An Object Group-Based Component Model*

Michaël Lienhardt, Mario Bravetti, and Davide Sangiorgi

Focus Team, University of Bologna, Italy
{lienhard,bravetti,davide.sangiorgi}@cs.unibo.it

Abstract. Dynamic reconfiguration, i.e. changing at runtime the communication pattern of a program is challenging for most programs as it is generally impossible to ensure that such modifications won't disrupt current computations. In this paper, we propose a new approach for the integration of components in an object-oriented language that allows *safe* dynamic reconfiguration. Our approach is built upon *futures* and *object-groups* to which we add: i) output ports to represent variability points, ii) critical sections to control when updates of the software can be made and iii) hierarchy to model locations and distribution. These different notions work together to allow dynamic and safe update of a system. We illustrate our approach with a few examples.

1 Introduction

Components are an intuitive tool to achieve unplanned dynamic reconfigurations. In a component system, an application is structured into several distinct pieces called *components*. Each of these components has dependencies towards functionalities located in other components; such dependencies are collected into *output ports*. The component itself, however, offers functionalities to the other components, and these are collected into *input ports*. Communication from an output port to an input port is possible when a *binding* between the two ports exists. Dynamic reconfiguration in such a system is then achieved by adding and removing components, and by replacing bindings. Thus updates or modifications of parts of an application are possible without stopping it.

Related Work. While the idea of components is simple, bringing it into a concrete programming language is not easy. The informal description of components talks about the structure of a system, and how this structure can change at runtime, but does not mention program execution. As a matter of fact, many implementations of components [1, 3, 5, 15, 2, 11, 13] do not merge into one coherent model i) the execution of the program, generally implemented using a classic object-oriented language like Java or C++, and ii) the component structure, generally described in an annex Architecture Description Language (ADL). This approach makes it simple to add components to an existing standard program. However, unplanned dynamic reconfigurations become hard, as

* Partly funded by the EU project FP7-231620 HATS.

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part I, LNCS 7609, pp. 64–78, 2012.

it is difficult to express modifications of the component structure using objects (since these are rather supposed to describe the execution of the programs). For instance, models like Click [13] do not allow runtime modifications while OSGi [1] allows addition of new classes and objects, but no component deletions or binding modifications. In this respect, a more flexible model is Fractal [3], which reifies components and ports into objects. Using an API, in Fractal it is possible to modify bindings at runtime and to add new components; Fractal is however rather complex, and it is informally presented, without a well-defined model.

Formal approaches to component models have been studied e.g., [4, 8, 14, 12, 10, 9]. These models have the advantage of having a precise semantics, which clearly defines what is a component, a port and a binding (when such a construct is included). This helps understanding how dynamic reconfigurations can be implemented and how they interact with the normal execution of the program. In particular, Oz/K [10] and COMP [9] propose a way to integrate in a unified model both components and objects. However, Oz/K has a complex communication pattern, and deals with adaptation via the use of *passivation*, which, as commented in [7], is a tricky operator — in the current state of the art it breaks most techniques for behavioral analysis. In contrast, COMP offers support for dynamic reconfiguration, but its integration into objects appears complex.

Our Approach. Most component models have a notion of component that is distinct from the objects used to represent the data and the main execution of the software. The resulting language is thus structured in two different layers, one using objects for the main execution of the program, one using components for the dynamic reconfiguration. Even though such separation seems natural, it makes difficult the integration of the different requests for reconfiguration into the program's workflow. In contrast, in our approach we tried to have a uniform description of objects and components. In particular, we aim at adding components on top of the *Abstract Behavioral Specification* (ABS) language [6], developed within the EU project HATS. Core ingredients of ABS are objects, futures and object groups to control concurrency. Our goal is to enhance objects and object groups with the basic elements of components (ports, bindings, consistency and hierarchy) and hence enable dynamic reconfigurations.

We try to achieve this by exploiting the similarities between objects and object groups with components. Most importantly, the methods of an object closely resemble the input ports of a component. In contrast, objects do not have explicit output ports. The dependencies of an object can be stored in internal fields, thus rebinding an output port corresponds to the assignment of a new value to the field. Objects, however, lack mechanisms for ensuring the consistency of the rebinding. Indeed, suppose we wished to treat certain object fields as output ports: we could add methods to the object for their rebinding; but it would be difficult, in presence of concurrency, to ensure that a call to one of these methods does not harm ongoing computations. For instance, if we need to update a field (like the driver of a printer), then we would want to wait first that all current execution using that field (like some printing jobs) to finish first. This way we ensure that the update will not break those computations.

