Oz/K: A Kernel Language for Component-Based Open Programming

Michaël Lienhardt, Alan Schmitt, Jean-Bernard Stefani

OZ/K: A Kernel Language for Component-Based Open Programming

Michael Lienhardt
Université Grenoble I, France
michael.lienhardt@inrialpes.fr

Alan Schmitt
INRIA, France
alan.schmitt@inrialpes.fr

Jean-Bernard Stefani
INRIA, France
jean-bernard.stefani@inrialpes.fr

Abstract

Programming in an open environment remains challenging because it requires combining modularity, security, concurrency, distribution, and dynamicity. In this paper, we propose an approach to open distributed programming that exploits the notion of locality, which has been used in the past decade as a basis for several distributed process calculi such as Mobile Ambients, Dπ, and Seal. We use the locality concept as a form of component that serves as a unit of modularity, of isolation, and of passivation. Specifically, we introduce in this paper OZ/K, a kernel programming language, that adds to the OZ computation model a notion of locality borrowed from the Kell calculus. We present an operational semantics for the language and several examples to illustrate how OZ/K supports open distributed programming.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Components, Locality, Open Programming

1. Introduction

Open environments involve distributed users that access and combine multiple services. These services interact, fail, and evolve constantly. Programming in such environments remains challenging because it requires, as pointed out in [24] by the designers of the Alice programming language, the combination of several features, notably: (i) *modularity*, the ability to build systems by combining and composing multiple elements; (ii) *security*, the ability to deal with unknown and untrusted system elements, and to enforce if necessary their isolation from the rest of the system; (iii) *distribution*, the ability to build systems out of multiple elements executing separately on multiple interconnected machines, which operate at different speed and under different capacity constraints, and which may fail independently; (iv) *concurrency*, the ability to deal with multiple concurrent events, and non-sequential tasks; and (v) *dynamicity*, the ability to introduce new systems, as well as to remove, update, and modify existing ones, possibly during their execution.

Each of these features has been, and continues to be, the subject of active research on its own. Combining them into a coherent and practical programming language, however, is still an open question, despite interesting developments in the past two decades, including languages such as Acute [29], Alice [24], ArchJava [1], Classages [20], Erlang [3], Java [4], JoCaml [13], Nomadic Pict [35], OZ [34]. Among these, Acute, Alice, and OZ (with its environment MOZART [14, 25]) provide the most extensive support for open programming, but they still fall short, we argue below, of providing enough support for isolation and dynamic reconfiguration.

In this paper, we propose an approach to open programming that exploits the notion of *locality*. This notion has been studied in several families of process calculi such as Mobile Ambients [9], Dπ [15], Klaim [5], or the Seal calculus [10]. We suggest to use the locality concept as a primitive form of *component* that can be used simultaneously as a unit of *modularity*, of *isolation*, and of *passivation* (we call passivation the ability to freeze and marshall a component during its execution). Conflating these different kinds of units into a single notion provides a way to address the different concerns of open programming with few programming constructs. Specifically, we introduce the OZ/K kernel programming language that extends the OZ kernel language with a notion of locality, called *kell*[1], borrowed from the Kell calculus [27], together with a passivation operation, borrowed from the M-calculus [26].

With respect to OZ and MOZART, OZ/K makes a number of contributions: (i) it generalizes the pickling operation in MOZART (i.e. the ability to make values in the language persistent – e.g. for storing them in a file or for sending them in a message) to cover not only stateless values but also complete execution structures; (ii) it allows to define different distributed programming abstractions without depending on a single, pre-defined distribution semantics for the different language entities as is currently the case in MOZART; (iii) it enhances security in OZ through first-class isolation units, and the ability to program sandboxes and security wrappers; (iv) it extends the classical exception handling mechanisms in OZ with failure handling facilities that operate at the component level; and (v) it provides basic support for strong mobility and dynamic reconfiguration through passivation.

Technically, the main contributions of this paper are: (i) the introduction of an extension of the *kell* concept from the Kell calculus [27] and the Kell calculus with sharing [16], with the ability to control communication channels of subordinate kells; (ii) the introduction of a passivation operation, called *packing*, which generalizes the passivation operator of the M-calculus [26] to an execution model with a shared store and logic variables; (iii) the introduction of operations on *packed values* (values resulting from the packing of kells) that provide support for dynamic linking and component replacement; (iv) the introduction of failure handling mechanisms that can deal with thread and component-level failures; (v) a formal operational semantics for the above constructs.

[1] Localities are called *kells* in a loose analogy to biological *cells*.

GPCE'07, October 1–3, 2007, Salzburg, Austria.

The paper is organized as follows. Section 2 discusses current OZ limitations and introduces our approach. Section 3 presents informally the main concepts behind OZ/K and its syntax. Section 4 illustrates, via several examples, how OZ/K supports open distributed programming. Section 5 defines a formal operational semantics for OZ/K. Section 6 discusses related work and Section 7 concludes the paper.

2. Extending OZ for open programming

The OZ language and its MOZART environment already provide several features for open programming. These include in particular: first class *modules* (records that group together related language entities such as procedures) and *functors* (functions that take modules and functors as arguments, and return modules); *module managers*, that allow access to modules referenced by URLs; *pickles*, that can be used to save complete values (i.e. values that do not contain unbound variables) to files; *tickets*, that constitute references to arbitrary language entities; *connections*, that support the establishment of communication links between remote sites using tickets for cross-site references; a distributed semantics (described in [14, 25]) that assigns sites to certain languages entities such as variables, and cells, together with associated *communication protocols* tailored for achieving network transparency with the different kinds of language entities, namely stateless entities (e.g. base values, records, procedures, functors), and stateful entities (e.g. variables, cells). Despite these features, we can single out three main areas where OZ and MOZART fall short of supporting open programming: isolation, support for dynamic reconfiguration, and distribution semantics.

Isolation. Systems operating in an open environment should be ready to deal with unknown, potentially malicious components. A basic strategy to deal with untrusted components is to set up *sandboxes*, as formalized e.g. by the notion of *wrappers* in the Boxed-π calculus [31]. A sandbox is an execution context that isolates encapsulated computations from the rest of their environment, and that prevents unwanted or suspicious communication attempts. More generally, isolating different parts of a running system from one another is required for performance isolation and for preventing denial of service (e.g. to prevent a component interfering with the execution of another one merely through inordinate resource consumption).

The current OZ language and its MOZART environment fail to support sandboxes formalized as Boxed-π wrappers, which allow a strict control of communications between a module or component and its environment. For instance, while it is possible, through the subclassing of the base MOZART module manager, to forbid a downloaded module to access local resources on installation, it is not possible to control the communication of a module with its environment while it executes, and thus to prevent it from discovering – and accessing – forbidden resources in the process.

Support for dynamic reconfiguration. An open distributed environment is a highly dynamic one, where failures, updates, adaptations, and unplanned changes can occur all the time. A language for open distributed programming should provide the means to change a system's structure and behavior on-the-fly, with no need to stop the whole system in order to perform modifications. Dynamic reconfiguration typically involves: the ability to circumscribe the part of a system which needs changing (the *target*); the ability to suspend the execution of the target in a well-defined state; the ability to replace the suspended target by a different subsystem.

The higher-order character of the OZ language allows to program systems as collections of components (e.g. in the form of *port objects* as described in [34]), and to program these components so that their behavior include some operation to change their state (see for instance the upgradable compute server in Chapter 11 of [34]). However, it is not possible to suspend the execution of a component or to delete it (e.g. if some unwanted behavior like unwarranted resource consumption is detected), unless such behavior is already part of the component program. Thus replacing a faulty or malicious component that does not support the appropriate update behavior is not possible in OZ. In addition, it is not possible to capture as a value the state of an ongoing execution (e.g. to take a checkpoint or to reinstate a failed system from a saved checkpoint).

Distribution semantics. An open environment is essentially heterogeneous, with a wide variety of networks and protocols, supporting different communication semantics and providing different guarantees. Furthermore, depending on the application, different levels of distribution transparency and different views of a networked infrastructure may need to be provided. For instance, a deployment application will likely require an explicit view of the individual sites in the target network, so as to control the placement, installation, and configuration of different software components on different sites. This view may be quite detailed, depending on deployment requirements. For instance, one could consider separate spaces for different users, separate component containers for different applications, different tiers in site clusters, with different interconnection schemas, different sub-networks for fault-tolerance and enhanced performance, etc.

