language which is an instance of Boudol's parametric calculus of membranes with the TyCO process calculus as the contents of the membrane. The goal of this work is to provide a prototype implementation of a membrane-based programming language that would validate the model introduced in the WP1. The target language of the actual compiler for MiKO is MIL (Multithreaded Intermediate Language, a fine grained assembly language for implementing concurrent calculi).

The third document, prepared by the INRIA unit [BSS05], describes the implementation of a member of the family of Kell process calculi. This family is intended as as basis for studying distributed component-based programming while dedicating specific attention to the proof of correctness of the implementations. The document presents an abstract machine for an instance of the calculus, a proof of its correctness, and

a prototype OCaml implementation. The originality of the abstract machine is that it does not mandate a particular physical configuration (e.g., mapping of localities to physical sites), and it is independent of any supporting network services. This allows to separate the proof of correctness of the abstract machine per se, from the proof of correctness of higher-level communication and migration protocols which can be implemented on the machine.

The last document, prepared by France Telecom [Lac05], discusses the notion of security membrane as a basic linguistic primitive of secure languages for programming global computers. Membranes separate the computational behavior of a site from the security code controlling access to the resources available at that site. The document discusses some of the challenges which arise when trying to implement security membranes on the top of an execution platform. Four main design issues are considered that are related to the choice of a security model; the type of architecture for the execution environment; the layer at which to place security mechanisms; and the assurance level of the platform. For all issues, the possible trade-offs between security, flexibility, simplicity, and trustworthiness are discussed.

References

[BFDL05] L. Bettini, D. Falassi, R. De Nicola, and M. Loreti. Implementing simple calculi with IMC. 2005.

[BSS05] P. Bidinger, A. Schmitt, and J-B. Stefani. An Abstract Machine for the Kell Calculus. 2005.

[Lac05] M. Lacoste. A Survey of Some Implementation Techniques for Security Membranes. 2005.

[LTV05] L. Lopes, B. Teixeira, and V.T. Vasconcelos. MiKO Programming Language Definition and Implementation (Progress Report). 2005.

Implementing simple calculi with IMC

L. Bettini, D. Falassi, R. De Nicola, M. Loreti
Dipartimento di Sistemi ed Informatica, Universitá di Firenze

Abstract

In this document we briefly describe how IMC has been used to implement two well known calculi for mobile processes. First, we describe how the run time support for KLAIM has been re-implemented by using some components of IMC; the previous implementation was only taking advantage of the mobility package. Then, we describe the actual implementation of two variant of $D\pi$, one very close to the original language, another one considering handling of node connections and the possibility of connection failures.

1 Re-implementing KLAVA using IMC

X-KLAIM [2, 5, 6] is a mobile code programming language where communication takes place through multiple tuple spaces (as in Linda [11]) distributed on sites; sites are accessible through their localities (e.g., IP addresses). The reserved locality `self` can be used to access the local execution site. A tuple t can be inserted in a tuple space located at locality ℓ with the operation **out**$(t)@\ell$ and removed (resp. read) with **in** (resp. **read**). Pattern matching is used for selecting tuples. X-KLAIM is based on the kernel language KLAIM [9].

The implementation of X-KLAIM is based on KLAVA [7], a Java package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the X-KLAIM paradigm. With this respect, by using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and implement a more fine-grained kind of mobility, as shown in [7]. X-KLAIM provides both *weak mobility* (via operation **eval**$(P)@\ell$, where P is a process) and *strong mobility* (via operation **go**$@\ell$, that we do not describe here). Conversely, KLAVA supports weak mobility only; indeed, Java does not allow to save and restore the execution state. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is presented in detail in [2, 7].

Processes are the basic computational units. The class `KlavaProcess` is an abstract class that must be specialized to create processes. The derived classes must implement the method `execute`. This method will be invoked when a process is executed (just like `run` for threads). A process must be executed within a node, which will be its *execution environment*. `KlavaProcess` also offers all the methods to access tuple spaces. In KLAVA, processes can be sent as part of a message and executed at the destination site, where however their Java classes, i.e., their code, may be unknown. It is then necessary to make such code available for execution at remote hosts.

The KLAVA package has been modified, by removing all the code concerning code mobility, since those parts are now dealt with by relying on the `mobility` package completely. First of all the class `KlavaProcess`, that used to extend the class `java.lang.Thread` directly, now extends `JavaMigratingCode` and all the code for class collecting has been removed (since it is basically inherited from `JavaMigratingCode`). We have to ensure that the classes of the package KLAVA are present in all the sites visited by KLAVA agents, so such classes must not be dispatched together with migrating code. For this reason, in the constructor of the `KlavaProcess` we use the method `addExcludePackage`, of the package `mobility`:

```
public class KlavaProcess extends JavaMigratingCode {
 public KlavaProcess() {
  ...
```

```
   addExcludePackage("Klava.");
   addExcludePackage("momi.");
  }
}
```

In [4] KLAIM was extended with mobile object-oriented features according to the approach of MOMI [3]. These mobile object-oriented run-time features are implemented in the Java package `momi` [1], thus we have to exclude also this package apart from the KLAVA package itself.

A `JavaMigratingPacket` already contains all the information to deliver and restore a migrating object (e.g., its serialized form and the byte code of all the needed classes). However, a `KlavaProcess` has to carry also other information, e.g., the environment to translate localities into physical addresses. For this reason, objects of class `KlavaMigratingPacket`, a derived class from `JavaMigratingPacket`, containing other information that are useful to migrate a KLAVA process, are used to dispatch migrating code. When a node receives a `KlavaProcessPacket`, it can recover the original process by using the unmarshaller provided by the `mobility` package

```
KlavaProcess P = (KlavaProcess) unmarshaller.unmarshal(pack) ;
```

restore the additional information included in the package (e.g., the allocation environment for translating localities into physical addresses):

```
if ( pack.environment != null )
  P.setEnvironment( pack.environment ) ;
```

and finally, simply start the received process for local execution:

```
P.start() ;
```

All the details about unmarshalling and dynamic class loading are dealt with directly and transparently by the unmarshaller of the package `mobility`. With this respect, KLAVA does not need any further functionalities about dynamic class loading, thus it can completely rely on the standard features provided by the `mobility` package itself, without providing any specialized implementation.

On-going Work

The re-engineering of the KLAVA by using the `mobility` package of the IMC framework is only the first step. Since KLAVA strongly relies on code mobility this was the most interesting point to experiment with. Currently, we are re-engineering KLAVA as follows:

- Use the `protocols` package to re-implement the communication related parts of KLAVA. The new communication protocol should be independent from Java, so that it can be used also by KLAVA clients written in other languages (such as C, C++, etc.) and possibly also manually, e.g., from a telnet session. Of course, some parts will be still be dependent from Java (e.g.,

code mobility, since we will move Java objects), but, by using an abstract object communication layer, one may think to use XML as a common process exchange language (however, this issue is not considered at the moment).

Thanks to the abstraction provided by the package `protocols`, once the KLAVA communication protocol is re-implemented in the above direction, it will be automatically independent from the lower levels of network communications: this will allow us to easily and smoothly decide whether to use TCP, or UDP (or a mixture) for communications among distributed nodes (possibly by also relying on multicast as a means for node discovery in a network).

- Use the `topology` and `naming` packages to implement the new hierarchical model of KLAIM based on [8], that is already implemented directly in the KLAVA package. Here the main point is to derive the KLAVA `Node` main class (representing a generic node of a KLAVA net) from the `Node` class of the `topology` package. The latter class already provides all the generic functionalities for both flat and hierarchical networks, and the specialization in KLAVA will only need to provide the features that are specific of a KLAVA net. All the other functionalities, such as routing and forwarding a message in a hierarchical net, and resolving a logical name into a physical name will be inherited by the `topology` and `naming` classes.

2 Implementing $D\pi$

$D\pi$, introduced by Hennesy and Riely [12], is a locality-based extension of the π-calculus [13] where processes are distributed over a set of *nodes* (or *locations*) each of which is univocally identified by a *name* (or *locality*).

Like in π calculus, processes interact via message passing over *channels*. However, only local communication is permitted: Two processes can interact only if they are located at the same node.

A process can change their execution environment performing action **go**.

$D\pi$ does not assume a specific network topology. Indeed, this is aspect is subspecified in the standard calculus. However, a variant of $D\pi$, DPIF, has been introduced in [10] to consider explicit connections between nodes.

In the rest of this section we will show how, using IMC, a Java framework, named $JD\pi$ that implements a runtime for $D\pi$, is described.

2.1 Nodes and Localities

A $D\pi$ node provides an abstraction for a computational environment that host process executions and provides basic functionalities for process interactions via channel communication.

In $JD\pi$, a general implementation of a $D\pi$ node is provided by the class `JDpiAbstactNode`. This class extends the class `Node` of package `topology` by providing new primitives for processes interactions and for mobility. However, since any specific network topology is considered, `JDpiAbstractNode` is an abstract class. Subclasses have to provide an implementation for a given topology.

That follow are the signatures of the methods for message passing:

```
public <T> void out(JDpiChannelName<T> c, T v)

public <T> T in(JDpiChannelName<T> c)
```

Where channels are referenced using class `JDpiChannelName<T>` that permits identifying a channel of type `T`.

Methods for channel interaction are parametrized with respect to the type `T` of the used channel. Method `out` permits sending an object of type `T` from channel referenced by `c`. Conversely, method `in` permits retrieving an object of type `T` from the channel referenced by `c`.

Process mobility is implemented by the following method:

```
public JDpiReply go( JDpiProcess p , JDpiLocality l )
```

This method, permits spawning a `JDpiProcess` (see below) to be evaluated remotely at the node referenced by the `JDpiLocality l`.

`JDpiLocality` implements an abstraction for univocally addressing $D\pi$ nodes. Please notice that in $D\pi$ processes use only localities that *really* exists. For this reason, the use of an unknown or an unreachable locality throws a Java exception.

2.2 Processes

$D\pi$ processes are implemented using (abstract) class `JDpiProcess`, which is a subclass of `NodeProcess` that implements `JavaMigratingCode`. All the implementation of code migration is demanded to the IMC `mobility` package.

Each process implementation has to provide method `execute()` that describes behaviour of the implemented process.

Each `JDpiProcess` interacts with the hosting node by means of a `JDpiNodeProxy` (that is assigned to the process when it is added to the node). This approach permits a process to use the methods for channel interactions and code mobility without interacting with the hosting node directly.

For instance the following $D\pi$ process that, after reading a locality from channel `ex`, spawns itself at the read locality:

$$\textbf{rec}\ X.ex?(l).go@l.X$$

can be implemented as follows:

```
public void execute() {
 JDpiChannelName<JDpiLocality> inC = new JDpiChannelName<JDpiLocality>("ex");
 JDpiLocality l = in( inC );
 go( this , l);
}
```

2.3 Topology

In this section we describe how $D\pi$ node topology can be implemented. We present two different extensions of class `JDpiAbstractNode` described in Section 2.1.

2.3.1 A flat topology

Since $D\pi$ does not consider a specific implementation for nodes topology, a network can be implemented by a single server that accepts connections from nodes. This approach implements a flat topology.

Class `JDpiDomain` implements the central server that accept connections from the node involved in the network. This class, which extends class `Node` in IMC `topology` package, keeps track of all the nodes available in a system. Moreover, all the communications in a system pass through the domain.

$D\pi$ nodes that use this flat topology are implemented by means of class `JDpiNode`. This class extends the `JDpiAbstractNode` in such a way that:

- no incoming connections are accepted;
- only one outgoing connection to a `JDpiDomain` can be created.

For instance, in Figure 1 the topology of net described by the following $D\pi$ term can be found.

$$\ell_1 [\![P]\!] \parallel \kappa_1 [\![Q]\!] \parallel \ell_1 [\![R]\!] \parallel \kappa_2 [\![V]\!] \parallel \ell_2 [\![S]\!] \tag{1}$$

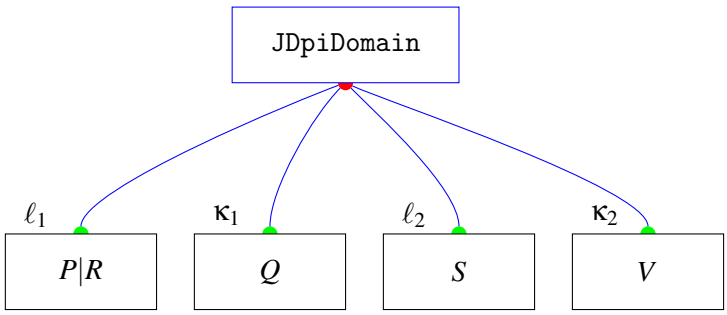


Figure 1: $D\pi$ flat topology

2.3.2 DPIF topology

Recently, Hennessy and Francalanza introduced in [10] a variant of $D\pi$ in which individual nodes may fail, or the links among them may be created and broken. The original language, $D\pi$, is extended with a new construct for detecting and reacting to these failures together with constructs that induce failure.

In DPIF an explicit notion of link is introduced. Indeed, a network is *evaluated* considering a given topology Δ that stores the connections between nodes. For instance, for the system of Equation (1), a possible topology is specified as follows:

$$\ell_1 \leftrightarrow \kappa 2, \ell_2 \leftrightarrow \kappa 1, \ell_2 \leftrightarrow \kappa 2$$

node ℓ_1 is connected to κ_2 while ℓ_2 is connected to both κ_1 and κ_2.

To model this kind of topology class `JDpiFNode` is introduced. This class extends `JDpiAbstractNode`. A `JDpiFNode` can get connected to and accept connection from different nodes.

Following the same interaction model proposed in $JD\pi$, two nodes can interact if and only if they are directly connected.

In Figure 2 a schema of a topology is presented.

2.4 Protocol

In $JD\pi$ two kinds of interactions can occur:

- a node ℓ spawns a process to node κ;

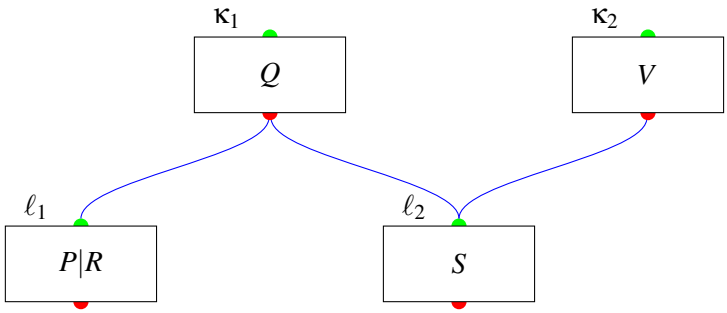


Figure 2: DPIF topology

- a node ℓ disconnects from another node.

The class `JDpiProtocol`, which extends class `Protocol` of package `protocols`, permits handling this kind of interactions.

When a process to be evaluated at κ is received, a specific *handler* is invoked. However, the different kinds of nodes we have described in previous sections implement different kinds of handlers. For instance, a `JDpiDomain` will submit the process to the corresponding node (if this exists) or notify the sender that the location κ is not reachable.

References

[1] L. Bettini. A Java package for class and mixin mobility in a distributed setting. In *Proc. of FIDJI'03*, volume 2952 of *LNCS*, pages 12–22. Springer-Verlag, 2003.

[2] 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`.

[3] L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, volume 2315 of *LNCS*, pages 56–71. Springer, 2002.

[4] L. Bettini, V. Bono, and B. Venneri. O'KLAIM: a coordination language with mobile mixins. In *Proc. of Coordination 2004*, volume 2949 of *LNCS*, pages 20–37. Springer, 2004.

[5] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th IEEE WETICE*, pages 110–115. IEEE Computer Society Press, 1998.

[6] L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In M. Lenisa and M. Miculan, editors, *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.

[7] 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.

[8] L. Bettini, M. Loreti, and R. Pugliese. An Infrastructure Language for Open Nets. In *Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM, 2002.

[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] A. Francalanza and M. Hennessy. Location and link failure in a distributed pi-calculus. 1, Department of Informatics, University of Sussex, 2005.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3, pages 3–17. Elsevier Science Publishers, 1998.

[13] R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

The MiKO Programming Language Definition and Implementation

Luís Lopes, Bruno Teixeira
Department of Computer Science
Faculty of Sciences, University of Porto

Vasco T. Vasconcelos
Department of Informatics
Faculty of Sciences, University of Lisbon

1 The MiKO Programming Language

The MiKO programming language is an instance of Boudol's parametric calculus of membranes [2] with the TyCO process calculus as the contents of the membrane itself, and was developed to provide a full prototype implementation of a membrane-based programming language that would validate the model introduced in the WP1.

The development of the language was made in two steps. First, the language's syntax, type system and operational semantics were established as described in [3]. The implementation of the compiler and of the virtual machine support required followed. Currently (beginning of February 2005) we are finalising the implementation of the back-end of the compiler, namely the final code generation.

The target language is MIL (Multithreaded Intermediate Language [8]), a fine grained assembly language for implementing concurrent calculi.

2 The Language Definition

MiKO is a higher order instance of TyCO-calculus based on Gerard Boudol's membrane model [2]. All domains have a *membrane* to control the migration of code that passes through its boundary, and a *contents* where the process execution really happens. The *contents* is referred as the *process* of the domain. To obtain the syntax of the MiKO calculus we use Boudol's parametric form and we extend the syntax of the TyCO calculus with new primitives to send and receive messages from the network. Since MiKO is a higher-order calculus, messages may carry process abstractions.