In Java, such consistency can be achieved using the *synchronized* keyword, but this solution is very costly as it forbids the interleaving of parallel executions, thus impairing the efficiency of the program. In ABS, object groups offer a mechanism for consistency, by ensuring that there is at most one task running in an object group. This does ensure some consistency, but is insufficient in situations involving several method calls. A further difference between objects and components is that only the latter talks about *locations*. Locations structure a system, possibly hierarchically, and can be used to express dynamic addition or removal of code, as well as distribution of a program over several computers.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS core language:

1. A notion of output port distinct from the object's fields. The former (identified with the keyword **port**) corresponds to the objects' dependencies and can be modified only when the object is in a *safe* state, while the latter corresponds to the inner state of the objects and can be modified with the ordinary assignments.
2. The possibility of annotating methods with the keyword **critical**: this specifies that the object, while an instance of the method is executing, is not in a safe state.
3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.
4. A hierarchy of locations. Thus an ABS program is structured into a tree of locations that can contain object groups, and that can move within the hierarchy. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location being a different computer) and code mobility (by moving a sub-location from a computer to another one).

The resulting language remains close to the underlying ABS language. Indeed, the language is a conservative extension of ABS (i.e., an ABS program is a valid program in our language and its semantics is unchanged), and, as shown in our following example, introducing the new primitives into an ABS program is simple. In contrast with previous component models, our language does not drastically separate objects and components. Three major features of the informal notion of component — ports, consistency, and location — are incorporated into the language as follows: (i) output ports are taken care of at the level of our enhanced objects; (ii) consistency is taken care of at the level of object groups; (iii) the information about locations is added separately.

We believe that the separation between output ports and fields is meaningful for various reasons:

- Output ports represent dependencies of an object towards its environment (functionalities needed by the object and implemented outside it, and that moreover might change during the object life time). As such they are logically different from the internal state of the object (values that the object may have to consult to perform its expected computation).

$$
\begin{array}{lcl@{\qquad}lcl}
P & ::= & I\ P \mid C\ P \mid \{\ \overline{T\ x};\ s\ \} & F & ::= & T\ x \\
T & ::= & \texttt{I} \mid \texttt{Fut}\langle T\rangle & S & ::= & T\ \texttt{m}(\overline{T\ x}) \\
I & ::= & \textbf{interface}\ \texttt{I}\ \{\ \overline{S}\ \} & M & ::= & S\{\ \overline{T\ x};\ s\ \} \\
C & ::= & \textbf{class}\ \texttt{C}(\overline{T\ x})\ [\textbf{implements}\ \overline{\texttt{I}}]\ \{\ \overline{F}\ \overline{M}\ \} & & & \\
s & ::= & \textbf{skip} \mid s;s \mid e \mid x = e \mid \textbf{await}(g) \mid \textbf{if}\ e\ \{\ s\ \}\ \textbf{else}\ \{\ s\ \} & & & \\
 & \mid & \textbf{while}\ e\ \{\ s\ \} \mid \textbf{return}\ e & & & \\
e & ::= & v \mid x \mid \textbf{this} \mid \textbf{new}\ [\textbf{cog}]\ \texttt{C}\ (\overline{e}) \mid e.\texttt{m}(\overline{e}) \mid e!\texttt{m}(\overline{e}) \mid \textbf{get}(e) & & & \\
v & ::= & \textbf{null} \mid \texttt{true} \mid \texttt{false} \mid 1 \mid \ldots & & & \\
g & ::= & e \mid e? \mid g \wedge g & & &
\end{array}
$$

Fig. 1. Core ABS Language

- The separation of output ports allows us to have special constructs for them. Examples are the constructs for consistency mentioned above. Moreover, different policies may be used for updating fields and output ports. For instance, in our model while a field of an object `o` may be updated only by *o*, an output port of `o` may be modified by objects in the same group as `o`. This difference of policy is motivated in Section 3.1
- The separation of output ports could be profitable in reasoning, in particular in techniques of static analysis.
- The presence of output ports may be useful in the deployment phase of a system facilitating, for instance, the connection to local communication resources.

Roadmap. §2 describes the core ABS language. §3 presents our extension to the ABS language. §4 presents the semantics of the language. The main features of core ABS and our extensions are illustrated along the document with several examples.