It is this very diversity that has lead the designers of the Acute language to abstain from incorporating in their language any specific means of remote interaction. In their words, "a general-purpose distributed programming language should not have a built-in commitment to any particular means of interaction" [30, 28]. The current MOZART environment relies on a predefined distribution semantics. We wish to avoid that dependency to keep in line with the above philosophy, and, in contrast to OZ and MOZART, to allow the definition of a distribution semantics and its supporting protocols within the language itself.

Our approach. To deal with the above issues, we extend the OZ kernel language with a locality construct. The aim is to provide a small and uniform formal basis for open programming capabilities that subsume those of the MOZART environment. As a consequence, open programming features in MOZART which are not expressible in the OZ kernel language (e.g. distribution protocols, or module placement), can now be defined in OZ/K. The OZ kernel language is built using a layered approach, with successive layers adding expressive power and capabilities. The first layer combines logic variables and higher-order procedures. The second layer adds explicit concurrency, in the form of threads. The third layer adds explicit state, in the form of updatable memory cells. The last layer adds lazy execution, in the form of by-need triggers. Our approach adds a new layer to the language, consisting of three main features: (i) a primitive form of component, which we call *kell*; (ii) a primitive operation for passivating kells, which we call *packing*; and (iii) a set of primitive operations for communication between kells, and for manipulating packed values.

A kell acts as a unit of modularity (kells encapsulate data and behavior behind well defined interfaces, called *gates*), a unit of isolation (a kell may fail independently of other kells, and a kell can act as a sandbox for its *subkells*, i.e. for kells that it contains), and a unit of reconfiguration (a kell can be passivated, independently from other kells, then moved, replaced, or deleted). The conflation of these different units in the single notion of kell is the key element of our approach. A kell encapsulates both *activity*, in the form of threads and other (sub) kells, and *state*, in the form of a private data store. Kells can thus be understood as hierarchically organized components, with the same granularity as port objects or active objects in OZ.

In order to achieve isolation, means of communication between kells are restricted to the emission and receipt of messages on *gates*, which are similar to channels in the (synchronous) π-calculus. As a consequence, logic variables, memory cells, and by-need triggers remain private to a kell and cannot be shared between different kells. This design choice is similar to the one made in the Erlang language, where processes, which are the unit of modularity and isolation, only communicate through mailboxes. It is also similar to the one made in the E language, where vats, which are units of concurrency and isolation, only communicate through asynchronous message exchanges (with futures). There are several reasons for this choice, including those well-documented in disfavor of shared state concurrency (see e.g. [18], [2] for a discussion in the context of the Erlang language). The overarching consideration in OZ/K is to avoid any form of shared state between kells to guarantee isolation.

OZ/K does not come equipped with a predefined distribution semantics. Instead, kells provide a basic notion of separation, from which different forms of remote interaction can be built, in line with the Acute philosophy discussed above. Communication on gates, which takes the form of atomic rendez-vous, should thus be seen as local communication. Remote interaction in OZ/K can be modeled by a program mediating communications between two or more peer kells (communication can take place via gates between a thread situated in a kell and a thread situated in the immediate parent kell).

One may ask why we did not consider adding this last layer to OZ as a library instead of language extension. The reason can be given as a three-pronged argument: (i) we wish to have a simple formal semantics for our kernel language; (ii) we consider that a library ought to be programmable (even if not actually implemented) in terms of its host language, so as to avoid introducing constructs that are not definable in the host language semantics; (iii) the isolation achieved by kells, and the passivation operation cannot strictly be expressed in OZ. Consideration (ii) ensures that different forms of remote interaction can be defined and understood by OZ/K programmers as programs that relay information between peer kells.

3. Syntax and overview

The OZ/K kernel language is built as a conservative extension of the OZ kernel language as formally described in chapter 13 of [34]. We first recall briefly the main constructs of the OZ kernel language, and then present the OZ/K-specific constructs. For more information on OZ and its supporting MOZART environment, please refer to [34].

3.1 OZ core

The basis for OZ/K is the OZ kernel language [14, 25, 34], featuring logic variables (single assignment variables), higher-order procedures, memory cells (which support multiple assignments), exception handling, concurrent threads, and by-need synchronization. We just present here the constructs of OZ which we use in our programming examples, the full language description may be found in the references above. The OZ execution model consists of *dataflow threads* that operate on a *shared store*. Threads contain *statement* sequences and communicate through shared references in the store.

The syntax of the OZ kernel language constructs we use in this paper is given in Table 1, where `S` and its decorated variants denote statements; `P`, `X`, `Y`, `C`, and their decorated variants denote variable identifiers; `v` denotes base values (integers and literals – i.e. names or atoms); and `J` denotes patterns.

Variables and values. The store includes *logic variables* (or *variables*, for brevity) that can be bound or unbound. An unbound variable does not yet refer to a value. A bound variable `X` refers to a definite value, which can be a base value (an integer, an atom, or a name), or a record. Atoms are values whose identity is determined by a sequence of printable characters. A record takes the form `l(f1:X1 ..... fn:Xn)`, where `l` is the *label* of the record, `f1,...,fn` are the *features* of the record, and variables `X1,...,Xn` (which can be bound or unbound) are the *fields* of the record. Assuming `R` is a record with feature `f`, record selection is written `R.f` (i.e. if `R = l(... f:X ...)`, then `R.f` evaluates to `X`). Tuples are records with consecutive integer features, starting with `1`. A tuple `lab(X1 ... Xn)` corresponds precisely to the record `lab(1:X1 ... n:Xn)`.

S ::=	**skip**	*empty statement*
|	S1 S2	*sequential composition*
|	**thread**{X} S **end**	*thread creation*
|	**local** X1 ... Xn **in** S **end**	*variable introduction*
|	X = Y	*imposing equality*
|	X = v	*binding to base value*
|	X = l(f1:X1 ... fn:Xn)	*binding to record*
|	**if** X **then** S1 **else** S2 **end**	*branch statement*
|	**case** X **of** J **then** S1 **else** S2 **end**	*pattern matching*
|	{NewName X}	*name creation*
|	**proc**{P X1 ... Xn} S **end**	*procedure definition*
|	{P X1 ... Xn}	*procedure call*
|	...	

Table 1. Syntax: OZ core

New variables are introduced with the statement:

```
local X1 ... Xn in S end
```

where `S` is an arbitrary statement, and `X1,...,Xn` are the n new variables being introduced[2]. Variables are introduced unbound. To bind a variable to a value, one can use equality statements of the forms (where `v` is an arbitrary base value, and `X1,...,Xq` are variables):

```
X = v                          X = l(f1:X1 ... fq:Xq)
```

Note that in statement: `X = l(f1:X1 ... fq:Xq)` variables `X1,...,Xq` may be unbound. Two variables `X` and `Y` can be constrained to be bound to the same value through the statement `X = Y` (unifying the two values if `X` and `Y` are already bound).

Names. Names are unforgeable constants that are typically used to identify, and refer to, various execution entities, such as procedures and threads. There are three special names with reserved keywords: **`unit`**, **`true`**, and **`false`**. Names **`true`** and **`false`** denote the boolean values true and false, respectively. The name **`unit`** is typically used as a synchronization token. Names are created with the statement:`{NewName N}` which binds the variable `N` to a fresh name, guaranteed to be unique among OZ/K computations.

Procedural abstraction. A procedure definition takes the form:

```
proc{P X1 ... Xn} S end
```

where the variable `P` is bound to the newly created procedure, the identifiers `X1 ... Xn` correspond to the formal parameters of the procedure, and `S` is a statement that constitutes the body of the newly created procedure. The scope of the identifiers `X1 ... Xn` is the body `S` of the procedure. Note that formal parameters of a

[2] We adopt the OZ syntactic constraint that variables start with an upper case letter. A lexical token that is not a keyword and that starts with a lower case letter is deemed to be an atom. Thus `Var` is an identifier for a variable, whereas `var` denotes the atom `'var'`. Keywords are written in boldface.

procedure can be input parameters (variables bound prior to the procedure execution) or output parameters (variables bound during the procedure execution). This means that a procedure may return any number of results, including none. A call to the procedure named `P` takes the form: `{P A1 ... An}`, where `A1 ... An` are variables corresponding to the actual parameters of the call. Note that procedures can be higher-order, i.e. take variables bound to procedures as parameters.

