Component Model for Concurrent Object Groups: Theory and Practice *

Mario Bravetti
University of Bologna – INRIA Focus Team, Italy

Michael Lienhardt
PPS, Paris Diderot, France

Peter Y. H. Wong
SDL Fredhopper, Amsterdam, The Netherlands

January 31, 2013

Abstract

Dynamic reconfiguration, that is, changing the communication pattern of a program at runtime, 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 component model on top of the *Abstract Behavioral Specification* (ABS) language to provide a seamless integration of components to an object-oriented setting that allows *safe* dynamic reconfigurations. ABS has a cooperative scheduling concurrency model in which objects reside in concurrent object groups (cogs) and communicate via asynchronous calls with futures. Our goal is to enhance this model with the following elements of components to enable dynamic reconfigurations: i) output ports to represent an object's dependencies to the environment and can only be *rebound* new values when the object is in a *safe state*, ii) critical methods to denote the object to be not in a safe state if one or more such methods are being executed, and iii) a new primitive to wait for an object to be in a safe state. We formalize the notion of an object's safe state and the correctness of the component model to the rebinding of output ports and provide an executable implementation of the component model in ABS. We show the applicability of the component model via an industrial case study. Furthermore, we introduce a model of a hierarchy of locations to component model to specify movements of components between locations and discuss how this can be applied to the case study.

1 Introduction

Components are an intuitive tool to achieve unplanned dynamic reconfigurations. In a component system, an application is structured into several distinct

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

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 [17, 2, 5, 21, 1, 13, 15] do not merge the following 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 Architecture Description Language (ADL). This approach makes it simple to add components to an existing standard program. However, unplanned dynamic reconfigurations become hard, as 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 [15] do not allow runtime modifications while OSGi [17] allows addition of new classes and objects, but no safe component deletions or binding modifications. In this respect, a more flexible model is Fractal [2], 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 does not have a well-defined model.

Formal approaches to component models have been studied, for example, [3, 10, 20, 14, 12, 11]. These models have the advantage of having a precise semantics, which clearly defines what is a component, a port and a binding (when such constructs are 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 [12] and COMP [11] propose a way to integrate both components and objects in a unified model. However, Oz/K has a complex communication pattern, and deals with adaptation via the use of *passivation*, which, as commented in [9], 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, and 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 try 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 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 first wait for all current executions that use the field (like some printing jobs) to finish. This way we ensure that the update will not break those computations. In Java, one way such consistency can be achieved is by 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.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add three elements to the core ABS 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 correspond to the inner state of the objects and can be modified using ordinary assignments.

2. The possibility of annotating methods with the keyword **critical**: this specifies that the object is not in a safe state while these methods are being executed.

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.

The resulting language remains close to the underlying core ABS language. Indeed, the language is a conservative extension of core 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 core ABS program is simple. In contrast with previous component models, our language does not drastically separate objects and components. Two major features of the informal notion of component — ports and consistency — 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.

Theoretical Results In the paper we present some theretical results showing the main properties of the behaviour of our component based model. In particular, that the three elements above that we add to the core ABS language work correctly. The main results are expressed via two theorems. The first theorem basically shows that it is impossible to rebind a port of an object while one of its critical method is under execution, the second one shows that, by using the new primitive for awaiting for (multiple) objects to be in a safe state, we can safely rebind multiple ports of multiple objects in a safe and atomic way.

Applications and Tools We have extended the Maude back-end of the ABS tool suite [22], encompassing the new elements added to the core ABS language, to produce executable code that makes use of the component model. In order to show applicability of the component model and to improve paper's readability and its presentation, we make use of an industrial case study, officially considered in the EU project HATS, based on Fredhopper Access Server (FAS): a distributed concurrent object-oriented system developed by SDL Fredhopper that provides search and merchandising services to eCommerce companies. FAS consists of multiple environments for processing client queries and receiving data updates. To maintain data consistency across these environments, FAS relies on a replication protocol, which is implemented by a component of FAS known as the Replication System. The replication protocol may specify one of the two types of replication job policies: sequential or concurrent. With the original implementation switching between these job policies requires shutting down and restarting the Replication System. In the paper we show how the component model we propose can be used to obtain an implementation of the Replication System that switches between job policies during the runtime of FAS.

Extension to Mobility Considering again the notion of component, another conceptual difference between objects and components is that, for the latter, a notion of *location* can be important to model. 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. In this paper we also propose another extension to the core ABS language which endows it with a hierarchy of locations. In particular an core ABS program is structured into a tree of locations that can contain object groups (as leaves of the tree), 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

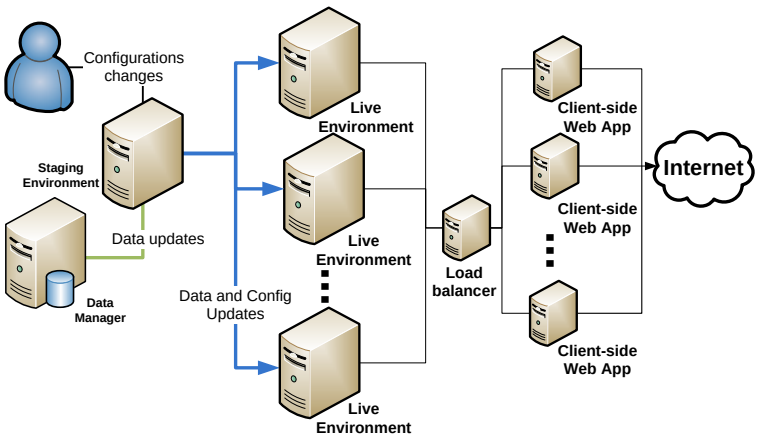

Figure 1: An example FAS deployment

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).

Paper Outline In Section 2 we introduce the industrial case study of the replication system, a conceptual introduction to core ABS and the significant fragments of core ABS code modeling the case study in the original version (without the component model we introduce here). In Section 3 we present the elements addeded to the core ABS language realizing the component model: the syntax and the semantics of ABS extended whith such elements and the theoratical results showing them to work correctly. In Section 4 we present the modifications to the model of the case study realizing the runtime switching between job policies using the component model and the related extension of the Maude backend of the ABS tool suite. In Section 5 we present the further extension to the core ABS language endowing it with locations expressing mobility behaviours. Finally in Section 6 we report some concluding remarks and discuss how the proposed model of locations can be applied in the case study.

2 Industrial Case Study

Fredhopper provides the Fredhopper Access Server (FAS). It is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Fig 1 shows an example FAS deployment.

FAS consists of a set of live environments and a single staging environment.

A live environment processes queries from client web applications via web services. FAS aims to provide a constant query throughput to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the *Replication Protocol*. The Replication Protocol is implemented by the *Replication System*. The Replication System consists of a *SyncServer* in the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication, as well as its content, while a SyncClient receives data and configuration updates according to the schedule.

2.1 Replication Protocol

The SyncServer communicates to SyncClients by creating *ConnectionThread* objects. ConnectionThreads serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. Each instance of the ClientJob defines a single *replication session*. When transferring data between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering with read/write access of the staging environment's underlying file system, the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed.