The Calculus

Consider the following set of names: $\mathcal{N}$ for names ranged over by x, and a set $\mathcal{L}$ for labels ranged over by l. The grammar of the MiKO calculus is as follows, subdivided into networks, denoted as N, and processes, denoted as P:

$$
\begin{aligned}
N &::= \mathsf{inaction} \mid x\{m\natural P\}[P] \mid x\,!\,M \mid N \mid N \mid \mathsf{new}\ x\ N && \text{(networks)}\\
P &::= \mathsf{inaction} \mid x\,!\,M \mid x?m \mid x?{*}m \mid P \mid P \mid \mathsf{new}\ x\ P \mid && \text{(processes)}\\
&\qquad A\vec{V} \mid \mathsf{in}[P] \mid \mathsf{out}[x, M] \mid \mathsf{mkdom}[x, m\natural P, P]\ \mathsf{in}\ P\\
M &::= l[\vec{V}] && \text{(messages)}\\
m &::= \{l_i = A_i\}_{i\in I} && \text{(methods)}\\
A &::= (\vec{x})P && \text{(abstractions)}\\
V &::= x \mid A && \text{(values)}
\end{aligned}
$$

The Language

The language definition of MiKO is very similar to the TyCO's language definition [6, 7]. We have decided to split the programming of the membrane and of the contents of a domain in two different files distinguished by their extensions. Given a site $x\{m\natural P\}[P]$ we create two files: "file.mkm" for the membrane $m\natural P$ and "file.mkp" for the contents P. This allows, for example, to reuse membrane implementations with distinct contents for a domain. At run-time the compiled images of these files will behave as the MiKO network:

$$\mathsf{new}\ x\ x\{\{\mathsf{myDomain}/x\}\ m\natural P\}[\{\mathsf{myDomain}/x\}\ P]$$

myDomain is a reserved word that is instantiated with the domain's name. Using the same set of names and labels we write the syntax of MiKO programs as follows:

Guardian	G	::=	$m\{S\} \mid \mathsf{new}\ x\ G$
Membrs/Conts	S, P	::=	$\mathsf{inaction} \mid S \mid S \mid \{S\} \mid \mathsf{new}\ x\ S \mid$
			$x?m \mid x\,!\,M \mid \mathsf{def}\ D\ \mathsf{in}\ S \mid$
			$X[\vec{e}] \mid A\vec{V} \mid \mathsf{if}\ \{e\}\ \mathsf{then}\ \{S\}\ \mathsf{else}\ \{S\} \mid$
			$\mathsf{in}[P] \mid\ \mathsf{out}[x, M] \mid \mathsf{mkdom}[x, G, P]\ \mathsf{in}\ S \mid$
			$\mathsf{import}\ x\ S$
Methods	m	::=	$\{l_i(\vec{x}_i) = S_i\}_{i \in I}$
Definitions	D	::=	$\{X_i(\vec{x}_i) = S_i\}_{i \in I}$
Messages	M	::=	$l[\vec{e}]$
Expressions	e	::=	$e\ Binop\ e \mid Unop\ e \mid (e) \mid V$
Abstractions	A	::=	$(\vec{x})S \mid x$
Values	V	::=	$x \mid A$
Binary Op.	$Binop$	::=	$+ \mid - \mid * \mid / \mid = \mid /= \mid < \mid >$
			$\mid <= \mid >= \mid \mathsf{and} \mid \mathsf{or}$
Unary Op.	$Unop$	::=	$- \mid \mathsf{not}$

It is noteworthy that names are global in a MiKO network, and thus, everytime a membrane or contents execute a new process they create a new channel that is implicitly exported to the whole network. However, to use this channel, a client must first obtain a binding for it. We assume that any free occurrence of a variable in a program is actually an implicit import/bind operation that takes a channel identifier, locates it in the network and creates a local proxy to it. To inforce a programming discipline where free names are correctly identified, and aren't just a by product of a programmer's mistake, we force programmers to import free identifiers before any local use can be made of them.

A more complete description of the MiKO language, its semantics and programming examples may be found in [3].

3 Compiler Implementation

The structure of the compiler is fairly standard, consisting of three main stages: *syntactic analysis*, *semantic analysis* (*scope analysis* plus *type inference*) and *code generation*.

The syntactic analysis is implemented using the JAVA tools, CUP and JLex [4], for automatic generation of the parser. The result is a parse tree that is then passed to the next stages of the compilation and is incrementaly

decorated with compile time information (during scope analysis and type-inference). The code generation then uses this information to translate the annotated parse tree into a MIL data-structure (and file).

The compiler is implemented using the *visitor* pattern [5] separating data structures from implementation. This makes it easier to develop the compiler incrementally and to add new features later.

4 Virtual Machine Support

The MIL virtual machine was modified to allow for two computations to run in the same site: the membrane and the contents of the membrane. This implies launching two threads each running the emulator for the MiKO instructions. One runs the code for the membrane and the other for the contents. Both threads share the same heap in such a way that membrane-contents communication is highly simplified.

The virtual machine was also extended in a modular way by implementing the new constructors: import, in, out and mkdom, using the machine's `external` feature for procedure calls. In the case of in and out, their implementation requires the communication between distinct threads of the MIL virtual machine (that running the membrane and that running the contents). mkdom implies the creation of a new domain with new contents and membrane and is thus a more computationaly intensive task. import requires interaction with an external name server to establish a proxy for a remote channel or domain. These functionalities were added without changing the core MIL language.

5 Network Infra-Structure Support

Currently, the TyCO language uses a custom implementation for providing the required network infra-structure to run TyCO applications. Our first approach to MiKO in this regard will be to use TyCO's run-time system as the base infra-structure for running MiKO programs. This will be done at almost no cost since most of the changes to MiKO happen at the virtual machine level and not the network infra-structure. A future development would be the implementation of this infra-structure in WP3's IMC (Implementing Mobile Calculi [1]) infra-structure, designed to allow fast prototyping of just this kind of system.

References

[1] Lorenzo Bettini, Rocco De Nicola, Daniele Falassi, Marc Lacoste, Luís Lopes, Licínio Oliveira, Hervé Paulino, and Vasco T. Vasconcelos. A Software Framework for Rapid Prototyping of Run-Time Systems for

Mobile Calculi. In *Contributions from Global Computing 2004*, Lecture Notes on Computer Science. Springer-Verlag, 2004. to appear.

[2] G. Boudol. A parametric model of migration and mobility, release 1. Mikado Deliverable D1.2.1, 2003.

[3] G. Boudol. An instance of the MIKADO migration model. Mikado Deliverable D1.2.2, `http://mikado.di.fc.ul.pt/repository/D1.2.2.pdf`, 2005.

[4] Scott E. Hudson et. al. *CUP, Parser Generator for Java*. available at `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[6] Vasco T. Vasconcelos. Core-TyCO Appendix to the Language Definition, Version 0.2. Technical report, Faculty of Sciences of the University of Lisbon, 2001.

[7] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO The Language Definition, Version 0.1. Technical report, Faculty of Sciences of the University of Lisbon, 1998.

[8] Vasco T. Vasconcelos, Luís Lopes, and Paulo Rafael. The MIL Virtual Machine Specification. Technical report, Faculty of Sciences of the University of Lisbon, 2005.

An Abstract Machine for the Kell Calculus

Philippe Bidinger *, Alan Schmitt, and Jean-Bernard Stefani

INRIA Rhône-Alpes, 38334 St Ismier, France
firstname.lastname@inrialpes.fr

Abstract. The Kell Calculus is a family of process calculi intended as as basis for studying distributed component-based programming. This paper presents an abstract machine for an instance of this calculus, a proof of its correctness, and a prototype OCaml implementation. The main originality of our abstract machine is that it does not mandate a particular physical configuration (e.g. mapping of localities to physical sites), and it is independent of any supporting network services. This allows to separate the proof of correctness of the abstract machine per se, from the proof of correctness of higher-level communication and migration protocols which can be implemented on the machine.

1 Introduction

The Kell calculus [18,17] is a family of higher-order process calculi with hierarchical localities and locality passivation, which is indexed by the pattern language used in input constructs. It has been introduced to study programming models for wide-area distributed systems and component-based systems. Two of the main design principles for the calculus are to keep all the actions "local" in order to facilitate its distributed implementation, and to allow different forms of localities to coexist. A consequence of the locality principle is that the calculus allows different forms of networks to be modeled (by different processes). Thus, on the one hand, an implementation of the calculus should not need to consider atomic actions occurring across wide-area networks. On the other hand, an implementation of the calculus should not imply the use of purely asynchronous communications between localities: one can have legitimate implementations of the calculus that exploit and rely on the synchronous or quasi-synchronous properties of specific environments (e.g. a local machine with different processes, a high-performance, low-latency local area network for homogeneous PC clusters).

We present in this paper a distributed abstract machine for an instance of the Kell calculus, and its implementation in OCaml. The original feature of our abstract machine is that, in contrast to other works on abstract machines for distributed process calculi, it does not depend on a given network model, and can be used to implement the calculus in different physical configurations. Let us explain in more detail what this means. An implementation of our abstract machine typically comprises two distinct parts:

– An implementation of the abstract machine specification per se, that conforms to the reduction rules given in section 3 below.

* Partly supported by EU IHP Marie Curie Training Site '**DisCo**: Semantic Foundations of Distributed Computation', contract HPMT-CT-2001-00290.

– Libraries, in the chosen implementation language, that provides access to network services, and that conform to a Kell calculus model of these services (i.e. a Kell calculus process).

For instance, assume that one wants to realize a physical configuration consisting of a network N, that interconnects two computers m_1 and m_2, that each run an implementation of the Kell calculus abstract machine, and a Kell calculus program (P_1 and P_2, respectively). This configuration would be modelled in the Kell calculus by the process

$$C \triangleq N[\texttt{Net} \mid m_1[\texttt{NetLib} \mid P_1] \mid m_2[\texttt{NetLib} \mid P_2]]$$

where the process `Net` models the behavior of network N, and where the process `NetLib` models the presence, at each site, of a library providing access to the network services modeled by `Net`. From the point of view of the Kell calculus abstract machine, the library `NetLib` is just a standard Kell calculus process, but whose communications will have side-effects (i.e. accessing the actual network services modelled by `Net`) outside of the abstract machine implementation.

The interesting aspect of our approach is the fact that we can thus provide implementations for different environments which all rely on the same abstract machine description and implementation. Consider for instance the physical configuration consisting of a network N, that interconnects two computers m_1 and m_2, that each run two separate processes, p_i^1 and p_i^2 ($i = 1, 2$). Each process p_i^j runs an implementation of the abstract machine, with a program Q_i^j. This configuration can be modelled by

$$C' \triangleq N[\texttt{Net} \mid M_1 \mid M_2]$$
$$M_1 \triangleq m_1[\texttt{NetOS} \mid \texttt{Ipc} \mid p_1^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^1] \mid p_1^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^2]]$$
$$M_2 \triangleq m_2[\texttt{NetOS} \mid \texttt{Ipc} \mid p_2^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^1] \mid p_2^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^2]]$$

where the process `NetOS` models the presence, at each site m_i, of some means (e.g. an operating system library) to access the network services modeled by `Net`, where the process `Ipc` models the presence, at each site, of a local communication library (e.g. an interprocess communication library provided by the local operating system), and where the processes `NetLib` and `IpcLib` model the presence, at each process p_i^j, of interfaces for accessing the different communication services provided, respectively, by the combination `Net` and `NetOS`, and by `Ipc`. Again, `NetLib` and `IpcLib` both appear as standard Kell calculus processes from the point of view of the abstract machine (i.e. they communicate with other processes by message exchange and can become passivated with their enclosing locality). However, the communication services they give access to can have very different semantics, if only in terms of reliability, latency, or security. The important point to note is that different communication services can coexist in the same implementation, and can be used selectively by application processes.

An important benefit of the independence of our abstract machine specification from any supporting network services, made possible by the local character of primitives in the Kell calculus, is the simplification of its proof of correctness. Indeed, the proof of correctness of our abstract machine does not involve the proof of a non-trivial distributed migration protocol, as is the case, for instance, with the JoCaml implementation of the distributed Join calculus [5], or with various abstract machines for ambient cal-

culi [7,10,5,14] [1]. Furthermore, the correctness of the machine is ensured, regardless of the network services used for the actual implementation.

The abstract machine described in this paper constitutes a first step in a potential series of more and more refined abstract machines, getting us closer to a provably correct implementation of the calculus. Such a progressive approach aims at breaking up the proof of correctness of an abstract machine close to implementation into more tractable steps. For this reason, our abstract machine remains non-deterministic, and still has a number of high-level constructs such as variable substitution. Compared to the calculus, the abstract machine realizes three important functions: (1) it handles names and name restriction; (2) it "flattens" a Kell calculus process with nested localities into a configuration of non-nested localities with dependency pointers; (3) it makes explicit high-level process marshalling and unmarshalling functions which are involved in the implementation of the locality passivation construct of the Kell calculus.

The correctness of the abstract machine is stated, following [14], as barbed bisimilarity between a process of the calculus and its abstract machine interpretation. However, the results we obtain are in fact stronger than pure barbed bisimilarity as they involve some form of contextual equivalence. The results are stated using a strong form of bisimilarity, for we use a notion of sub-reduction to abstract away purely administrative reductions.

The paper is organized as follows. Section 2 presents the instance of the Kell calculus we use in this paper. Section 3 specifies our abstract machine for the calculus. In Section 4 we give a correctness result for the abstract machine. In Section 5, we discuss an Ocaml prototype implementation of our abstract machine. In Section 6, we discuss related works. Section 7 concludes the paper with a discussion of future work. Proofs of the correctness properties are presented in the appendix.

2 The Kell calculus: syntax and operational semantics

2.1 Syntax

We now define the instance of the kell calculus we use in this paper. We allow five kinds of input patterns: *kell patterns*, that match a subkell, *local patterns*, that match a local message, *up patterns*, that match a message in the parent kell, and two kinds of *down patterns*, that match a message from a subkell. The syntax of the Kell calculus, together with the syntax of evaluation contexts, is given below:

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid P \mid P \mid a[P] \mid a\langle \widetilde{P} \rangle \\
P_* & ::= & \mathbf{0} \mid x \mid \xi \triangleright P \mid P_* \mid P_* \mid a[P_*] \mid a\langle \widetilde{P} \rangle \\
\xi & ::= & a\langle \widetilde{u} \rangle \mid a\langle \widetilde{u} \rangle^{\downarrow} \mid a\langle \widetilde{u} \rangle^{\downarrow^a} \mid a\langle \widetilde{u} \rangle^{\uparrow} \mid a[x] \\
u & ::= & x \mid (x) \\
\mathbf{E} & ::= & \cdot \mid \nu a.\mathbf{E} \mid a[\mathbf{E}] \mid P \mid \mathbf{E}
\end{array}
$$

[1] Note that the Channel Ambient abstract machine presented in [13] assumes that ambients may synchronize, for instance to run an `in` primitive. This assumption might be difficult to implement in an asynchronous distributed setting.

Filling the hole $\cdot$ in an evaluation context $\mathbf{E}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{E}\{Q\}$.

We assume an infinite set N of *names*. We let a, b, x, y and their decorated variants range over N. Note that names in the kell calculus act both as name constants and as (name or process) variables. We use $\widetilde{V}$ to denote finite vectors $(V_1, \ldots, V_q)$. Abusing notation, we equate $\widetilde{V}$ with the word $V_1 \ldots V_n$ and the set $\{V_1, \ldots, V_n\}$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q, R and their decorated variants range over processes. We say that a process is in *normal form* when it does not contain any name restriction operator. We use P_*, Q_*, R_* and their decorated variants to denote these processes. We call *message* a process of the form $a\langle\widetilde{P}\rangle$. We call *kell* a process of the form $a[P]$, with a called the name of the kell. In a kell of the form $a[\ldots \mid a_j[P_j] \mid \ldots \mid Q_k \mid \ldots]$ we call *subkells* the processes $a_j[P_j]$. 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 an *up pattern* $a\langle\widetilde{u}\rangle^{\uparrow}$, a *down pattern* $a\langle\widetilde{u}\rangle^{\downarrow^b}$ or $a\langle\widetilde{u}\rangle^{\downarrow}$, a *local pattern* $a\langle\widetilde{u}\rangle$, or a *control pattern* $a[x]$. A down pattern $a\langle\widetilde{u}\rangle^{\downarrow^b}$ matches a message on channel a coming from a subkell named b. A down pattern $a\langle\widetilde{u}\rangle^{\downarrow}$ matches a message on channel a coming from any subkell.

In a term $\nu a.P$, the scope extends as far to the right as possible. In a term $\xi \triangleright P$, the scope of $\triangleright$ extends as far to the right as possible. Thus, $a\langle c\rangle \triangleright P \mid Q$ stands for $a\langle c\rangle \triangleright (P \mid Q)$. We 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 $\nu\widetilde{a}.P$ if $\widetilde{a} = (a_1 \ldots a_q)$. By convention, if the name vector $\widetilde{a}$ is empty, then $\nu\widetilde{a}.P \triangleq P$. We also note $\prod_{i\in I} P_i$, $I = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $I = \emptyset$, then $\prod_{i\in I} P_i \triangleq \mathbf{0}$.

A pattern ξ acts as a binder in the calculus. All names x that do not occur within parenthesis $()$ in a pattern ξ are bound by the pattern. We call *pattern variables* (or *variables*, for short) such bound names in a pattern. Variables occurring in a pattern are supposed to be linear, i.e. there is only one occurrence of each variable in a given pattern. Names occurring in a pattern ξ under parenthesis (i.e. occurrences of the form (x) in ξ) are *not* bound in the pattern. We call them free pattern names (or free names, for short). We assumes that bound names of a pattern are disjoint from free names. The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Free names ($\mathtt{fn}$), bound names ($\mathtt{bn}$), free pattern variables ($\mathtt{fpn}$), and bound pattern names ($\mathtt{bpn}$) are defined as usual. We just point out the handling of free pattern names:

$$\mathtt{fpn}(a\langle\widetilde{u}\rangle) = \{a\} \cup \{x \in \mathsf{N} \mid (x) \in \widetilde{u}\} \qquad \mathtt{bpn}(a\langle\widetilde{u}\rangle) = \{x \in \mathsf{N} \mid x \in \widetilde{u}\}$$

We call *substitution* a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{K}$ from names to names and Kell calculus processes that is the identity except on a finite set of names. We note $\mathtt{supp}$ the support of a substitution (*i.e.* $\mathtt{supp}(\theta) = \{i \in \mathsf{N} \mid \theta(i) \neq i\}$). We assume when writing $\xi\theta$ that $\mathtt{fpn}(\xi) \cap \mathtt{supp}(\theta) = \emptyset$ and that $\mathtt{supp}(\theta) \subseteq \mathtt{bpn}(\xi)$.

We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

Formally, the reduction rules in section 2.2 could yield terms of the form $P[Q]$, which are not legal Kell calculus terms (i.e. the syntax does not distinguish between

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

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q}\ [\text{S.NU.PAR}] \qquad \frac{P =_\alpha Q}{P \equiv Q}\ [\text{S.}\alpha] \qquad \frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}}\ [\text{S.CONTEXT}]$$

Fig. 1. Structural equivalence

$$\frac{a \neq b}{a[\nu b.P] \stackrel{\equiv}{\to} \nu b.a[P]}\ [\text{SR.NEW}] \qquad \frac{P \stackrel{\equiv}{\to} P'}{E[P] \stackrel{\equiv}{\to} E[P']}\ [\text{SR.CONTEXT}]$$

$$\frac{P' \equiv P \qquad P \stackrel{\equiv}{\to} Q \qquad Q \equiv Q'}{P' \stackrel{\equiv}{\to} Q'}\ [\text{SR.STRUCT}]$$

$$\frac{}{c\langle \widetilde{u}\varphi\rangle \mid b[R \mid (c\langle \widetilde{u}\rangle^{\uparrow} \triangleright Q)] \to b[R \mid Q\varphi]}\ [\text{R.IN}] \qquad \frac{}{c\langle \widetilde{u}\varphi\rangle \mid (c\langle \widetilde{u}\rangle \triangleright Q) \to Q\varphi}\ [\text{R.LOCAL}]$$

$$\frac{\downarrow^{\bullet} = \downarrow^{b} \ \mathtt{or}\ \downarrow^{\bullet} = \downarrow}{b[c\langle \widetilde{u}\varphi\rangle \mid R] \mid (c\langle \widetilde{u}\rangle^{\downarrow^{\bullet}} \triangleright Q) \to b[R] \mid Q\varphi}\ [\text{R.OUT}]$$

$$\frac{}{a[P_*] \mid (a[x] \triangleright Q) \to Q\{P_*/x\}}\ [\text{R.PASS}] \qquad \frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}\ [\text{R.CONTEXT}]$$

$$\frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}\ [\text{R.STRUCT}] \qquad \frac{P' \stackrel{\equiv}{\to}{}^{*}\ P \qquad P \to Q}{P' \to Q}\ [\text{R.STRUCT.EXTR}]$$

Fig. 2. Reduction Relation

names playing the role of name variables, and names playing the role of process variables). However, a simple type system can be used to rule out such illegal terms.

