Component-Oriented Programming with Sharing: Containment is not Ownership

Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, Jean-Bernard Stefani

Component-Oriented Programming with Sharing: Containment is not Ownership

Daniel Hirschkoff[1], Tom Hirschowitz[1], Damien Pous[1], Alan Schmitt[2], Jean-Bernard Stefani[2]

(1) LIP ENS Lyon, 46, Allée d'Italie 69364 Lyon Cedex 07 - France
(2) INRIA Rhône-Alpes, 655 Avenue de l'Europe, 38334 St Ismier, France

Abstract. Component-oriented programming yields a tension between higher-order features (deployment, reconfiguration, passivation), encapsulation, and component sharing. We propose a discipline for component-oriented programming to address this issue, and we define a process calculus whose operational semantics embodies this programming discipline. We present several examples that illustrate how the calculus supports component sharing, while allowing strong encapsulation and higher-order primitives.

1 Introduction

Wide-area distributed systems and their applications are increasingly built as heterogeneous, dynamic assemblages of software components. This modular structure persists during execution: such systems provide the means to control their run-time modular configuration, which encompasses automatic deployment, unanticipated evolution, passivation, run-time reconfiguration, and introspection. This expressive power conflicts with the strong encapsulation properties generally expected from modular programs.

A key tension point is *component sharing*, which allows two remote components to encapsulate a common component, as depicted in Figure 1, where the component L (e.g. a software library) is shared among C and D. How does one preserve encapsulation in this case? In particular, what happens to L and D if A removes C from the configuration? How can C replace L by L', without necessarily impacting D? Essentially, the difficulty lies in combining (1) encapsulation with fine-grain, objective control over communications, (2) locality passivation, migration, and replication, and (3) access to shared components with simple communication rules.

Previous models of component-oriented programming do not completely address these three requirements. Models that do not address requirement (3) comprise process calculi with hierarchical localities that feature local communications only (i.e., no direct communication between arbitrarily distant localities in the locality forest) [5, 4, 14, 3, 8, 18]. Indeed, sharing is representable in such models, but at the expense of complex routing rules which are difficult to maintain. Models that do not have this routing complexity problem, but are weak on requirement (1), include the Cell calculus [15] and process calculi with localities that do not restrict communications betwen localities [10, 12, 20, 19, 11, 1, 16]. The tKlaim calculus [9] is a recent variant of Klaim that allows the establishment of different communication topologies between localities.

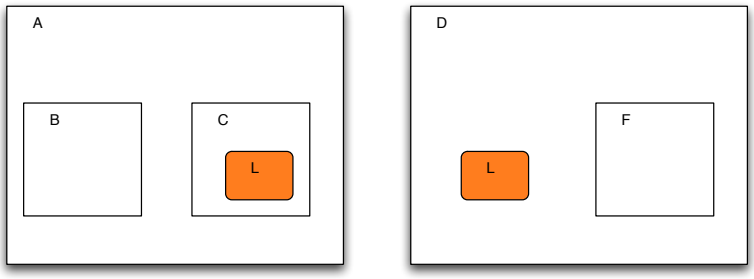

Fig. 1. A configuration with sharing

However, such calculus still falls short of full encapsulation of sub localities, since there is no objective control over process migration and execution.

Our starting point to solve the issue of component sharing is that, from the standpoint of the latter kind of models (weak on (1)), the problem is reminiscent of the *aliasing problem* in object-oriented languages [13]: sharing is easy, but encapsulation is problematic. To solve this problem, Clarke et al. introduce *ownership types* [6, 7], which attribute to each object o an *owner* that controls the references to o. We adapt this idea of ownership to the setting of process calculi. However, instead of designing a type system to preserve encapsulation, we enforce it at the level of the operational semantics, as follows. We split the usual hierarchical forest of localities into two graphs: the *ownership forest* and the *containment graph*. Locality passivation must be local for the ownership forest, and communication must be local for the containment graph. As in type systems for ownership, we require, by scoping constraints in our semantics, that owners be dominators: the owner of a component c dominates (in the ownership forest) all the components holding references to c. Owing to this condition, the aliasing problem does not arise: when updating a component c, its owner has access to all references to c. Moreover, the containment graph may be an arbitrary directed graph, which allows component sharing. The resulting language, an extension of the Kell calculus [18], turns out to be an interesting model of component-oriented programming, as we show by encoding key aspects of the Fractal component model [2].

Our main contributions are as follows: (1) we propose a programming discipline for component-oriented programming to address the issue of component sharing, while preserving encapsulation and higher-order features; (2) we define a process calculus whose operational semantics embodies this programming discipline; (3) we argue that our calculus is suitable to represent most idioms of component-oriented programming, by reviewing key concepts from a concrete component model; (4) additionally, we propose a new, more modular, presentation of the Kell calculus.

The paper is organized as follows. §2 briefly presents the Fractal model, discusses the modeling of Fractal components in the Kell calculus, and introduces informally several examples of component sharing. §3 extends the Kell calculus with primitive component sharing. §4 shows how to program several component sharing examples within the obtained calculus. §5 concludes the paper with a discussion of future research.

2 Components and the Kell calculus

After giving an informal description of the Kell calculus [18], which is our starting point, we present the main elements of a concrete component model, the Fractal model [2]. We discuss to which extent Fractal component configurations without sharing can be interpreted as processes of the Kell calculus. We then present various examples of component configurations with sharing, and explain informally how we extend the Kell calculus with sharing to deal with these examples.

2.1 The Kell calculus

The Kell calculus is a higher-order process calculus with hierarchical localities (called *kells*), local communication, and locality passivation. Actions in the Kell calculus are communication actions and passivation actions. Communication is said to be *local* as it may occur only within a kell, between a kell and its sub kells, or between a kell and its immediate parent, as illustrated below.

1. Receipt of local message $a\langle Q\rangle.T$ on port a bearing process Q and continuation T by local trigger (input construct) $a\langle x\rangle \triangleright P$.

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

2. Receipt of message $a\langle Q\rangle.T$ residing in sub kell b by local trigger $a^{\downarrow}\langle x\rangle \triangleright P$.

$$b[a\langle Q\rangle.T].S \mid (a^{\downarrow}\langle x\rangle \triangleright P) \rightarrow b[T].S \mid P\{Q/x\}$$

   In pattern $a^{\downarrow}\langle x\rangle$, the arrow $\downarrow$ denotes a message that should come from a sub kell.
3. Receipt of message $a\langle Q\rangle.T$ residing out of the enclosing kell by local trigger $a^{\uparrow}\langle x\rangle \triangleright P$.