Concurrency. Sequential composition of statement `S1` with statement `S2` is written `S1 S2`. The empty statement is **`skip`**. Statement **`thread`**`{T} S` **`end`** creates a new thread that executes statement `S`, and binds its (freshly generated) name to variable `T`.

Control flow. Two conditional statements from OZ are used in our examples:

```
if X then S1 else S2 end
case X of J then S1 else S2 end
```

Both constructs block until the variable `X` is bound. When `X` is bound to a value `v`, this value is matched against **`true`** (for the **`if`** construct) or `J` (for the **`case`** construct). If the match succeeds, statement `S1` is executed, otherwise statement `S2` is executed. For instance, if `X` is bound to the record `rec(a:V1 b:V2)`, then the statement

```
case X of rec(a:X1 b:X2)
  then {P X1 X2} else skip end
```

evaluates to `{P V1 V2}` (pattern variables `X1` and `X2` are bound during pattern matching to `V1` and `V2`, respectively).

3.2 OZ/K constructs

To the OZ core, OZ/K adds three main elements: *kells*, *gates*, and *packing*. The syntax for the OZ/K-specific constructs is given in Table 2, where `K`, `X`, `Y`, `Z`, `G` denote variable identifiers.

`S` ::=	...	
|	**`kell`**`{K} S` **`end`**	*kell creation*
|	`{NewGate X}`	*gate creation*
|	`{Send G X}`	*emitting message X on gate G*
|	`{Receive G X}`	*receiving message X on gate G*
|	`{Open K G}`	*grant kell K access to gate G*
|	`{Close K G}`	*revoke access to gate G for kell K*
|	`{Pack K X}`	*packing kell K*
|	`{Unpack X Y}`	*unpacking packed value X*
|	`{Mark X Y Z}`	*marking X with names in Y*
|	`{Status K X}`	*get status of thread K*

Table 2. Syntax: OZ/K extensions

Kells. A *kell* is a computational location, i.e. a form of concurrent component, which associates a *named locality* to part of an OZ/K computation. Localities are organized in a tree where each node contains a (logically) private store and several running threads. Kells are created via statements of the form:

```
kell{K} S end
```

where `K` is bound to the (freshly generated) name of the newly created kell. Statement `S` corresponds to the body of the kell. Upon creation of kell `K`, the execution of `S` starts in a new thread running within `K`. In order to ensure isolation, `S` must contain only *strict* variables (except for `K` which is bound during the creation of the kell). Strict variables are variables which are bound to strict values, i.e. values which, recursively, do not contain unbound variables. In effect, kells partition OZ/K computations into isolated subsets that can only communicate through *gates*. Note that strictness is not difficult to implement since it involves the same traversal of the value graph than unification.

Gates and communication. A *gate* in OZ/K denotes an interaction point for a kell. It is similar to a π-calculus channel: two kells can only communicate through a gate, and gate names can be sent across gates. A gate can be created via a call of the form: `{NewGate G}`. Once the gate has been created, it can be used to send values (`{Send G X}`) or to receive them (`{Receive G X}`).

Communication through gates is by atomic rendez-vous: a `Receive` statement is successful only if there is a matching `Send` statement available in a different thread. This mode of communication on gates, together with the isolation property, allows locality passivation (packing) to take place at any point in time during an execution. Having an atomic rendez-vous as a primitive form of communication allows to derive other forms of interaction, including ones that implement flow control between emitters and receivers. In particular, component connectors can be realized as kells that mediate communication between two or more peer kells. Only *strict* values can be sent through a gate. This restriction ensures that kells remain isolated during execution, and that gates form the only means of communication between kells. Communication on gates should be understood as local, i.e. as taking place on a single machine. Remote communication in OZ/K can be *modeled*, as illustrated in the next section, by programs that relay information between two or more peer kells, using two or more gates.

Controlling communication. In order to support sandboxing, kell boundaries can impose restrictions on communications. By default, communication may cross at most one kell boundary: it is allowed within a kell, and between a kell and its parent-kell. Direct communication on some gate `G` between two kells separated by more than one boundary is only allowed if every kell boundary crossed by the communication has this gate *open*. A gate `G` can be opened in the boundary of a kell `K` by its parent-kell using the procedure call `{Open K G}`. Hence, to allow two sibling kells `K1` and `K2`, children of kell `K`, to communicate directly on a gate `G`, one has to open `G` for both `K1` and `K2` via this statement in kell `K`:

```
{Open K1 G} {Open K2 G}
```

To make a parent kell *transparent* for some or all of its child-kells, one can use the key-word **`all`** to reference all the child-kells, or all the gates in a kell. For instance, the statement `{Open` **`all all`**`}` opens all the gates for all the children-kells of the current kell.

Packing and unpacking. `{Pack K V}` is the statement implementing passivation. It suspends the execution of the child `K` of the current kell and marshalls it, together with the relevant portion of the store, in a *packed value* bound to the variable `V`. Packed values can be modified using the `Mark` operation. Specifically, `{Mark V1 R V2}` returns in `V2` the packed value `V1` modified according to the instruction given by tuple `R`. If `R=gate(G1 G2)`, the gate `G1` is replaced in `V2` by `G2`. If `R=prc(P Q)`, the procedure `P` is replaced by `Q`. If `R=top(K)`, the top-kell of the packed value is replaced by `K`. A side-effect of `Mark` is that it prevents *marked* names to be changed during unpacking of the packed value. Thus, the statement `{Mark V1 gate(G G) V2}` only marks gate `G` to prevent it being renamed when unpacking `V2`.

The statement `{Unpack V R}` can be used to unpack a packed value `V`. Unpacking creates an execution structure similar to the one which has been packed, with new names for its gates, kells, and procedures, with the exception of the ones which have been marked. These new names are returned in the record `R`, which consists of old names as features with corresponding new names as fields. A `Mark` operation on a packed value can be understood as a

dynamic linking operation that connects a kell about to be unpacked to its new environment.

Failure handling. OZ has only classical exception handling. In our context, we need to deal with thread-level and kell-level failures. This is made possible by the detection of thread failures, via the `Status` statement, which is explained in Section 5.

4. Open programming in OZ/K: examples

In this section, we present some simple OZ/K programs illustrating open programming in OZ/K. In the code fragments presented below, we often replace "**`local`** `X` **`in`** `S` **`end`**" by "`X` **`in`** `S`" (a licit OZ syntactic convenience).

Distribution As explained above, OZ/K has no built-in support for remote communications. However, because of their inherent separation, kells can be used to *model* different sites, communicating using different communication semantics.

Here is for instance a simple configuration, with two sites `Site1` and `Site2`, running programs `P1` and `P2` respectively. The kell `Net` acts like an interconnecting asynchronous network, relaying messages from one site to another: we suppose that `S1` (resp. `S2`) listen on the gate `In1` (resp. `In2`) and emit on `Out1` (resp. `Out2`).

```
kell{Site1} {P1} end
kell{Site2} {P2} end
kell{Net}
  Relay in
    proc{Relay G1 G2}
      M in
      {Receive G1 M} thread {Send G2 M} end
      {Relay G1 G2}
    end
    thread {Relay Out1 In2} end
    thread {Relay Out2 In1} end
end

{Open Net all}
{Open Site1 In1}{Open Site1 Out1}
{Open Site2 In2}{Open Site2 Out2}
```

Note that the statement `{Open Net all}` ensures that the `Net` is allowed to communicate on all the gates (`In1`, `In2`, `Out1`, `Out2`), with its sibling kells `Site1` and `Site2`. On the other hand, `Site1` is only allowed to communicate with its sibling `Net` on gates `In1` and `Out1`. This prevents its direct communication with `Site2`, which is only allowed to communicate on gates `In2` and `Out2`.