2.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [1], via a structural equivalence relation and a reduction relation. The structural equivalence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 1 and that makes the parallel operator $\mid$ associative and commutative, with $\mathbf{0}$ as a neutral element.

The reduction relation $\to$ is the smallest binary relation on K^2 that satisfies the rules given in Figure 2.

Notice that we do not have structural equivalence rules that deal with scope extrusion beyond a kell boundary (i.e we do not have the Mobile Ambient rule $a[\nu b.P] \equiv \nu b.a[P]$, provided $b \neq a$). This is to avoid phenomena as illustrated below:

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

However, such name extrusion is still needed to allow communication across kell boundaries. The solution adopted here is to allow only scope *extrusion* across kell boundaries and to restrict passivation to processes without name restriction in evaluation context. Formally, this is achieved by requiring a process to be in normal form (P_*) in rule R.PASS and by adding a scope extrusion sub-relation reduction $\xrightarrow{\equiv}$.

Rules R.IN and R.OUT govern the crossing of kell boundaries. Only messages may cross a kell boundary. In rule R.IN, a trigger receives a message from the outside of the enclosing kell. In rule R.OUT, a trigger receives a message from a subkell.

3 Abstract Machine

3.1 Syntax

Following [14], our abstract machine is specified in the form of a process calculus whose terms, called *machine terms*, correspond to abstract machine states. Intuitively, a machine term consists in a set of localities, each executing a different program, organized in a tree by means of pointers between localities.

The syntax of the abstract machine calculus is given below:

$$\begin{array}{ll} M ::= \mathbf{0} \mid L \mid M \mid M & M_* ::= \mathbf{0} \mid L_* \mid M_* \mid M_* \\ L ::= h : m[P]_{k,S} & L_* ::= h : m[P_*]_{k,S} \\ S ::= \emptyset \mid h \mid S,\, S & \end{array}$$

$$\begin{array}{l} P ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid P \mid P \mid a[P] \mid a\langle \widetilde{P} \rangle \mid \mathbf{reify}(k, M_*) \\ P_* ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid P_* \mid P_* \mid a\langle \widetilde{P} \rangle \\ \xi ::= a\langle \widetilde{u} \rangle \mid a\langle \widetilde{u} \rangle^{\downarrow} \mid a\langle \widetilde{u} \rangle^{\downarrow^a} \mid a\langle \widetilde{u} \rangle^{\uparrow} \mid a[x] \\ u ::= x \mid (x) \qquad x \in \mathsf{N} \qquad h, k, l \in \mathsf{MN} \qquad a, m \in \mathsf{N} \cup \mathsf{MN} \end{array}$$

Terms generated by the productions M, M_* in the abstract machine grammar are called *machine terms* (or *machines* for short, when no ambiguity arises), and are ranged over by M, N and their decorated variants. We designate their set by M. Machine terms make use of two sorts of names: the set N and a disjoint infinite set MN whose elements are called *machine names*. We call *locality* a machine term of the form $h : m[P]_{k,S}$. In a locality $h : m[P]_{k,S}$, m is the name of the kell the locality represents, h is the machine name of the locality, k is the machine name of its parent locality, S is the set of the machine names of its sublocalities, and P is the *machine process* being run at locality h. We use three particular machine names: $\mathbf{r}$, $\mathbf{rn}$ and $\mathbf{rp}$, which denote, respectively, the machine name of the root locality, the name of the root kell (associated with the root locality), and the machine name of the (virtual) root parent locality. Machine names appearing in a machine terms are all unique (in contrast to kell names).

We call MK the set of machine processes (i.e. terms generated via the productions P, P_* in the abstract machine grammar), and we have $\mathsf{K} \subseteq \mathsf{MK}$. The machine processes are slightly different from Kell calculus processes. First a new term $\mathbf{reify}(k, M_*)$ is introduced to represent a passivated machine. The term M_* is a tree of machines encoded as a parallel composition of localities and k is the machine name of the root of this tree.

$$\frac{M =_\alpha N}{M \equiv N}\ [\text{M.SE.}\alpha] \qquad \frac{P \equiv P' \qquad S \equiv S'}{l : h[P]_{k,S} \equiv l : h[P']_{k,S'}}\ [\text{M.SE.CTX}]$$

Fig. 3. Structural equivalence for machines

Secondly, the names that can be used by a machine process belong to $\mathsf{N} \cup \mathsf{MN}$. This point will be made clear in the next subsection. A machine process is in normal form, written P_*, when it has no name restriction operator nor kells in evaluation context. A machine is in normal form when all machine processes in its localities are in normal form. We use the $._*$ suffix to denote machines and processes in normal form. The definitions and conventions given in section 2 extend to machine processes. Note that we use the same meta-variables to denote processes and machine processes. When it is not clear from the context, we will precise whether a variable denote a process or a machine process.

3.2 Reduction semantics

The reduction relation is defined as for the calculus via a structural congruence relation and a reduction relation.

First, we define two equivalence relations (both denoted by $\equiv$), on machine processes and sets of localities, respectively, as the smallest relations that make the parallel operator $|$ (resp. the , operator) associative and commutative with $\mathbf{0}$ (resp. $\emptyset$) as a neutral element. Then, we define the structural congruence $\equiv$ on machines as the smallest equivalence relation that verifies the rules in figure 3 and that makes the parallel operator $|$ associative and commutative with $\mathbf{0}$ as a neutral element.

This structural equivalence, together with the rules M.S.CTX and M.S.STR, allows us to view machines as sets of localities and terms S as sets of machine names. Note that the equivalence relation on machine processes is different from the one on kell calculus processes as it does not contain rules dealing with restriction. This is because restriction is handled by the abstract machine as a name creation operator (rule M.S.NEW).

The reduction relation is defined as the smallest relation that satisfies the rules in Figures 4 and 5. It uses a subreduction relation $\overset{\equiv}{\rightarrow}$. The first subreduction rule, M.S.NEW, deals with restriction, which is interpreted as name creation. The reason the rule imposes the newly created name to be a machine name is related to the correctness proof, where we need to distinguish between restricted and free Kell names. Rule M.S.CELL creates a new locality when a kell is in the locality process. Rule M.S.ACT activates a passivated machine. Activation involves releasing the process held in the root locality of the passivated machine in the current locality, and releasing the sublocalities of the passivated machine as new sublocalities of the current locality.

The reduction rules M.IN, M.OUT, M.LOCAL, and M.PASS are the direct equivalent of the Kell calculus rules R.IN, R.OUT, R.LOCAL, and R.PASS, respectively. In rule M.PASS, the localities passivated are in normal form.

$$\frac{l\ fresh \in \mathsf{MN}}{h : n[(\nu a.P) \mid Q]_{k,S} \xrightarrow{\equiv} h : n[P\{l/a\} \mid Q]_{k,S}}\ [\text{M.S.NEW}]$$

$$\frac{h'\ fresh \in \mathsf{MN}}{h : n[m[P] \mid Q]_{k,S} \xrightarrow{\equiv} h : n[Q]_{k,(S,h')} \mid h' : m[P]_{h,\emptyset}}\ [\text{M.S.CELL}]$$

$$\frac{\begin{array}{c} M_* = l : n[R_*]_{l',S'} \mid M'_* \\ \mathbf{locnames}(M'_*) = \{l_i / i \in I\} \qquad k_i\ fresh \in \mathsf{MN},\ i \in I \end{array}}{h : m[\mathbf{reify}(l, M_*) \mid P]_{k,S} \xrightarrow{\equiv} h : m[R_* \mid P]_{k,(S,S'\{k_i/l_i\}_{i\in I}} \mid M'_*\{h/l\}\{k_i/l_i\}_{i\in I}}\ [\text{M.S.ACT}]$$

$$\frac{M \xrightarrow{\equiv} M'}{M \mid N \xrightarrow{\equiv} M' \mid N}\ [\text{M.S.CTX}] \qquad \frac{M \equiv M' \qquad M' \xrightarrow{\equiv} M'' \qquad M'' \equiv M'''}{M \xrightarrow{\equiv} M'''}\ [\text{M.S.STR}]$$

Fig. 4. Sub-reduction for machines

The reduction rules use the auxiliary function $\mathbf{locnames}$, the predicate $\mathbf{tree}$, and the notion of well-formed machine, which we now define.

The predicate $\mathbf{tree}(M, l, a, p)$ is defined as follows (where S may be empty):

$$\mathbf{tree}(M, l, a, p) = (M \equiv l : a[P]_{p,S} \mid \prod_{j\in S} M_j) \wedge_{j\in S} \mathbf{tree}(M_i, l_j, a_j, p_j))$$

with the additional condition that l, p, l_j, p_j are all distinct.

The function $\mathbf{locnames}(M)$ designates the set of locality names of all localities present in a machine M.

We say that a machine M is *well-formed* if we have $\mathbf{tree}(M, \mathbf{r}, \mathbf{rn}, \mathbf{rp})$. The set of well-formed machines is noted WFM. Finally, we will need the relation $\cong$ defined as follows: $M \cong N$ if and only if $\mathbf{tree}(M, l, m, p)$ and $M\sigma \equiv N\sigma'$ where σ and σ' are injective renaming of machine names such that $\sigma(l) = \sigma'(l) = l$ and $\sigma(p) = \sigma'(p) = p$ and if $m \in \mathsf{MN}$, $\sigma(m) = \sigma'(m) = m$.

4 Correctness

We establish the correctness of our abstract machine by establishing a strong bisimilarity result between Kell calculus processes and their interpretation by the abstract machine. The notion of equivalence we adopt is strong barbed bisimulation [15], which we denote by $\sim$. This notion of bisimulation can be used to compare different transition systems, provided that they are equipped with observability predicates and a reduction relation. An originality of our correctness result is that it relies on a strong form of barbed bisimilarity, instead of a weak one. This is possible because we abstract away administrative reduction rules through the subreduction relations in both the calculus and the abstract machine semantics. Our main result is the following:

Theorem 1 (Correctness). *For any Kell calculus process P, we have $[\![P]\!] \sim P$.*

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\ [\text{M.PAR}] \qquad \frac{\xi = c\langle \widetilde{u} \rangle}{\xi\varphi \mid (\xi \triangleright Q) \to Q\varphi}\ [\text{M.LOCAL}]$$

$$\frac{\xi = c\langle \widetilde{u} \rangle^{\uparrow}}{h : a[\xi\varphi \mid P]_{k,S} \mid h' : b[(\xi \triangleright Q) \mid R]_{h,S'} \mapsto h : a[P]_{k,S} \mid h' : b[Q\varphi \mid R]_{h,S'}}\ [\text{M.IN}]$$

$$\frac{\xi = c\langle \widetilde{u} \rangle^{\downarrow} \ \vee \ \xi = c\langle \widetilde{u} \rangle^{\downarrow^a}}{h : a[\xi\varphi \mid P]_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'} \mapsto h : a[P]_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}}\ [\text{M.OUT}]$$

$$\frac{M_* = l : a[R_*]_{h,S'} \mid M'_* \qquad \mathbf{tree}(M_*, l, a, h)}{h : m[(a[x] \triangleright P) \mid Q]_{k,S} \mid M_* \mapsto h : m[P\{\mathbf{reify}(l, M_*)/x\} \mid Q]_{k,S \setminus l}}\ [\text{M.PASS}]$$

$$\frac{M \mapsto M'}{M \mid N \mapsto M' \mid N}\ [\text{M.CTX}] \qquad \frac{M \equiv M' \qquad M' \mapsto M'' \qquad M'' \equiv M'''}{M \mapsto M'''}\ [\text{M.STR}]$$

$$\frac{P \to P'}{h : m[P]_{k,S} \mapsto h : m[P']_{k,S}}\ [\text{M.RED}] \qquad \frac{M \stackrel{\equiv}{\to}^* M' \qquad M' \mapsto M''}{M \to M''}\ [\text{M.NORM}]$$

Fig. 5. Reduction for machines

This theorem asserts the equivalence of any Kell calculus process P with its translation in the abstract machine calculus. In the rest of this section we give the main definitions and intermediate results that intervene in the proof of Theorem 1.

We first define the translation of a Kell calculus process in the abstract machine calculus.

Definition 1. $[\![P]\!] = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset}$

A first important property of our model is to ensure that the tree structure of the machine is preserved through reduction.

Proposition 1 (Well-Formedness). *If* $\mathbf{tree}(M, l, a, p)$ *and* $M \cong M'$, $M \stackrel{\equiv}{\to} M'$, $M \mapsto M'$, *or* $M \to M'$, *then* $\mathbf{tree}(M', l, a, p)$. *In particular, well-formedness is preserved by reduction. Moreover, for any process* P, $[\![P]\!]$ *is well-formed.*

From now on, unless otherwise stated, we only consider machine terms M such that $\mathbf{tree}(M, l, a, p)$ for some names l, a, p. The definitions of strong barbed bisimulation and strong barbed bisimilarity are classical [15]. We reproduce them below.

Definition 2 (Strong barbed bisimulation). *Let* TS_1 *and* TS_2 *be two sets of transition systems equipped with the same observability predicates* $\downarrow_a$, $a \in \mathsf{N}$. *A relation* $R \subseteq TS_1 \times TS_2$ *is a* strong barbed simulation *if whenever* $(A, B) \in R$, *we have*

- *If* $A \downarrow_a$ *then* $B \downarrow_a$
- *If* $A \to A'$ *then there exists* B' *such that* $B \to B'$ *and* $(A', B') \in R'$

A relation R is a strong barbed bisimulation if R and R^{-1} are both strong barbed simulations.

Definition 3 (Strong barbed bisimilarity). *Two transition systems A and B are said to be* strongly barbed bisimilar*, noted $A \sim B$, if there exists a strong barbed bisimulation R such that $(A, B) \in R$.*

To define strong bisimilarity for Kell calculus processes and machines we rely on the following observability predicates.

Definition 4 (Observability predicate for processes). *If P is a Kell calculus process, $P \downarrow_a$ holds if one of the following cases holds:*

- $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.a\langle\widetilde{P}\rangle \mid R \mid P'$*, with* $a \notin \widetilde{c}$
- $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.m[a\langle\widetilde{P}\rangle \mid R] \mid P'$*, with* $a \notin \widetilde{c}$
- $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.a[P] \mid P'$*, with* $a \notin \widetilde{c}$

Definition 5 (Observability predicate for machines). *If M is a well-formed machine and $a \in \mathsf{N}$, $M \downarrow_a$ holds if one of the following cases holds:*

- $M \equiv \stackrel{\equiv}{\rightarrow}^* \mathbf{r} : \mathbf{rn}[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{rp},S} \mid M'$
- $M \equiv \stackrel{\equiv}{\rightarrow}^* h : m[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{r},S} \mid M'$
- $M \equiv \stackrel{\equiv}{\rightarrow}^* h : a[P]_{\mathbf{r},S} \mid M'$

We now define two equivalence relations over machines that we use to state correctness properties. The first one identifies two machines that have the same normal form. The second one corresponds to a form of strong barbed congruence. Note that the second one is defined on well-formed machine only.

Lemma 1 (Normal form). *If M is a machine term, then there exists M'_* such that $M \stackrel{\equiv}{\rightarrow}^* M'_*$. Moreover, if $M \equiv \stackrel{\equiv}{\rightarrow}^* M''_*$ then $M'_* \cong M''_*$. Besides, $M \stackrel{\equiv}{\not\rightarrow}$ if and only if $M = M'_*$ for some M'_*.*

Definition 6 (Equivalence). *Two machines M and N are said to be* equivalent*, noted $M \doteq N$, if they have the same normal form (up to $\cong$).*

From now on, we will use the same notation M_* for a normal form of M (*i.e.* $M \stackrel{\equiv}{\rightarrow}^* M_* \stackrel{\equiv}{\not\rightarrow}$), and for an arbitrary term in normal form.

Definition 7. *Let $M = l : n[P]_{p,S} \mid M'$ be a machine such that $\mathbf{tree}(M, l, n, p)$ and h a fresh machine name. We define:*

$$\begin{aligned} M \mid Q &= l : n[P \mid Q]_{p,S} \mid M' \\ a[M] &= l : n[\mathbf{0}]_{p,h} \mid h : a[P]_{l,S} \mid M'\{h/l\} \\ \nu a.M &= M\{h/a\} \end{aligned}$$

We extend these definitions to any contexts of the following form:

$$\mathbf{E} ::= . \quad | \quad (R \mid \mathbf{E}) \quad | \quad a[\mathbf{E}] \quad | \quad \nu a.\mathbf{E}$$

Definition 8 (Contextual equivalence for machines). *Two well-formed machines M and N are contextually equivalent ($M \sim_c N$) if and only if $\forall \mathbf{E}, \mathbf{E}[M] \sim \mathbf{E}[N]$.*

We check easily that $\sim_c$ is the largest relation over machines included in strong barbed bisimilarity that is a preserved by $a[.]$, $\nu a..$ and $. \mid R$.

Lemma 2. *$\sim_c$, $\doteq$, $\cong$ and $\equiv$ are equivalence relations.*

Lemma 3. *We have $\equiv \subseteq \cong \subseteq \doteq$ and if we consider the restrictions of these relations to well-formed machines, they are all strong barbed bisimulation and $\doteq \subseteq \sim_c$.*

We now state two properties that relate machine reductions to process reductions (soundness), and process reductions to machine reductions (completeness).

Proposition 2 (Soundness). *$[\![P]\!] \to M \implies P \to P'$ with $[\![P']\!] \sim_c M$.*

Proof. We give here an outline of the proof, which is given in full in the appendix. We first define by induction an inverse translation function $[\![.]\!]^{mac}$ from machines to processes. This function has three roles: to expand the "reified" processes, to rebuild the tree structure of the term, and to recreate restricted names from machine names.