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

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

These constructs may be combined using *join patterns* [10] that are triggered only when the required messages are simultaneously present, as in the following example (note that $\mid$ has higher precedence than $\triangleright$).

$$a\langle Q\rangle.T \mid b[c\langle R\rangle.U \mid (a^{\uparrow}\langle x\rangle \mid c\langle y\rangle \triangleright P)].S \rightarrow T \mid b[U \mid P\{Q/x, R/y\}].S$$

Communication with other localities has to be explicitly programmed in the language. For instance, in order to exchange messages, two sibling kells need the help of their common parent, as depicted in the following example.

$$\begin{array}{rlll} & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) & \mid b[c\langle P\rangle.Q] & \mid e[(c^{\uparrow}\langle x\rangle \triangleright T)]] \\ \rightarrow & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) \mid c\langle P\rangle & \mid b[Q] & \mid e[(c^{\uparrow}\langle x\rangle \triangleright T)]] \\ \rightarrow & a[(c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle) & \mid b[Q] & \mid e[T\{P/x\}] \quad] \end{array}$$

In this example, the parent locality contains a permanent forwarder $c^{\downarrow}\langle x\rangle \diamond c\langle x\rangle$ that pulls messages of the shape $c\langle P\rangle$ out of its sub kells. This allows sub kells to receive these messages using an *up pattern* $c^{\uparrow}\langle x\rangle$. The construction $(\xi \diamond P)$ denotes a replicated trigger, i.e., a trigger which persists after a reaction, and is in fact a shorthand for $\nu t.Y_{t,\xi,P} \mid t\langle Y_{t,\xi,P}\rangle$, where $Y_{t,\xi,P} = (t\langle y\rangle \mid \xi \triangleright P \mid y \mid t\langle y\rangle)$.

Passivation in the Kell calculus is depicted in the following example, where the kell named a is destroyed, and the process Q it contains is used in the guarded process P.

$$a[Q].T \mid (a[x] \triangleright P) \rightarrow T \mid P\{Q/x\}$$

Assume, for instance, that we want to model the dynamic update of component b, where the new version P of the component program is received on channel a. We could do so, in one atomic action, using the following join pattern where the new version $b[P]$ is spawned, replacing the previous b component.

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

2.2 The Fractal component model and its interpretation in the Kell calculus

Fractal is a general component model which is intended to implement, deploy, monitor, and dynamically configure complex software systems, including in particular operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), introspection capabilities (to monitor and control the execution of a running system), and re-configuration capabilities (to deploy and dynamically configure a system).

A Fractal component is a run-time entity which is encapsulated, which has a distinct identity, and which is either primitive or composite (built from other components). Bindings between components are described explicitly, either by local, *primitive* bindings, using explicit component interfaces, or by remote, *composite* bindings, using components whose role is to embody communication paths. Features like encapsulation and interfaces are rather standard. The originality of the Fractal model lies in its reflective features and in its ability to define component configurations with sharing. In order to allow for well scoped dynamic reconfiguration, components in Fractal can be endowed with controllers, which provide a meta-level access to a component internals, allowing for component introspection and the control of component behaviour. A Fractal component consists of two parts: *contents*, that correspond to its internal components, and a *membrane*, which provides so-called *control interfaces* to introspect and reconfigure the internal features of the component. The membrane of a component is typically composed of several controllers.

Representing a Fractal component (without sharing) in the Kell calculus is relatively straightforward. A component named a, takes the form $a[P \mid Q]$, where process P corresponds to the membrane of the component, and process Q, of the form $c_1[Q_1] \mid \ldots \mid c_n[Q_n]$, corresponds to the contents of the component, with n sub components c_1 to c_n. Interfaces of a component can be interpreted as channels on which a component can emit or receive messages. The membrane P is composed of controllers implementing the control interfaces of the component.

The Fractal model specifies several useful forms of controllers, which can be combined and extended to yield components with different reflective features. Let us briefly describe some of them, and sketch their interpretation in the Kell calculus.

An *attribute* of a component is a configurable property that can be manipulated by the means of an *attribute controller*. It can be interpreted as some value held in a memory cell by a component membrane. A membrane providing an attribute controller interface is easy to program, by emitting the current value of the attribute on a private channel and by providing channels to read and update this value.

$$\nu s.(\mathtt{get}^{\uparrow}\langle r\rangle \mid s\langle v\rangle \diamond s\langle v\rangle \mid r\langle v\rangle) \mid (\mathtt{set}^{\uparrow}\langle v'\rangle \mid s\langle v\rangle \diamond s\langle v'\rangle) \mid s\langle 0\rangle$$

A *contents controller* supports an interface to list, add, and remove sub components in the contents of a component. A membrane providing a simplistic contents controller interface could be of the form $\mathtt{Add} \mid \mathtt{Rmv} \mid \ldots$, with the following definitions (in which the contents controller interface is manifested by the `cc` channel carrying the request type (where $\backslash add$ means a name that is exactly add), the name c of the targetted component, and either the program of the added component (including both membrane and contents) or a channel r to return the removed component).

$$\mathtt{Add} = (\mathtt{cc}^{\uparrow}\langle \backslash add, c, x\rangle \diamond c[x]) \qquad \mathtt{Rmv} = (\mathtt{cc}^{\uparrow}\langle \backslash rmv, c, r\rangle \diamond (c[x] \triangleright r\langle c, x\rangle))$$

A less simplistic encoding would take into account additional details, such as exception conditions (e.g, the possible absence of a component to remove). However, the above definitions convey the essence of the contents controller.