This (evidently simplistic) example illustrates how the separation between different loci of computation can be used to model a networked environment. Note that we encapsulated the network in a separate kell: this would allow us, for instance, to model failures of the `Net` component independent from failures of sites. A programmer can thus be provided with a semantics for distributed computation in terms of the OZ/K computation model. Importantly, this semantics can be adapted to different network environments, and arbitrary details of the supporting infrastructure revealed to the programmer, without having to change the language semantics. One can thus provide different abstractions to distributed programmers, depending on their needs, the network environment considered, and the level of distribution transparency required, as in [6].

Modules and pickling. The notion of kell unifies notions of software modules and components, and packing generalizes the pickling construct provided by the MOZART environment. Consider for instance the following code, where `G` is a gate:

```
kell{Mod}
  Proc Rec T in
    proc{Proc Param} ... end
    proc{T X} {Send G X}{T X} end
    Rec = m(op:Proc)
    {T Rec}
end
```

This code fragment creates a new kell which defines a unary procedure, `Proc`, and puts it in a record `Rec` which is continuously available on gate `G`. In effect, `Rec` corresponds to a simple software module that consists of just one procedure, accessed through the feature `op`.

Using the module is straightforward, one just has to bind to the output gate to retrieve the module and call the module's procedure:

```
{Receive G Y}
{Y.op M}
```

Importantly, kell `Mod` can be packed and sent to a different location (another kell), so that the module can be made available there. For instance, assuming `Out` is a gate on a channel to a different site, as in the distribution example, then the program

```
{Pack Mod Z}
{Send Out Z}
```

illustrates how to marshall the kell `Mod` using the packing operation, and how to send the resulting packed value for use of the module at a different site. Note that programs that retrieved the module prior to its packing can still use its procedure, since it has been communicated prior to packing.

Strong mobility and dynamic linking. The previous module example works fine as long as the client program does not move. Indeed, assume the client was running on the site `S1` and is transferred to some other site `S2`. The client may now be required to use the module's implementation local to site `S2`.

We show how to change the definition of the module, which will ensure that each copy of the module dynamically retrieves the local implementation upon each call, so as to take into account possible moves of clients of the module. The module code now is:

```
kell{Mod}
 Proc Rec T P in
     proc{Proc Param} ... end
     proc{T X} {Send G X}{T X} end
     proc{P Param}
       Z in {Receive G Z}{Z.loc Param}
     end
     Rec = m(op:P loc:Proc)
     {T Rec}
end
```

As before, the procedure `Proc` is the local implementation of the module's functionality, and the procedure `P` corresponds to the front-end of the module, that retrieves at each call the local implementation of the module, and then executes the proper procedure. Using the module `Mod` in this case remains similar to the previous example: just access the procedure through the module's `op` feature. The `loc` feature, storing the local implementation of the procedure and is not supposed to be directly used by the client[3].

Now, as the client moves from the site `S1` to the site `S2`, the gate where the module is available changes, raising the necessity to modify the reference of the gate in the client's code. This modification of the client's code is done using the procedure `mark`, as shown below (we suppose that the module is available at `G1` – resp. `G2` – at site `S1` – resp. `S2`):

[3] The privacy of the `loc` feature can be enforced using names and procedures. See chapter 3 in [34]. A slightly more complex version would use two gates, one to send the module containing `op`, the other to send the local implementation.

```
%% Site 1
{Pack Client Z}
{Send Out1 msg(service:G1 pack:Z)}

%% Site 2
Mes K Z1 Z2 List in
 {Receive In2 Mes}
 case Mes of msg(service:G pack:Z) then
   kell{K}
     {Mark Z gate(G G2) Z1}
     {Mark Z1 top(K) Z2}
     {Unpack Z2 List}
   end
 else skip end
end
```

Isolation The kell construct allows to build configurable sandboxes. Consider the case of a plug-in of dubious origin. It is possible to isolate it in different ways.

A first approach is a straightforward application of marking and communication control. In this case, the only communication allowed by the `Sandbox` is on gate `G`, unknown initially to the plug-in. Thus the plug-in must be configured to use this gate. To this end, the output gate used by the plug-in is specified under the `gate` feature. Using the `mark` procedure, it is replaced by gate `G`. Then the plug-in is unpacked inside a kell `K`, itself inside the sandbox. This double inclusion prevents any communication of the plug-in with the environment of the sandbox, apart from the ones explicitly allowed on `G`. This gate is opened for both kells `K` and `Sandbox`.

```
Sandbox Mes in
  {Receive In Mes}
  case Mes of msg(gate:PG plugin:Z) then
    kell{Sandbox}
     K Z1 Z2 List in
      kell{K}
        {Mark Z gate(PG G) Z1}{Mark Z1 top(K) Z2}
        {Unpack Z2 List}
      end
      {Open K G}
    end
    {Open Sandbox G}
  else skip end
```

The behavior of a sandbox can be more complex. For instance, we may allow the plug-in to request the opening of some gate for communication. The program can then check the security of such an opening, using the procedure `Check`, and allow it or not. In the following program, the plug-in can request a gate opening by sending on channel `G` a record of the form `r(gate:G1 Resp:R)`, where `G1` is the gate to open and `R` is the gate used to give the plug-in the answer of the `Check` procedure. The procedure `Control` receives the message and calls the `Check` procedure to validate the request. As before, the gate has to be open for kells `K` and `Sandbox`, sending a message on `G0` for the later. The installation of the plug-in is done as in the previous example, launching an additional `Control` thread.

```
Check Control Sandbox OpenSandbox Mes G0 in
 {NewGate G0}
 proc{OpenSandbox}
   {Receive G0 GateName}
   {Open Sandbox GateName}
   {OpenSandbox}
 end
 thread {OpenSandbox} end

 proc{Check K G1 Resp} ... end

 proc{Control K G}
   Mes in
     {Receive G Mes}
     case Mes of r(gate:G1 resp:R)
     then B in
       {Check K G1 Resp}
       if Resp then {Open K G1} {Send G0 G1} {Send R ok}
       else {Send R nok} end
     else skip end
     {Control K G}
 end

 {Receive In Mes}
 case Mes of msg(gate:PG plugin:Z) then
   kell{Sandbox}
    K Z1 Z2 List G in
     {NewGate G}
     kell{K}
       {Mark Z gate(PG G) Z1}{Mark Z1 top(K) Z2}
       {Unpack Z2 List}
     end
     thread {Control K G} end
   end
 else skip end
```

The encapsulation realized by the kell construct allows in particular to build wrappers as in the Boxed-π calculus [31]. For instance, we can build a simple *filtering wrapper* for some untrusted plugin, where the used service is made available on gate `SV`.

```
Filter Mes Sandbox in
  proc{Filter G1 G2} ... end
  {Receive In Mes}
  case Mes of msg(gate:PG plugin:Z) then
    kell{Sandbox}
      K Z1 Z2 List G in
        {NewGate G}
        kell{K}
          {Mark Z gate(PG G) Z1}{Mark Z1 top(K) Z2}
          {Unpack Z2 List}
        end
        thread {Filter G SV} end
    end
  else skip end
```

In this example, the procedure `Filter` acts as a partial relay between the gates `G1` and `G2`, transmitting only *valid* messages and erasing the others.

Handling failures Failure handling in OZ/K bears some strong similarity with failure handling in Erlang [2], and with a recent proposal for enhanced failure handling in OZ [11]. Units of failure in OZ/K are threads and kells. Handling a failure in a thread or a kell requires setting up an independent thread that can monitor state changes in the supervised thread or kell. Setting up a monitoring thread can be done as in the following program:

```
proc {NewMonThread Body Gate}
 Th Monitor in
 thread{Th} {Body} end
 thread{Monitor}
   S in {Status Th S}
   case S of failed(Z) then {Send Gate failed(Th Z)}
   else skip end
 end
end
```

The above program creates two threads, the monitor thread `Monitor`, and the monitored thread `Th`. The behavior of `Monitor` is simple: it waits for `Th` to fail, and then notifies this failure on gate `Gate`. Notice that the case statement here matches the thread name with the *execution status* of the thread (see Section 5 for a definition of the execution status of a thread or of a kell). Using `Pack`, it is also possible to force a kell to abort upon the occurrence of some failure

in one of its threads, thereby obtaining a similar effect to *process linking* in Erlang, which causes a group of Erlang processes to fail together if one of the processes in the group fails. In our case, we can link threads by placing them in a kell and setting up an appropriate monitoring structure. This is illustrated by the following fragment, where two threads are linked in a kell, which is aborted as soon as one the two threads fails:

```
K Gate Mes Z in
 {NewGate Gate}
 kell{K}
   {NewMonThread Body1 G} {NewMonThread Body2 G}
 end
 {Receive G Mes}
 case Mes of failed(T Z) then
   {Pack K Z} {Send MG failed(K Z)}
 else skip end
```

Note also that when the kell `K` fails, a failure message is sent on the monitoring gate `MG`. This illustrates how kells can be monitored for failure as well.