2 Core ABS

We present in Figure 1 the object core of the ABS language. For the full description of the language, including its *functional* aspect, see [6]. We assume an overlined element to be any finite sequence of such element. A program P is defined as a set of interface and class declarations I and C, with a main function $\{\ \overline{T\ x};\ s\ \}$. The production T types objects with interface names `I` and futures with future types $\texttt{Fut}\langle T\rangle$, where T is the type of the value returned by an asynchronous method call of the kind $e!\texttt{m}(\overline{e})$ (versus $e.\texttt{m}(\overline{e})$ representing synchronous calls): the actual value of a future variable can be read with a **get**. An interface I has a name `I` and a body declaring a set of method headers S. A class C has a name `C`, may implement several interfaces, and declares in its body its fields with F and its methods with M. In the following examples: for simplicity we will omit "?" in await guards (in ABS "e?" guards are used for expressions "e" returning a future, instead simple "e" guards are used for boolean expressions) and we will follow the ABS practice to declare the class constructor like a method, named *init*.

```
class Printer {
 Status s;

 int print(File f) {
  int id = s.addToQueue(f);
  await(s!isCurrent(id));

  int code = this.printPhy(f);
  await(s!popFromQueue(id));
  return code;
 }

  int printPhy(File f) {...}
  Status getStatus() { return s; }
 }

 class Status {
  ...
  int addToQueue(File f) {...}
  void popFromQueue(int id){...}
  void isCurrent(int id) {...}
  void isCurrentFile(File f) {...}
 }
```

Fig. 2. Example, the class Printer

Object Groups and Futures. One of the main features of ABS is its concurrency model which aims to solve data races. Objects in ABS are structured into different groups called *cogs* which are created with the `new cog` command. These cogs define the concurrency of a program in two ways: i) inside one cog, at most one object can be active (i.e. execute a method); ii) all cogs are concurrent (i.e. all method calls between objects of different cogs must be *asynchronous*). Concurrency inside a cog is achieved with cooperative multitasking using the `await` statement, and synchronization between concurrent executions is achieved with the `await` and `get` statements, based on futures.

We illustrate this concurrency model with a simple class `Printer` in Figure 2, modeling a printer driver with a job queue stored in a `Status s`. The principle of the `print` method of `Printer` is as follow: i) the printing request is added to the queue of jobs, which returns the identifier for that new job; ii) the method *waits* until all previous jobs have been processed; iii) the method does the actual printing (using the method `printPhy`) and waits for its completion, which returns a code describing if the printing was successful or not; and iv) the job is removed from the queue and the code is returned to the user.

3 Component Model

3.1 Ports and Bindings

The ABS concurrency model as it is cannot properly deal with runtime modifications of a system, in particular with unplanned modifications. Let us consider the client presented in Figure 3. This class offers a little abstraction over the `Printer` class with three extra features: i) the possibility to change printer; ii) some notification messages giving the current status of the printing job (`count` being the identifier of the job); and iii) the possibility to get the number of jobs handled by this object.

```
class PrintClient {
 Printer p;
 int count;

 void setPrinter(Printer pr) { p = pr }

 void print(File f) {
  Fut<int> err = p!print(f);
  count = count + 1;

  System.out.println("Job " + count + ": Waiting to begin");
  await ((get(p!getStatus())!isCurrentFile(f));
  System.out.println("Job " + count + ": Being processed");
  await err;
  System.out.println("Job " + count
          + ": Completed with error code = " + (get(err)));
 }

 int GetNumberOfJobs() { return count; }

 void init() { count = 0; }
}
```

Fig. 3. An evolved Printing Client

This class is actually erroneous: let us consider the scenario where a printing job is requested, followed by the modification of the printer. The `print` method sends the job to the first printer p_1, then waits for the notification from p_1's status. While waiting, the printer gets modified into p_2: the following requests will fail as they will be directed to p_2 and not p_1. A possible solution would be to forbid the interleaving of different methods execution by replacing the `await`s by `get`s, which corresponds to the *synchronized* in Java.

We overcome this inconsistency problem by forbidding the modification of the field `p` while it is in use. For this, we combine the notions of *output port* (from components) and of *critical section*. Basically the field `p`, which references an external service that can change at runtime, is an *output port*; the `print` method that needs stability over this port, creates a critical section to avoid the modification of `p` while it is executing; the `count` field and the `GetNumberOfJobs` method, that have no link to an external service, remain unchanged.