The soundness proposition results from the following lemmas:

Lemma 4. *If M is well-formed and $M \xrightarrow{\equiv} N$ then $[\![M]\!]^{mac} \xrightarrow{\equiv}{}^{*} \equiv [\![N]\!]^{mac}$.*

Lemma 5. *If M is well-formed and $M \mapsto N$ then $[\![M]\!]^{mac} \mapsto [\![N]\!]^{mac}$.*

Lemma 6. *If M is a well-formed machine, then $[\![[\![M]\!]^{mac}]\!] \sim_c M$. If P is a process, then $[\![[\![P]\!]]\!]^{mac} \equiv P$.*

Proposition 3 (Completeness). *$P \to P' \implies [\![P]\!] \to \sim_c [\![P']\!]$*

Proof (Sketch).

The proof of this proposition is on induction on the derivation of $P \to P'$ and need the two following lemmas:

Lemma 7. *If $P \equiv P'$ then $[\![P]\!] \doteq [\![P']\!]$. If $P \xrightarrow{\equiv} P'$ then $[\![P]\!] \doteq [\![P']\!]$.*

Lemma 8. *Let P_* be a process and M_* a machine such that $\mathbf{tree}(M_*, p, a, r)$. If we have $p : a[P_*]_{p',\emptyset} \xrightarrow{\equiv}{}^{*} \cong M_*$ then for any machine N we have $N\{\mathbf{reify}(p, M_*)/x\} \sim_c N\{P_*/x\}$.*

The proof of Theorem 1 then results immediately from Propositions 2 and 3 by showing that the relation $\{\langle [\![P]\!], P\rangle \mid P \in \mathsf{K}\}$ is a strong barbed bisimulation up to $\sim$.

5 Implementation

We have implemented a prototype of our abstract machine in OCaml, which realizes a Kell calculus interpreter, and is available at [11]. The source language for the interpreter (called `kcl`) is essentially a typed extension of the calculus presented in this paper, with values. Values are either basic (integers, lists, strings), higher-order (process abstractions, passivated processes) or expressions built upon classical operators such as arithmetic operators or marshalling/unmarshalling primitives.

User programs are first parsed and typed-checked using a simple type inference algorithm. Then, they are executed by a runtime that follows closely the reductions of the abstract machine. Unlike the abstract machine, the runtime is deterministic (we do not detail here the particular reduction strategy we use). Moreover, we use environments in order to avoid the use of substitutions. The freshness conditions in the rules M.S.CELL, M.S.ACT and M.S.NEW are implemented either through the use of runtime pointers for locality names, or by a global fresh identifier generator for names created by a *new* instruction.

An independent part of the interpreter allows user programs to access various services as library functions, which may also be modeled as Kell Calculus processes. More precisely, we can see an interpreter as a context $\mathtt{vmid}[\mathtt{Lib} \mid u[\,\cdot\,]]$ executing a user program P (filling the hole) according to the rules of the abstract machine. The program P can use services specified in `Lib` that correspond to OCaml functions, but are accessed transparently from P like any other receiver. Similarly, these functions can generate messages in the `vmid` locality that can be received by P. In the implementation, messages sent from the top level of P are treated differently whether they are addressed to a receiver in `Lib` or not. A very simple library could be $\mathtt{Lib} = (\mathtt{echo}^{\downarrow}\langle x\rangle \diamond Q)$, where Q specifies the output of the string x on the standard output, and where $\diamond$ denotes to a replicated input construct (which can be encoded in the Kell calculus as shown in [17]).

A distributed configuration of interpreters can be specified as follows. If we run the programs $P_0, \ldots, P_n$ on different interpreters, the resulting behavior is specified by the following term

$$\mathtt{Net} \mid \mathtt{vmid}_0[\mathtt{Lib}(\mathtt{vmid}_0) \mid u[P_0]] \mid \ldots \mid \mathtt{vmid}_n[\mathtt{Lib}(\mathtt{vmid}_n) \mid u[P_n]]$$

where we assume `vmid` names to be distinct. The processes `Lib` model the local libraries and `Net` the network. In our implementation they are mainly defined as follows (omitting the type annotations):

$$\mathtt{Lib}(\mathtt{vmid}) = (\mathtt{send}^{\downarrow}\langle x, y\rangle \diamond \mathtt{send}\langle x, y\rangle \mid (\mathtt{recv}^{\uparrow}\langle(\mathtt{vmid}), y\rangle \diamond \mathtt{msg}\langle x\rangle \mid (\mathtt{echo}^{\downarrow}\langle x\rangle \diamond Q)$$

$$\mathtt{Net} = \mathtt{send}^{\downarrow}\langle x, y\rangle \diamond \mathtt{rcv}\langle x, y\rangle$$

These processes specify an environment allowing the exchange of asynchronous messages between interpreters, and providing some output capability. The `vmid` name allows to send messages to uniquely deignated kells. In addition, marshalling and unmarshalling functions allow to send arbitrary values over the network.

We give in Figure 6 the code of a distributed application consisting of a client and a server that simply executes the code that it receives. `vm` is a constructor that builds an identifier for a virtual machine (typically to locate a name server) from an address and

a port. `thisloc` is bound to the identifier of the machine in which it is evaluated. The construct `def in` corresponds to an input ($\xi \triangleright P$) and `rdef` to a replicated input. We use marshalling and unmarshalling functions that convert arbitrary values to string and conversely.

client.kcl

```
new a in new b in new c in
let serverid = vm ("plutonium.inrialpes.fr", 6000) in
let myid = thisloc in
  ( def a [ X ] in send < serverid, marshall(X) > )
| ( def b [ X ] in X )
| ( def c [ X ] in X | X)
| a [ send < myid, marshall ( echo <"good"> | b[c[echo <"bye">]] ) >
      | echo < "hello" > ]
|  rdef msg up < X > in unmarshall(X) as proc
```

server.kcl

```
rdef msg up < X > in X
```

Fig. 6. kcl example

The execution of the server and the client on two different machines gives the following result.

```
plutonium:~/kcl-0.1/bidinger$ kcl server.kcl -p 6000
hello

californium:~/kcl-0.1/bidinger$ kcl client.kcl -p 7000
good
bye
bye
```

6 Related work

There has been a number of recent papers devoted to the description and implementation of abstract machines for distributed process calculi. One can cite notably the Jocaml distributed implementation of the Join calculus [6,5], the Join calculus implementation of Mobile Ambients [7], Nomadic Pict [21,19], the abstract machine for the M-calculus [9], the Fusion Machine [8], the PAN and GCPAN abstract machines for Safe Ambients [14,10], the CAM abstract machine for Channel Ambients [13]. In addition, there have been also implementations of distributed calculi such as the Seal calculus [20], Klaim [2], or DiTyCO [12].

Our abstract machine specification has been designed to be independent from the actual implementation environment and the network services it provides. It thus can be used in widely different configurations. For instance, one is not limited to mapping top-level localities to physical sites as in [7,5,9], or does not need to introduce physical sites

as a different locality abstractions than that of the supported calculus as in [10,14]. This separation between abstract machine behavior and network semantics is not present in other abstract machines for distributed process calculi.

The Seal calculus [4] and the M-calculus [16] are the only calculi that share with the Kell calculus a combination of local actions and hierarchical localities, and could thus achieve a similar independence between abstract machine and network services. No abstract machine is described for the Seal calculus, however (only an implementation is mentioned in [20]), and the M-calculus abstract machine described in [9] relies on a fixed network model and a mapping of top-level localities to physical sites. Calculi which rely on an explicit flat network model such as Nomadic Pict, DiTyCO, Klaim have abstract machines and implementations which presuppose a given physical configuration and its supporting network model.

The Fusion Machine implements the general fusion calculus, where no localities are present, but the abstract machine itself is based on a fixed asynchronous network model. Furthermore, because of the nature of communications in Fusion, the Fusion machine relies on a non-trivial migration protocol for achieving synchronization in presence of multiple sites. In contrast to our calculus and abstract machine, this prevents distributed Fusion programs to directly, and at no cost, exploit low-level network services such as a basic datagram service.

Abstract machines and implementations for distributed process calculi with hierarchical localities other than the Seal calculus and the M-calculus, namely the Join calculus and Ambient calculi, must implement migration primitives, which forces a dependence on a given network model. For instance, the JoCaml abstract machine for the distributed join calculus [5] depends on an asynchronous message passing network model and on a specific interpretation of the locality hierarchy (top level localities are interpreted as physical sites). The PAN [14] and GCPAN [10] abstract machines for Safe Mobile Ambients depend as well on an asynchronous message passing network model for specifying the migration of ambients between sites (corresponding to the interpretation of the Ambient primitive `open`), and on the introduction of a notion of execution site, not related to ambients. The Channel Ambient abstract machine [13] leaves in fact the realization of its `in` and `out` migration primitives unspecified.

7 Conclusion

We have presented an abstract machine for an instance of the Kell calculus, and discussed briefly its OCaml implementation. The originality of our abstract machine lies in the fact that it is independent from any network services that could be used for a distributed implementation. Indeed, as our simple OCaml implementation illustrates, we can isolate network services provided by a given environment in language libraries that can be reified as standard Kell calculus processes for use by Kell calculus programs. While this means that our abstract machine, just as the Kell calculus, does not embody any sophisticated abstraction for distributed programming, it demonstrates that the calculus and its associated machine provide a very flexible basis for developing these abstractions. Furthermore, this independence has the advantage to simplify the

proof of correctness of our abstract machine, as it does not depend on the correctness proof of a sophisticated distributed protocol.

Much work remains of course towards a provably correct implementation of the calculus. Our non-deterministic abstract machine remains too abstract in a number of dimensions to be the basis for an efficient implementation of the calculus. First, truly local actions can only be realized, and efficiency obtained, if there is some determinacy in routing messages to triggers (as it is enforced in our OCaml implementation). One can think of applying a type system similar to that reported in [3], which guarantees the unicity of kell names, to obtain linearity conditions ensuring the unicity of message destinations. Secondly, an efficient machine would require a more deterministic behavior. Here we face the prospect of a more difficult proof of correctness, and more difficulty in stating the correctness conditions, which must probably relate the non-determinism at the calculus level with the determinism of the abstract machine through some sort of fairness condition.

References

1. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science, vol. 96*, 1992.
2. L. Bettini, M. Loreti, and R. Pugliese. Srtuctured nets in klaim. In *Proceedings of the 2000 ACM Symposium on Applied Computing, ACM Press*, 2000.
3. P. Bidinger and J.B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. 6th IFIP FMOODS International Conference*, volume 2884 of *LNCS*. Springer, 2003.
4. G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th FST-TCS*, number 2556 in LNCS. Springer, 2002.
5. Fabrice Le Fessant. *JoCaml: Conception et Implantation d'un Langage à Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2001.
6. 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.
7. 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.
8. Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine. In *CONCUR 2002*, volume 2421 of *LNCS*. Springer-Verlag, 2002.
9. F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.
10. D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients, 2004. Unpublished. Available at: http://www.cs.unibo.it/∼sangio/DOC_public/gcpan.ps.gz.
11. The Kell calculus page. http://sardes.inrialpes.fr/kells/.
12. 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.
13. A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of ESOP 2004*, LNCS. Springer-Verlag, April 2004.
14. D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th ICALP*, volume 2076 of *LNCS*. Springer-Verlag, 2001.

15. D. Sangiorgi and S. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
16. 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.
17. A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In P. Quaglia, editor, *Global Computing*, volume 3267 of *LNCS*. Springer, 2004.
18. J.B. Stefani. A Calculus of Kells. In *Proceedings 2nd International Workshop on Foundations of Global Computing*, 2003.
19. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, 2001.
20. J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA, Lecture Notes in Computer Science 1686, Springer*, 1998.
21. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

A Proofs

We start by giving some additional definitions. We define the functions $\mathbf{names}$ that returns the set of machine names of a machine, and $\mathbf{mnames}$, the set of machine names that are not locality names. In particular, we have $\mathbf{names}(M) = \mathbf{mnames}(M) \uplus \mathbf{locnames}(M)$.

General lemmas

Proof (Proof of lemma 2). Immediate.

Lemma 9. *If $M \stackrel{\equiv}{\rightarrow} M'$ then $M \equiv L \mid M''$ with $L \stackrel{\equiv}{\rightarrow} M'''$ and $M' \equiv M''' \mid M''$. Moreover $L \stackrel{\equiv}{\rightarrow} M'''$ is one of the axiom rules defining $\stackrel{\equiv}{\rightarrow}$.*

Proof. Immediate.

Lemma 10. *If $M \mapsto N$ then $M' \equiv M_0 \mid M''$, $M_0 \mapsto M_0'$ and $N \equiv M_0' \mid M''$. Moreover $M_0 \mapsto M_0'$ is one of the axiom rules defining $\mapsto$.*

Proof. Immediate.

Lemma 11. *If $M \cong N$ and $M \stackrel{\equiv}{\rightarrow} M'$, $M \mapsto M'$ or $M \rightarrow M'$ then $N \stackrel{\equiv}{\rightarrow} N'$, $N \mapsto N'$, $N \rightarrow N'$ respectively, with $M' \cong N'$.*

Proof. First suppose $\mathbf{tree}(M, l, n, p)$. We have from proposition 1 $\mathbf{tree}(N, l, n, p)$ and $\mathbf{tree}(M', l, n, p)$.

By lemma 9, we have $M \equiv L_0 \mid M_0''$ with $L_0 \stackrel{\equiv}{\rightarrow} M_0'''$. We consider the case where $L_0 \stackrel{\equiv}{\rightarrow} M_0'''$ corresponds to the M.S.NEW rule. We have

$$\frac{l\ fresh \in \mathsf{MN}}{h : n[(\nu a.P) \mid Q]_{k,S} \stackrel{\equiv}{\rightarrow} h : n[P\{l/a\} \mid Q]_{k,S}}$$

Besides, we have $M\sigma \equiv N\sigma'$ with σ and σ' injective renamings of machine names such leaving invariant l, p and n. We can write $N \equiv L_1 \mid M_1''$ with $L_0\sigma \equiv L_1\sigma'$ and $M_0''\sigma \equiv M_1''\sigma'$ and

$$L_1 \equiv h' : n[(\nu a.P') \mid Q']_{k',S'}$$

Hence, we have

$$\frac{l'\ fresh \in \mathsf{MN}}{h' : n[(\nu a.P') \mid Q']_{k',S'} \stackrel{\equiv}{\rightarrow} h' : n[P'\{l'/a\} \mid Q']_{k',S'} = M_1'''}$$

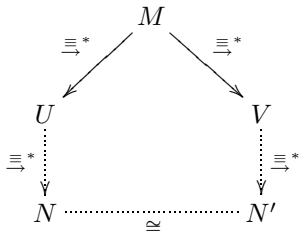
and $N \stackrel{\equiv}{\rightarrow} M_1''' \mid M_1'' = N'$. Moreover, we can suppose $l \notin \sigma(\mathbf{names}(M))$, $\sigma(l) = l$, $l' \notin \sigma'(\mathbf{names}(M'))$ and $\sigma'(l') = l'$ (or more precisely, we can define new renamings with these properties). If l'' is a fresh name, we define $\tau = \sigma\{l''/l\}$ and $\tau' = \sigma'\{l''/l'\}$. we then have $N'\tau \equiv M'\tau'$ with τ and τ' injective renamings (and leaving invariant l, n and p) and we deduce $M' \cong N'$. The two other cases are similar.

We proceed similarly for $\mapsto$. The result for $\rightarrow$ is then a consequence of rule M.NORM.

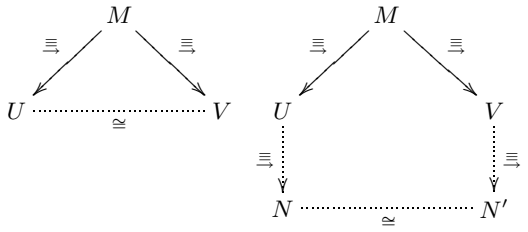
Lemma 12.

- *If* $M \stackrel{\equiv}{\rightarrow}^* U$ *and* $M \stackrel{\equiv}{\rightarrow}^* V$ *then* $U \stackrel{\equiv}{\rightarrow}^* N$ *and* $V \stackrel{\equiv}{\rightarrow}^* N'$ *with* $N \cong N'$.
- *If* $M \mapsto M'$ *and* $M \stackrel{\equiv}{\rightarrow}^* M''$ *then, for some* N, $M'' \mapsto N$ *and* $M' \stackrel{\equiv}{\rightarrow}^* \cong N$.

Proof. Let us prove the first part. It can be illustrated with the following diagram.



We first prove that if $M \stackrel{\equiv}{\rightarrow} U$ and $M \stackrel{\equiv}{\rightarrow} V$ then $U \cong V$ or $U \stackrel{\equiv}{\rightarrow} N$ and $V \stackrel{\equiv}{\rightarrow} N'$ with $N \cong N'$.



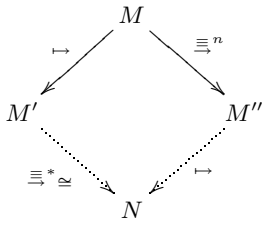
We reason by case on the reduction rules that lead to U and V. We can apply lemma 9: $M \equiv L_0 \mid M_0'$ and $L_0 \stackrel{\equiv}{\rightarrow} M_0''$ where the latter reduction corresponds to one of the axioms defining $\stackrel{\equiv}{\rightarrow}$. Similarly, $M \equiv L_1 \mid M_1'$ and $L_1 \stackrel{\equiv}{\rightarrow} M_1''$. If $L_0 \not\equiv L_1$ then $M \equiv L_0 \mid L_1 \mid M_2'$ and the result is a consequence of rule R.ST.CONTEXT (and maybe a renaming of machine names). If $L_0 \equiv L_1$, we have nine easy cases to consider, depending on the axioms used for each reduction.

For the general case, we consider the following predicate:

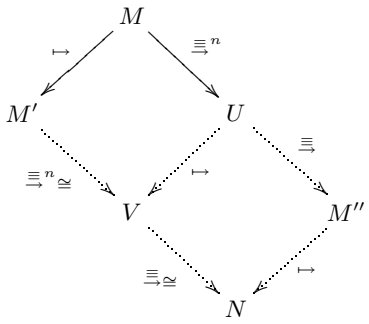
$$H(n,p) \stackrel{\Delta}{=} (M \stackrel{\equiv}{\rightarrow}^n U \wedge M \stackrel{\equiv}{\rightarrow}^p V) \implies$$
$$(U \stackrel{\equiv}{\rightarrow}^{n'} N \wedge V \stackrel{\equiv}{\rightarrow}^{p'} N' \text{ with } N \cong N', n' \leq \mathbf{max}(n,p), p' \leq \mathbf{max}(n,p))$$

The result follows from the proposition $\forall n, p \geq 0, H(n,p)$ which can be proved by induction on $N = \mathbf{max}(n,p)$ (using the previous case and the fact that $\cong$ is reduction-closed for the relation $\stackrel{\equiv}{\rightarrow}$).