Figure 2 shows how for each scheduled replication session, a `ClientJob` object is created to interact with the staging environment via a `ConnectionThread` object. We first provide an informal description of the interaction:

> For each scheduled replication session, the SyncClient creates a ClientJob. This ClientJob first connects to an `Acceptor`, the Acceptor then creates a new `ConnectionThread` object (`Acceptor.getConnection`). The ClientJob then acquires from the ConnectionThread the next schedule for this replication session (`ClientJob.getSchedules()`) and notifies the SyncClient (`SyncClient.schedule()`). The SyncClient *waits* for the current ClientJob to terminate before proceeding with the next scheduled replication session. In the meantime the current ClientJob receives replication updates from the ConnectionThread `ClientJob.sendItems()`.

2.2 Job Policies

Fredhopper deploys a FAS installation to each customer on an agreed Service Level Agreement (SLA). This may consist of the query throughput, the query

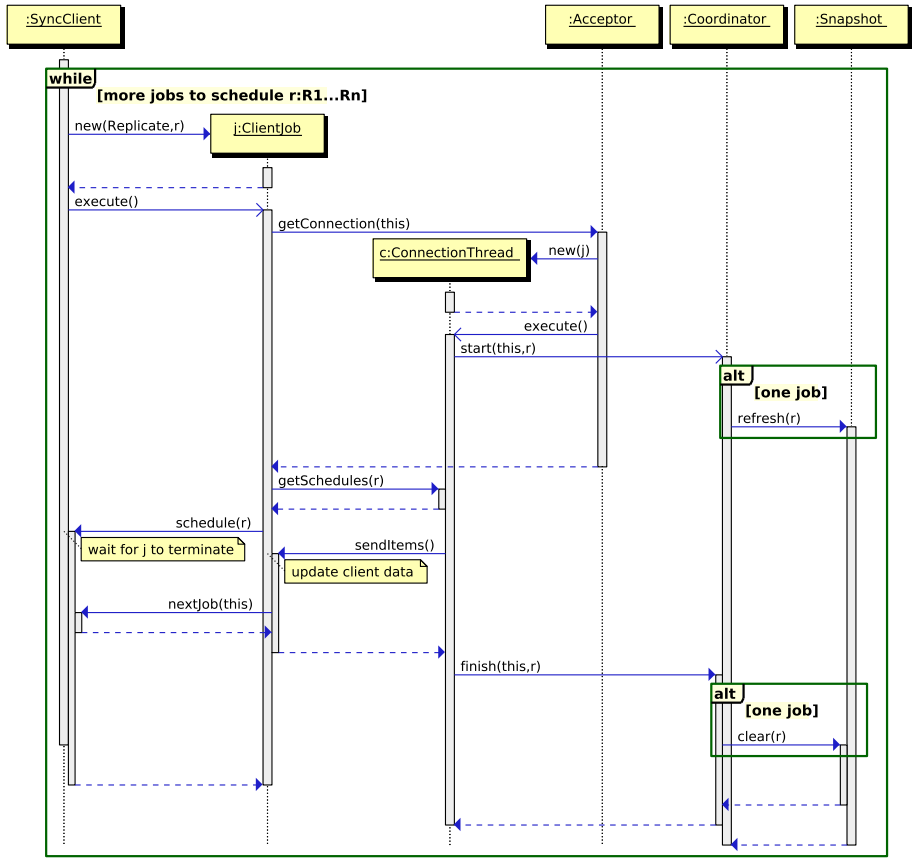

Figure 2: UML sequence diagram of a replication job

response time and the frequency of configuration and index updates. In particular, to cater for varying update frequencies, FAS offers two types of replication job policies: *sequential* and *concurrent*. In the sequential job policy, replication sessions that are scheduled by a single SyncClient must be run sequentially. On the other hand, in the concurrent job policy, replication sessions that are scheduled by a single SyncClient may be executed concurrently. However, to ensure the consistency of live environment's underlying configuration and index, two replication sessions may be run in parallel if and only if they do not *interfere*. Two sessions interfere if they write to the same index or configuration. Interference can cause index inconsistency since multiple sessions may write to the same index structure at the same time. We therefore associate a *type* to each replication session and enforce the following constraint:

Definition 1. *Two sessions of different types do not write to the same index structure and configuration.*

Given this constraint, the Replication System ensures that replication sessions of the same type are run sequentially, while sessions of different types may be run concurrently. Currently switching between job policies requires shutting down and restarting the Replication System.

2.3 Modeling the Replication System

In this section we first introduce the core ABS language via the case study, focusing on the core parts of the sequential job policy. We then present the core parts of the concurrent job policy, highlighting the main differences between the policies. The complete ABS model of the Replication System can be found at `http://www.hats-project.eu/sites/default/files/replication-system-dynamic-policy.zip`.

core ABS is an abstract, executable, object-oriented modeling language with a formal semantics [6], targeting distributed systems. core ABS is designed with a layered architecture, at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) Java. core ABS generalizes the concurrency model of Creol [7] from single concurrent objects to concurrent object groups (cogs). As in [19] cogs encapsulate synchronous, interleaving, shared state computation on a single processor. An essential difference to thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows to write concurrent programs in a much less error-prone way than in a thread-based model and makes core ABS models suitable for static analysis.

We now describe the core constructs of core ABS used in this paper in some detail. Specifically, we describe

- algebraic data types and functions;
- interfaces and classes;
- synchronous method calls and objects creation;
- asynchronous method calls and concurrent object group creation;
- cooperative scheduling using **await** statements.

Data Types and Functions core ABS supports *algebraic data types* (ADT) to model data in a software system. ADTs abstract away from implementation details such as hardware environment, file content, or operating system specifics. For example in the Replication System, the following ADT `Schedule` models a schedule specification of a replication session, recording the type of the session, the time to wait before starting the session and the file location from which to retrieve updates.

```
data Schedule = Schedule(Type stype, Int time, String location);
```

core ABS supports first-order functional programming with ADT. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features. For example, the following function `stype` is a selector function that returns the type of replication session specified in the given schedule.

```
def Type stype(Schedule s) = case s { Schedule(x,y,z) => x; };
```

Interfaces core ABS has a nominal type system with interface-based subtyping. Interfaces define types for objects. They have a name, and define a set of method signatures, that is, the names and types of callable methods. The following shows a simplified version of interface `SyncClient` that models a SyncClient, focusing on parts of the SyncClient relevant for implementing job policies.

```
interface SyncClient {
  Unit nextJob(Type tp);
  Unit scheduleJob(Schedule schedule);
  Unit requestShutDown();
  Bool isShutdownRequested();
}
```

The method `scheduleJob` is invoked to schedule a replication session according to the information given by `schedule`; Method `nextJob` is invoked to notify the next session of a given type may commence; Method `requestShutDown` is invoked to request the SyncClient to shut down and that no further session may run, and Method `isShutdownRequested` checks whether or not a request has been made to shut down the SyncClient.