The syntax for our manipulation of output port and critical section is as follows.

$$
\begin{array}{rcll}
F & ::= & \ldots & \mid \textbf{port}\ T\ \texttt{f} \\
S & ::= & \ldots & \mid \textbf{critical}\ T\ \texttt{m}(\overline{T\ x}) \\
s & ::= & \ldots & \mid \textbf{rebind}\ e.x = e \\
g & ::= & \ldots & \mid \|e\|
\end{array}
$$

Here, a field can be annotated with the keyword **port**, which makes it an output port, supposedly connected to an external service that can be modified at

```
class PrintClient {
 port Printer p;
 int count;

 void setPrinter(Printer pr) {
  await (‖this‖);
  rebind p = pr
 }

 critical void print(File f) { ... }

 int GetNumberOfJobs() { return count; }

 void init() { count = 0 }
}
```

Fig. 4. An improved Printing Client

runtime. Moreover, methods can be annotated with the keyword **critical**, which ensures that, during the execution of that method, the output ports of the object will not be modified.

Output ports differ from ordinary fields in two aspects:

1. output ports cannot be freely modified. Instead one has to use the **rebind** statement that checks if the object has an open critical section before changing the value stored in the port. If there are no open critical sections, the modification is applied; otherwise an error in a form of a dead-lock is raised;
2. output ports of an object o can be modified (using the **rebind** statement) by *any* object in the same object-group of o. This capacity is not in opposition to the classic object-oriented design of not showing the inner implementation of an object: indeed, a port does not correspond to an inner implementation but exposes the relationship the object has with independent services. Moreover, this capacity helps achieving consistency as shown in the next examples.

Finally, to avoid errors while modifying an output port, one should first ensure that the object has no open critical sections. This is done using the new guard $\|e\|$ that waits for the object e not to be in critical section. Basically, if an object o wants to modify output ports stored in different objects o_i, it first waits for them to close all their critical section, and then can apply the modifications using `rebind`.

3.1.1 Examples

Printing Client. In Figure 4 we show how to solve our previous example (from Figure 3). The changes are simple: i) we specify that the field `p` is a port; ii) we annotate the method `print` with **critical** (to protect its usage of the port `p`); and iii) we change the method `setPrinter` that now waits for the object to be in a consistent state *before* rebinding its output port `p`.

```
class OperatorFrontEnd {
 port Operator _op;

 critical Document modify(Document doc) { ... }

 void init(Operator op) { rebind _op = op; }
}

class WFController {
 port Document _doc;
 port Printer _p;
 OperatorFrontEnd _opfe;

 critical void newInstanceWF() { ... }

 void changeOperator(Operator op) {
  await(‖this‖ ∧ ‖_opfe‖);
  rebind _opfe._op = op;
 }

 void init(Document doc, Operator op, Printer p) {
  rebind _doc = doc;
  rebind _p = p;
  _opfe = new OperatorFrontEnd(op);
 }
}
```

Fig. 5. Dynamic Reconfiguration Example

Workflow Controller. For the purpose of this example, we suppose we want to define a workflow that takes a document (modeled by an instance of the class `Document`), modifies it using an `Operator` and then sends it to a `Printer`. We suppose that the protocol used by `Operator` objects is complex, so we isolate it into a dedicated class. Finally, we want to be able to change protocol at runtime, without disrupting the execution of previous instances of the workflow. Such a workflow is presented in Figure 5.

We thus have two classes: the class `OperatorFrontEnd` implements the protocol in the method `modify`; the class `WFController` encodes the workflow. The elements `_op`, `_doc` and `_p` are *ports*, and correspond to dependencies to external resources. In consequence they are annotated as **port**. It is only possible to modify their value using the construct **rebind**, which checks if the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods `modify` and `newInstanceWF` make use of these ports in their code, and are thus annotated as **critical** as it would be dangerous to rebind ports during their execution.

The key operations of our component model are shown in the two lines of code describing the method `changeOperator`. First is the **await** statement, which waits for the objects **this** and `_opfe` to be in a safe state. By construction, these objects

are in a safe state only when there are no running instances of the workflow: it is then safe to modify the ports. Second is the **rebind** statement; the statement will succeed since the concurrency model of object-groups ensures that no workflow instance can be spawned between the end of the **await** and the end of the method. Moreover, the second line shows that it is possible to rebind a port of another object, provided that this object is in the same group as the one doing the rebinding.