Let us prove now the second part. It is enough to prove the following diagram:



We prove it by induction on n. Suppose the result is true for n, $M \mapsto M'$ and $M \stackrel{\equiv}{\rightarrow}^{n+1} M''$. We have $M \stackrel{\equiv}{\rightarrow}^{n} U \stackrel{\equiv}{\rightarrow} M''$ for some machine U. By induction, we have $U \mapsto V$ and $M' \stackrel{\equiv}{\rightarrow}^{*} V' \cong V$ for some machines V and V'. Furthermore, since $U \stackrel{\equiv}{\rightarrow} M''$ we can apply the case $n = 1$ and we have $V \stackrel{\equiv}{\rightarrow} \cong N$ and $M'' \mapsto N$ for some machine N. From $M' \stackrel{\equiv}{\rightarrow}^{*} \cong V$, $V \stackrel{\equiv}{\rightarrow}^{*} \cong N$ and lemma 11, we deduce that $M' \stackrel{\equiv}{\rightarrow}^{*} \cong N$.



It remains to prove the result for $n = 1$. By lemmas 9 and 10, we have $M \equiv M_0 \mid M_1$ and $M \equiv M_0' \mid M_1'$ with $M_0 \stackrel{\equiv}{\rightarrow} N_0$ and $M_0' \mapsto N_0'$ where only the axioms rules are used in these two reductions. If M_0 and M_0' correspond to different localities, then the result is immediate. Otherwise, there are several cases to consider, depending on the axioms used to derive the reductions. All the cases being similar, we only consider one case corresponding to the axioms M.S.NEW and M.OUT where the locality reduced by M.S.NEW is the one containing the message in M.OUT. We have

$$M_0 \equiv h : a[(\nu c.P) \mid P']_{h',S} \stackrel{\equiv}{\rightarrow} h : a[P\{l/c\} \mid P']_{h',S}$$

$$M_0' \equiv h : a[\xi\varphi \mid P'']_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'} \mapsto h : a[P'']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$$

and

$$M_0 \equiv h : a[(\nu c.P) \mid P']_{h',S} \equiv h : a[\xi\varphi \mid P'']_{h',S}$$

We deduce that M_0 and M_0' can be written

$$M_0 \equiv h : a[(\nu c.P) \mid \xi\varphi \mid R']_{h',S}$$

$$M_0' \equiv h : a[\xi\varphi \mid (\nu c.P) \mid R']_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'}$$

Moreover, we have

$$N_0 \equiv h : a[P\{l/c\} \mid \xi\varphi \mid R']_{h',S}$$
$$N_0' \equiv h : a[(\nu c.P) \mid R']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$$

By the rules M.OUT, M.S.NEW and M.S.CXT we have

$$h : a[P\{l/c\} \mid \xi\varphi \mid R']_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'}$$
$$\mapsto h : a[P\{l/c\} \mid R']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$$
$$h : a[(\nu c.P) \mid R']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$$
$$\stackrel{\equiv}{\rightarrow} h : a[P\{l'/c\} \mid R']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$$

Finally, if $N_0'' = h : a[P\{l/c\} \mid R']_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}$ and $N = N_0'' \mid M_1'$, we have $N_0 \mid M_1 \mapsto N$ and $N_0' \mid M_1' \stackrel{\equiv}{\rightarrow}\cong N$.

Lemma 13. *A relation $\mathcal{R}$ is contextual if $M\mathcal{R}N$ implies that $\mathbf{E}\{M\}\mathcal{R}\mathbf{E}\{N\}$ for any context $\mathbf{E}$. If M is a machine such that $\mathbf{tree}(M, l, n, p)$ and $\mathbf{E}$ is a context, we have $\mathbf{tree}(\mathbf{E}\{M\}, l, n, p)$. Moreover, the relations $\equiv$, $\cong$, $\mapsto$, $\stackrel{\equiv}{\rightarrow}$, $\rightarrow$ and $\doteq$ are contextual.*

Proof. We prove the preservation of the tree structure and the contextuality of $\equiv$, $\mapsto$, $\cong$ and $\stackrel{\equiv}{\rightarrow}$ by induction on the structure of $\mathbf{E}$. We detail the proof for $\cong$. Suppose $M \cong N$ with $\mathbf{tree}(M, l, n, p)$. If $\mathbf{E} = .$, the result is immediate. Suppose that for a context $\mathbf{E}$ we have $\mathbf{E}\{M\}\sigma \equiv \mathbf{E}\{N\}\sigma'$ with σ, σ' injective renamings such that $\sigma(l) = \sigma'(l) = l$ and $\sigma(p) = \sigma'(p) = p$. We can write $\mathbf{E}\{M\} = l : n[P]_{p,S} \mid M'$ and $\mathbf{E}\{N\} = l : n[P']_{p,S'} \mid N'$.

$\mathbf{E} = a[\mathbf{E}]$

$$a[\mathbf{E}\{M\}\sigma] = l : n[\mathbf{0}]_{p,h} \mid h : a[P\sigma]_{l,S\sigma} \mid M'\sigma\{h/l\}$$
$$a[\mathbf{E}\{N\}\sigma'] = l : n[\mathbf{0}]_{p,h'} \mid h' : a[P'\sigma']_{l,S'\sigma'} \mid N'\sigma'\{h'/l\}$$

We notice that $a \notin \mathbf{names}(\mathbf{E}\{M\}) \cup \mathbf{names}(\mathbf{E}\{N\})$. Moreover, we can choose σ and σ' such that $h \notin \sigma(\mathbf{names}(\mathbf{E}\{M\}))$, $\sigma(h) = h$, $h' \notin \sigma'(\mathbf{names}(M'))$ and $\sigma'(h') = h'$. Hence, if $\tau = \sigma\{h''/h\}$ and $\tau' = \sigma'\{h''/h'\}$, we have $a[\mathbf{E}\{M\}]\tau \equiv a[\mathbf{E}\{N\}]\tau'$ with τ and τ' injective renamings.

$\mathbf{E} = \mathbf{E} \mid Q$

$$\mathbf{E}\{M\}\sigma \mid Q = l : n[P\sigma \mid Q]_{p,S\sigma} \mid M'\sigma$$
$$\mathbf{E}\{N\}\sigma' \mid Q = l : n[P'\sigma' \mid Q]_{p,S'\sigma'} \mid N'\sigma'$$

We conclude that $(\mathbf{E}\{M\} \mid Q)\sigma \equiv (\mathbf{E}\{N\} \mid Q)\sigma'$ by noticing that $\mathbf{names}(Q) \cap \mathbf{names}(\mathbf{E}\{M\}) = \emptyset$ and $\mathbf{names}(Q) \cap \mathbf{names}(M') = \emptyset$.

$\mathbf{E} = \nu c.\mathbf{E}$ Similar to the previous cases.

$\mapsto$ and $\stackrel{\equiv}{\rightarrow}$ are contextual, consequently $\rightarrow$ is contextual by rule M.NORM. $\doteq$ is contextual as a consequence of corollary 1 and the contextuality of $\stackrel{\equiv}{\rightarrow}$ and $\cong$.

Lemma 14. *For any well-formed machine M we have, $M \downarrow_a$ if and only if $M_* \cong \downarrow_a$.*

Proof. Let us prove first the direct implication. First notice that if M and N are well-formed machine we have

$$M \cong N \implies (\forall a \in \mathsf{N}, M \downarrow_a \iff N \downarrow_a)$$

The reason is that M and N differ only by a renaming of locality names other than $\mathbf{r}$ and $\mathbf{rp}$, and that these names do not intervene in the definition of the observability predicate. It is then enough to show that if M is a machine such that $M \xrightarrow{\equiv} M'$ and $M \downarrow_a$ then $M' \downarrow_a$. We show it easily with lemma 9, by considering the three possible cases. The converse implication follows immediately from definition 4.

Proof (Proof of lemma 1). We first prove that $\xrightarrow{\equiv}$ is terminating. Given a machine term M, we define $\mathbf{size}(M)$ as the cardinal of the multiset

$$\{P \in \mathsf{MK}, (M \equiv h : a[P \mid R]_{h',S} \mid M') \wedge (P = \nu b.Q \vee P = b[Q] \vee P = \mathbf{reify}(k, M_*))\}$$

where we identify structurally equivalent processes. It is easy to see that $M \xrightarrow{\equiv} M'$ for some M' if and only if $\mathbf{size}(M) > 0$ and then $\mathbf{size}(M') < \mathbf{size}(M)$. Besides, $\mathbf{size}(M) = 0$ if and only if M is in normal form. We deduce immediately that $\xrightarrow{\equiv}$ is terminating. Furthermore, the unicity up to $\cong$ of the normal form is a consequence of the first part of lemma 12.

Corollary 1. *If $M \xrightarrow{\equiv}{}^* \cong M'$, then $M_* \cong M'_*$.*

Proof. Suppose $M \xrightarrow{\equiv}{}^* N \cong M'$. By lemma 1 we have, $N \xrightarrow{\equiv}{}^* N_*$ and $M_* \cong N_*$. By lemma 11, we have $M' \xrightarrow{\equiv}{}^* M''$ with $M'' \cong N_*$. It is easy to see that M'' is a machine in normal form. Thus, by lemma 1, $M'' \cong M'_*$ and finally, $M_* \cong M'_*$.

Lemma 15.

- *If $M \to N$ then $M_* \to \xrightarrow{\equiv}{}^* \cong N_*$.*
- *If $M_* \to N$ then $M \to \xrightarrow{\equiv}{}^* N_*$.*

Proof. The second part is a consequence of lemma 1 and rule M.NORM. For the first part, we have $M \xrightarrow{\equiv}{}^* M' \xrightarrow{\equiv}{}^* M_*$ and $M' \mapsto N$. If follows from lemma 12 that $M_* \mapsto \xrightarrow{\equiv}{}^* \cong N'$ with $N \xrightarrow{\equiv}{}^* N'$. In particular, $N_* \cong N'_*$. Besides, by definition $N' \xrightarrow{\equiv}{}^* N'_*$ and $\cong$ is reduction-closed for $\xrightarrow{\equiv}$ (lemma 11) , hence $M_* \mapsto \xrightarrow{\equiv}{}^* \cong N_*$.

Proof (Proof of lemma 3). $\equiv \subseteq \cong$ follows from the definition of $\cong$. From corollary 1, we have $\cong \subseteq \doteq$. Now we will consider the restrictions of these relations to well-formed machines. They are all barb-preserving (cf. lemma 14). $\equiv$ is reduction-closed as a consequence of rules M.STR and M.S.STR. $\cong$ is reduction-closed by lemma 11. Let us show that $\doteq$ is reduction-closed: suppose we have $M \doteq N$ and $M \to M'$. By the first part of lemma 15, we have $M_* \to U \xrightarrow{\equiv}{}^* \cong M'_*$ and we deduce $N_* \to V \cong U$ ($\cong$ is reduction-closed). Then the second part of lemma 15 tells us that $N \to V' \xrightarrow{\equiv}{}^* V_*$. Finally, we have $N \to V'$ with $V' \doteq M'$.

We conclude that $\equiv$, $\cong$ and $\dot{=}$ are barb-preserving and reduction-closed, hence strong barbed bisimulation. Finally, $\dot{=}$ is preserved by every evaluation context $\mathbf{E}$ by lemma 13. $\sim_c$ is the largest strong bisimulation preserved by every context, hence $\dot{=} \subseteq \sim_c$.

Completeness

Lemma 16.

- *If* $P \equiv P'$ *then* $[\![P]\!] \dot{=} [\![P']\!]$
- *If* $P \stackrel{\equiv}{\rightarrow} P'$ *then* $[\![P]\!] \dot{=} [\![P']\!]$

Proof. We first generalize definition 1 as follows:

$$[\![P]\!]_{l,n,p} = l : n[P]_{p,\emptyset}$$

We prove the more general result that for any name n and distinct machine names l and p we have that if $P \equiv P'$ then $[\![P]\!]_{l,n,p} \dot{=} [\![P]\!]_{l,n,p}$. Let us proceed by induction on the derivation of $P \equiv P'$. The $\equiv$ relation over machine processes is still monoidal for the parallel composition with $\mathbf{0}$ as a neutral element. Hence, using M.SE.CTX we show easily the result when $P \equiv P'$ corresponds to one of these monoidal rules (the argument is the same for the S.α rule). Let us detail the other rules.

S.NU.NIL $P = \nu a.\mathbf{0}$ and $P' = \mathbf{0}$. $[\![P]\!]_{l,n,p} \stackrel{\equiv}{\rightarrow} l : n[\mathbf{0}]_{p,\emptyset} = [\![P]\!]_{l,n,p}$. We conclude by corollary 1 that $[\![P]\!]_{l,n,p} \dot{=} [\![P]\!]_{l,n,p}$.

S.NU.COMM, S.NU.PAR Similar to the previous case.

S.CONTEXT It is enough to show it for the three basic contexts $a[\,\cdot\,]$, $\nu a.\cdot$ and $\cdot \mid R$. Let us prove the third case, the two others being similar. By induction, we know that $[\![P]\!]_{l,n,p} \stackrel{\equiv}{\rightarrow}^* N_*$ and $[\![P]\!]_{l,n,p} \stackrel{\equiv}{\rightarrow}^* N'_*$ with $N_* \cong N'_*$. We have $[\![P]\!]_{l,n,p} = [\![P]\!]_{l,n,p} \mid R \stackrel{\equiv}{\rightarrow}^* N_* \mid R$ and similarly $[\![P]\!]_{l,n,p} = \stackrel{\equiv}{\rightarrow}^* N'_* \mid R$. Hence $[\![P]\!]_{l,n,p} \dot{=} [\![P]\!]_{l,n,p}$ by corollary 1.

The proof of the second part is similar.

Corollary 2.

- *If* $P \equiv P'$ *then* $[\![P]\!] \sim_c [\![P']\!]$
- *If* $P \stackrel{\equiv}{\rightarrow} P'$ *then* $[\![P]\!] \sim_c [\![P']\!]$

Proof (Proof of Lemma 8). We first prove that the relation

$$S = \{(N\{\mathbf{reify}(p, M_*)/x\}, N\{P_*/x\}), N \in \mathsf{WFM}\}$$

is a bisimulation up to $\dot{=}$. We will need the following definition and sub-lemmas.

Definition 9. *An occurrence of a free variable x is* guarded *in M if it does not appear in the continuation of a pattern or in a message. A free variable is said to be guarded if all of its occurrences are guarded.*

Lemma 17. *If (U, V) is in S, then there is N_* with x guarded in N_* such that*

$$U_* \cong N_*\{\mathbf{reify}(p, M_*)/x\}$$

$$V_* \cong N_*\{P_*/x\}$$

Proof. Let (U, V) be in S. We consider two cases. First, suppose that x is guarded in N. It is easy to see that $U_* \cong N_*\{\mathbf{reify}(p, M_*)/x\}$ and $V_* \cong N_*\{P_*/x\}$. The reason is that a guarded occurrence of a variable cannot become unguarded after a structural reduction. In the second case, we define a term N' by renaming all guarded occurrences of x in N by a fresh variable name y. We then have

$$U = N'\{\mathbf{reify}(p, M_*)/y\}\{\mathbf{reify}(p, M_*)/x\}$$

$$V = N'\{P_*/y\}\{P_*/x\}$$

We will show that $U \overset{\equiv}{\to}^* \cong N''\{\mathbf{reify}(p, M_*)/y\}$ and $V \overset{\equiv}{\to}^* \cong N''\{P_*/y\}$ with all occurrences of y guarded. Hence, we can conclude with the previous case. To simplify, we suppose that N' has only one (not guarded) occurrence of x. The reasoning is the same as in the general case (by induction on the number of occurrences of x). We assume that $M_* = l : n[R_*]_{l',S'} \mid M'_*$. First notice that, $N' \overset{\equiv}{\to}^* M' = h : m[x \mid Q]_{k,S} \mid M''$. We then have

$$\begin{aligned} N'\{\mathbf{reify}(p, M_*)/x\} &\overset{\equiv}{\to} h : m[\mathbf{reify}(l, M_*) \mid Q]_{k,S} \mid M'' \\ &\overset{\equiv}{\to} h : m[R_* \mid Q]_{k,(S,S'\{k_i/l_i\}_{i\in I})} \mid M'_*\{h/l\}\{k_i/l_i\}_{i\in I} \mid M'' \\ &= U' \end{aligned}$$

$$\begin{aligned} N'\{P_*/x\} &\overset{\equiv}{\to} h : m[P_* \mid Q]_{k,S} \mid M'' \\ &\overset{\equiv}{\to}^* h : m[R_* \mid Q]_{k,(S,S'\{k'_i/l_i\}_{i\in I})} \mid M'_*\{h/l\}\{k'_i/l_i\}_{i\in I} \mid M'' \\ &= V' \end{aligned}$$

We deduce the result where $N' = U'\sigma = V'\sigma'$ with σ and σ' injective renamings such that $\sigma(k_i) = \sigma'(k'_i)$.

Lemma 18. *If $(U, V) \in S$ and $U \to U'$, there is a V' such that $V \to V'$ and $(U''_*, V''_*) \in S$ with $U''_* \cong U'_*$ and $V''_* \cong V'_*$.*

Proof. First, as a consequence of lemma 15 and by definition of $\to$, we know that there is a A such that $U \overset{\equiv}{\to}^* U_* \mapsto A \overset{\equiv}{\to} \cong U'_*$. Secondly, from lemma 17, $U_* \cong B_*\{\mathbf{reify}(p, M_*)/x\}$ and $V_* \cong B_*\{P_*/x\}$ for some machine term B_* with x guarded in B_*. Using this latter fact, we obtain

$$U_* \to U' \iff \begin{cases} B_*(x) \to B'(x) \\ U' \cong B'\{\mathbf{reify}(p, M_*)/x\} \end{cases}$$

We deduce that there is B' such that $U' \cong B'\{\mathbf{reify}(p, M_*)/x\}$ and $V_* \to V' \cong B'\{P_*/x\}$. By rule M.NORM, we conclude $V \to V'$.

If (U, V) is in S and $U \to U'$, we deduce immediately from lemma 18 that $V \to V'$ for some V' such that $U' \doteq S \doteq V'$. Furthermore we need to show that $U \downarrow a \iff V \downarrow a$. By lemma 14, it is enough to prove that $U_* \downarrow a \iff V_* \downarrow a$. By lemma 17 we know that $U_* = N_*\{\mathbf{reify}(p, M_*)/x\}$ and $V_* = N_*\{P_*/x\}$ with x guarded in N_*. We deduce that U and V have the same barbs. Finally, we have proved that S is a strong barbed bisimulation up to $\doteq$ and

$$\forall N \in \mathsf{WFM}, N\{\mathbf{reify}(p, M_*)/x\} \sim N\{P_*/x\}$$

Now, let us prove the result for $\sim_c$. Let N be a well-formed machine, it is enough to show that for any context $\mathbf{E}$, we have

$$\mathbf{E}\{N\{\mathbf{reify}(p, M_*)/x\}\} \sim \mathbf{E}\{N\{P_*/x\}\}$$

Let $\mathbf{E}$ be a context, Q a machine process and y a fresh variable, we have

$$\begin{aligned}\mathbf{E}\{N\{Q/x\}\} &= \mathbf{E}\{N\{y/x\}\{Q/y\}\} \\ &= \mathbf{E}\{N\{y/x\}\}\{Q/y\} \\ &= N'\{Q/y\} \qquad\qquad \text{with } N' = \mathbf{E}\{N\{y/x\}\}\end{aligned}$$

But we know that $N'\{\mathbf{reify}(p, M_*)/y\} \sim N'\{P_*/y\}$ from the previous result. Hence we deduce $\mathbf{E}\{N\{\mathbf{reify}(p, M_*)/x\}\} \sim \mathbf{E}\{N\{P_*/x\}\}$.