A *life-cycle controller* allows an explicit control over the execution of a component. As an illustration, we can define a membrane P providing a simple interface to suspend and resume execution of sub components (where the life-cycle interface is manifested by the `lfc` channel, and a sub component c is supended by turning it into a message on a channel of the same name as the component).

$$P = \mathtt{Suspend} \mid \mathtt{Resume} \mid \ldots \qquad \mathtt{Suspend} = (\mathtt{lfc}^{\uparrow}\langle \backslash suspend, c\rangle \diamond (c[x] \triangleright c\langle x\rangle))$$
$$\mathtt{Resume} = (\mathtt{lfc}^{\uparrow}\langle \backslash resume, c\rangle \diamond (c\langle x\rangle \triangleright c[x]))$$

Again, a more realistic implementation would be more complex, but this section only aims to show that capturing the operational essence of a reflective component model such as Fractal (without component sharing) is relatively direct using the Kell calculus. For another example, Schmitt and Stefani [17] provide an interpretation of a *binding controller*, allowing a component to bind and unbind its client interfaces to server interfaces.

2.3 Component sharing

Component sharing arises in situations where some resource must be accessed by several client components. A first example of such a situation is that of a log service, which merely provides client components the ability to register status information. Figure 1 depicts an example configuration, where L is the log service component, and C and D are client components. In this case, communications are unidirectional, from the client

$$\prod_{i\in I;j\in J_i} m_{ij}\left[(msg\langle k,l,x\rangle \diamond n_i\langle k,l,x\rangle) \mid (n_i^{\uparrow}\langle \backslash i,\backslash j,x\rangle \diamond x) \mid P_{ij}\right]$$

$$\mid \quad R\left[\prod_{i,j\in I;i\neq j}(n_i^{\uparrow}\langle \backslash j,l,x\rangle \diamond n_j\langle j,l,x\rangle)\right] \quad \mid \quad \prod_{i\in I}(n_i^{\downarrow}\langle k,l,x\rangle \diamond n_i\langle k,l,x\rangle)$$

Fig. 2. A router configuration

components to the shared component, and the log service maintains its own mutable state. Passivation of a client does not affect the execution of the log service or the processing of logging requests previously sent by that client.

Figure 1 can illustrate as well a second example of component sharing, that of a shared programming library or module. In this case, the communication between client components C and D and the library L is bidirectional (typically, a request/response style of communication). The expected behavior in presence of passivation is different from the first example: if a client is passivated, requests to functions in the library should be suspended along with the rest of the client activity.

As a third example, consider a database service used by several components of a system (for instance, a directory service), which can again be depicted as in Figure 1. Here, the communications between clients and the service are bidirectional, but they are no longer independent as in the previous example, for the database service maintains a mutable state that can be viewed and updated by each client component.

The previous examples correspond to pure software architectures and describe configurations on a single machine. One can also consider mixed software/hardware configurations. For instance, consider the case of a router R connecting several networks N_i with $i \in I$. Each network N_i connects machines m_{ij} with $j \in J_i$. There are several ways to model such a configuration in the Kell calculus without sharing. If one wants to model the networks as components and have messages be directly exchanged between machines and the networks, and between the networks and the router, then the locality of communications and the tree structure of kells impose the following shape:

$$R\left[\prod_{i\in I} N_i\left[\prod_{j\in J_i} m_{ij}\,[\ldots]\right]\right]$$

where $\prod_{i\in I} P_i$ means the parallel composition of the processes P_i.

Such an approach is not satisfactory because the passivation of the router or of a network, e.g., to model their failure, implies the passivation of several machines. A solution consists of modelling a network N_i by a channel n_i, as in Fig.2. Machines send outbound messages on the channel msg with the destination machine address k, l, where k is the destination network, and the message to deliver x. Each machine m_{ij} contains a rule that forwards such messages to the local network n_i. Each network N_i is represented by the replicated pulling of messages on n_i out of sub kells. The

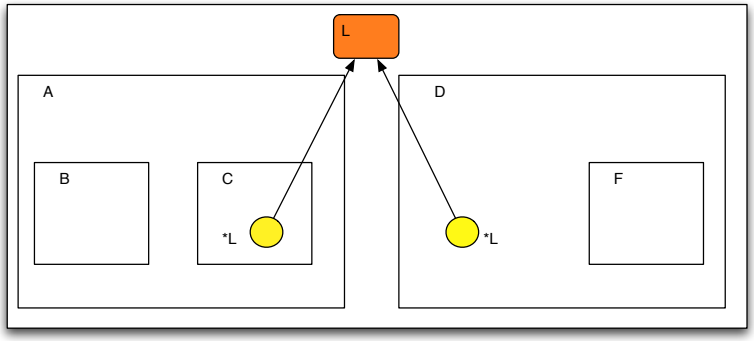

Fig. 3. A Kell calculus configuration with sharing

router pulls messages that are in a network different from their destination—$n_i\langle k, l, P\rangle$ with $i \neq k$—and routes them to the correct network. Finally, every machine m_{ij} picks from the local network the messages that target it using the pattern $n_i^{\uparrow}\langle \backslash i, \backslash j, x\rangle$. This encoding, however, does not model the fact that the networks are disjoint resources shared by the machines and connected by a router.

In this paper, we extend the Kell calculus with explicit sharing, following the ideas sketched in §1. Technically, in our extension, the ownership forest is captured by the locality hierarchy. For instance, in configuration $C = a[b[P] \mid c[R \mid d[Q]]]$, component a is the owner of components b and c, while c is the owner of component d. The containment graph is captured via *references* to shared components: thus the process $*a$ denotes a reference to the component named a. For instance, in the configuration $D = a[H \mid b[P \mid *e] \mid c[R \mid *e] \mid e[Q]]$, component a is the owner of component e, which is shared by components b and c as each of them holds a reference $*e$. The scope of a component e, where it is accessible by references $*e$, is the sub tree rooted at the owner of e, unless there is a deeper component named e whose scope encompasses the reference. Note that the scope does not include e itself. In our extension, a reference $*e$ can be created and communicated, exactly as a name. In the latter case, note that references may escape their original scope: for instance, in the configuration D above, if H passivates component b, and sends it outside of a, then the reference to the shared component e will escape its scope. Allowing a component reference to escape its scope makes it possible to model in a simple way a primitive form of dynamic binding for shared components. The example of Fig.1 may then be represented in the Kell calculus with sharing as in Fig.3.