Classes core ABS also supports class-based, object-oriented programming with standard imperative constructs. Classes define the implementation of objects. In contrast to Java, for example, classes do *not* define a type. Classes can implement arbitrarily many interfaces, which then define types of a new instance of that class. A class has to implement all methods of all its implementing interfaces. Instead of constructors, classes in core ABS have *class parameters*, which are instance fields and an *initialisation method*, which is invoked on object creation. In addition, a class may further define additional instance fields. The following class `SequentialSyncClient` implements `SyncClient` with the sequential job policy:

```
class SequentialSyncClient(Network network) implements SyncClient {
  Bool next = True; Bool shutDown = False;
  Bool isShutdownRequested() { .. }
```

```
  Unit nextJob(Type tp) { .. }
  Unit requestShutDown() { .. }
  Unit waitFor(Type tp) { .. }
  Unit scheduleJob(Schedule schedule) { .. }
}
```

It defines class parameter `network` of `Network`. The parameter `Network` models a network in which the SyncClient resides. It further defines instance fields `next` and `shutDown`. In addition it defines a private method `waitFor` that is not exposed through the interface `SyncClient`.

Thread-based computation Basic statements describing the flow of control of a single thread include the usual (synchronous) method invocations, object creation, and field and variable reads and assignments. These statements can be composed by the standard control structures (sequential composition, conditional and iteration constructs). The following expands the implementation `SequentialSyncClient` to look at the implementation of method `nextJob`.

```
class SequentialSyncClient(Network network) implements SyncClient {
  Bool next = True; Bool shutDown = False;
  Bool isShutdownRequested() { return shutDown; }
  Unit nextJob(Type tp) { if (~shutDown) { next = True; }}
  Unit requestShutDown() { .. }
  Unit waitFor(Type tp) { .. }
  Unit scheduleJob(Schedule schedule) { .. }
}
```

The method `nextJob` checks whether the SyncClient has been requested to shut down and sets the flag `next`, which indicates the next job can be executed, to `True`. Note that for sequential job policy, only one job may be executed at any time, hence the input value `tp` is not taken into account.

Asynchronous communications The concurrency model of core ABS is based on the concept of *Concurrent Object Groups* (object groups). A typical ABS system consists of multiple, concurrently running object groups at runtime. Object groups can be regarded as autonomous runtime components that are executed concurrently, share no state and communicate via method calls. A new object group is created using the **new cog** expression. It takes as argument a class name and optional parameters and returns a reference to the initial object of the new group. Communication between groups may be done solely via *asynchronous method calls*. The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot. The following expands the fragment of `SequentialSyncClient` shown earlier to illustrate group creation and asynchronous communications.

```
class SequentialSyncClient(Network network) implements SyncClient {
  Bool next = True; Bool shutDown = False;
  Bool isShutdownRequested() { .. }
  Unit nextJob(Type tp) { .. }
  Unit requestShutDown() {
    shutDown = True;
    network!shutDown(this); }
  Unit waitFor(Type tp) { .. }
  Unit scheduleJob(Schedule schedule) {
    this.waitFor(stype(schedule));
    if (~shutDown) {
      ClientJob job = new cog ClientJobImpl(this, schedule);
      next = False; job!executeJob(); }}}
```

In the method `scheduleJob`, we see that the SyncClient first invokes method `waitFor` and then if the SyncClient has not been requested to shut down, it sets `next` to `False` and creates a new concurrent object group with the initial object to be a new `ClientJobImpl`. After which it asynchronously invokes the ClientJob's method `executeJob`.

Cooperative scheduling Each asynchronous method call results in a *task* in the object group of the target object. Tasks are scheduled *cooperatively* within the scope of a object group. Cooperative scheduling means that switching between tasks of the same object group happens only at specific *scheduling points* during program execution and that at no point two tasks in the same group are active at the same time. With the **await** statement, one can create a *conditional* scheduling point, where the running task is suspended until after the specified condition becomes true. The **await** statement can be used to suspend a task until a future becomes resolved or a Boolean condition over the object state becomes true. While waiting for a future, other tasks can run. The following further expands the implementation of `SequentialSyncClient`.

```
class SequentialSyncClient(Network network) implements SyncClient {
  Bool next = True; Bool shutDown = False;
  Bool isShutdownRequested() { return shutDown; }
  Unit nextJob(Type tp) { if (~shutDown) { next = True; }}
  Unit requestShutDown() { .. }
  Unit waitFor(Type tp) { await next || shutDown; }
  Unit scheduleJob(Schedule schedule) {
    this.waitFor();
    if (~shutDown) {
      ClientJob job = new cog ClientJobImpl(this, schedule);
      next = False; job!executeJob(); }}}
```

The method `waitFor` invoked by `scheduleJob` before creating a new ClientJob suspends itself (the running task) until either the next job is allowed to proceed (`next == True`) or the SyncClient has been requested to shut down

(`shutDown == True`). Note that due to this concurrency model, multiple invocations of this method of the same instance cannot be executed at the same time, but instead they can be interleaved.

2.3.1 Concurrent Job Policy

To implement the concurrent job policy, FAS must ensure that: *Two concurrently running replication sessions do not* interfere*, that is, they do not write to the same index or configuration.* To ensure non-interference replication sessions of the same type are run sequentially while sessions of different types may be run concurrently. In the core ABS model, `Type` denotes the type of the session and the selector function `stype` returns the type specified in a given schedule. The following class `ConcurrentSyncClient` implements `SyncClient` with the concurrent job policy.

```
class ConcurrentSyncClient(Network network) implements SyncClient {
  Map<Type,Bool> nexts = EmptyMap; Bool shutDown = False; Int jc = 0;
  Bool isShutdownRequested() { return shutDown; }
  Unit nextJob(Type tp) { if (~shutDown) { nexts = put(nexts,tp,True); }}
  Unit requestShutDown() { .. }
  Unit waitFor(Type tp) {
    await lookupDefault(nexts,tp,True) || shutDown; }
  Unit scheduleJob(Schedule schedule) {
    Type tp = stype(schedule);
    this.waitFor(tp);
    if (~shutDown) {
      ClientJob job = new cog ClientJobImpl(this, schedule);
      nexts = put(nexts,tp,False);
      job!executeJob(); }}}
```

It defines a map `nexts` that records for which replication type a session may start. The method `scheduleJob` ensures to start a session if the session's replication type does not already have a running session (`lookupDefault(nexts,tp,True) == True`). After creating a ClientJob, the method blocks subsequent replication session of the same type from running `nexts = put(nexts,tp,False)`.