Proof (Proof of proposition 3). By induction on the derivation of $P \to P'$.

R.OUT We have $P = \xi \triangleright Q \mid b[R \mid Q\varphi]$ and $P' = Q\varphi \mid b[R]$.

$$\begin{aligned}[\![P]\!] &= \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset} \\ &\overset{\equiv}{\to} \mathbf{r} : \mathbf{rn}[\xi \triangleright Q]_{\mathbf{rp},p} \mid p : b[R \mid Q\varphi]_{\mathbf{r},\emptyset} \qquad \text{(M.S.CELL)}\end{aligned}$$

$$\begin{aligned}[\![P']\!] &= \mathbf{r} : \mathbf{rn}[Q\varphi \mid b[R]]_{\mathbf{r},\emptyset} \\ &\overset{\equiv}{\to} \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},q} \mid q : b[R]_{\mathbf{r},\emptyset} \qquad \text{(M.S.CELL)}\end{aligned}$$

Besides, we have by definition

$$p : b[R \mid Q\varphi]_{\mathbf{r},\emptyset} = p : b[R]_{\mathbf{r},\emptyset} \mid Q\varphi$$

By lemma 1

$$p : b[R]_{r,\emptyset} \overset{\equiv}{\to}^* p : b[R'_*]_{r,S} \mid M_*$$

$$q : b[R]_{r,\emptyset} \overset{\equiv}{\to}^* q : b[R''_*]_{r,S'} \mid N_*$$

From lemma 13 we deduce

$$p : b[R \mid Q\varphi]_{r,\emptyset} \overset{\equiv}{\to}^* p : b[R'_* \mid Q\varphi]_{r,S} \mid M_*$$

Finally, we have

$$\begin{aligned}[\![P]\!] &\xrightarrow{\equiv}{}^{*} \mathbf{r} : \mathbf{rn}[\xi \triangleright Q]_{\mathbf{rp},p} \mid p : b[R'_* \mid Q\varphi]_{\mathbf{r},S} \mid M_* \\ &\to \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},p} \mid p : b[R'_*]_{\mathbf{r},S} \mid M_* \qquad \text{(M.OUT)} \\ &= U\end{aligned}$$

$$\begin{aligned}[\![P']\!] &\xrightarrow{\equiv}{}^{*} \mathbf{r} : \mathbf{rn}[Q\varphi]_{\mathbf{rp},q} \mid q : b[R''_*]_{\mathbf{r},S'} \mid N_* \\ &= V\end{aligned}$$

We conclude by remarking that $U \cong V$ and $[\![P']\!] \doteq V$. By lemmas 2 and 3, we deduce $U \sim_c [\![P']\!]$.

R.IN**,** R.LOCAL These two cases are handled in the same way as case R.OUT.

R.PASS

$$\begin{aligned}[\![P]\!] &= \mathbf{r} : \mathbf{rn}[a[P_*] \mid (a[x] \triangleright Q)]_{\mathbf{rp},\emptyset} \\ &\xrightarrow{\equiv} \mathbf{r} : \mathbf{rn}[a[x] \triangleright Q]_{\mathbf{rp},p} \mid p : a[P_*]_{\mathbf{r},\emptyset} \qquad \text{(M.S.CELL)}\end{aligned}$$

By lemma 1 and proposition 1, we have $p : a[P_*]_{r,\emptyset} \xrightarrow{\equiv}{}^{*} M_*$ with $\mathbf{tree}(M_*, p, a, r)$. We deduce

$$\begin{aligned}[\![P]\!] &\xrightarrow{\equiv}{}^{*} \mathbf{r} : \mathbf{rn}[a[x] \triangleright Q]_{\mathbf{rp},p} \mid M_* \\ &\to \mathbf{r} : \mathbf{rn}[Q\{\mathbf{reify}(p, M_*)/x\}]_{\mathbf{rp},\emptyset} \qquad \text{(M.PASS)} \\ &= U\end{aligned}$$

Moreover, since we have $[\![P']\!] = \mathbf{r} : \mathbf{rn}[Q\{P_*/x\}]_{\mathbf{rp},\emptyset}$ we can apply lemma 8 and conclude $[\![P']\!] \sim_c U$.

R.CONTEXT By induction, we have $[\![P]\!] \to M$ and $M \sim_c M' = [\![P']\!]$. We want to prove that for any context E such that $\mathbf{E}\{P\} \to \mathbf{E}\{P'\}$ we have $[\![\mathbf{E}\{P\}]\!] \to N$ and $N \sim_c N' = [\![\mathbf{E}\{P'\}]\!]$. We proceed by (sub)induction on the context E.

$\mathbf{E} = .$ Immediate by induction.

$\mathbf{E} = a[\mathbf{E}]$

$$\begin{aligned}[\![a[\mathbf{E}\{P\}]]\!] &= \mathbf{r} : \mathbf{rn}[a[\mathbf{E}\{P\}]]_{\mathbf{rp},\emptyset} \\ &\xrightarrow{\equiv}\cong a[[\![\mathbf{E}\{P\}]\!]] \\ &\to a[N] \\ &\sim_c a[N']\end{aligned}$$

Furthermore, $[\![a[\mathbf{E}\{P'\}]]\!] \xrightarrow{\equiv}\cong a[[\![\mathbf{E}\{P'\}]\!]]$ and by corollary 1, $[\![a[\mathbf{E}\{P'\}]]\!] \doteq a[[\![\mathbf{E}\{P'\}]\!]]$ and hence $[\![a[\mathbf{E}\{P'\}]]\!] \sim_c a[[\![\mathbf{E}\{P'\}]\!]]$. Besides, $a[[\![\mathbf{E}\{P'\}]\!]] \sim_c a[N']$ because $\sim_c$ is a congruence. We conclude $[\![a[\mathbf{E}\{P'\}]]\!] \sim_c a[N']$.

$\mathbf{E} = \mathbf{E} \mid Q$ As in the previous case, we have

$$\begin{aligned}
[\![\mathbf{E}\{P\} \mid Q]\!] &= \mathbf{r} : \mathbf{rn}[\mathbf{E}\{P\} \mid Q]_{\mathbf{rp},\emptyset} \\
&= [\![\mathbf{E}\{P\}]\!] \mid Q \\
&\to N \mid Q \\
&\sim_c N' \mid Q
\end{aligned}$$

Besides, we have $[\![\mathbf{E}\{P'\} \mid Q]\!] = [\![\mathbf{E}\{P'\}]\!] \mid Q \sim_c N' \mid Q$.

$\mathbf{E} = \nu c.\mathbf{E}$ As before

$$\begin{aligned}
[\![\nu c.\mathbf{E}\{P\}]\!] &= \mathbf{r} : \mathbf{rn}[\nu c.\mathbf{E}\{P\}]_{\mathbf{rp},\emptyset} \\
&\xrightarrow{\equiv} \mathbf{r} : \mathbf{rn}[\mathbf{E}\{P\}\{i/c\}]_{\mathbf{rp},\emptyset} \\
&\cong \nu c.[\![\mathbf{E}\{P\}]\!] \\
&\to \nu c.N \\
&\sim_c \nu c.N' \\
&\sim_c [\![\nu c.\mathbf{E}\{P'\}]\!]
\end{aligned}$$

Hence, we have $[\![\nu c.\mathbf{E}\{P\}]\!] \to \sim_c [\![\nu c.\mathbf{E}\{P'\}]\!]$.

R.STRUCT By lemma 16, $[\![P]\!] \doteq [\![P']\!]$ and $[\![Q]\!] \doteq [\![Q']\!]$. Besides, by induction, we have $[\![P]\!] \to M \sim_c [\![Q]\!]$. Since $\doteq$ is a bisimulation (lemma 3), we have $[\![P']\!] \to M'$ with $M' \doteq M$. Finally, since $\doteq \subseteq \sim_c$ (lemma 3), and $\sim_c$ is an equivalence (lemma 2), we deduce that $M' \sim_c [\![Q']\!]$.

R.STRUCT.EXTR Same reasoning as R.STRUCT.

Soundness We will now prove proposition 2. For that purpose, we need a second translation function $[\![.]\!]^{mac}$ from machines to processes. The definition uses two mutually recursive functions:

- $[\![.]\!]^{proc}$ from machine processes to machine processes.
- $[\![.]\!]^{loc}_{.}$ from machines M such that $\mathbf{tree}(M, l, a, p)$ to machine processes.

The first one is almost the identity, except that it "expands" reified processes using the second one. The latter has two arguments: a machine, and the locality name in this machine corresponding to the root of the tree. The machine processes returned by this two functions may use some machine names, and thus may not be kell calculus processes. The $[\![.]\!]^{mac}$ function abstracts these machine names (to be precise, these names should be renamed into fresh process names).

Definition 10.

$$
\begin{aligned}
[\![\mathbf{0}]\!]^{proc} &= \mathbf{0} \\
[\![x]\!]^{proc} &= x \\
[\![\xi \triangleright P]\!]^{proc} &= \xi \triangleright [\![P]\!]^{proc} \\
[\![\nu a.P]\!]^{proc} &= \nu a.[\![P]\!]^{proc} \\
[\![P \mid P']\!]^{proc} &= [\![P]\!]^{proc} \mid [\![P']\!]^{proc} \\
[\![a[P]]\!]^{proc} &= a[[\![P]\!]^{proc}] \\
[\![a\langle \widetilde{P} \rangle]\!]^{proc} &= a\langle \widetilde{[\![P]\!]^{proc}} \rangle \\
[\![\mathbf{reify}(k, M_*)]\!]^{proc} &= [\![M_*]\!]^{loc}_k
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{0}]\!]^{loc}_p &= \mathbf{0} \\
[\![M]\!]^{loc}_l &= [\![P]\!]^{proc} \mid \Pi_{i \in 1..n} n_i[[\![M_i]\!]^{loc}_{l_i}] \\
&\quad \textit{if } M \equiv l : a[P]_{p,S} \mid M_1 \mid \ldots \mid M_n \textit{ with } \mathbf{tree}(M_i, l_i, n_i, l)
\end{aligned}
$$

$$
[\![M]\!]^{mac} = \nu \widetilde{c}.[\![M]\!]^{loc}_{\mathbf{r}} \ (\widetilde{c} = \mathbf{mnames}(M))
$$

The following lemma will be central for the proof of the soundness proposition. intuitively, it allows us to isolate the part of a tree that can react.

Lemma 19. *If M is machine term such that*

- $\mathbf{tree}(M, u, a, v)$
- $M \equiv l : n[P]_{k,S} \mid M' \mid M''$
- $\mathbf{tree}(l : n[P]_{k,S'} \mid M', l, n, k)$
- $S' \subseteq S$

then there exists a (machine process) context E and a machine process Q such that

$$
[\![M]\!]^{loc}_u \equiv E\{[\![l : n[P \mid Q]_{k,S'} \mid M']\!]^{loc}_l\}
$$

Moreover, E and Q do not depend on $l : n[P]_{k,S'} \mid M'$.

Proof. We prove the result by induction on the size of the tree corresponding to M. If $M = u : a[P]_{v,\emptyset}$ then $a = n$, $u = l$, $v = k$ and $S = S' = \emptyset$. We can choose $E = \cdot$ and $Q = \mathbf{0}$. In the general case, assuming that

$$
M \equiv u : a[T]_{v,R} \mid M_1 \mid \ldots \mid M_n \text{ with } \mathbf{tree}(M_i, l_i, p_i, u)
$$

we have

$$
[\![M]\!]^{loc}_u \equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]^{loc}_{l_1}] \mid \ldots p_n[[\![M_n]\!]^{loc}_{l_n}]
$$

We can distinguish two cases: $u = l$, or the node l belongs to one of the subtrees of root l_i. In the first case, we have $a = n$, $u = l$, $v = k$, $S = R$ and we can suppose that $S' = 1..m$ with $m \leq n$.

$$\begin{aligned}
[\![M]\!]_u^{loc} &\equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]_{l_1}^{loc}] \mid \ldots \mid p_m[[\![M_m]\!]_{l_m}^{loc}] \mid \ldots \mid p_n[[\![M_n]\!]_{l_n}^{loc}] \\
&\equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]_{l_1}^{loc}] \mid \ldots p_m[[\![M_m]\!]_{l_m}^{loc}] \mid Q \\
&\equiv [\![T]\!]^{proc} \mid [\![Q]\!]^{proc} \mid p_1[[\![M_1]\!]_{l_1}^{loc}] \mid \ldots p_m[[\![M_m]\!]_{l_m}^{loc}] \\
&\equiv [\![T \mid Q]\!]^{proc} \mid p_1[[\![M_1]\!]_{l_1}^{loc}] \mid \ldots p_m[[\![M_m]\!]_{l_m}^{loc}] \\
&\equiv [\![l : n[T \mid Q]_{k,S'} \mid M_1 \mid \ldots \mid M_m]\!]_u^{loc}
\end{aligned}$$

We have the result with $E = \cdot$ and $Q = p_1[[\![M_{m+1}]\!]_{l_{m+1}}^{loc}] \mid \ldots \mid p_m[[\![M_n]\!]_{l_n}^{loc}]$.
In the second case, we can suppose that the node l belongs for instance to the subtree of root l_n. By induction, we have

$$[\![M_n]\!]_{l_n}^{loc} \equiv E\{[\![l : n[P \mid Q]_{k,S'} \mid M']\!]_l^{loc}\}$$

We deduce that

$$[\![M]\!]_u^{loc} \equiv E'\{[\![l : n[P \mid Q]_{k,S'} \mid M']\!]_l^{loc}\}$$

with

$$E' \equiv [\![T]\!]^{proc} \mid p_1[[\![M_1]\!]_{l_1}^{loc}] \mid \ldots \mid p_{n-1}[[\![M_{n-1}]\!]_{l_{n-1}}^{loc}] \mid E$$

Corollary 3. *If M is machine term such that*

- $M \equiv l : n[P]_{k,S} \mid M' \mid M''$
- $\mathbf{tree}(l : n[P]_{k,S'} \mid M', l, n, k)$
- $S' \subseteq S$

then there exists a context E and a machine process Q such that

$$[\![M]\!]^{mac} \equiv \nu\widetilde{c}.E\{[\![l : n[P \mid Q]_{k,S'} \mid M']\!]_l^{loc}\}$$

with $\widetilde{c} = \mathbf{mnames}(l : n[P]_{k,S'} \mid M')$. Moreover, E and Q do not depend on $l : n[P]_{k,S'} \mid M'$.

Proof (proof of Lemma 4). According to lemma 9, it must be that $M \equiv L \mid M'$ with $L = l : n[P]_{k,S'}$ and $L \stackrel{\equiv}{\to} M''$. We reason by case on the form of $L \stackrel{\equiv}{\to} M''$.

M.S.CELL We have $L = h : n[m[P] \mid Q]_{k,S}$ and $M'' = h : n[Q]_{k,(S,h')} \mid h' : m[P]_{h,\emptyset}$. By lemma 19 we deduce

$$[\![M]\!]^{mac} = E\{[\![h : n[m[P] \mid Q]_{k,\emptyset}]\!]_h^{loc}\}$$

$$[\![N]\!]^{mac} = E\{[\![h : n[Q]_{k,\{h'\}} \mid h' : m[P]_{h,\emptyset}]\!]_h^{loc}\}$$

Then, with the definition of function $[\![.]\!]_.^{loc}$ it is easy to check that $[\![M]\!]^{mac} \equiv [\![N]\!]^{mac}$.

M.S.NEW In that case, $L = h : n[(\nu c.P) \mid Q]_{k,S}$ and $M'' = h : n[P\{l/c\} \mid Q]_{k,S}$.
We can apply lemma 19

$$[\![M]\!]^{mac} = E\{[\![h : n[(\nu c.P) \mid Q]_{k,\emptyset}]\!]^{loc}_h\}$$

$$[\![N]\!]^{mac} = \nu l.E\{[\![h : n[P\{l/m\} \mid Q]_{k,\emptyset}]\!]^{loc}_h\}$$

It is then easy to show that $[\![M]\!]^{mac} \stackrel{\equiv}{\rightarrow}^{*} [\![N]\!]^{mac}$.

M.S.ACT As before, we apply lemma 19

$$[\![M]\!]^{mac} = E\{[\![h : m[\mathbf{reify}(l, M_*) \mid P]_{k,\emptyset}]\!]^{loc}_h\}$$

$$[\![N]\!]^{mac} = E\{[\![h : m[R_* \mid P]_{k,S'\{k_i/l_i\}_{i\in I}} \mid M'_*\{h/l\}\{k_i/l_i\}_{i\in I}]\!]^{loc}_h\}$$

and we check that $[\![M]\!]^{mac} \equiv [\![N]\!]^{mac}$.

Proof (proof of 5). We reason by case on the derivation $L \rightarrow L'$ using lemma 10.

M.OUT

$$M \equiv h : a[\xi\varphi \mid P]_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'} \mid M'$$

$$N \equiv h : a[P]_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'} \mid M'$$

We can apply lemma 19:

$$[\![M]\!]^{mac} = \mathbf{E}\{[\![h' : b[(\xi \triangleright Q) \mid R]_{k',\{h\}} \mid h : a[\xi\varphi \mid P]_{h',S} \mid M']\!]^{loc}_{k'}\}$$

$$[\![N]\!]^{mac} = \mathbf{E}\{[\![h' : b[Q\varphi \mid R]_{k',\{h\}} \mid h : a[P]_{h',S} \mid M']\!]^{loc}_{k'}\}$$

Then we check that $[\![M]\!] \rightarrow [\![N]\!]$.

M.IN, M.LOCAL Similar to the previous case.

M.PASS We proceed as before

$$M \equiv h : m[(a[x] \triangleright P) \mid Q]_{k,S} \mid M_* \mid M'$$

$$N \equiv h : m[P\{\mathbf{reify}(l, M_*)/x\} \mid Q]_{k,S\setminus l} \mid M'$$

Then by lemma 19,

$$[\![M]\!]^{mac} = \mathbf{E}\{[\![h : m[(a[x] \triangleright P) \mid Q]_{k,\{l\}} \mid M_* \mid M']\!]^{loc}_k\}$$

$$[\![N]\!]^{mac} = \mathbf{E}\{[\![h : m[P\{\mathbf{reify}(l, M_*)/x\} \mid Q]_{k,\emptyset} \mid M']\!]^{loc}_k\}$$

We verify that $[\![M]\!] \equiv [\![N]\!]$.