5. Operational semantics

To facilitate comparison and cross-reference, we adopt for the OZ/K semantics the same approach as that in Chapter 13 of [34]. The OZ/K operational semantics is given in terms of a reduction relation $\rightarrow \subseteq (\texttt{Store} \times \texttt{Task})^2$. We call *execution structure* an element of $\texttt{Store} \times \texttt{Task}$, i.e. a pair consisting of a store and a task. We assume given the following infinite countable and mutually disjoint sets: `Ident`, the set of variable identifiers, `Var` the set of logical variables, `Name` the set of names, `Atom` the set of atoms.

Statements. For the purpose of the presentation of the operational semantics, we consider extended statements where logical variables can be substituted to some or all variable identifiers in a statement S. The effect of such a substitution is defined classically, the binding constructs including variable introduction, procedure definition, and pattern matching.

Tasks. The set of tasks, `Task`, consists of elements $\mathcal{T}$ given by the following grammar (where τ denotes a thread name):

$$\mathcal{T} ::= \tau T \quad | \quad \mathcal{T}\ \mathcal{T} \qquad \textit{tasks}$$
$$T ::= \langle\rangle \quad | \quad \langle S\ T\rangle \qquad \textit{thread stacks}$$

Intuitively, a task $\mathcal{T}$ is a set (parallel composition) of named threads τT.

Stores. A store consists of a conjunction (noted $\wedge$) of primitive assertions. Primitive assertions comprise:

- Variable bindings, of the form $x = M$, where x is a variable and M can either be $\bot$, meaning that x is unbound, or some value (integer, atom, name, record, or packed value).
- Name bindings, of the form $\xi : N$, where N is some semantical value such as a procedure, a gate, or a kell.
- Additional assertions, of the form $\texttt{pred}(\ldots)$, where `pred` is some predicate qualifying or relating names or variables. For instance, $\texttt{read}(r)$ means that r is a read-only variable.

A packed value $\texttt{pack}(\kappa, \mathcal{T}, \sigma, \mu)$ consists of four elements: the name κ of the kell that has been packed, a suspended thread set $\mathcal{T}$, its associated store σ, and the set of names μ that have been *marked* with a call to the procedure `Mark`: these names must not be changed during unpacking.

A name binding $\tau : \texttt{thread}(x)$ refers to a thread named τ, whose *execution status* is given by the variable x. While the thread is running, variable x remains unbound. If the thread terminates normally, then x becomes bound to the value `terminated`. If the thread fails because of an uncaught exception, x becomes bound to a failed value of the form $\texttt{failed}(y)$, where y is the exception that caused the thread to fail. The status of a thread `K` can be obtained using the statement `{Status K X}`, which returns the status of `K` in variable `X`. A name binding of the form $\kappa : \texttt{kell}(\pi, x)$ states that κ is a kell name. In this statement, π is a set of pairs of the form $\kappa' \cdot \gamma$, defining that the gate γ is open for the child-kell κ'. The variable x defines the execution status of the kell: it is either unbound when kell is *active* or bound to the value `packed` when the kell is passivated.

The predicate $\texttt{in}(\kappa, \kappa')$ indicates that the kell κ' is located inside kell κ. The predicate $\texttt{inth}(\kappa, \tau)$ indicates that the thread τ is located inside kell κ. We note $\sigma \models \phi$, where ϕ is conjunction of primitive assertions, to indicate that $\sigma = \sigma' \wedge \phi$, for some σ'.

5.1 Reduction relation

We can now define the reduction relation $\rightarrow$ as the smallest relation that satisfies the set of inference rules given below, together with the inference rules in chapter 13 of [34] (that give the operational semantics of the OZ core).

To facilitate the comparison with the original OZ operational semantics, we use the same notational conventions as in [34], noting $(T, \sigma) \rightarrow (T', \sigma')$ as $\dfrac{T \;\|\; T'}{\sigma \;\|\; \sigma'}$. The reduction rules take the form of inference rules of the form

$$\frac{T \;\|\; T'}{\sigma \;\|\; \sigma'} \quad \text{if } C$$

where C is some condition on T, σ, T', and σ'.

Before giving the reduction rules[4], we define informally some auxiliary functions and predicates. The appendix provides additional formal details. The function `dom` takes a store σ as parameter and returns the set of all the names and variables used in σ. The predicate $\texttt{strict}_\sigma(\texttt{v})$ is true if `v` is a *strict* value in the store σ. We extend this predicate on variables x and statements `S`. The predicate $\texttt{strict}_\sigma(\texttt{S}, y)$ is true if all the free variables in extended statement `S`, except y, are strict. The assertion $\texttt{access}_\sigma(\gamma, \kappa, \kappa')$ means that gate γ is accessible for communication between the kells κ and κ'. For this to be true, γ must have been opened for communication for all the kells on the path that connects κ_1 and κ_2 in the kell tree, unless they are separated by at most one kell boundary. The function $\texttt{grant}_\sigma$ associates to the pair of variables (k, g) a set: the singleton pair corresponding to their names if k denotes a kell and g denotes a gate, and $\emptyset$ otherwise. The last auxiliary functions are $\texttt{subk}_\sigma$ and $\texttt{subth}_\sigma$: $\texttt{subth}_\sigma(\kappa)$ returns the set of names of all the threads contained by the kell κ and all its descendant kells, $\texttt{subk}_\sigma(\kappa)$ returns the set of names of all the descendant kells of kell κ.

Contextual rules The contextual rules define reductions under parallel task contexts and for equivalent configurations. For lack of space, we only give the rule pertaining to parallel contexts.

$$[\textsc{Par}]\ \frac{\mathcal{T}_1\ \mathcal{T}_2 \;\|\; \mathcal{T}_1'\ \mathcal{T}_2}{\sigma \;\|\; \sigma'} \text{ if } \frac{\mathcal{T}_1 \;\|\; \mathcal{T}_1'}{\sigma \;\|\; \sigma'}$$

Kell abstraction The rules pertaining to the kell abstraction deal with the creation and the replacement of kells. The rule for kell creation is given below.

$$[\textsc{NewK}]\ \frac{\tau\langle \texttt{kell}\{y\}\ S\ \texttt{end}\ \ T\rangle \;\|\; \tau T\ \ \tau'\langle S\ \ \langle\rangle\rangle}{\sigma \;\|\; \sigma \wedge \sigma'} \text{ if } C$$

[4] For lack of space, we only give the main rules. Rules which have been left out, which can be found in the full paper [**?**], include the rule for kell replacement, the rules that cater to various exception conditions – in particular dynamic typing errors –, the rule of thread creation, the rule for `Status`, and rules that give access to certain names held in a packed value.

$$C \equiv \kappa', \tau', w, r \notin \mathtt{dom}(\sigma) \wedge \mathtt{strict}_\sigma(S, \{y\}) \wedge y = \bot \wedge \mathtt{inth}(\kappa, \tau)$$
$$\sigma' \equiv y = \kappa' \wedge \kappa' : \mathtt{kell}(\emptyset, w) \wedge \mathtt{read}(w) \wedge \tau' : \mathtt{thread}(r) \wedge \mathtt{read}(r) \wedge \mathtt{inth}(\kappa', \tau') \wedge \mathtt{in}(\kappa, \kappa')$$

Gate abstraction The rule for the creation of new gates follows.