2.4 Meeting Varying Requirements

In the recent years, there are increasing demands on more flexible SLA. For example, retail customers might expect large throughput during seasonal sales. In these periods, much larger amounts of purchases will be made and stock units must be updated very frequently so that customers do not receive incorrect information. Higher throughput requires a larger number of live environments, while more frequent updates requires more frequent replication sessions. In both cases, more resources such as memory and processing power would be required. Furthermore, customers would like to be able to change their expected

P	::=	$\overline{Dl}\ \{\ \overline{T\ x;}\ s\ \}$	Program
Dl	::=	$D \mid F \mid I \mid C$	Declarations
T	::=	$\texttt{V} \mid \texttt{D} < \overline{T} > \mid \texttt{I}$	Type
D	::=	$\textbf{data}\ \texttt{D} < \overline{V} >= \texttt{Co}[(\overline{T})][\mid \overline{\texttt{Co}[(\overline{T})]}]$	Data Type
F	::=	$\textbf{def}\ T\ \texttt{fun}[< \overline{T} >](\overline{T\ x}) = e$	Function
I	::=	$\textbf{interface}\ \texttt{I}\ \{\ \overline{S}\ \}$	Interface
C	::=	$\textbf{class}\ \texttt{C}(\overline{T\ x})\ [\textbf{implements}\ \overline{\texttt{I}}]\ \{\ \overline{Fl}\ \overline{M}\ \}$	Class
Fl	::=	$T\ x$	Field Declatation
S	::=	$T\ \texttt{m}(\overline{T\ x})$	Method Signature
M	::=	$S\{\ \overline{T\ x;}\ s\ \}$	Method Definition
s	::=		Statement
		$\textbf{skip} \mid s;s \mid x = z \mid \textbf{await}\ g$	
	|	$\textbf{if}\ e\ \{\ s\ \}\ \textbf{else}\ \{\ s\ \} \mid \textbf{while}\ e\ \{\ s\ \} \mid \textbf{return}\ e$	
z	::=	$e \mid \textbf{new}\ [\textbf{cog}]\ \texttt{C}\ (\overline{e}) \mid e.\texttt{m}(\overline{e}) \mid e!\texttt{m}(\overline{e}) \mid e.\textbf{get}$	Expression with Side Effects
e	::=	$v \mid x \mid \textbf{this} \mid \texttt{fun}(\overline{e}) \mid \textbf{case}\ e\ \{\overline{p \Rightarrow e}\}$	Expression
v	::=	$\textbf{null} \mid \texttt{Co}[(\overline{v})]$	Value
p	::=	$\_ \mid x \mid \textbf{null} \mid \texttt{Co}[(\overline{p})]$	Pattern
g	::=	$x \mid x? \mid g \wedge g$	Guard

Figure 3: Core ABS Language

throughput and update frequency *without* shutting down and restarting running instances of FAS. To cater for these flexible changes to the update frequency without introducing unnecessary resource cost, it must be possible to switch between sequential and concurrent job policies on-demand *at runtime*. Moreover, the Replication System must be able to *safely* switch between sequential and concurrent jobs. To safely switch between sequential and concurrent jobs during the runtime of FAS, the Replication System must satisfy the following property:

Definition 2. *The Replication System must be able to switch between sequential and concurrent replication jobs only when there is no running session.*

3 Component Model

In this section, we first present the formal syntax of the core ABS language, as it is used in the case study. In a second part, we extend this core language with simple primitives, inspired from component models, that enable *safe* dynamic reconfiguration. We finally present the operational semantics of the resulting language and prove some of its properties.

3.1 Core ABS

The syntax of the core ABS language is presented in Figure 3. For the sake of readability, we use the following notations in our presentation of the syntax: an overlined element corresponds to any finite sequence of such element; and an element between square brackets is optional.

A program in core ABS, defined by the entry P of our syntax, is constructed as a list of declarations Dl, followed by a main function $\{\,\overline{T\ x\texttt{;}}\ s\,\}$. Core ABS includes four kind of declarations: *data-types* D and *functions* F constitute the functional part of the language, while *interfaces* I and *classes* C constitute its object-oriented part. A type annotation T can have three forms: it can either be a type variable `V` used for polymorphism, a datatype with parameters $\texttt{D} < \overline{T} >$ (to type structured data), or the name of an interface `I` (to type objects). A data type D has a name `D` and a set of parameters $\overline{V}$, and is constructed as a set of type constructors `Co` with possible parameters $\overline{T}$. Note that data types include types like `Integer`, `Bool`, `String` and $\texttt{Fut} < T >$ which is used to type futures. A function F has a return type T, a name `fun`, can be polymorphic with a set of types in parameter, has a set of parameters $\overline{T\ x}$, and returns the expression e after evaluation. 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 Fl and its methods M.

Statements s are as presented in Section 2, with the empty statement **skip**, the sequential composition $s; s$, the assignment $x = z$, the conditional wait **await** g, the conditional **if**, the loop **while**, and the return **return** e,. The z production represents expressions that may change the state of the system, like an object or cog creation **new** [**cog**] `C` $(\overline{e})$, a method call $e.\texttt{m}(\overline{e})$ or $e!\texttt{m}(\overline{e})$, or a get on a future e.**get**. On the opposite, the e production represents *functional* expressions, like pure value v, variable x, function application $\texttt{fun}(\overline{e})$, or a pattern matching **case** e $\{\overline{p \Rightarrow e}\}$. Values include the **null** object, and structured data $\texttt{Co}[(\overline{v})]$, while patterns p extend these values with variables x and anonymous variables _. Finally, guards for await statements include boolean variables x that need to be true to resume the execution of the method, future lookup $x?$ that need the future x to be completed to resume the execution of the method, or the conjunction of guards $g \wedge g$.

3.2 Component Model

As discussed in Section 2, 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 4. This client, upon the call of the method `useService`, performs, asynchronously for efficiency, three operations on the server. Moreover, to allow the update of the server at runtime, the client offers the method `setServer`. However, this dynamic reconfiguration can lead to an inconsistent use of the server: let us consider the scenario where `useService` starts, while `setServer` is waiting to be executed. The method `doFirstThing` is called on the server, followed by the **await** `f1?` statement which allows `setServer` to change the reference to the server. Then `useService` starts again and calls `doSecondThing` on the new server: this call is inconsistent as the new server didn't execute the method `doFirstThing`. Hence, dynamic reconfiguration in core ABS raises a problem of consistency, which is in general difficult to solve using only the primitives of core ABS.

```
class Client {
 Server s;
 Integer count;

 Unit setServer(Server newS) { s = newS; }

 Unit useService(Parameter p) {
  count = count + 1;

  Fut<Unit> f1 = s!doFirstThing(p); await f1?;
  Fut<Unit> f2 = s!doSecondThing(p); await f2?;
  Fut<Unit> f3 = s!doLastThing(p); await f3?;

  count = count - 1;
 }

 Integer getNumberOfJobs() { return count; }

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

Figure 4: Client with Inconsistent Dynamic Update

3.2.1 Extending Core ABS

We overcome the problem of inconsistent dynamic updates by forbidding the modification of important data while they are in use. For this, we combine the notions of *output port* (from components) and *critical section*. Basically fields which reference an external service that can change at runtime have a special status: they are called *output ports*; and the methods that use these ports and thus need them not to change during their execution, create a critical section to forbid their modification. Formally, these two notions are integrated in core ABS with the following syntax extension:

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

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 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 the 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 $\mathtt{o}_i$, it first waits for them to close all their critical sections, and then applies the modifications using rebind.

3.2.2 Examples

1. The `Client` class, Revisited In Figure 5 we show how to solve the consistency issue of the `Client` class (from Figure 4). The changes are simple: i) we specify that the field `s` is a port; ii) we annotate the method `useService` with **critical** (to protect its usage of the port `s`); and iii) we change the method `setServer` so that it now waits for the object to be in a consistent state *before* rebinding its output port `s`.

2. Coordiating Dynamic Updates We previously stated that it is useful to allow output ports to be modified by objects different from their owners. Indeed, let's consider the following scenario: suppose that we have several clients, working together in a specific workflow, and using a central server for their communication. Updating the server is a difficult task, as it requires to update the reference in all clients at the same time, to avoid communication failures. This task can easily be achieved using our model, as shown in Figure 6. This figure presents the class `Controler` that manages the server update of all the clients $\mathtt{c}_i$ with the method `updateServer`. This method first waits for all clients to be in a state with no open critical section: they thus all allow their reference to the server to be modified; and then, their references are updated one by one.

3.3 Semantics

The semantics of our model is a simple extension of core-ABS's semantics, which is based on two elements: a runtime syntax consisting of the language extended with constructs needed for the computations, such as the runtime representation