3.2 Locations

The final layer of our language introduces *locations* that are used to model the different elements of our virtual office, like printers, computers, rooms and buildings. The idea is that components stand at a certain location. Thus every location, e.g. a room, is endowed with its own resources/services, e.g. printers, scanners, etc. . . , and a worker computer that stands at a certain location may exploit the location information to use resources at the same location.

Locations themselves are structured into trees according to a sublocation relation, such that we can have several locations at the top level (roots of trees) and object groups can only occur as leaves of such trees (and not as intermediate nodes).

We modify slightly the syntax of our previous calculus to introduce locations in it. We use l to represent location names. We represent with $(l, \mathtt{g})$ and (l, l') the father-to-son sublocation relation where object groups can only appear as leaves of the location tree. We use $l_\perp$ to stand for a name which is either $\perp$ or l, where $\perp$ is used to represent absence of a father, i.e. $(\perp, \mathtt{g})$ and $(\perp, l)$ mean that g and l, respectively, do not have a father. We also use n to represent node names which can be location names l or group names $\mathtt{g}$.

The additions are presented as follows.

$$\begin{array}{rcl} s & ::= & \ldots \mid \textbf{move}\ e\ \textbf{in}\ e \\ e & ::= & \ldots \mid \textbf{new loc} \end{array}$$

First, we add the possibility to create a new location (with a fresh name l) with a command **new loc**, then we add the possibility of modifying the father of a location/group n returned by an expression (or to establish a father in the case n does not possess one, or to remove the father of n) with the command **move** n **in** $l_\perp$: the new father becomes the location $l_\perp$ (returned by an expression). Technically, we also introduce a new type for location values, called `location`, which is added to the syntax of types T.

3.2.1 Examples

In the Virtual Office case study we use locations to express the movement of a worker from a location to another one. The worker moves with his laptop, in which we suppose a workflow document has been previously downloaded. The worker component has a set of output ports for connection to the services at the current worker location, which are needed to execute the downloaded workflow. Therefore the worker movement from a location to another one requires rebinding all such output ports, which can only be done if the workflow (a critical method) is not executing. Therefore, compared to previous examples, we need to model simultaneous rebinding of multiple output ports.

Example 1. We represent the movement of a worker to a different environment as the movement of the worker to a new location, which includes:

- a set of object groups representing the devices that the worker needs to perform the workflow (here represented by services "*ServiceA*" and "*ServiceB*")
- possibly, a local registry component, providing to the worker laptop component the links to the devices above; this will be modeled in Example 2.

More precisely, whenever the worker moves to a location l, first we wait for possible current workflow executions to be terminated, then we rebind to the (possibly discovered, see Example 2) new devices in the new location.

We represent the worker component as an object group composed by two objects:

- a "*ServiceFrontEnd*" object endowed by all the required output ports (here ports "*a*" and "*b*" for services "*ServiceA*" and "*ServiceB*", respectively),
- a "manager" object, called "*WorkerFrontEnd*" which: changes the ports in the "*ServiceFrontEnd*" object (possibly performing the service discovery enquiring the local service registry, see Example 2).

Finally, in the example code below, we make use of a primitive function "*group*" which is supposed to yield the group of a given object.

```
class ServiceA { ... }
class ServiceB { ... }

class ServiceFrontEnd {
  port ServiceA a;
  port ServiceB b;
  critical void workflow() { ... }
}

class WorkerFrontEnd {

  ServiceFrontEnd s;

  void changeLocation(location l2, ServiceA a2, ServiceB b2) {
    await ||s||;
    move group(this) in l2;
    rebind s.a = a2;
    rebind s.b = b2;
  }

  void init(location l, ServiceA a, ServiceB b) {
    move group(this) in l;
    s = new ServiceFrontEnd();
    rebind s.a = a;
    rebind s.b = b;
  }
}
```

Example 2. In this example we also model the local registry component for each location, providing links to the local devices for the worker component, and the global root registry (which has a known address) which, given a location, provides the link to the local register at that location.