Passivation in the new calculus takes place just as in the Kell calculus without sharing. However, communications across kell boundaries now require a reference to that kell to receive a message from it or to send a message to it.

$$b[a\langle Q\rangle.T].S \mid (a^{\downarrow}\langle x\rangle \triangleright P) \mid *b \rightarrow b[T].S \mid P\{Q/x\} \mid *b$$
$$a\langle Q\rangle.T \mid *b \mid b[a^{\uparrow}\langle x\rangle \triangleright P].S \rightarrow T \mid *b \mid b[P\{Q/x\}].S$$

Process:	$P ::= 0 \mid x \mid P\|Q \mid \nu a.P \mid a\langle\widetilde{P}\rangle.Q \mid a[P].Q \mid *a \mid \tilde{\xi} \triangleright P$
Pattern:	$\xi ::= a[x] \mid a^{\alpha}\langle\widetilde{\eta}\rangle \mid *a$
Argument pattern:	$\eta ::= x \mid a \mid \backslash a \mid a \mathrel{!}= b$
Place pattern:	$\alpha ::= \bullet \mid \uparrow \mid \downarrow$
Formula:	$F ::= \epsilon \mid r \mid r^{\perp} \mid F\|F$
Resource:	$r ::= \widetilde{M} \mid a^{\downarrow}(\widetilde{M}) \mid a^{\uparrow}(\widetilde{M}) \mid a[P] \mid *a \mid a \mid s$
Spot:	$s ::= \triangleright \mid a[\triangleright] \mid [\triangleright]$
Message:	$M ::= a\langle\widetilde{P}\rangle$

Fig. 4. Processes and formulas

3 The calculus

The syntax of the calculus is depicted in Figure 4. It is based on a denumerable set of *variables* x and a denumerable set of *names* a. *Processes* include the standard null process 0, variables x, parallel composition $P|Q$, and name creation $\nu a.P$, plus some less standard constructs. Messages have the shape $a\langle\widetilde{P}\rangle.Q$, where $\widetilde{P}$ is a list of processes (we use $\widetilde{\cdot}$ in the following to denote a list of $\cdot$'s). In $a\langle\widetilde{P}\rangle.Q$, Q is called a *continuation*, because it is triggered synchronously upon consumption of the message. *Kells* have the shape $a[P].Q$, where a is the name of the kell, P is its contents, and, as for messages, Q is its continuation. The calculus admits references $*a$ as processes, for referencing remote kells named a, as informally described in §2.3. References are also used to send names in messages, as illustrated in matching rules M-NAME, M-CST, and M-NEG below. Finally, the calculus features first-class reduction rules, called *triggers*, which are written $\tilde{\xi} \triangleright P$. Here, $\tilde{\xi}$ denotes a list of *patterns*, where each variable and name is bound at most once (see the definition of scoping below). A pattern ξ may be a kell pattern $a[x]$ for passivation of active kells, a reference $*a$, for suppression of containment links, or a message pattern $a^{\alpha}\langle\widetilde{\eta}\rangle$, for plain communication. In the message pattern, $\widetilde{\eta}$ denotes a list of argument patterns of the shape $x, a, \backslash a$, or $a \mathrel{!}= b$. The first two kinds of argument patterns respectively represent input of processes and names. The third kind $\backslash a$ tests the equality of the corresponding message argument with a. The last kind $a \mathrel{!}= b$ checks that the argument is different from b, and inputs it as a. The direction α indicates where the received message should come from: $\uparrow$ messages come from a parent kell, $\bullet$ messages come from the current kell, and $\downarrow$ messages come from a sub kell.

Processes are scoped as follows. Name restriction is a binder, as usual. Moreover, given a trigger $\tilde{\xi} \triangleright P$, the *defined identifiers* $\mathsf{DI}(\tilde{\xi})$ of $\tilde{\xi}$ bind in P. We define $\mathsf{DI}(\tilde{\xi})$ as follows. Given an argument pattern η, define $\mathsf{DI}(x) \triangleq \{x\}$, $\mathsf{DI}(a) \triangleq \mathsf{DI}(a \mathrel{!}= b) \triangleq \{a\}$, and $\mathsf{DI}(\backslash a) \triangleq \emptyset$. Then, let $\mathsf{DI}(\tilde{\xi})$ be the disjoint union of all $\mathsf{DI}(\xi)$, for ξ in $\tilde{\xi}$, with $\mathsf{DI}(a[x]) = \{x\}$, $\mathsf{DI}(*a) = \emptyset$, and $\mathsf{DI}(a^{\alpha}\langle\widetilde{\eta}\rangle)$ the disjoint union of all $\mathsf{DI}(\eta)$, for η in $\widetilde{\eta}$. Let structural congruence $\equiv$ be the smallest congruence including, as usual,

associativity and commutativity of parallel composition, neutrality of 0 w.r.t. $|$, extrusion of name restriction above $|$, ν, and $a[\cdot].P$, and renaming of bound variables and names.

Resources The reduction relation is based on a labelled transition system (LTS), whose labels represent a trade of *resources* r. As discussed below, such a trade is typically written $F_1 \multimap F_2$, where F_1 and F_2 are *formulas*, to express that the process undergoing the transition offers the resources described by F_2, provided the environment provides the resources in F_1. In particular, the reacting trigger $\tilde{\xi} \triangleright P$ trades some basic resources (messages, passivated kells) against a *reaction token* written $\triangleright$: if the environment provides the expected resources, then the trigger reacts. When composing processes, the corresponding transitions are composed, which may involve the annihilation of some resource requests and corresponding offers, in case they meet.

As defined in Figure 4, there are two kinds of resources. *Basic resources* include messages ($\widetilde{M} \mid a^{\downarrow}(\widetilde{M}) \mid a^{\uparrow}(\widetilde{M})$), where $M ::= a\langle\widetilde{P}\rangle$, passivated kells ($a[P]$), consumed references ($*a$), and permissions ($a$). They are generated directly from processes. For example, a message $a\langle P\rangle.Q$ trades a reaction token against $a\langle P\rangle$, yielding the transition $a\langle P\rangle.Q \xrightarrow{\triangleright \multimap a\langle P\rangle} Q$. As explained in §2.1, we want to control the locality of communications, so this transition should happen in the same kell as the transition involving the reacting trigger, and trades involving $\triangleright$ should only take place at the same level as the reaction.

On the other hand, we cannot completely restrict trades to the level of the reaction, e.g., because the consumed resources may come from shared kells, which syntactically may reside far above the reaction site. This leads us to consider several kinds of reaction tokens, each of them determining the position of the considered transition relatively to the reacting trigger. These reaction tokens are called *spots* $s \in \mathsf{Spots}$.