$$[\textsc{NewG}]\ \frac{\tau\langle\{\mathtt{NewGate}\ \ x\}\ \ T\rangle \ \Big\|\ \tau T}{\sigma \ \Big\|\ \sigma \wedge x = \gamma \wedge \gamma : \mathtt{gate}} \quad \text{if } \gamma \notin \mathtt{dom}(\sigma) \wedge \sigma \models x = \bot$$

The rule COM below governs communication through gates.

$$[\textsc{Com}]\quad \frac{\tau\langle\{\mathtt{Send}\ g\ x\}\ \ T\rangle \quad \tau'\langle\{\mathtt{Receive}\ h\ y\}\ \ T'\rangle \ \Big\|\ \tau T \quad \tau' T'}{\sigma \ \Big\|\ \sigma \wedge y = x} \ \text{if } C$$

$$C \equiv \mathtt{strict}_\sigma(x) \wedge \mathtt{access}_\sigma(\gamma, \kappa, \kappa')$$
$$\wedge\, \sigma \models y = \bot \wedge g = \gamma \wedge h = \gamma \wedge \gamma : \mathtt{gate}$$
$$\wedge\, \sigma \models \mathtt{inth}(\kappa, \tau) \wedge \mathtt{inth}(\kappa', \tau')$$

Opening and closing The next two rules define the semantics of operations `Open` and `Close`.

$$[\textsc{Open}]\ \frac{\tau\langle\{\mathtt{Open}\ k\ g\}\ \ T\rangle \ \Big\|\ \tau T}{\sigma \wedge \kappa : \mathtt{kell}(\pi, w) \ \Big\|\ \sigma \wedge \kappa : \mathtt{kell}(\pi \cup \mathtt{grant}_\sigma(k, g), w)} \quad \text{if } \mathtt{grant}_\sigma(k, g) \neq \emptyset \wedge \sigma \models \mathtt{inth}(\kappa, \tau)$$

$$[\textsc{Close}]\quad \frac{\tau\langle\{\mathtt{Close}\ k\ g\}\ \ T\rangle \ \Big\|\ \tau T}{\sigma \wedge \kappa : \mathtt{kell}(\pi, w) \ \Big\|\ \sigma \wedge \kappa : \mathtt{kell}(\pi \setminus \mathtt{grant}_\sigma(k, g), w)} \quad \text{if } \mathtt{grant}_\sigma(k, g) \neq \emptyset \wedge \sigma \models \mathtt{inth}(\kappa, \tau)$$

Packed values The operation `Mark` allows to modify a packed value, by replacing a gate, a procedure, or the top level kell in the packed value, with an existing gate, an existing procedure, or an existing kell, respectively. We only present below the rule concerning the replacement of gates. The rules concerning the replacement of the top level kell or a procedure in the packed value are similar.

$$[\textsc{MarkG}]\ \frac{\tau\langle\{\mathtt{Mark}\ \ z\ \ \mathtt{gate}(x\ \ y)\ \ p\}\ \ T\rangle \ \Big\|\ \tau T}{\sigma \ \Big\|\ \sigma \wedge \sigma'} \ \text{if } C$$

$$C \equiv \sigma \models \phi \wedge p = \bot \wedge \sigma'' \models \gamma : \mathtt{gate} \qquad \theta = \{\gamma \to \gamma'\}$$
$$\phi \equiv z = \mathtt{pack}(\omega, \mathcal{T}, \sigma'', \mu) \wedge x = \gamma \wedge \gamma : \mathtt{gate} \wedge y = \gamma' \wedge \gamma' : \mathtt{gate}$$
$$\sigma' \equiv p = \mathtt{pack}(\omega, \mathcal{T}\theta, \sigma''\theta, \mu \cup \{\gamma'\})$$

Packing The rule for packing is given below. Notice that packing implies passivating the target kell, together with all of its subkells. Packing produces a *packed value*, which encapsulates the part of the current execution structure corresponding to the target kell. The set of marks of the resulting packed value is initially empty.

$$[\textsc{Pack}]\ \frac{\tau\langle\{\mathtt{Pack}\ x\ y\}\ \ T\rangle \quad \tau_1 T_1 \ldots \tau_n T_n \ \Big\|\ \tau T}{\sigma \ \Big\|\ \sigma \wedge \sigma'} \ \text{if } C$$

$$C \equiv \mathtt{subth}_\sigma(\kappa) = \{\tau_1, \ldots, \tau_n\} \wedge \mathtt{subk}_\sigma(\kappa) = \{\kappa_1, \ldots, \kappa_m\}$$
$$\wedge\sigma \models \bigwedge_{i=1}^{m} \phi_i \wedge y = \bot \wedge \mathtt{inth}(\kappa', \tau) \wedge \kappa' : \mathtt{kell}(\pi', z') \wedge z' = \bot \wedge \phi$$
$$\phi \equiv x = \kappa \wedge \kappa : \mathtt{kell}(\pi, z) \wedge z = \bot \wedge \mathtt{in}(\kappa', \kappa)$$
$$\phi_i \equiv \kappa_i : \mathtt{kell}(\pi_i, w_i) \wedge w_i = \bot$$
$$\sigma' \equiv \bigwedge_{i=1}^{m} w_i = \mathtt{packed} \wedge y = \mathtt{pack}(\kappa, \mathcal{T}, \sigma, \emptyset) \wedge z = \mathtt{packed}$$
$$\mathcal{T} \equiv \tau_1 T_1 \ \ldots \ \tau_n T_n$$

The rule for unpacking is given below. Unpacking creates an execution structure which is similar to the packed one, except all the variables and all the non-marked names in the packed structure are renamed (substitution θ below) to avoid any potential conflict between the current store, σ, and the packed one, σ'. In addition, unpacking returns a tuple whose elements are pairs of the form $(\xi, \theta(\xi))$, where ξ is a name in the packed store σ' and $\theta(\xi)$ is the corresponding new name after unpacking. Condition C implies a renaming of the top-level kell so that it becomes that of the current kell. Care must be taken if the top kell in the packed value has been previously marked: in this case, the top-level kell in the packed value must match the current kell. Finally, $\mathtt{tkn}_{\sigma'}(\mathcal{T})$ denotes all the children kells of kell κ' in task $\mathcal{T}$.

$$[\textsc{Unpack}]\ \frac{\tau\langle\{\mathtt{Unpack}\ y\ x\}\ \ T\rangle \ \Big\|\ \tau T \quad \mathcal{T}\theta\theta'}{\sigma \ \Big\|\ \sigma \wedge \sigma''} \ \text{if } C$$

$$C \equiv \sigma \models \kappa : \mathtt{kell}(\pi, z) \wedge x = \bot \wedge y = \mathtt{pack}(\kappa', \mathcal{T}, \sigma', \mu) \wedge \mathtt{inth}(\kappa, \tau)$$
$$\wedge\, (\sigma' \equiv \sigma'' \wedge \kappa' : \mathtt{kell}(\pi', z')) \wedge (\kappa' = \kappa \implies \pi = \pi' \wedge z = z')$$
$$\wedge\, (\mathtt{dom}(\theta) = \mathtt{dom}(\sigma') \setminus \mu) \wedge \forall l \in \mathtt{ran}(\theta), l \notin \mathtt{dom}(\sigma \wedge \sigma')$$
$$\wedge\, \theta' = \{\kappa' \to \kappa\}$$
$$\sigma'' \equiv \bigwedge_{\eta \in \mathtt{tkn}_{\sigma'}(\mathcal{T})} \mathtt{in}(\kappa, \eta\theta) \wedge x = \mathtt{n}((\xi_1\ \theta(\xi_1)) \ldots (\xi_n\ \theta(\xi_n))) \wedge \sigma''\theta\theta'$$

5.2 OZ/K properties

We give here two properties of the OZ/K operational semantics. We let $\to^*$ denote the reflexive and transitive closure of the reduction relation $\to$. We say that an execution structure $(\sigma, \mathcal{T})$ results from the execution of an OZ/K statement, if there exists a OZ/K statement S such that $(\sigma_0, \tau\langle S\ \ \langle\rangle\rangle) \to^* (\sigma, \mathcal{T})$, where $\sigma_0 \equiv \tau : \mathtt{thread}(w) \wedge \mathtt{inth}(\top, \tau)$, and $\top$ is the name of the top-level kell (i.e. the root of the kell tree). We say that a set of threads $\mathcal{T}$ belongs to a kell κ if for all names τ of threads in $\mathcal{T}$, we either have $\mathtt{inth}(\kappa, \tau)$ or $\mathtt{inth}(\kappa', \tau)$, where κ' is a descendant kell of the kell κ. We note $\mathtt{v}(\mathcal{T}, \sigma)$ the set of variables in store σ that are reachable by $\mathcal{T}$.