More precisely, whenever the worker moves to a location l, first we have a discovery phase via a global root register so to obtain the local registry at location l, then we wait for possible current workflow executions to be terminated, then a discovery phase via the registry component of the new location, and finally a rebinding to the discovered devices in the new location.

```
class ServiceA { ... }
class ServiceB { ... }

class Register {
  ServiceA discoverA() { ... }
  ServiceB discoverB() { ... }
}

class RootRegister {
  Register discoverR(location l) { ... }
}

class ServiceFrontEnd {
  port ServiceA a;
  port ServiceB b;
  critical void workflow() { ... }
}

class WorkerFrontEnd {

  RootRegister rr;
  ServiceFrontEnd s;

  void changeLocation(location l2) {
    Fut<Register> fr=rr!discoverR(l2); await(fr); Register r=get(fr);
    await ‖s‖;
    move group(this) in l2;
    rebind s.a = get(r!discoverA());
    rebind s.b = get(r!discoverB());
  }

  void init(location l, RootRegister rr2) {
    rr = rr2;
    Fut<Register> fr=rr!discoverR(l); await(fr); Register r=get(fr);
    move group(this) in l;
    s = new ServiceFrontEnd();
    rebind s.a = get(r!discoverA());
    rebind s.b = get(r!discoverB());
  }
}
```

$$
\begin{array}{rcl}
N & ::= & \epsilon \mid I \mid C \mid N\ N \\
 & \mid & ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q) \\
 & \mid & cog(\mathtt{c}, \mathtt{o}_\varepsilon) \\
 & \mid & fut(\mathtt{f}, v_\perp) \\
 & \mid & invoc(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}) \\
 & \mid & (\gamma_\perp, \gamma) \\
Q & ::= & \varepsilon \mid K \mid Q\ Q \\
K & ::= & \{\ \sigma, s\ \} \\
v & ::= & \mathbf{null} \mid \mathtt{o} \mid \mathtt{f} \mid 1 \mid \ldots
\end{array}
\qquad
\begin{array}{rcl}
\sigma & ::= & \varepsilon \mid \sigma; T\ x\ v \\
 & \mid & \sigma; \mathbf{this}\ \mathtt{o} \\
 & \mid & \sigma; \mathbf{class}\ \mathtt{C} \\
 & \mid & \sigma; \mathbf{cog}\ \mathtt{c} \\
 & \mid & \sigma; \mathtt{nb}_{cr}\ v \\
v_\perp & ::= & v \mid \perp \\
\mathtt{o}_\varepsilon & ::= & \mathtt{o} \mid \varepsilon \\
K_{\mathbf{idle}} & ::= & K \mid \mathbf{idle} \\
\gamma_\perp & ::= & \gamma \mid \perp
\end{array}
$$

Fig. 6. Runtime Syntax; here o, f and c are object, future, and cog names

4 Semantics

We present in this section the semantics of our language. Our semantics is described as a virtual machine based on i) a runtime syntax that extends the basic language; ii) some functions and relations to manipulate that syntax; and iii) a set of reduction rules describing the evolution of a term.

4.1 Runtime Syntax

The runtime syntax consists of the language extended with constructs needed for the computations, like the runtime representation of objects, groups, and tasks. Figure 6 presents the global runtime syntax. Configurations N are sets of classes, interfaces, objects, concurrent object groups (cogs), futures, invocation messages and hierarchy statements between components. The associative and commutative union operator on configurations is denoted by a whitespace and the empty configuration by ϵ. An object is a term of the form $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q)$ where o is the object's identifier, σ is a substitution representing the object's fields, $K_{\mathbf{idle}}$ is the active *task* of the object (or $K_{\mathbf{idle}} = \mathbf{idle}$, when the object is idle and it is not executing anything), and Q is the queue of waiting tasks (the union of such queue, denoted by the whitespace, is associative with ε as the neutral element). A cog is a term of the form $cog(\mathtt{c}, \mathtt{o}_\varepsilon)$ where c is the cog's identifier, $\mathtt{o}_\varepsilon$ is either ε, which means that there is nothing currently executing in the cog, or an object identifier, in which case there is one task of the object o executing in c. A future is a pair of the name of the future f and a place $v_\perp$ where to store the value computed for this future. An invocation message $invoc(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$ specifies that some task called the method m on the object o with the parameters $\overline{v}$, this call corresponding to the future f. An hierarchy statement $(\gamma_\perp, \gamma)$ states that the component γ is a child of the component $\gamma_\perp$ ($\perp$ being the name of the top level component). A task K consists of a pair with a substitution σ of local variable bindings, and a statement s to execute. A substitution σ is a mapping from variable names to values. For convenience, we associate the declared type of the variable with the binding, and, in case of substitutions directly included in objects, we also use substitutions to store, the "this" reference, the class, the cog of an object and an integer denoted by $\mathtt{nb}_{cr}$ which, as we will see, will be used for critical section management. Finally, we extend the values v with object and future identifiers.