More precisely, consider a process $S|b[a[(\xi \triangleright P)|Q]|R]$, where the reacting trigger is $\xi \triangleright P$. We have just seen that resources matching the reaction token $\triangleright$ provided by $\xi \triangleright P$ may only come from Q. Immediately above a, i.e., in R, trades may use the information that the reaction lies in some sub kell named a. Thus, in R, $\triangleright$ is viewed as the *sub reaction* token $a[\triangleright]$. Further above a, e.g., from S, it becomes the less precise *internal reaction* token $[\triangleright]$, which only indicates that the reaction lies in some sub kell.

Formulas Formulas are the labels of our LTS. Intuitively, they match the resources offered and requested by the considered process. Formally, *formulas* are defined as in Figure 4, and considered equivalent modulo the following equation schemes:

$$F_1|F_2 = F_2|F_1 \quad (1) \qquad F|\epsilon = \epsilon|F = F \quad (2) \qquad \frac{r \notin \mathsf{Spots}}{r|r^{\perp} = \epsilon} \quad (3) \qquad s|s^{\perp} = s \quad (4)\,.$$

Equation (3) specifies that basic resources (non-spots) are used linearly: they may be consumed only once; (4) specifies that one spot may satisfy several requests, as a join pattern consumes several messages.

Transitions The LTS is defined in Figure 5. Rule MATCH describes reaction, using the notion of *matching* defined below, which is a three arguments judgement written $\xi : F \to \Theta$, where Θ is a *substitution*. A substitution is an element of ($\mathsf{Vars} \to_{\mathsf{fin}}$

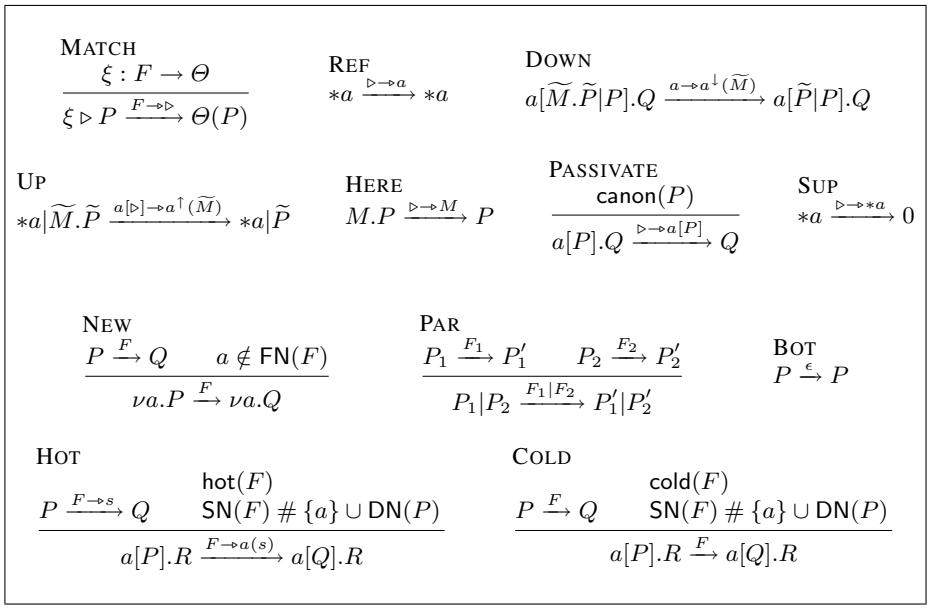

Fig. 5. The LTS

Processes) $\times$ (Names $\to_{\mathsf{fin}}$ Names), i.e., a pair of a finite map from variables to processes and a finite map from names to names. Capture-avoiding substitution is defined as usual on processes, and written $\Theta(P)$. Define the negation $F^{\perp}$ of a formula F by distributing it over resources, given that ${r^{\perp}}^{\perp} = r$. Let $F_1 \multimap F_2$ denote $F_1^{\perp}|F_2$. The rule states that if $\xi : F \to \Theta$, then the trigger $\xi \triangleright P$ has a transition to $\Theta(P)$, under the label $F \multimap \triangleright$. Thus, the reaction happens only if the environment provides the resources F (recall that spot $\triangleright$ stands here for the firing of the trigger).

By rule REF, at the level of a reaction, a reference may generate a permission to receive messages from the kell it points to. This permission is then used in rule DOWN to actually consume the corresponding messages. By rule UP, a reference to the reacting kell allows the reaction to consume messages from the kell holding the reference. By rules HERE, PASSIVATE, and SUP, a reaction may consume messages, active kells, and references at its top-level. In rule PASSIVATE, we use the notation $\mathsf{canon}(P)$ to mean that P has no active ν. This means that such ν's must have been extruded before by structural congruence. Formally, a *context* $\mathbb{C}$ is a process with exactly one occurrence of the special variable $\Box$. Textual replacement of $\Box$ with some process P (possibly with capture) is written $\mathbb{C}\{P\}$. A process P is in *canonical form*, written $\mathsf{canon}(P)$, iff for all context $\mathbb{C} \neq \Box$, if $P = \mathbb{C}\{\nu a.Q\}$, then $\mathbb{C}\{\nu a.Q\} \not\equiv \nu a.\mathbb{C}\{Q\}$.

The other rules specify how the transition relation is closed under active contexts. Rule NEW handles the case of ν. Rule PAR combines the resources of several parts of the process. If one argument provides the resources requested by the other, then the trade occurs. Formally, two derivations having an occurrence of the MATCH rule can be put together using this rule: the restriction to only one active trigger per reaction is

enforced by the rule for reduction, presented below. Rule BOT closes transitions under parallel composition with spectator processes.