The first proposition establishes the separation property for OZ/K computation. It asserts that two distinct kells in an execution structure cannot hold references to the same unbound variable (either directly, or indirectly, through cells, procedures, etc).

PROPOSITION 1. *Assume $(\mathcal{T}, \sigma)$, with $\mathcal{T} \equiv \mathcal{T}_1\ \mathcal{T}_2\ \mathcal{T}'$, is an execution structure that result from the execution of an* OZ/K *statement, where $\mathcal{T}_1$ belongs to kell κ_1, $\mathcal{T}_2$ belongs to kell κ_2, and $\kappa_1 \neq \kappa_2$. If $\sigma = \sigma' \wedge x = \bot$, for some σ', and $x \in \mathtt{v}(\mathcal{T}_1, \sigma)$, then $x \notin \mathtt{v}(\mathcal{T}_2, \sigma)$.*

The second proposition asserts a form of *perfect firewall* property for OZ/K, namely, that there exists an execution structure where a task can be completely isolated from the rest of the other tasks in the execution structure. Let $(\mathcal{T}, \sigma)$ be an execution structure. We say that κ *appears at the top level* in $(\mathcal{T}, \sigma)$, if $\sigma = \sigma' \wedge \mathtt{in}(\top, \kappa)$, for some σ'. We also say that κ is *not referenced in* $\mathcal{T}$ if there exists no variable x such that $x \in \mathtt{v}(\mathcal{T}, \sigma)$ and $\sigma = \sigma' \wedge x = \kappa$, for some σ'.

PROPOSITION 2. *Let $(\mathcal{T}\ \ \mathcal{T}_\kappa, \sigma)$ be an execution structure that results from the execution of an* OZ/K *statement, where: κ appears at the top level; $\mathcal{T}_\kappa$ is the set of all threads that belong to κ; κ is not referenced in $\mathcal{T}$; there is no thread named τ such that $\sigma \models \mathtt{inth}(\kappa, \tau)$; and $\sigma = \sigma_0 \wedge \kappa : \mathtt{kell}(\emptyset, w)$, for some σ_0, w. The reductions possible from $\langle \sigma, \mathcal{T}\ \ \mathcal{T}_\eta \rangle$ can only be of one of the following two forms:*

$$\frac{\mathcal{T}\ \ \mathcal{T}_\kappa \ \Big\|\ \mathcal{T}'\ \ \mathcal{T}_\kappa}{\sigma \ \Big\|\ \sigma'} \ \textit{or} \ \frac{\mathcal{T}\ \ \mathcal{T}_\kappa \ \Big\|\ \mathcal{T}\ \ \mathcal{T}'_\kappa}{\sigma \ \Big\|\ \sigma'}$$

where $\mathcal{T}'_\kappa$ is the set of threads that belong to κ in execution structure $(\mathcal{T}\ \ \mathcal{T}'_\kappa, \sigma')$, and σ' is such that there is no τ such that $\sigma' \models \mathtt{inth}(\kappa, \tau)$, and $\sigma' = \sigma'_0 \wedge \kappa : \mathtt{kell}(\emptyset, w)$, for some σ'_0.

Informally, if we denote by $\kappa[\mathcal{T}]$ a task $\mathcal{T}$ whose threads belong to κ, the proposition asserts that a kell structure of the form $\kappa[\kappa_1[\mathcal{T}_1] \ldots \kappa_n[\mathcal{T}_n]]$ at the top level, where κ is not referenced outside of $\kappa[\ldots]$ (and thus cannot be packed), constitutes a perfect firewall for the tasks $\mathcal{T}_1, \ldots, \mathcal{T}_n$. This can be understood intuitively since there is no thread in kell κ (condition *there is no thread named* τ *such that* $\sigma \models \texttt{inth}(\kappa, \tau)$) that can act as a relay of communication between threads in $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and the outer environment, and since there is no gate opened in κ for such communication (condition $\sigma = \sigma_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$).

6. Related work

OZ/K is related to work in several areas: programming languages, architecture description languages, process calculi with localities, and component-based programming models. Here we consider only related work on programming languages. A more detailed analysis of related work is available in the full paper [**?**].

The reference language for open programming is the Java language [4], with its comprehensive environment. However, Java still suffers from important limitations in relation to open programming: limited form of modules and marshalling, no serialization of code, no generic pickling mechanism, dynamic linking and sandboxing available through complex APIs with no formal semantics.

A few programming languages are built around a notion of locality, notably JoCaml [13], Nomadic Pict [35], O'Klaim [5], ULM [7]. None of these languages provide the ability to build sandboxes with strong isolation properties as OZ/K provides. Except for JoCaml (which supports hierarchical localities and strong mobility), localities in these languages essentially represent execution sites.

ArchJava [1] and Classages [20] consider a notion of component close to that of OZ/K, and which allows an explicit control other component communications (through notions of connectors). However, in contrast to OZ/K, ArchJava or Classages components are not units of fault isolation, and rely on the underlying class loading Java mechanisms for handling code modules and shipping code. As a result, AchJava and Classages do not properly support strong mobility, and they do not provide the ability to build proper sandboxes, as OZ/K does.

Alice [23] and Acute [29], target open programming, with its whole range of issues. Alice can be understood as an extension of Standard ML [22] that offers higher-order modules, packages (essentially, an extension of the notion of dynamics, which combines a higher-module with its dynamic signature), pickles (marshalled forms of packages), components, and concurrency with futures and laziness. The Alice notion of component (or dynamic module) can be understood, following [23] as a function, taking packages as arguments (imports), and that evaluates to a package (containing the export module). The Alice notions of packages, pickles and components, formalize, in a strongly typed setting, similar notions of notions of functors and pickles that appear in OZ. Still, compared to OZ/K, Alice does not provide support for passivation, and support for isolation and sandboxing in Alice, is as limited as in OZ.

Acute is also a language in the ML family, with extensive support for open distributed programming in a strongly typed setting, including explicit marshalling, dynamic linking, dynamic modules, support for versioning constraints, support for concurrency through threads, and even a form of passivation through the ability to thunkify running threads. Acute also introduces the notion of *mark* to control the extent of dynamic linking in module. Compared to OZ/K, Acute does not provide support for programmable sandboxing and isolation, and it supports open programming through a relatively complex set of mechanisms that are subsumed in OZ/K by a smaller set of constructs (namely via *kells*, *gates*, and *packing*).

The Sing# language [12], developed as part of the Microsoft Singularity operating system, that extends the C# programming language with isolated processes and asynchronous message passing communication. Processes in Sing# are isolated by virtue of their code being unmodifiable at run-time, and by communicating only through message passing with other processes. However, Sing# does not provide the sandboxing and capabilities of OZ/K, nor its dynamic update capabilities.

Our work on OZ/K is also related to work on OZ. The notion of locality can be seen as a generalization of the notion of computation space in Oz, introduced by C. Schulte to program constraint services. An attempt at exploiting a locality concept inspired by the Kell calculus was made in [17]. In this paper, localities (named "membranes") are finer grained than kells in OZ/K, but they are used only for communication control (confinement), and do not constitute units of failure isolation, or of passivation. The kell construct in OZ/K seems in line with the proposed design guidelines for a *secure* OZ, presented in [32].

Like OZ, OZ/K is an essentially untyped language. This is in contrast to most of the languages cited above, which are statically typed (with forms of *dynamics* for some, such as Acute and Alice). Static type checking has well-known advantages, however an untyped setting provides more flexibility when trying to combine different forms of programming, as OZ and OZ/K attempt to do. Actually, OZ/K demonstrates that it is possible to achieve strong isolation and proper sandboxing of untrusted components without relying on static type checking.