Lemma 20. *If $M \rightarrow N$ then $[\![M]\!]^{mac} \rightarrow [\![N]\!]^{mac}$.*

Proof. Immediate using lemmas 5 and 4.

Lemma 21. *For any machine M and N_* such that $\mathbf{tree}(N_*, p, n, q)$ and M is well-formed, we have*

$$M\{\mathbf{reify}(p, N_*)/x\} \sim_c M\{[\![N_*]\!]^{loc}_p/x\}$$

Proof. By lemma 8, it is enough to show that $p : n[[\![N_*]\!]_p^{loc}]_{q,\emptyset} \stackrel{\equiv}{\rightarrow}^* \cong N_*$. We prove it by induction on the size of the tree N_*. If N_* consists in only one locality, $N_* = p : n[P_*]_{q,\emptyset}$ and $[\![N_*]\!]_p^{loc} = [\![P_*]\!]^{proc} = P_*$ and $p : n[[\![N_*]\!]_p^{loc}]_{q,\emptyset} = N_*$. Otherwise, $N_* = p : n[P_*]_{q,S} \mid N_*^1 \mid \ldots \mid N_*^n$ with $\mathbf{tree}(N_*^i, l_i, n_i, l)$.

$$\begin{aligned} p : n[[\![N_*]\!]_p^{loc}]_{q,\emptyset} &= p : n[[\![P_*]\!]^{proc} \mid \Pi_{i\in 1..n} n_i[[\![N_*^i]\!]_{l_i}^{loc}]]_{q,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* p : n[[\![P_*]\!]^{proc}]_{q,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} l'_i : n_i[[\![N_*^i]\!]_{l_i}^{loc}]]_{p,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* p : n[P_*]_{q,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} N_*'^i \end{aligned}$$

Moreover, $N_*'^i \cong N_*^i$. We deduce that $p : n[P_*]_{q,\{l'_1,\ldots,l'_n\}} \mid \Pi_{i\in 1..n} N_*'^i \cong N_*$.

Lemma 22. *If* $\mathbf{tree}(M, p, n, q)$ *and there is no reified process in* M*, then* $[\![[\![M]\!]_p^{loc}]\!]_{p,n,q} \doteq M$*. Moreover, if* M *is well-formed* $[\![[\![M]\!]^{mac}]\!] \doteq M$*.*

Proof. We reason by induction on the size of the tree. We can write $M \equiv l : n[P]_{p,S} \mid M^1 \mid \ldots \mid M^n$ with $\mathbf{tree}(M^i, l_i, n_i, l)$.

$$\begin{aligned} [\![[\![M]\!]_p^{loc}]\!]_{p,n,q} &= [\![[\![P]\!]^{proc} \mid \Pi_{i\in 1..n} n_i[[\![M^i]\!]_{l_i}^{loc}]\!]_{p,n,q} \\ &= p : n[P \mid \Pi_{i\in 1..n} n_i[[\![M^i]\!]_{l_i}^{loc}]]_{q,\emptyset} \\ &\stackrel{\equiv}{\rightarrow}^* p : n[P]_{q,\{l_1,\ldots,l_n\}} \mid \Pi_{i\in 1..n} l_i : n_i[[\![M^i]\!]_{l_i}^{loc}]_{p,\emptyset} \\ &= p : n[P]_{q,S} \mid \Pi_{i\in 1..n} [\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,n_i,p} \end{aligned}$$

We can apply the induction hypothesis to $[\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,n_i,p}$. We have

$$[\![[\![M^i]\!]_{l_i}^{loc}]\!]_{l_i,n_i,p} \stackrel{\equiv}{\rightarrow}^* M'^i$$

with $M'^i \cong M_*^i$ and

$$[\![[\![M]\!]_p^{loc}]\!]_{p,n,q} \stackrel{\equiv}{\rightarrow}^* p : n[P]_{q,S} \mid \Pi_{i\in 1..n} M'^i = M'$$

Besides, we notice that $M \stackrel{\equiv}{\rightarrow}^* \cong M'$. So we deduce $[\![[\![M]\!]_p^{loc}]\!]_{p,n,q} \doteq M$. If M is well-formed and $\widetilde{c} = \mathbf{mnames}(M)$ we have

$$\begin{aligned} [\![[\![M]\!]^{mac}]\!] &= [\![\nu\widetilde{c}.[\![M]\!]_{\mathbf{r}}^{loc}]\!] \\ &\stackrel{\equiv}{\rightarrow}^* [\![[\![M\{\widetilde{m}/\widetilde{c}\}]\!]_{\mathbf{r}}^{loc}]\!] \qquad \text{rule M.S.NEW with } \widetilde{m} \text{ fresh} \end{aligned}$$

By corollary 1 we have

$$[\![[\![M]\!]^{mac}]\!] \doteq [\![[\![M\{\widetilde{m}/\widetilde{c}\}]\!]_{\mathbf{r}}^{loc}]\!]$$

Besides, by definition we have $M\{\widetilde{m}/\widetilde{c}\} \cong M$ and we have just shown before that

$$[\![[\![M\{\widetilde{m}/\widetilde{c}\}]\!]_{\mathbf{r}}^{loc}]\!] \doteq M\{\widetilde{m}/\widetilde{c}\}$$

So finally, we deduce that $[\![[\![M]\!]^{mac}]\!] \doteq M$.

Proof (proof of lemma 6). The second part is immediate. For the first part, we define a machine N as M where we replace every occurrence of reified processes by a fresh variable x_i, such that $M = N\{\mathbf{reify}(l_i, M^i_*)/x_i\}$. By definition of $[\![.]\!]^{mac}$, it is easy to see that $[\![M]\!]^{mac} = [\![N\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\}]\!]^{mac}$. Besides, from lemma 22 we have $[\![[\![N]\!]^{mac}]\!] \stackrel{\equiv}{\rightarrow}^* U$ and $N \stackrel{\equiv}{\rightarrow}^* U'$ with $U \cong U'$. In particular,

$$[\![[\![N\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\}]\!]^{mac}]\!] \stackrel{\equiv}{\rightarrow}^* U\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\}$$

$$N\{\mathbf{reify}(l_i, M^i_*)/x_i\} \stackrel{\equiv}{\rightarrow}^* U'\{\mathbf{reify}(l_i, M^i_*)/x_i\}$$

and then

$$[\![[\![N\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\}]\!]^{mac}]\!] \doteq U\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\}$$

$$N\{\mathbf{reify}(l_i, M^i_*)/x_i\} \doteq U'\{\mathbf{reify}(l_i, M^i_*)/x_i\}$$

Moreover, we have $U'\{\mathbf{reify}(l_i, M^i_*)/x_i\} \cong U\{\mathbf{reify}(l_i, M^i_*)/x_i\}$ and by lemma 21

$$U\{[\![M^i_*]\!]^{loc}_{l_i}/x_i\} \sim_c U\{\mathbf{reify}(l_i, M^i_*)/x_i\}$$

Proof (Proof of proposition 2). Suppose that $[\![P]\!] \rightarrow M$. Then we have by lemma 20 that $[\![[\![P]\!]]\!]^{mac} \rightarrow [\![M]\!]^{mac} = P'$. If we apply the second part of lemma 6, we have $[\![[\![P]\!]]\!]^{mac} \equiv P$. Moreover, $[\![P']\!] = [\![[\![M]\!]^{mac}]\!] \sim_c M$ by the first part of lemma 6. We conclude that $P \rightarrow P'$ with $[\![P']\!] \sim_c M$.

A Survey of Some Implementation Techniques for Security Membranes ⋆

Marc Lacoste

Distributed Systems Architecture Department, France Télécom R& D.
marc.lacoste@rd.francetelecom.com

Abstract. The notion of security membrane appears as an emerging concept in the design of secure languages for global computing. Membranes separate the computational behavior of a site from the security code controlling access to site-located resources. We provide a survey of some of the challenges which arise when trying to implement security membranes in an execution platform such as an operating system. We identify four main design issues: the choice of a security model; the type of architecture for the execution environment; the layer at which to place security mechanisms; and the assurance level of the platform. In each case, we discuss possible trade-offs between security, flexibility, simplicity, and trustworthiness. We then show how applying a component-based approach to design and implement the execution environment can help to reach an acceptable compromise between such properties.

1 Introduction

The notion of *security membrane* is increasingly attracting attention for the design of secure languages for global computing environments, as witnessed by a number of recent papers [11, 23, 38, 45]. The aim of a membrane is to clearly separate the computational entities of a location, generally modeled as a number of processes running in parallel, from a generic controller, the *membrane* (also called *guardian*) which supervises the behavior those entities at run-time, and acts as a unique gateway with the outside of the location [11, 45]. Thus, a location named ℓ can be modeled as $\ell\ (\ M\ )\{\ P\ \}$, where M is a specification of the membrane behavior, and P is a parallel composition of processes. A membrane should be seen as an enhanced container that can provide fine-grained control over the semantics of communication or mobility for localized processes.

A membrane can also be used for security-specific purposes to protect the location, by filtering incoming and outgoing messages. This allows to realize a security domain, by selectively controlling access to a group of processes which share a common security policy. This policy is often expressed as a type system [13, 16, 24, 28], against which processes will be type-checked, either statically or dynamically, to ensure safe behavior. Location ℓ can then be written $\ell\ (\ \pi\ ,\ RM\ )\{\ P\ \}$, where π is a set of types defining the location security policy, and *RM* is a *reference monitor* [2], abstract machine in charge

⋆ This work has been partially supported by EU FET Global Computing initiative, project MIKADO IST-2001-32222.

of enforcing the authorization policy π. In general, a clear separation is made between the programmer who implements the process behaviors, and the network administrator who defines the security policy, for a site or a group of sites.

So far, research on the protection of location resources in a global computing environment has mainly focused on language-based security, for instance by defining numerous type systems for distributed process calculi with locations. But is this enough? What is needed in practice to implement a security membrane? In existing approaches, little is said about effective implementation requirements. The emphasis is put on proving theoretical properties, like type safety results, leaving enforcement in an execution platform to traditional code verification techniques, like model checking or proof-carrying code [41]. Yet, some of these methods still lack maturity. For instance, most automated verification tools for mobile code, like model-checkers and theorem provers, reach their limits for large-size systems, and a human intervention is often needed to carry out the proof of safety. Still, providing strong end-to-end security guarantees in software infrastructures is critical to protect assets located within network locations against unauthorized disclosure and improper modification, while ensuring service availability to legitimate users. Such security requirements call for strong protection mechanisms down to the lowest software layers like the operating system, which serves as a foundation to guarantee security at higher levels.

We address this problem by studying in this paper what are the practical implementation challenges for the realization of a security membrane, encompassing not only the language aspects, but also the platform-related issues, in particular in the operating system. The objective is to acquire an overall perspective of some of the critical design parameters for the effective implementation of such a membrane.

A first sensible design principle for controlling access to location resources is to clearly separate the security policy from the enforcement mechanisms. The former captures the protection requirements to be satisfied, for instance gathered after a vulnerability analysis. The latter should be understood as a Trusted Computing Base (TCB), defined in the Orange Book [50] as "the totality of protection mechanisms within a computer system – including hardware, firmware and software – the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system." Hence the need for a bird's eye view of security membrane design, which should not be limited just to language issues.

The above separation enables access control mechanisms to be reused to enforce different security policies. As such, it has a direct impact on the overall flexibility of the security membrane. This requirement is also motivated by the need to federate several authorization policies in global computing networks, seen as superpositions of multiple heterogeneous subsystems, each endowed with its own security policy.

A membrane should (at least) guarantee the following properties:

Security: no illegitimate access to resources should be possible; access control mechanisms should guarantee complete mediation, i.e., they should be non-circumventable; they should also be tamper-proof, to preserve integrity; finally, the principle of least privilege should be enforced to avoid abusive propagation of delegated access rights.

Flexibility: the membrane should support multiple authorization policies, and allow fine-grained access control; it should also enable dynamic management of access rights to deal with unpredictable changes in the environment of the location; finally, it should provide mechanisms to manage the delegation of privileges.

Performance: the impact of security checks on the performance of running processes should be unobtrusive, or, at least, acceptable.

Simplicity: a principle of economy of mechanism should guide the membrane design, so that it may be simple to implement, to use, and to administer.

Trustworthiness: the membrane implementation should be small enough to be subject to analysis, in order to provide evidence to independent evaluators like trusted-third parties that the protection mechanisms fulfill in a correct way the identified security requirements for the location.

It seems difficult to find the right balance between such often conflicting properties. We identify the following important criteria in the design of security membrane, which may help to reach a compromise:

- *The choice of the security model:* what security property (i.e., confidentiality, integrity, least privilege, etc.) is the membrane trying to enforce?
- *The type of operating system kernel:* operating systems architecture has greatly evolved from monolithic and micro-kernels to more extensible systems. Which architecture allows to reach the best compromise between the properties mentioned above?
- *The choice and location of security mechanisms:* which protection mechanisms should be used? Is language-based security sufficient? Should the membrane also rely on operating system protection, or even on hardware security? What are the right levels of abstraction and the appropriate software layers to place security mechanisms in the system?
- *The level of assurance of the platform:* what guarantees can the platform provide of correct policy enforcement? What metric should be used to assess the strength of the security membrane? How trustworthy really is the access controller?

The paper provides an overview of the involved trade-offs for each key design issue identified previously. Based on that quick survey, we then consider a possible solution for the architecture of security membranes, by applying a component-based approach to its design. We argue that this approach enables to build secure yet adaptable security membranes, where flexible access control mechanisms can be easily designed and implemented, allowing support for multiple authorization policies.

The rest of the paper is structured as follows: we first give a survey of possible trade-offs between security, flexibility, performance, simplicity, and trustworthiness for a security membrane, regarding: the choice of a security model (section 2), the type of architecture for the execution environment (section 3), the layers at which to place security mechanisms (section 4), and the platform assurance level (section 5). We then discuss in section 6 the benefits of a component-oriented design for implementing flexible security membranes.

2 Choosing a Security Model

A security model is an abstract description of a security policy, which forms the theoretical setting for the expression of the policy, and provides a basis for reasoning to prove security properties.

2.1 A Brief Overview of Existing Security Models

Overcoming the Safety Problem. The *access control matrix* [34] is a foundational model describing the protection state of an authorization system. Unfortunately, proving system safety with this model was shown to be undecidable [27]. To overcome this difficulty, the focus of recent research on access control models has shifted towards the following directions:

- The study of less expressive models, which can be implemented more easily, and on which reasoning becomes possible. This restriction is at the expense of flexibility, since in those limited models, security policies are generally static, i.e., the assignment of access rights from subjects to objects cannot vary in time, or has to be done by trusted principals [3, 4].
- The extension of the access matrix with *constraints* to facilitate the verification of security properties. Those models also capture more dynamic aspects of a security policy, like the changing of privileges over time [12, 22, 47]. Unfortunately, constraints-based models are hard to implement due to the complexity of the logical languages used to express the constraints.

Thus, a compromise has to be reached between: the expressiveness of the security model, which is a first metric of its genericity and flexibility; the simplicity of implementation; and the easiness of enforcing and verifying security properties within the model.

Which Security Policy? Access control policies may be classified according to the security properties one wishes to prove on a system. Discretionary access control schemes, like those found in traditional UNIX security, reach their limits for systems with strong security requirements. A mandatory approach to security is then needed, so that authorization mechanisms may not be circumvented. The covered spectrum is quite broad, since the explored concepts range from simple properties like confidentiality [4] and integrity [9, 10], to more advanced principles like least privilege and separation of duties [12, 22, 47]. For instance, see [7] for a detailed comparison of some access control models and their expressive power.

Confidentiality Policies. A number of lattice-based models, like the one from Bell and Lapadula [4], focus on preventing unauthorized disclosure of information, notably for multi-level military-grade security systems. These models enable extended analysis of information flows.

Integrity Policies. Other models try to preserve the integrity of systems. For instance, the Biba integrity model [9] is the exact dual of the Bell-Lapadula one, where security levels classify the integrity of information instead of its confidentiality. Clark and Wilson [14] provide a model more adapted to commercial environments by introducing integrity constraints. Boebert and Yain [10] study non-hierarchical mandatory integrity policies, and introduce the notion of *type-enforcement*, used for instance in the DTE (Domain and Type Enforcement) model for UNIX security [3]: *types* are assigned to resources, and processes are grouped into *domains*, which grant a number of access rights over specific types.

Least Privilege Policies. This family of policies tries to enforce the concept of *separation of duties* (SoD): each task in a sequence of operations has to be performed by a different subject [47]. A variant is the Chinese Wall [12] policy, where a history of accesses to resources is kept to resolve conflicts of interests in commercial systems. A formal study of different types of least privilege policies can be found in [22].

Role-Based Policies. By introducing the abstraction of *role*, the RBAC (Role-Based Access Control) model [44] makes administration of access rights easier. Users are assigned *roles* which confer permissions over a set of resources. This model can represent several classes of security policies [42], and is a first step towards policy-neutral access control (cf. section 2.2). The price to pay is a greater complexity: for instance, the NIST RBAC standard [19] distinguishes no less than four variants of the model, such as *flat* RBAC, which defines assignments of roles to users and of permissions to roles, *hierarchical* RBAC where roles are structured into a hierarchy, and *constrained* RBAC which enables expression of additional constraints to enforce separation of duties.

2.2 Towards Policy-Neutral Access Control

Principles. This multiplicity of security models contributed to an absence of agreement as to what should be the security model to implement in a distributed platform or operating system kernel, since each system supports his own classes of security policies based on a platform-specific security model. This motivated a new approach to authorization called *policy-neutral access control*, where authorization mechanisms should be agnostic vis-à-vis a security model. Since no single policy can claim to capture all the system protection requirements for different execution environments [37], the main advantage of policy-neutral authorization is to support multiple security models with a single mechanism. Thus, a wide range of models and policies can be enforced without need to change the security infrastructure, which is also more modular. In the case of wide-area networking, this approach allows to federate multiple security policies with a unified authorization mechanism. Indeed, federation of security policies is a key issue when implementing a secure infrastructure for global computing, where multiple heterogeneous networks, each governed by its own security policy, must be connected transparently.

Policy Description and Composition Languages. Several languages have been designed, both to describe authorization policies in a policy-neutral manner, or to reconcile different security policies. For instance: the ASL [31] authorization language supports SoD and Chinese Wall security models; the ORION++ [8] system enforces different policies in database environments. In each case, enforcement mechanisms are clearly separated from the security policy, a single security server supporting multiple authorization policies. Policy reconciliation languages such as [51] define policy algebras, abstract descriptions independent of enforcement mechanisms, which allow to decentralize policy specifications, and possibly to support unknown policies, expressed as incomplete specifications. Another approach is that of [46], where policy composition amounts to the composition of automata.