Rule HOT allows to wrap an already existing reaction inside some parent kell: a transition $P \xrightarrow{F \multimap s} Q$ is seen from the enclosing kell as $a[P].R \xrightarrow{F \multimap a(s)} a[Q].R$, where the operation $a(s)$ over spots is defined by $a(\triangleright) \triangleq a[\triangleright]$, and $a(b[\triangleright]) \triangleq a([\triangleright]) \triangleq [\triangleright]$. The rule is subject to two side conditions. First, F must be *hot*, written $\mathsf{hot}(F)$, which means that F matches the syntax $F ::= \epsilon \mid b^{\perp} \mid b^{\downarrow}(\widetilde{M}) \mid b^{\uparrow}(\widetilde{M}) \mid F|F$. Second one must have $\mathsf{SN}(F) \# \mathsf{DN}(P)$ (see below). Intuitively, the presence of s in the label of the conclusion imposes that the reaction occurs in P, so the side condition means that a reacting kell only has three kinds of interactions with its context: 1) it (partially) specifies the place of reaction; 2) it exhibits authorizations to access shared kells; 3) it consumes messages through references to shared kells (in both directions). The second side condition enforces the fact that references point to the closest kell in the hierarchy, as informally stated in §2.3. We call the *defined names* $\mathsf{DN}(P)$ of a process P the set of all a's such that $P \equiv \nu\widetilde{b}.Q|a[R].$ for some $Q, R, \widetilde{b}$, with $a \notin \widetilde{b}$. Moreover, a formula is in *canonical form* iff, for each resource r, it does not contain both r and $r^{\perp}$. We define the *scoped names* $\mathsf{SN}(F)$ of a formula F in canonical form as follows: for resources r of the shape $a^{\downarrow}(\widetilde{M}), a^{\uparrow}(\widetilde{M}), a$, and $a[\triangleright]$, let $\mathsf{SN}(r) \triangleq \{a\}$; for other resources r, let $\mathsf{SN}(r) \triangleq \emptyset$. Additionally, let $\mathsf{SN}(F_1|F_2) \triangleq \mathsf{SN}(F_1) \cup \mathsf{SN}(F_2)$ and $\mathsf{SN}(F^{\perp}) \triangleq \mathsf{SN}(F)$. The rule prevents resources consumed through a reference $*a$ to escape the scope of any kell named a. For instance, a request for a message of the shape $a^{\downarrow}(b\langle P\rangle)$ through a reference $*a$ is supposed to be consumed in (one of) the closest kell(s) named a. Such a request leads to the formula $a^{\perp}|a^{\downarrow}(b\langle P\rangle) \multimap s$: if a down message is found in a, using formula $a \multimap a^{\downarrow}(b\langle P\rangle)$, then the reaction occurs. However, if $P_1 \xrightarrow{a^{\perp}|a^{\downarrow}(b\langle P\rangle) \multimap s} P_2$, then we do not want $c[P_1|a[Q_1]] \xrightarrow{a^{\perp}|a^{\downarrow}(b\langle P\rangle) \multimap s} c[P_2|a[Q_1]]$ to hold, because the message ought to be found in Q_1. Here, $\mathsf{DN}(a[Q_1]) = \{a\}$ which is not disjoint from $\mathsf{SN}(a^{\perp}|a^{\downarrow}(b\langle P\rangle)) = \{a\}$. Note that this check is done only when crossing kell boundaries. Indeed, we allow the presence of more than one kell named a in parallel to the reacting trigger.

Symmetrically to rule HOT, rule COLD allows to transfer resources from kells containing references $*a$ to the reacting kell a, which may be syntactically distant. Let F be cold, written $\mathsf{cold}(F)$, iff F matches the syntax $F ::= b[\triangleright]^{\perp} \mid b^{\uparrow}(\widetilde{M}) \mid F|F$. Rule COLD says that any transition with a cold label is viewed identically from outside the ambient kell, provided the scoping conditions are met. In practice, rule COLD is only used to transfer the consumption of up messages (created by rule UP) through kells.

Matching Figure 6 defines the matching relation. Rule M-PAR states that matching a pattern $\xi_1|\xi_2$ is like matching ξ_1 and ξ_2 separately, and then combining the result. In the rule, $+$ denotes the union of finite maps with disjoint domains. By rule M-HERE, matching a pattern $a^{\bullet}\langle\widetilde{\eta}\rangle$ against a resource $a\langle\widetilde{P}\rangle$ boils down to match $\widetilde{\eta}$ against $\widetilde{P}$ (as defined below). Rule M-ELSEWHERE handles the cases of down and up messages. Given a pair ζ consisting of a name a and argument patterns $\widetilde{\eta}$, we let ζ^{α} stand for $a^{\alpha}\langle\widetilde{\eta}\rangle$. Similarly, given a list $\widetilde{\zeta} = \zeta_1|\ldots|\zeta_n$, let $\widetilde{\zeta}^{\alpha} = \zeta_1^{\alpha}|\ldots|\zeta_n^{\alpha}$. The rule tunes the directions (up or down) in order to allow rule M-HERE to apply coherently. Rules M-PASSIVATE

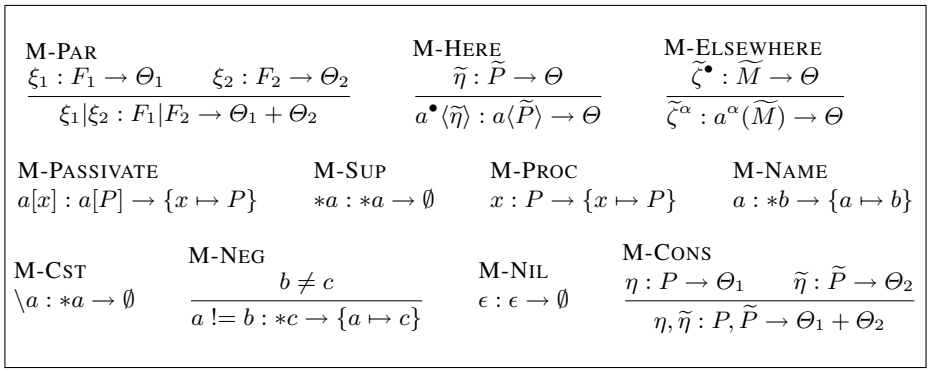

Fig. 6. Matching

and M-SUP are straightforward. For message contents, Rule M-CST states that an escaped pattern $\backslash a$ matches itself, yielding no substitution. Rules M-NAME, M-NEG, and M-PROC handle the input of names and variables. Rules M-NIL and M-CONS dispatch the results.

Reduction Finally, reduction, written $\to$, is the smallest relation satisfying the rule

$$\frac{P \equiv P' \qquad P' \xrightarrow{s} Q' \qquad Q' \equiv Q}{P \to Q}\,.$$

As exactly one spot is allowed, this rule guarantees that exactly one trigger fires.

4 Examples

Let us first present a simple example.