```
class Client {
 port Server s;
 Integer count;

 Unit setServer(Server newS) {
  await |this|;
  rebind this:s = newS;
 }

 critical Unit useService(Parameter f) {
  count = count + 1;

  Fut<Unit> f1 = s!doFirstThing(f); await f1?;
  Fut<Unit> f2 = s!doSecondThing(f); await f2?;
  Fut<Unit> f3 = s!doLastThing(f); await f3?;

  count = count - 1;
 }

 Integer getNumberOfJobs() { return count; }

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

Figure 5: A Client with Consistent Dynamic Update

of objects, groups, and tasks; and an operational semantics, defined as a set of reduction rules, which represents the effect of each expression and statement at runtime.

3.3.1 Runtime syntax

Figure 7 presents the global runtime syntax. Configurations N are sets of classes, interfaces, objects, concurrent object groups (cogs), futures and invocation messages. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ϵ. An object is a term of the form $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q)$ where $\mathtt{o}$ 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 set of waiting tasks (the union of such queue, denoted by the whitespace, is commutative, associative with ε as the neutral element). A cog is a term of the form $cog(\mathtt{c}, \mathtt{o}_\varepsilon)$ where $\mathtt{c}$ is the cog's identifier, and $\mathtt{o}_\varepsilon$ 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 $\mathtt{o}$ executing in $\mathtt{c}$. A future is a pair of the name of the future $\mathtt{f}$ 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

```
interface Server { ... }
interface Client { port Server s; ... }

class Controler {
 Client c1, c2, ... cn;

 Unit updateServer(Server s2) {
  await |c1| /\ |c2| /\ ... /\ |cn|;
  rebind c1:s = s2;
  rebind c2:s = s2;
  ...
  rebind cn:s = s2;
 }
}
```

Figure 6: A Dynamic Update Controler

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

Figure 7: Runtime Syntax; here `o`, `f` and `c` are object, future, and cog names

called the method `m` on the object `o` with the parameters $\overline{v}$, this call corresponding to the future `f`. 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, we also use substitutions to store: i) in case of substitutions directly included in objects, their **this** reference, their class **class**, their cog **cog**, and an integer denoted by $\mathbf{nb}_{cr}$ which *counts* how many open critical sections there are in an object; and ii) in case of substitution directly included in tasks, their **destiny** future where they will put their return value. Finally, we extend values v with object and future identifiers (`o` and `f`), and statments with continuation **cont**(`f`), used for synchronous calls.

3.3.2 Initial Configuration

To be able to execute a program written in our language, one must first translate it in a configuration that can be executed. This translation is given by the 'Configuration' function, described below:

$$
\begin{array}{l}
\text{Configuration}(\overline{D}\ \overline{F}\ \overline{C}\ \overline{I}\ \{\, \overline{T\ x;}\ s \,\}) \\
\triangleq \left(\begin{array}{c} \overline{C}\ cog(\mathtt{c}_{main}, \mathtt{o}_{main})\ fut(\mathtt{f}_{main}, \perp) \\ ob(\mathtt{o}_{main}, \varepsilon; \mathbf{cog}\ \mathtt{c}_{main}; \mathbf{nb}_{cr}\ 0, \{\, \overline{T\ x\ \text{default}(T)}; \mathbf{destiny}\ \mathtt{f}_{main}, s \,\}, \varepsilon) \end{array} \right)
\end{array}
$$

$$\textsc{RedInCons}\ \frac{\sigma \vdash e_i \rightsquigarrow e_i'}{\sigma \vdash \texttt{Co}(e_1,\ldots,e_i,\ldots,e_n) \rightsquigarrow \texttt{Co}(e_1,\ldots,e_i',\ldots,e_n)} \qquad \textsc{RedInFun}\ \frac{\sigma \vdash e_i \rightsquigarrow e_i'}{\sigma \vdash \texttt{fun}(e_1,\ldots,e_i,\ldots,e_n) \rightsquigarrow \texttt{fun}(e_1,\ldots,e_i',\ldots,e_n)}$$

$$\textsc{RedFun}\ \frac{F(\texttt{fun}) = (\overline{T\ x})(e)}{\sigma \vdash \texttt{fun}(\overline{v}) \rightsquigarrow e[\overline{x} \mapsto \overline{v}]} \qquad \textsc{RedVar}\ \frac{}{\sigma \vdash x \rightsquigarrow \sigma(x)}$$

$$\textsc{RedCase1}\ \frac{\sigma \vdash e \rightsquigarrow e'}{\sigma \vdash \textbf{case}\ e\ \{\overline{p \Rightarrow e}\} \rightsquigarrow \textbf{case}\ e'\ \{\overline{p \Rightarrow e}\}} \qquad \textsc{RedCase2}\ \frac{\text{match}(\sigma(p), v) = \bot}{\sigma \vdash \textbf{case}\ v\ \{p \Rightarrow e; \overline{p' \Rightarrow e'}\} \rightsquigarrow \textbf{case}\ v\ \{\overline{p' \Rightarrow e'}\}}$$

$$\textsc{RedCase3}\ \frac{\text{match}(\sigma(p), v) = \sigma'}{\sigma \vdash \textbf{case}\ v\ \{p \Rightarrow e; \overline{p' \Rightarrow e'}\} \rightsquigarrow \sigma'(e)}$$

Figure 8: Semantics of Expression

This function creates a configuration from a program by: extracting the classes from the program; creating an initial cog, called $\texttt{c}_{main}$; creating an initial dummy object called $\texttt{o}_{main}$ executing the main method of the program in a task whose future is $\texttt{f}_{main}$. Local variables of the main are initialized to a default value of the right type using the auxiliary function 'default'. Function declarations are not included to the configuration, but in an auxiliary function, called 'Ffun' that maps each function name to a pair of its formal parameters and main expression. This function will be used to give the semantics of function application. The other data in the program, like interfaces and data types, are useful to ensure the type safety of the program and do not take part to its runtime semantics.

3.3.3 Reduction relation

The semantics of the component model is an extension of the semantics of core ABS in [6]. It is defined in four parts: i) Figure 8 presents the semantics of functional expressions; ii) Figure 9 presents the semantics of guards; iii) Figure 10 presents the semantics of expressions with side effects; and iv) Figure 11 presents the semantics of statements.