Some Implementations Some policy-neutral access control mechanisms have already been implemented for different types of kernels. For instance: the DTE-based confinement mechanisms for UNIX processes [3]; the support of multiple security models implemented in Linux as kernel modules [52]; or, the SELINUX architecture [48]. In the MACH micro-kernel [39], access request interception is realized by a policy-neutral security server inside the micro-kernel, while policy-decision is implemented outside the micro-kernel as an security server, which is dependent on a specific security model, and which can be replaced. Other implementations can be found in extensible [25] or component-based [32] kernels.

3 Selecting the Type of Kernel

The operating system provides applications with an abstract view of resources contained in a location: CPU, memory, file system, network bandwidth, etc. It offers primitives to manage resource sharing and protection. The system is generally composed of two types of components, which may, or may not reside within the same protection domain:

- The *kernel* has access to all hardware resources: there, the CPU can execute security-sensitive instructions in privileged mode.
- Higher-level *system services* are directly used by applications.

3.1 First Trends

General laws stating the impact of an OS architecture on security or performance are hazardous, since the drawn conclusions may be contradicted by the design of some specific systems. Nevertheless, in what follows, we try to sketch some general directions for trade-offs.

The security of the architecture strongly depends on whether services are located within the kernel protection domain, or are implemented as separate processes. Creating several distinct domains is in favor of stronger security: each application remains confined within its own address space, and cannot corrupt other parts of the system. However, this separation is made at the expense of system performance: each access request to hardware resources coming from an application-level domain requires a system call, costly due to multiple context switching with the kernel protection domain.

3.2 Different Types of Kernels

Monolithic Kernels. Traditional *"monolithic" kernels* like Linux or PALMOS integrate system services within the kernel protection domain. This solution results in a large-size and unstructured OS providing little flexibility. The isolation of a specific component within the kernel remains difficult. Upgrades often require recompiling the complete kernel. Due to its complexity, the OS contains many security loopholes: attacks on a kernel component are almost impossible to confine and will, more often than not, take control of the overall system. Despite such limitations, these systems generally present good performance results, since execution remains confined within the kernel address space, without need for context switching.

Micro-Kernels. In *micro-kernels* like QNX [29], services are run as separated processes which communicate via IPCs (Inter-Process Communication). Both the protection domains, which behave as confinement units, and the reduced kernel size make those systems much safer. They are also more flexible: the modular kernel structure allows to dynamically insert new services within the OS. Reduced performance may be the inherent limitation of this type of architecture, due to the cost of IPC mechanisms. Note that the overhead of an access control mechanism is lower when the authorization server is located outside the kernel, since the number of cross-domain invocations is then reduced. The trade-off of performance is in that case a weaker security.

Extensible Kernels. *Extensible kernels* [17, 20] are currently preferred for embedded systems, notably those with a single address space like SPIN [6]. Eliminating from the architecture unnecessary abstractions – like heavyweight processes, which are replaced by threads – improves performance, since cross-domain context switching is less costly. These minimal kernels only contain the absolutely essential services to multiplex hardware resources. Hence, the TCB is smaller and easier to certify, resulting in higher-assurance kernels. However, these systems are quite vulnerable to attacks without any additional protection mechanism [1], since they can be easily reconfigured by dynamically downloading modules within the kernel which may contain malicious code.

Component-Based Kernels. Finally, *component-based kernels* like THINK [18] are extensible kernels which allow a greater flexibility thanks to a more uniform architectural model : the entire kernel is built from an assembly of elementary units of reconfiguration called *components*, which are composable and reusable to design minimal dedicated systems. Compared to traditional systems, performance results show no significant degradation due to componentization.

[1] Extensible kernels are often written using strongly-typed programming languages: well-defined language semantics allows to produce safety proofs, which can be checked before downloading code into the kernel.

A Comparison. The above discussion is summarized in figure 1. Note that as the structure of the kernel is more clearly defined, its architecture offers more implementation hooks for flexibility.

Property	Monolithic Kernel	Micro-Kernel	Extensible Kernel	Component-Based Kernel
Security	- -	+ + / -	- / + +	+ +
Performance	+ +	- - / +	+	+
Flexibility	- -	+	+ +	+ + +
Simplicity	- -	+	+ +	+ + +

Fig. 1. Comparison of Different Types of Kernels.

4 Placing Security Mechanisms at the Right Abstraction Level

4.1 Hardware Solutions

Protection mechanisms may be implemented at the *hardware level*. Such a solution is truly minimal, enforces complete mediation, and provides fail-safe and tamper-resistant protection against internal, unintentional, or malicious threats to system integrity. Historical protection domains were implemented as hardware mechanisms in MULTICS [4]. Other examples of trusted hardware protection include: the MMU (Memory Management Unit) which confines applications by defining separate address spaces; co-processors with a secure mode such as [36]; or bus-level exclusion mechanisms [26].

4.2 Language-Based Solutions

Protection may also be *language-based*, using type-safe programming languages such as JAVA. These languages do not allow direct memory addressing, and enforce complete mediation using the concept of *interface*, unique access point to system resources. Language-based security solutions are quite flexible, and allow to realize easily fine-grained access control. Yet, their security is somewhat weaker than OS-level protection mechanisms, the attacker often taking complete control of a vulnerable run-time system. These solutions also require additional software layers (compilers, virtual machine) to be trusted. Thus, the TCB size is increased, making the system possibly more difficult to certify. For instance, see [30] for a comparison between language and OS-based security mechanisms.

4.3 Some Implementation Techniques

Between those two extremes co-exist a wide range of authorization mechanisms which can be implemented from kernel to application levels. A detailed comparison is beyond the scope of this report. Traditional solutions for discretionary access control like capability-based systems [35] are well-known. They only provide limited protection to confine the execution of programs with root privileges. Different forms of interposition allow enforcement of mandatory security policies: system call interception [5, 43], the system call level providing a well-defined interface to enforce complete mediation of security-sensitive operations; authorization hooks in kernel space [52]; and software wrappers to sandbox applications [1] or kernel modules [21]. Other solutions, more related to dynamic optimization, like program shepherding [33] or return address protection [15] can also be used to thwart buffer overflow or code injection attacks within the kernel.

The invasiveness of an access control mechanism will generally decrease as it is placed closer to the kernel, since context switching between user and kernel spaces will be less frequent. Reducing the distance – in terms of number of software layers to cross – between the authorization mechanism and the system resources to protect will also improve security, making mechanisms less likely to be bypassed, and introducing fewer security flaws. In that case, however, security enhancements will more difficult to reconfigure, requiring extensive changes or kernel recompilation.

5 Evaluating the Assurance Level

After selecting a security model and a type of kernel, and designing a security architecture where security mechanisms are placed in the most vulnerable software layers, one key issue remains: to convince other parties of the trustworthiness of the security membrane implementation. For low assurance levels, testing methodologies may be sufficient, using both unit testing and system testing to ensure that individual components are properly integrated. Kernels aiming for higher assurance levels would need to be verified using formal evaluation methodologies like the Common Criteria (CC) [40]. Code verification techniques needed to reach a given level of assurance may also be used to verify code safety for modules downloaded within the security membrane [41]. Existing verification algorithms and tools are currently limited to evaluating small systems, partly due to the computational power needed to carry out large-scale verification. Thus, proving correctness of complex security membranes is not yet within reach.

6 Towards Component-Based Security Membranes

In the previous sections, we explored some of the main trade-offs between security, flexibility, performance, simplicity, and trustworthiness, for different design parameters when implementing a security membrane. We now concentrate on two key properties for membranes in a global computing environment, often conflicting, and which need to be reconciled: reconfigurability and security. We then explore the benefits of a component-oriented design for implementing flexible yet secure membranes.

6.1 Reconciling Security and Reconfigurability

The advent of global computing has required infrastructures to become more adaptable and reconfigurable, to cope with the increasing number of wireless networks and the heterogeneity of mobile terminals. At stake is the extreme dynamicity of those environments, where both users and executable code are mobile. Execution contexts are characterized by the addition of new features on the fly, frequent downloads of platform updates, and personalization of existing services. However, reconfigurability puts security at risk, for instance when untrusted components are downloaded into a security infrastructure.

These infrastructures are generally extremely vulnerable and unreliable. Attacks, of increasing strengths and numbers, take advantage of the multiplicity of available paths for information flow, in overly complex systems. They also thrive on the size of extremely decentralized infrastructures, where applying a uniform security policy is nearly impossible.

Unfortunately, so far, reconfigurability has been considered orthogonally to protection. Among many explanations:

- Designs for security architectures ensure safe component reconfiguration within the limits of a specific security model. Reconfigurability has been applied with limited success to the security architecture itself that often remains monolithic.
- Security has simply been "forgotten" in a number of adaptable middleware for safety-critical and real-time systems.
- Traditional protection techniques are difficult to adapt to the context of wide-area systems. The architecture may be too monolithic, or demand too many computational and networking resources.

A compromise between security and flexibility has to be reached to win the trust of end users. Can secure yet reconfigurable membranes be designed and implemented using the same abstractions for reconfiguration and security architectures?

6.2 Components: a Solution for Reconfigurable and Secure Membranes?

We propose to reach such a goal by adopting a component-based approach to membrane design. This choice is motivated by a need for an integrated vision of security in global computing environments: indeed, grafting isolated and perimetric protection mechanisms on existing infrastructures is highly inadequate, since many loopholes and security breaches in the system are overlooked. A major advantage of component-based technology is to obtain with a single design a system that is simultaneously reconfigurable and secure.

Components are usually described as entities encapsulating code and data, endowed with an identity, which appear in software systems as units of execution, configuration, administration or mobility: “a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition to third parties” [49]. Building a system according to the component paradigm amounts to composing reusable units in a recursive way, since the assembly of a group of components allows creating new components at a higher level of abstraction.

Component technology enables to master the complexity of software infrastructures in terms of specification and implementation. The architect can choose the right level of abstraction to describe and observe the system, and, at this level, is given a homogeneous view of the infrastructure. The key argument in favor of Component-Based Software Engineering (CBSE) is to reduce the costs of software development, distribution, and maintenance, by reuse or mutualization of existing components. CBSE is well adapted to the dynamic needs of global computing environments, since it allows adapting and dynamically extending infrastructures, by addition, replacement, upgrade, or reconfiguration of components.

A component is not only a unit of reconfiguration. It is also a unit of protection. Although security properties are not in general compositional, CBSE enables to preserve security properties by including additional security mechanisms, for instance, by the use of type-safe languages to avoid any forms of interference between components during the operation of composition.

6.3 Benefits of Components for Security Membranes

Components appear as an promising solution to some the main design issues we discussed in this paper:

- Component-based design is security policy-neutral. Decoupling security policy decision components from the components containing the protection enforcement mechanisms enables to adopt a policy-neutral approach to security membrane design: hence, the same infrastructure mechanisms can be reused to enforce different authorization policies.
- CBSE also allows building a great variety of types of platforms, for instance in the domain of operating systems, ranging from monolithic and extensible systems to micro- and exo-kernels. We already saw in section 3 that in general the performance impact of componentization remains reasonable.
- Furthermore, by offering a fine-grained view of system functionalities, CBSE leaves a lot of freedom to the localization of protection mechanisms, which can be placed closer to the hardware, if stronger security is needed; or closer to applications, if simplicity and compatibility with existing software, e.g., no kernel recompilation, are the leading design requirements.
- Finally, thanks to its ability to realize truly minimal kernels, CBSE allows to build platforms with potentially high assurance levels.

These arguments are in favor of a component-based solution to achieve a good compromise between the various criteria we identified for the design of a security membrane. Given the breadth of the problem, much research remains still to be done in order to fully validate this claim.

7 Conclusion

The notion of security membrane appears as an emerging concept in the design of secure languages for global computing environments. Membranes separate the computational behavior of a site from the security code controlling access to site-located resources. However, an integrated view of the involved design issues is not yet achieved.

This brings forth new challenges for a practical implementation, which cannot be addressed by mere restriction to language-based aspects. In this paper, we studied some of these challenges, putting the focus on platform-related issues, in particular at the operating system level. We identified four critical design parameters for an effective implementation, namely the choice of a security model, the type of architecture for the execution environment, the layers at which to place security mechanisms, and the assurance level of the platform. In each case, we discussed possible trade-offs between key properties for a security membrane such as security, flexibility, performance, simplicity, and trustworthiness.

The proposed solution to reach a compromise between these properties is to design and implement a security membrane using the component-based paradigm, which maintains a delicate balance between reconfigurability and security. This choice enables the creation of different types of infrastructures without real performance penalties. The concern for modularity allows exploring several security models, and placing the access control mechanisms at different abstraction levels. Furthermore, component-based design may create truly minimal infrastructures, which could help to reach high assurance levels, to win the trust of end users. This approach allows adapting straightforwardly the system to the changes which may occur during its life-cycle, without really endangering the security of the infrastructure, since a component is both a unit of reconfiguration and of security. To further explore these ideas, more studies of integration of component-based concepts in practical implementations of security membranes will need to be considered. Thus, we hope this promising approach will emerge in the near future as a solution for the design of secure yet reconfigurable membranes.

References

1. A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings USENIX Security Symposium*, 2000.
2. S. Ames, M. Gasser, and R. Schell. Security Kernel Design and Implementation: An Introduction. *IEEE Computer*, 16(7):14–22, 1983.
3. L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghinghat. Practical Domain and Type Enforcement for UNIX. In *Proceedings IEEE Symposium on Security and Privacy (S&P'95)*, 1995.
4. D. Bell and L. Lapadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, Electronics Systems Division, Bedford USAF Base, DoD, 1976.
5. M. Bernaschi, E. Gabrielli, and L. Mancini. REMUS: A Security-Enhanced Operating System. *Transactions of Information and System Security (TISSEC)*, 5(1):36–61, 2002.
6. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczyncki, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.
7. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A Logical Framework for Reasoning about Access Control Models. *ACM Transactions of Information and System Security*, 6(1):71–127, 2003.
8. E. Bertino, S. Jajodia, and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. In *Proceedings IEEE Symposium on Security and Privacy (S&P'96)*, 1996.

9. K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, Bedford USAF Base, DoD, 1997.
10. W. Boebert and R. Yain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings National Computer Security Conference*, 1985.
11. G. Boudol. A Generic Membrane Model. In *Proceedings Workshop on Global Computing (GC'04)*, 2004.
12. D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings IEEE Symposium on Security and Privacy (S&P'89)*, 1989.
13. L. Cardelli, G. Ghelli, and A. Gordon. Types for the Ambient Calculus. *Information and Computation*, 177:160–194, 2002.
14. D. Clark and D. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings IEEE Symposium on Security and Privacy (S&P'87)*, 1987.
15. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks. In *Proceedings USENIX Security Symposium*, 1998.
16. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
17. D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.
18. J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : a Software Framework for Component-Based Operating System Kernels. In *Proceedings USENIX Annual Technical Conference (USENIX'02)*, 2002.
19. D. Ferraiolo, R. Sandhu, S. Gavrila, and D. Kuhn. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions of Information and System Security*, 4(3):224–274, 2001.
20. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.
21. T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings IEEE Symposium on Security and Privacy (S&P'99)*, 1999.
22. V. Gligor, S. Gavrila, and D. Ferraiolo. On the Formal Definition of Separation of Duty Policies and their Composition. In *Proceedings IEEE Symposium on Security and Privacy (S&P'98)*, 1998.
23. D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proceedings Conference on Security in Pervasive Computing (SPC'03)*, 2003.
24. D. Gorla and R. Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *Proceedings International Colloquium on Automata, Languages and Programming (ICALP'03)*, 2003.
25. R. Grimm and B. Bershad. Providing Policy-Neutral and Transparent Access Control in Extensible Systems. Technical Report UW-CSE-98-02-02, University of Washington, 2002.
26. T. Halfhill. ARM Dons Armor : TrustZone Security Extensions Strengthen ARMv6 Architecture. Microprocessor Report, 2003. Document availble at the URL: `http://www.arm.com/miscPDFs/4136.pdf`.
27. M. Harrison, W. Ruzzo, and J. Ullman. Protection in Operating Systems. *Communication of the ACM*, 19(8):461–471, 1976.
28. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
29. D. Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Micro-Kernels and other Kernel Architectures*, 1992.
30. T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Proceedings USENIX Security Symposium*, 1998.

31. S. Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings IEEE Symposium on Security and Privacy (S&P'97)*, 1997.
32. T. Jarboui, J.-P. Fassino, and M. Lacoste. Applying Components to Access Control Design : Towards a Framework for OS Kernels. In *Proceedings International Conference on Dependable Systems and Networks (DSN'04)*, 2004.
33. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings USENIX Security Symposium*, 2002.
34. B. Lampson. A Note of the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
35. H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
36. D. Lie, C. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings ACM Symposium on Operating Systems Principles (SOSP'04)*, 2004.
37. P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, and J. Farell. The Inevitability of Failure: the Flawed Assumption of Security in Modern Computing Environments. In *Proceedings National Information Systems Security Conference*, 1998.
38. F. Martins and V. Vasconcelos. Controlling Security Policies in a Distributed Environment. Technical Report DI–FCUL–TR 04–01, Department of Computer Science, University of Lisbon, 2004.
39. S. Minear. Providing Policy Control over Object Operations in a Mach-Based System. In *Proceedings USENIX Security Symposium*, 1995.
40. National Institute Of Standards and Technology. Common Criteria for Information Technology Security Evaluation (CC). Technical Report CCIMB-99-031, 1999.
41. G. Necula. Proof-Carrying Code. In *Proceedings ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997.
42. S. Osborn, R. Sandhu, and Q. Munawer. Configuring Role Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and Systems Security*, 3(2), 2000.
43. N. Provos. Improving Host Security with System Call Policies. In *Proceedings USENIX Security Symposium*, 2003.
44. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
45. A. Schmitt and J.-B. Stefani. The M-calculus: a Higher-Order Distributed Process Calculus. In *Proceedings ACM Symposium on Principles of Programming Languages (POPL'03)*, 2003.
46. F. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
47. R. Simon and M. Zurko. Separation of Duty in Role-Based Environments. In *Proceedings IEEE Computer Security Foundations Workshop (CSFW'97)*, 1997.
48. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Ansersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings USENIX Security Symposium*, 1999.
49. C. Szyperski. *Component Software Systems*. Addison-Wesley, 2002.
50. U.S. Department of Defense. Trusted Computer Security Criteria (TCSEC). Technical report, 1985.
51. D. Wijesekera and S. Jajodia. Policy Algebras for Access Control - The Predicate Case. In *Proceedings ACM Conference on Computer and Communications Security (CCS'02)*, 2002.
52. C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings USENIX Security Symposium*, 2002.