Example 1 *Consider the following configuration.*

$$A = a[(e_1^\uparrow\langle x\rangle \mid e_2^\uparrow\langle y\rangle \triangleright P) \mid c\langle Q\rangle] \quad | \quad l_1[e_1\langle U\rangle \mid *a] \quad | \quad l_2[e_2\langle V\rangle \mid *a \mid (c^\downarrow\langle z\rangle \triangleright R)]$$

The component a can emit the message $c\langle Q\rangle$, which implies that a reference $*a$ to a can be used to access this message. Hence we have the following reduction where the rule in kell l_2 is triggered.

$$A \to a[(e_1^\uparrow\langle x\rangle \mid e_2^\uparrow\langle y\rangle \triangleright P)] \quad | \quad l_1[e_1\langle U\rangle \mid *a] \quad | \quad l_2[e_2\langle V\rangle \mid *a \mid R\{Q/z\}]$$

The component a can also receive messages from both components l_1 and l_2 since it is a shared sub component of both. Hence we have the following reduction where the rule in kell a is triggered.

$$A \to a[P\{U, V/x, y\} \mid c\langle Q\rangle] \quad | \quad l_1[*a] \quad | \quad l_2[*a \mid (c^\downarrow\langle z\rangle \triangleright R)]$$

Let us now give an example of dynamic binding and reconfiguration in the calculus.

Example 2 *Consider the following configuration, which models a running component receiving instructions to update its sub component c with a new code $P(d)$, which uses a service named d.*

$$A = update\langle c, P(d)\rangle \mid *a \mid a[(update^{\uparrow}\langle b, x\rangle \diamond (b[y] \triangleright b[x])) \mid c[P_c] \mid d[P_d]]$$

It reduces in two steps to $*a \mid a[(update^{\uparrow}\langle b, x\rangle \diamond (b[y] \triangleright b[x])) \mid c[P(d)] \mid d[P_d]]$, where the references to d in $P(d)$ have been dynamically bound to $d[P_d]$.

We now review the examples of §2.3 within our calculus. First, assume given two components $Queue[\ldots]$ and $Pair[\ldots]$, working as follows. They expect messages from their parent components, on channels $Queue.push$, $Queue.pop$, $Pair.fst$, and so on. The channels of these messages identify the action to execute. The messages contain a return channel name and the corresponding arguments. On the return channel, $Queue$ and $Pair$ send messages which have to be picked up as down messages by the client parent component. For convenience, we use the syntactic sugar $\mathsf{let}\ x = a(\widetilde{P})\ \mathsf{in}\ Q$ for $\nu b.a\langle b, \widetilde{P}\rangle | (b^{\downarrow}\langle x\rangle \triangleright Q)$, with some fresh b used as return channel. For instance, $\mathsf{let}\ x = Queue.push(P, Q)\ \mathsf{in}\ R$ uses the result x of pushing P on top of Q in R.

Example 3 *The log service example can be represented as follows (reproducing the configuration of Figure 1 with $L = Log$).*

$$\begin{array}{l} Log[*Queue \mid \ldots \text{code to actually log} \ldots \\ \qquad \mid (Log.log^{\uparrow}\langle x\rangle \mid state\langle y\rangle \diamond \mathsf{let}\ z = Queue.push(x, y)\ \mathsf{in}\ state\langle z\rangle)] \\ \mid \quad A[B[\ldots] \mid C[*Log \mid \ldots]] \quad \mid \quad D[*Log \mid F[\ldots]] \end{array}$$

In the rest of the program, the encapsulation links to Log are represented by occurrences of the reference $*Log$. The ownership of Log by, say, o is encoded by the fact that the sub component Log appears at the top-level in o. The implicit scope of Log, restricted to processes encapsulated in o, ensures that o is a dominator of Log.

Example 4 *The shared printer example can be represented as follows, where c stands for "client", and j stands for "job".*

$$\begin{array}{l} Printer[*Queue \mid *Pair \mid \ldots \text{code to actually print} \ldots \\ \qquad \mid (Printer.lpr^{\uparrow}(c, j) \mid state\langle q\rangle \diamond \mathsf{let}\ x = Pair.pair(c, j)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ q' = Queue.push(x, q)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad state\langle q'\rangle) \\ \qquad \mid (Printer.lpq^{\uparrow}(r) \mid state\langle q\rangle \diamond r\langle q\rangle \mid state\langle q\rangle)] \\ \mid \quad A[B[\ldots] \mid C[*Printer \mid \ldots]] \quad \mid \quad D[*Printer \mid F[\ldots]] \end{array}$$

The shared library example can be represented similarly. We can however emphasize the code server aspect of the example with a representation that only requires a unidirectional communication between the clients and the shared library. The shared library is thus modelled as a code server that allows an instance of the library code to be made available on request in the client component that requires it.

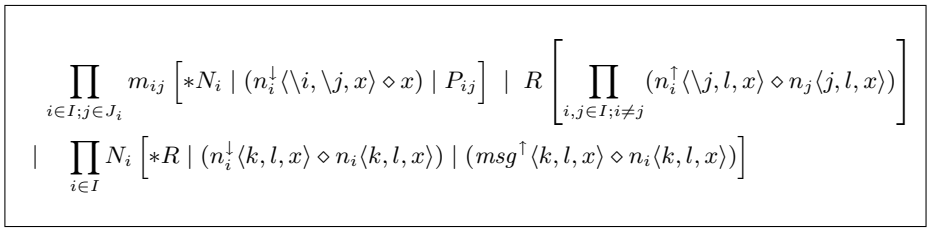

Fig. 7. A better router configuration

Example 5 *The shared library example can be represented as follows, where* $!a\langle P\rangle$ *stands for* $\nu b.(a\langle P\rangle.b\langle\rangle|(b\langle\rangle \diamond a\langle P\rangle.b\langle\rangle))$.

$$Lib[!Lib.get\langle P\rangle] \quad | \quad A[B[\ldots] \mid C[*Lib \mid \ldots]] \quad | \quad D[*Lib \mid F[\ldots]]$$

Finally, we review the router example from Figure 2, which is more direct than Examples 3 and 4 because it does not require any data structure: we just assume that names include integers.