Functional Expressions The evaluation of a functional expression e is unchanged from core ABS, and defined as a small step semantics, written $\sigma \vdash e \rightsquigarrow e'$, which means that in the context of σ mapping variables to values, e reduces to e'. This relation between functional expressions is defined as the transitive closure of the rules presented in Figure 8. The two rules RedInCons and RedInFun state that the reduction can occur in parameters of data constructors and function applications. Rule RedFun uses the auxiliary function 'Ffun' to reduce the application of `fun` with the real parameters $\overline{v}$ to the expression e, where we replaced the variables x with their real value. Rule RedVar simply replaces a variable by its value in the subsitution σ. Case expressions are handled by the rules RedCase1, RedCase2 and RedCase3. Rule RedCase1 states that the reduction can occur inside the parameter of the **case** construct. Rules RedCase2 and RedCase3 find a branch that matches the parameter v,

$$\textsc{GuardBool}\quad N, \sigma \vdash x \rightsquigarrow \sigma(x)$$

$$\textsc{GuardFut1}\quad \frac{\sigma(x) = \mathtt{f} \qquad \mathit{fut}(\mathtt{f}, v) \in N}{N, \sigma \vdash x? \rightsquigarrow \mathtt{true}}$$

$$\textsc{GuardFut2}\quad \frac{\sigma(x) = \mathtt{f} \qquad \mathit{fut}(\mathtt{f}, \bot) \in N}{N, \sigma \vdash x? \rightsquigarrow \mathtt{false}}$$

$$\textsc{GuardLand}\quad \frac{\sigma, N \vdash g_1 \rightsquigarrow g_1' \qquad \sigma, N \vdash g_2 \rightsquigarrow g_2'}{\sigma, N \vdash g_1 \wedge g_2 \rightsquigarrow g_1' \;\&\&\; g_2'}$$

$$\textsc{GuardCS1}\quad \frac{ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q) \in N \qquad \sigma(x) = \mathtt{o} \qquad \sigma_\mathtt{o}(\mathbf{nb}_{cr}) = 0}{\sigma, N \vdash |x| \rightsquigarrow \mathtt{true}}$$

$$\textsc{GuardCS2}\quad \frac{ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q) \in N \qquad \sigma(x) = \mathtt{o} \qquad \sigma_\mathtt{o}(\mathbf{nb}_{cr}) \neq 0}{\sigma, N \vdash |x| \rightsquigarrow \mathtt{false}}$$

Figure 9: Semantics of Guard

using the annex function $\mathrm{match}(p, v)$ that returns either the unique substitution σ such that $\sigma(p) = v$ and $\mathrm{dom}(\sigma)$ is limited to the variables of p; or $\bot$ when such a substitution does not exist.

Guards The evaluation of guards g is defined as a big step semantics, presented in Figure 9, and written $N, \sigma \vdash g \rightsquigarrow v$, which means that in the configuration N, using the substitution σ, the guard g reduces to v. The four first rules are identical as in core-ABS: variables are evaluated to their value (rule GuardBool); guards on completed futures are evaluated to `true` (rule GuardFut1); guards on undefined futures are evaluated to `false` (rule GuardFut2); and conjunctions of two guards are evaluated to the conjunction of their value (rule GuardLand). Rules GuardCS1 and GuardCS2 are new and deal with the new guard $|x|$. Informally, this guard must evaluate to `true` when the object x does not have any open critical sections, or to `false` otherwise. In our implementation, the integer $\mathbf{nb}_{cr}$ counts for each object the number of opened critical sections. Rule GuardCS1 thus looks at the $\mathbf{nb}_{cr}$ integer of the object `o`, sees that there are no open critical sections, and thus returns `true`. On the opposite, rule GuardCS2, looking at the $\mathbf{nb}_{cr}$ integer, sees that there are some open critical sections, and returns `false`.

Expressions with Side Effects As this kind of expression is always at the right hand side of an assignment statement $x = z$, we present its semantics the same way we will present the semantics of statements: the reduction rules, presented in Figure 10, presents a statement in the context of its executing object, and in parallel of elements from the configuration that are needed. These rules are identical as in core ABS, but the auxiliary functions they use, for the object creation and method calls, are modified to include critical section management. These functions will be presented as we describe the rules. Rule AsyncCall sends an invocation message to $\mathtt{o}'$ with a new, fresh future `f`, the method name `m` and its actual parameters $\overline{v}$. The return value of the call, stored in the future declaration $\mathit{fut}(\mathtt{f}, \bot)$ is undefined yet. Rule Bind consumes an invocation message and places the task generated by the function bind in the callee's task pool, with a terminating **skip** statement used during the termination of the task. The

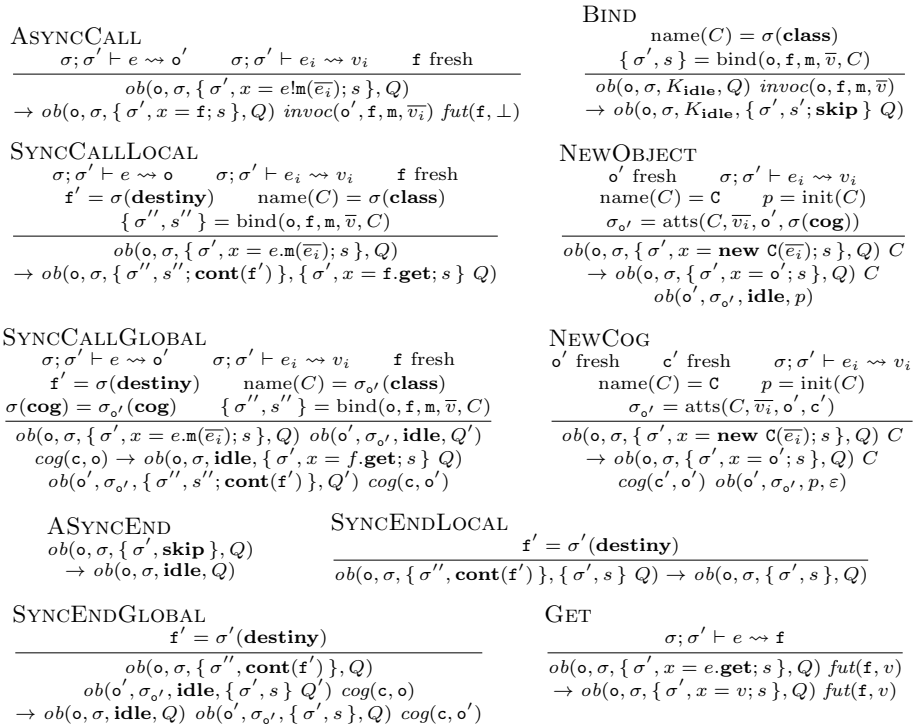

Figure 10: Semantics of Expressions with Side Effect

function bind, which transforms an invocation message into a task corresponding to the method activation, is described in the following two rules:

$$\textsc{SBind}\quad \frac{C = \mathbf{class}\ \mathtt{C} \ldots \{\, T\ \mathtt{m}(\overline{T}\ x)\{\,\overline{T'\ x'}\ s\,\}\ \ldots\,\}}{\mathrm{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},C) = \{\,\overline{T\ x\ v};\overline{T'\ x'\ \mathrm{default}(T')};\mathbf{destiny}\ \mathtt{f}, s\,\}}$$

$$\textsc{CBind}\quad \frac{C = \mathbf{class}\ \mathtt{C} \ldots \{\, \mathbf{critical}\ T\ \mathtt{m}(\overline{T}\ x)\{\,\overline{T'\ x'}\ s\,\}\ \ldots\,\} \quad s' = \mathbf{nb}_{cr} = \mathbf{nb}_{cr} + 1; s; \mathbf{nb}_{cr} = \mathbf{nb}_{cr} - 1}{\mathrm{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},C) = \{\,\overline{T\ x\ v};\overline{T'\ x'\ \mathrm{default}(T')};\mathbf{destiny}\ \mathtt{f}, s'\,\}}$$