4.2 Reduction Relation

The semantics of the component model is an extension of the semantics of core ABS in [6]. It uses a reduction relation $\rightarrow$ over configurations, $N \rightarrow N'$ meaning that, in one execution step, the configuration N can evolve into N'. We extend that relation in four different aspects. First, we extend the reduction definition with three reduction rules that define the semantics of the **Rebind** and **subloc** operator.

REBIND-LOCAL

$$\frac{\sigma(\mathtt{nb}_{cr}) = 0}{ob(\mathtt{o}, \sigma, \{\ \sigma', \textbf{rebind}\ \mathtt{o.f} = v; s\ \}, Q) \rightarrow ob(\mathtt{o}, \sigma[f \mapsto v], \{\ \sigma', s\ \}, Q)}$$

REBIND-GLOBAL

$$\frac{\sigma_{\mathtt{o}}(\mathtt{nb}_{cr}) = 0 \qquad \sigma_{\mathtt{o}}(\textbf{cog}) = \sigma_{\mathtt{o}'}(\textbf{cog})}{\begin{array}{c} ob(\mathtt{o}, \sigma_{\mathtt{o}}, K_{\textbf{idle}}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma', \textbf{rebind}\ \mathtt{o.f} = v; s\ \}, Q) \\ \rightarrow ob(\mathtt{o}, \sigma_{\mathtt{o}}[f \mapsto v], K_{\textbf{idle}}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma', s\ \}, Q) \end{array}}$$

LOC-MOVE

$$(\gamma_{\perp}, \gamma)\ ob(\mathtt{o}, \sigma, \{\ \sigma', \textbf{move}\ \gamma\ \textbf{in}\ \gamma'_{\perp}; s\ \}, Q) \rightarrow (\gamma'_{\perp}, \gamma)\ ob(\mathtt{o}, \sigma, \{\ \sigma', s\ \}, Q)$$

The rule REBIND-LOCAL is applied when an object rebinds one of its own ports. The rule first checks that the object is not in a critical section by testing the special field $\mathtt{nb}_{cr}$ for zero and then updates the value of the field. The rule REBIND-GLOBAL is applied when an object rebinds a port of another object and is similar to the previous one. The rule LOC-MOVE moves a location γ (initially put inside the location $\gamma_{\perp}$) inside another location $\gamma'_{\perp}$.

The second aspect of our extension defines the semantics of our new expression, the creation of location **new loc**. In [6], the reduction rules defining the semantics of expressions are written using statements of the form $\sigma \vdash e \rightarrow \sigma \vdash e'$ to say that in the context σ mapping some variables to their values, e reduces to e'. Because expression **new loc** has a side effect (adding the new location to the configuration), we extend this statement to include the configuration: $N, \sigma \vdash e \rightarrow N', \sigma \vdash e'$.

NEW-LOCATION

$$\frac{\gamma\ \text{fresh}}{N, \sigma \vdash \textbf{new loc} \rightarrow N\ (\perp, \gamma), \sigma \vdash \gamma}$$

That rule simply states that the **new loc** commands creates a new location and returns it.

The third aspect of our extension concerns method call. In our system, we indeed have two kinds of methods: normal ones and critical ones, the second ones creating a critical section on the callee. We model opened critical sections with the special hidden field $\mathtt{nb}_{cr}$, that is initialized to zero, incremented each time a critical section is opened, and decremented each time a critical section is closed. Then, when an object calls a method, it creates an *invoc* message describing who is the callee, the method to execute, the parameters and the

return future. This message is then reduced into a task in the queue of the callee using the function bind that basically replaces the method by its code. To give the semantics of our critical methods, we extend this bind function to add, to the code of a critical method, some statements that manipulate the $\mathtt{nb}_{cr}$ field.

$$\text{NM-BIND}\quad \frac{\textbf{class}\ \mathtt{C} \ldots \{\ T\ \mathtt{m}(\overline{T}\ x)\{\ \overline{T'\ x'}\ s\ \}\ \ldots\ \} \in N}{\mathrm{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},\mathtt{C}) = \{\ \overline{T\ x = v}; \overline{T'\ x' = \textbf{null}}; \textbf{this} = \mathtt{o}, s\ \}}$$

$$\text{CM-BIND}\quad \frac{\begin{array}{c}\textbf{class}\ \mathtt{C} \ldots \{\ \textbf{critical}\ T\ \mathtt{m}(\overline{T}\ x)\{\ \overline{T'\ x'}\ s\ \}\ \ldots\ \} \in N \\ s' = \mathtt{nb}_{cr} = \mathtt{nb}_{cr} + 1; s; \mathtt{nb}_{cr} = \mathtt{nb}_{cr} - 1\end{array}}{\mathrm{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},\mathtt{C}) = \{\ \overline{T\ x = v}; \overline{T'\ x' = \textbf{null}}; \textbf{this} = \mathtt{o}, s'\ \}}$$