7. Conclusion

We have presented OZ/K, a kernel language for open distributed programming, that extends the OZ computation model with localities and passivation, and its formal operational semantics. Introducing localities and passivation in OZ brings several benefits, chief among them is the ability to interpret, with a compact set of constructs, related notions such as modules, pickles, components, while providing strong sandboxing capabilities.

Much work remains ahead, however. First, one could improve the connection between loclaity constructs and OZ constructs, notably sharing logical variables between localities. How this can be done without sacrificing isolation remains to be seen. Apart from developing an efficient implementation, one of our ongoing work is to define higher-level abstractions for component-based programming, typically along the lines of object-oriented programming support in OZ (abstractions defined as syntactic sugar on the kernel language). The full paper [**?**] provide some indication by proposing an interpretation in OZ/K of the FRACTALreflective component model [8]. Another issue concerns support for dynamic reconfiguration. The passivation construct in OZ/K only constitutes a first basis, which needs to be completed with capabilities for analyzing and manipulating packed values. This should involve introspection capabilities for packed values, some form of multistage programming, and probably the development of an updateability analysis, e.g. along the lines of [33].

Acknowledgments This research has been partially funded by the European Commission FP6 IST Project Selfman, and FP6 NoE CoreGrid. The paper has benefited from useful comments from anonymous reviewers and Peter Van Roy, Yves Jaradin, Raphaël Collet, Per Brand and Seif Haridi.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In Springer, editor, *Proceedings 16th ECOOP*, volume 2374 of *LNCS*, 2002.

[2] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Stockholm, Sweden, 2003.

[3] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 4th edition*. Addison-Wesley, 2005.

[5] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In *Global Computing*, number 2874 in LNCS. Springer-Verlag, 2003.

[6] P. Bidinger, A. Schmitt, and J.-B. Stefani. An abstract machine for the kell calculus. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS), 7th IFIP WG 6.1 International Conference*, volume 3535 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] G. Boudol. Ulm: a core programming model for global computing. In *13th ESOP*, volume 2986 of *LNCS*, 2004.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[9] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

[10] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1), 2005.

[11] R. Collet and P. Van Roy. Failure Handling in a Network-Transparent Distributed Programming Language. In *Recent Advances in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *1st EuroSys Conference*. ACM, 2006.

[13] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: a language for concurrent, distributed and mobile programming. In *Summer Schol Adv. Functional Programming*, volume 2638 of *LNCS*, 2003.

[14] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Trans. Program. Lang. Syst.*, 21(3), 1999.

[15] M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: a language for controlling mobile code. *Acta Informatica*, 42(4-5), 2005.

[16] D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt, and J.B. Stefani. Component-oriented programming with sharing: Containment is not ownership. In *4th GPCE*, volume 3676 of *LNCS*. Springer, 2005.

[17] Y. Jaradin, F. Spiessens, and P. Van Roy. Capability Confinement by Membranes. Technical Report Research Report RR2005-03, Dep. of Comp. Science and Eng., UniversitéCatholique de Louvain, Belgium, 2005.

[18] E. Lee. The problem with threads. *IEEE Computer*, 39(5), 2006.

[19] M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/K: A Kernel Language for Component-Based Distributed Programming , 2007. http://sardes.inrialpes.fr/papers/files/rr-ozk.pdf.

[20] Y.D. Liu and S. Smith. Interaction-Based Programming with Classages. In *Proceedings OOPSLA*, 2005.

[21] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, baltimor, Maryland, USA, 2006.

[22] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[23] A. Rossberg. The Missing Link – Dynamic Components for ML. In *Int. Conf. Functional Programming (ICFP)*, 2006.

[24] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*. Intellect, 2006.

[25] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Trans. Program. Lang. Syst.*, 19(5), 1997.

[26] 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.

[27] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*. Springer, 2005.

[28] P. Sewell, J. Leifer, K. Wansbrough, M. Allen-Willians, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation – Design rationale and language definition. Technical Report RR-5329, INRIA, 2004.

[29] P. Sewell, J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Willians, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Int. Conf. Functional Programming*, 2005.

[30] P. Sewell, J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Willians, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *to appear Journal of Functional Programming*, 2006.

[31] P. Sewell and J. Vitek. Secure Composition of Untrusted Code: Box pi, Wrappers, and Causality. *Journal of Computer Security*, 11(2), 2003.

[32] F. Spiessens and P. Van Roy. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *2nd Int. Conf. Multiparadigm Programming in Mozart/Oz,*, volume 3389 of *LNCS*. Springer, 2005.

[33] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *32nd POPL*. ACM, 2005.

[34] P. van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, 2004.

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

A. Appendix

The predicate $\texttt{access}_\sigma$ is defined by cases as follows. The first case is when κ and κ' are the same. In this case, communication is always possible on any gate, hence we have $\texttt{access}_\sigma(\gamma, \kappa, \kappa)$, for all γ, κ. The second case is when κ is a child-kell of κ' or conversely, i.e. $\sigma \models \texttt{in}(\kappa, \kappa')$ or $\sigma \models \texttt{in}(\kappa', \kappa)$. In this case we have $\texttt{access}_\sigma(\gamma, \kappa, \kappa')$, and $\texttt{access}_\sigma(\gamma, \kappa', \kappa)$. The last case is when κ and κ' are different and are not an immediate child of one another. In this case, let $\kappa_0, \kappa_1, \ldots, \kappa_n, \kappa_{n+1}$, $n \geq 1$, be a sequence of kells, such that $\kappa_0 = \kappa$ and $\kappa_{n+1} = \kappa'$, and $\sigma \models \texttt{in}(\kappa_i, \kappa_{i+1})$ or $\sigma \models \texttt{in}(\kappa_{i+1}, \kappa_i)$ for all $i \in \{0, \ldots, n\}$ (i.e. $\kappa_0, \ldots, \kappa_{n+1}$ is a path from κ to κ' in the kell tree, where we assume a top-level kell exists). Let $\pi_0, \ldots, \pi_{n+1}$ be such that $\sigma \models \kappa_i : \texttt{kell}(\pi_i, x_i)$. We define $\texttt{access}_\sigma(\gamma, \kappa, \kappa')$ by:

$$\texttt{access}_\sigma(\gamma, \kappa, \kappa') \triangleq \bigwedge_{i=0}^{n} \texttt{auth}_\sigma(\gamma, \kappa_i, \kappa_{i+1})$$

$$\texttt{auth}_\sigma(\gamma, \kappa_i, \kappa_{i+1}) \triangleq \kappa_i \cdot \gamma \in \pi_{i+1} \qquad \text{if } \sigma \models \texttt{in}(\kappa_{i+1}, \kappa_i)$$

$$\texttt{auth}_\sigma(\gamma, \kappa_i, \kappa_{i+1}) \triangleq \kappa_{i+1} \cdot \gamma \in \pi_i \qquad \text{if } \sigma \models \texttt{in}(\kappa_i, \kappa_{i+1})$$

If the store σ is of the form $\sigma' \wedge \sigma''$, where σ'' is of the form

$$(\bigwedge_{i \in I} \texttt{inth}(\kappa, \tau_i) \wedge \tau_i : \texttt{thread}(x_i)) \wedge (\bigwedge_{j \in J} \texttt{in}(\kappa, \kappa_j) \wedge \kappa_j : \texttt{kell}(\pi_j, x_j))$$

and where σ' has no occurrence of predicates of the form $\texttt{in}(\kappa, \_)$ or $\texttt{inth}(\kappa, \_)$, then we can define inductively the function $\texttt{subth}_\sigma$ as:

$$\texttt{subth}_\sigma(\kappa) = \bigcup_{j \in J} \texttt{subth}_\sigma(\kappa_j) \cup \{\tau_i \mid i \in I\}$$

The function $\texttt{grant}_\sigma$ gives the set of pairs of names corresponding to a pair (k, g) denoting a kell and a gate. For lack of space, we do not give the full specification of the function here, but the simplest case is given below:

$$\texttt{grant}_\sigma(k, g) \triangleq \kappa \cdot \gamma \quad \text{if} \quad \sigma \models k = \kappa \wedge \kappa : \texttt{kell}(\ldots) \wedge g = \gamma \wedge \gamma : \texttt{gate}$$