Rule SBind corresponds to the normal semantics of the bind function, which simply creates a task using the code of the method, and a local store σ with the parameter of the method, its local variables mapped to a default value of the right type using the annex function 'default', and the **destiny** variable that stores a reference to the task's future. The rule CBind is the one used to bind a critical function. Basically, this rule extends the code of the method with some code for critical section management: the first thing a critical method does is to

increment the field $\mathbf{nb}_{cr}$, opening the critical section, and the last thing it does is to decrement the field, thus closing it. The store σ is created as in the SBind rule. Rules SyncCallLocal and SyncCallGlobal give the semantics of synchronous method calls, and are constructed as the combination of the rules AsyncCall and Bind, with three differences: i) the method call is replaced by a **get** on the new task future; ii) the current task is put back in the object's task pool, while the new task grabs the cog lock and starts executing; iii) the code of the new task is extended with a $\mathbf{cont}(\mathtt{f}')$ statement that will automatically start the old task at the end of the new one.

Rule NewObject creates an object with a unique identifier $\mathtt{o}'$. The object's fields are given default values by the function atts described in the following rule:

$$\textsc{Atts}\quad \frac{C = \mathbf{class}\ \mathtt{C}(\overline{T\ x}) \dots \{\, \overline{T'\ x'}\ \overline{\mathbf{port}\ T''\ x''} \dots \,\}}{\mathrm{atts}(C, \overline{v}, \mathtt{o}, \mathtt{c}) = \begin{pmatrix} \varepsilon; \overline{T\ x\ v}; \overline{T'\ x'\ \mathrm{default}(T')}; \overline{T''\ x''\ \mathrm{default}(T'')} \\ \mathbf{this}\ \mathtt{o}; \mathbf{class}\ \mathtt{C}; \mathbf{cog}\ \mathtt{c}; \mathbf{nb}_{cr}\ 0 \end{pmatrix}}$$

The fields of an object consist of the parameter of the class, initialized with the actual parameter of the object creation; the declared fields and ports initialized with a default value; the **this** field referencing the object itself; a reference to the class and the cog of the object; and the field $\mathbf{nb}_{cr}$ counting the number of open critical sections, and initialized to 0. To instanciate the remaining fields, the task set p, corresponding to the class constructor, is queued. Formally, this task is defined by the 'init' function, defined by the following rules, that are identical to their core ABS version:

$$\textsc{Init}\quad \frac{C = \mathbf{class}\ \mathtt{C} \dots \{\, \mathrm{Unit}\ \mathit{init}() \{\, \overline{T\ x}; s \,\} \dots \,\}}{\mathrm{init}(C) = \{\, \overline{T\ x\ \mathrm{default}(T)}, s \,\}} \qquad \textsc{InitEmpty}\quad \frac{\mathit{init} \notin C}{\mathrm{init}(C) = \varepsilon}$$

The task p is not directly scheduled in order to uphold the cog invariant (i.e. that only one object per cog is active), hence any scheduling policy managing the task set Q must take care to always schedule this task with the highest priority. Rule NewCog is similar to rule NewObject, except that a fresh cog is created with $\mathtt{o}'$ as its only object.

Rule AsyncEnd terminates an asynchronous method call: when its statement is reduced to a simple **skip**, the task is simply destroyed. Rules SyncEndLocal and SyncEndGlobal terminate synchronous method call when its statement is reduced to $\mathbf{cont}(\mathtt{f}')$, the task is destroyed and the task whose future is $\mathtt{f}'$ (i.e. the caller of the finishing one) is automatically started again. Finally, rule Get deferences the future $\mathtt{f}$ if it contains a value (note that if the future's value is still undefined, the **get** operator is blocked).

Statements The rules describing the operational semantics of statements, presented in Figure 11, are the same as in core ABS, extended with two rules for the rebind. Rule Skip consumes a **skip** statement, and produces no effet. Rule

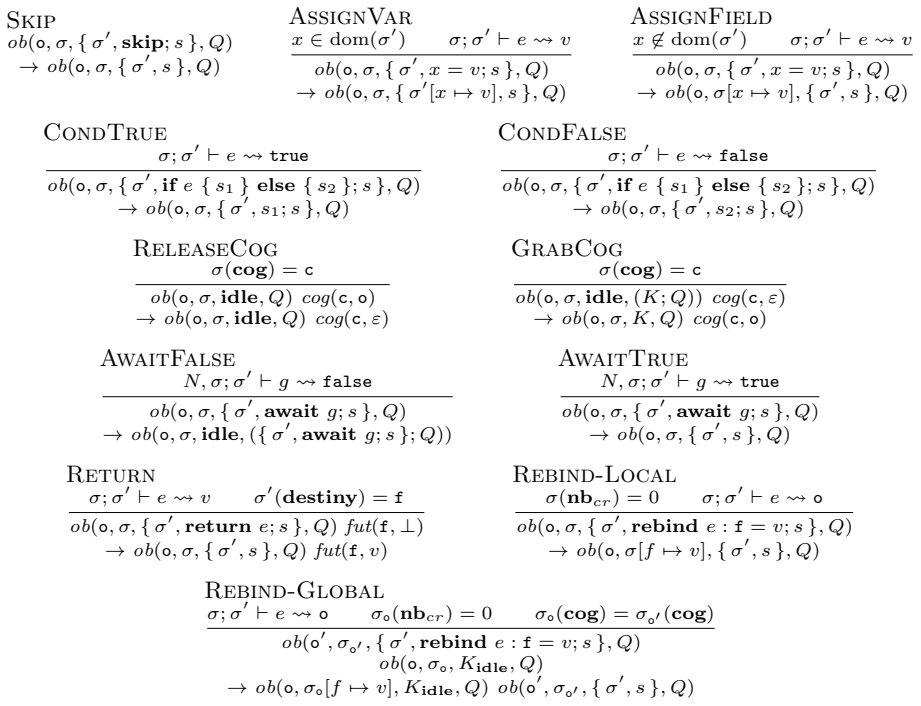

Figure 11: Semantics of Statements

AssignVar updates the value of a local variable of the current task, while rule AssignField updates the values of one of the object's fields. Rules CondFalse and CondTrue reduces the **if** statement, choosing a different branch depending on the value obtained by evaluating the expression e. Rule ReleaseCog makes the cog idle if its active object is idle. Rule GrabCog choose aribtrarily a task K from the task pool, grab the cog, and start executing K. These two previous rules ensure that there is at most one active object per cog, and that when a cog is idle, a task ready to be executed will always by scheduled. Rule AwaitFalse stops the execution of a task with an **await** statement whose guard is false, and put it in the object's task pool. Rule AwaitTrue consumes the **await** statement of the current task, when its guard is true, and continue the task execution. Rule Return places the return value into the call's associated future.