The rule NM-BIND corresponds to the normal semantics of the bind function, while the rule CM-BIND is the one used to bind a critical function. Basically, the first thing a critical method does is to increment the field $\mathtt{nb}_{cr}$, opening the critical section, and the last thing it does is to decrement the field, thus closing it.

Finally, the last aspect of our extension concerns our guard extension $\|e\|$.

$$\text{CSGUARD1}\quad \frac{N, \sigma \vdash e \rightsquigarrow N, \sigma \vdash \mathtt{o} \qquad ob(\mathtt{o}, \sigma_{\mathtt{o}}, K_{\textbf{idle}}, Q) \in N \qquad \sigma_{\mathtt{o}}(\mathtt{nb}_{cr}) = 0}{\sigma, N \vdash \|e\| \rightsquigarrow \sigma, N \vdash \mathtt{true}}$$

$$\text{CSGUARD2}\quad \frac{N, \sigma \vdash e \rightsquigarrow N, \sigma \vdash \mathtt{o} \qquad ob(\mathtt{o}, \sigma_{\mathtt{o}}, K_{\textbf{idle}}, Q) \in N \qquad \sigma_{\mathtt{o}}(\mathtt{nb}_{cr}) \neq 0}{\sigma, N \vdash \|e\| \rightsquigarrow \sigma, N \vdash \mathtt{false}}$$

These two rules simply state that, when the object $\mathtt{o}$ has its field $\mathtt{nb}_{cr}$ different from zero, it has a critical section opened.

4.3 Properties

Important properties that show the adequateness of our machinery for port rebinding are: (*i*) we never modify a port while being in a critical section (this property is a consequence of the reduction rule Rebind: the execution of the rebind expression can only occur when the object's lock is 0) and (*ii*) when await statements are not used in between, modification of several ports is atomic (due to cooperative concurrency in the object group model): this can be used, like in the second example of the location extension, to ensure consistency.

References

[1] OSGi Alliance. Osgi Service Platform, Release 3. IOS Press, Inc. (2003)
[2] Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: A system for constructing fine-grain configurable communication services. ACM Trans. Comput. Syst. 16(4) (1998)

[3] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and its Support in Java. Software - Practice and Experience 36(11-12) (2006)
[4] Castagna, G., Vitek, J., Nardelli, F.Z.: The Seal calculus. Inf. Comput. 201(1) (2005)
[5] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: OpenCOM v2: A Component Model for Building Systsms Software. In: Proceedings of IASTED Software Engineering and Applications, SEA 2004 (2004)
[6] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
[7] Lenglet, S., Schmitt, A., Stefani, J.-B.: Howe's Method for Calculi with Passivation. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 448–462. Springer, Heidelberg (2009)
[8] Levi, F., Sangiorgi, D.: Mobile safe ambients. ACM. Trans. Prog. Languages and Systems 25(1) (2003)
[9] Lienhardt, M., Lanese, I., Bravetti, M., Sangiorgi, D., Zavattaro, G., Welsch, Y., Schäfer, J., Poetzsch-Heffter, A.: A Component Model for the ABS Language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 165–183. Springer, Heidelberg (2011)
[10] Lienhardt, M., Schmitt, A., Stefani, J.-B.: Oz/K: A kernel language for component-based open programming. In: GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 43–52. ACM, New York (2007)
[11] Miranda, H., Pinto, A.S., Rodrigues, L.: Appia: A flexible protocol kernel supporting multiple coordinated channels. In: 21st International Conference on Distributed Computing Systems (ICDCS 2001). IEEE Computer Society (2001)
[12] Montesi, F., Sangiorgi, D.: A Model of Evolvable Components. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 153–171. Springer, Heidelberg (2010)
[13] Morris, R., Kohler, E., Jannotti, J., Frans Kaashoek, M.: The Click Modular Router. In: ACM Symposium on Operating Systems Principles (1999)
[14] Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
[15] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements (2006)