Example 6 *The router example is depicted in Fig.7. It is very similar to Fig.2: the router is identical and shared between the networks, the networks are now kells shared between machines and may directly pull messages out of machines and the router. This encoding allows the failure of the router or a network to only impact inter-machine communication, it also segregates messages in different networks.*

5 Conclusion

Component sharing, as experienced with component models providing it, is a feature that proves extremely useful when describing or programming software architectures or systems with shared resources. We have presented in this paper an extension of the Kell calculus that provides a direct, formal interpretation of component models with sharing. To our knowledge, this is the first calculus offering (1) encapsulation with fine-grain, objective control over communications, (2) locality passivation, migration and replication, and (3) access to shared components with simple communication rules. Our approach draws on a distinction between ownership and containment inspired by recent works on ownership types and the control of aliasing in object-oriented programming languages. In contrast to these works, however, our approach avoids the burden of a type system, by primitively distinguishing ownership from containment, thus enforcing the programming discipline directly in the operational semantics.

The work we have presented here is only preliminary, however. First, the standard issues appearing when one introduces a new process calculus remain to be dealt with, e.g., the development of a bisimulation-based behavioral theory, or of static analyses to ensure semantic properties of processes. Furthermore, it would be interesting to study

the exact relation between approaches to object containment and ownership in object-oriented languages and in the Kell calculus with sharing. At a minimum, we need to investigate the different benchmarks used in the object-oriented programming community and study how they are handled in our calculus.

Second, two important, inter-related questions remain, that pertain (1) to the control of communications with shared components, and (2) to the control over dynamic binding. The first issue concerns a potential security hole in our design. It can be succinctly stated as follows: in the extended Kell calculus presented here, the construct $\nu a.a[a[P]]$ is not a perfect firewall, while it is in the plain Kell calculus. This is due to the fact that P may have references to shared kells, which may in turn allow P to emit and receive messages from its environment. We see two posssible solutions to this problem.

First, one could annotate each kell construct $a[\cdot]$ with explicit sieves on communications with shared components. For instance, let us write $a[P]_A$, where $A ::= \emptyset \mid * \mid \widetilde{a} \mid \neg\widetilde{a}$ represents the names of shared components the present component is allowed to communicate with. Then, define the interpretation of annotations by $[\![\emptyset]\!] = \mathsf{Names}$, $[\![*]\!] = \emptyset$, $[\![\widetilde{a}]\!] = \mathsf{Names} \setminus \widetilde{a}$, and $[\![\neg\widetilde{a}]\!] = \widetilde{a}$. The semantics of these consructs is given by a simple modification of the rules HOT and COLD, given by adding textually the side condition $\mathsf{SN}(F) \# [\![A]\!]$ to both of them. With these new constructs and rules, we recover the perfect firewall equation for $\nu a.a[a[P]_\emptyset]_\emptyset$: P cannot communicate with the environment outside of a.

The second, more radical solution is to introduce a second ν operator, say ∇, that would not cross component boundaries. Channel names bound by ∇ would then represent communication channels, while free names and names bound by ν would represent global names. Distant communication would be restricted to channels, thus preventing an incoming piece of code to arbitrarily communicate with distant components. Global names would serve for matching against local messages. We conjecture that the presence of ν and ∇ avoids the need for directional patterns $(\uparrow, \downarrow, \bullet)$. The calculus thus collapses to a simpler version. The second solution might also turn out to solve the second problem (which is not the case of the first solution): the distinction between local channels and global names might give rise to a fine-grain account of dynamic binding, provided the pattern language is enriched adequately.

References

[1] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D.Gorla, M.Loreti, E.Moggi, R.Pugliese, E.Tuosto, B.Venneri. The KLAIM project: Theory and practice. In *GC*, vol. 2874 of *LNCS*. Springer, 2003.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani. An open component model and its support in Java. In *CBSE*, vol. 3054 of *LNCS*. Springer, 2004.

[3] M. Bugliesi, G. Castagna, S. Crafa. Boxed ambients. In *TACS*, vol. 2215 of *LNCS*. Springer, 2001.

[4] L. Cardelli, A. D. Gordon. Mobile ambients. In *FOSSACS*, vol. 1378 of *LNCS*. Springer, 1998.

[5] G. Castagna, F. Zappa Nardelli. The Seal calculus revisited: Contextual equivalence and bisimilarity. In *FSTTCS*, vol. 2556 of *LNCS*. Springer, 2002.

[6] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.

[7] D. Clarke, T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, vol. 2743 of *LNCS*. Springer, 2003.

[8] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *Computing: the Australasian Theory Symposium*, vol. 78 of *ENTCS*. Elsevier, 2003.

[9] R. De Nicola, D. Gorla, R. Pugliese. Global computing in a dynamic network of tuple spaces. In *COORD*, vol. 3454 of *LNCS*. Springer, 2005.

[10] C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *POPL*. ACM Press, 1996.

[11] M. Hennessy, J. Rathke, N. Yoshida. SafeDpi: a language for controlling mobile code. In *FOSSACS*, vol. 2987 of *LNCS*. Springer, 2004.

[12] M. Hennessy, J. Riely. Resource access control in systems of mobile agents. In *International Workshop on High-Level Concurrent Languages*, vol. 16(3) of *ENTCS*. Elsevier, 1998.

[13] J. Hogg, D. Lea, A. Wills, D. deChampeaux, R. Holt. The Geneva convention on the treatment of object aliasing, 1991.

[14] F. Levi, D. Sangiorgi. Controlling interference in ambients. In *POPL*. ACM Press, 2000.

[15] Y. D. Liu, S. F. Smith. Modules with interfaces for dynamic linking and communication. In *ECOOP*, vol. 3086 of *LNCS*. Springer, 2004.

[16] A. Ravara, A. Matos, V. Vasconcelos, L. Lopes. Lexically scoped distribution: what you see is what you get. In *FGC*, vol. 85(1) of *ENTCS*. Elsevier, 2003.

[17] A. Schmitt, J.-B. Stefani. The Kell calculus: A family of higher-order distributed process calculi. In *GC*, vol. 3267 of *LNCS*. Springer, 2005.

[18] J.-B. Stefani. A calculus of Kells. In *FGC*, vol. 85(1) of *ENTCS*. Elsevier, 2003.

[19] P. T. Wojciechowski, P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *Concurrency*, 8(2), 2000.

[20] N. Yoshida, M. Hennessy. Assigning types to processes. In *LICS*. IEEE, 2000.