Rule Rebind-Local is applied when an object rebinds one of its own ports: it first checks that the object is not in a critical section by testing the special field $\mathbf{nb}_{cr}$ for zero and then updates the value of the field. Finally, rule Rebind-Global is applied when an object rebinds a port of another object and is similar to the previous one.

3.4 Properties

Important properties that show the correctness of our component model for port rebinding are: (i) a port of an object can never be modified while one of its critical methods is under execution (this property is a consequence of the reduction rules for rebinding: the execution of the rebind expression can only occur when $\mathbf{nb}_{cr}$ is 0) and (ii) when await statements (or synchronous calls) are not used inbetween, modification of multiple ports of multiple objects o_i is atomic (due to cooperative concurrency in the object group model the cog lock is never released inbetween) thus, if they preceded by an await statement on new guards $|o_i|$ (and all o_i objects belong to the same cog as the object modifying their ports), all such rebindings are guaranteed to take effect. In the following we will provide formal theorems showing that these properties actually hold true.

We first show that property (i) holds true. In the following we will use $conf_{init}$ to denote the set of configurations N which are reachable from the initial configuration.

Definition 3. *Let* $\mathtt{o}, \mathtt{o}'$ *be objects. An* $\mathtt{o}'$*-rebinding-*$\mathtt{o}$ *configuration is a configuration* N *such that* $ob(\mathtt{o}', \sigma, \{\, \sigma', \mathbf{rebind}\ e : \mathtt{f} = e'; s \,\}, Q) \in N$ *and* $\sigma; \sigma' \vdash e \rightsquigarrow o$.

Definition 4. *An object* $\mathtt{o}$ *is rebindable by an object* $\mathtt{o}'$ *in a configuration* N *if* N *is a* $\mathtt{o}'$*-rebinding-*$\mathtt{o}$ *configuration, i.e.* $ob(\mathtt{o}', \sigma, \{\, \sigma', \mathbf{rebind}\ e : \mathtt{f} = e'; s \,\}, Q) \in N$ *such that* $\sigma; \sigma' \vdash e \rightsquigarrow o$*, and there exists* N' *such that:* $N \to N'$*,* $ob(\mathtt{o}', \sigma'', \{\, \sigma', s \,\}, Q) \in N'$ *and* $ob(\mathtt{o}, \sigma_{\mathtt{o}}, K_{\mathbf{idle}}, Q') \in N'$*, where* $\sigma_{\mathtt{o}}(\mathtt{f}) = v$*, with* $\sigma; \sigma' \vdash e' \rightsquigarrow v$.

Definition 5. *The number of critical tasks under execution by an object* $\mathtt{o}$ *in a configuration* N *such that* $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q) \in N$*,* $\#critical(\mathtt{o}, N)$*, is defined as follows. Considered tasks* K *under execution by* $\mathtt{o}$*, i.e. such that either* $K = K_{\mathbf{idle}}$ *or* $K \in Q$*, we have that* $\#critical(\mathtt{o}, N)$ *is the number of such tasks* K *that satisfy:* $K = \{\, \sigma', s; \mathbf{nb}_{cr} = \mathbf{nb}_{cr} - 1 \,\}$*, for some* s *which is not in the form* $\mathbf{nb}_{cr} = \mathbf{nb}_{cr} + 1; s'$*, or* $K = \{\, \sigma', \mathbf{nb}_{cr} = \mathbf{nb}_{cr} - 1 \,\}$.

Lemma 1. *The number of critical tasks under execution by an object* $\mathtt{o}$ *in a configuration* $N \in conf_{init}$ *such that* $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q) \in N$*,* $\#critical(\mathtt{o}, N)$*, is equal to the value of* $\sigma(\mathbf{nb}_{cr})$.

Proof. The assert is proven by induction on the length of any reduction sequence from the initial configuration (where both values are obviously 0). □

Theorem 1. *If the number of critical tasks under execution by an object* $\mathtt{o}$ *in a configuration* $N \in conf_{init}$*,* $\#critical(\mathtt{o}, N)$*, is greater than* 0 *then* $\mathtt{o}$ *is not rebindable by any object in* N.

Proof. By applying Lemma 1 we know that $\sigma(\mathbf{nb}_{cr}) > 0$, with $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q) \in N$, therefore the only two rules that could produce the desired reduction, i.e. rules Rebind-Local and Rebind-Global cannot be applied. □

We now show property (*ii*). We first need to present a proposition which characterizes atomic execution of multiple (composite) statements in the presence of cooperative concurrency of languages like ABS.

Definition 6. *A (composite) statement s is executed atomically if the following holds true. Let $N \in conf_{init}$ be such that it includes an object $\mathtt{o}$ executing a task with remaining code $s; s'$ and destiny $\mathtt{f}$, i.e. $ob(\mathtt{o}, \sigma, \{\, \sigma_t, s; s' \,\}, Q) \in N$, with $\sigma_t(\mathbf{destiny}) = \mathtt{f}$. Consider any possible reduction sequence $N = N_0 \to N_1 \to \ldots \to N_k$ from N. Let index i, with $0 \leq i \leq k$, be the minimum index such that in configuration N_i object $\mathtt{o}$ is executing a task with remaining code s'; let $i = k$ if there is no such a configuration. We have that for each configuration N_j, with $0 \leq j \leq i$: there is no task in execution by objects different from $\mathtt{o}$ that are in the same cog of $\mathtt{o}$, i.e. $ob(\mathtt{o}', \sigma', K_{\mathbf{idle}}, Q') \in N_j$ with $\mathtt{o}' \neq \mathtt{o}$ and $\sigma'(\mathbf{cog}) = \sigma(\mathbf{cog})$ implies $K_{\mathbf{idle}} = \mathbf{idle}$; and $\mathtt{o}$ is executing task with remaining code $s''; s'$ (or s') and the same destiny $\mathtt{f}$ (showing it to be the same task), i.e. $ob(\mathtt{o}, \sigma'', \{\, \sigma_{t'}, s''; s' \,\}, Q'') \in N_j$, with $\sigma_{t'}(\mathbf{destiny}) = \mathtt{f}$.*

Proposition 1. *Let s be a statement that does not include any "**await** g" statement or "$e.\mathtt{m}(\overline{e})$" expression, then s is executed atomically.*

Proof. Since object $\mathtt{o}$ in the initial configuration $N = N_0$ is not idle, we have that N must include $cog(\mathtt{c}, \mathtt{o})$, where $c = \sigma(\mathbf{cog})$. This because the only way for an object $\mathtt{o}$ to become non-idle is to grab the lock of its cog c via rule GrabCog or to directly receive it from another non-idle object in the same cog via rule SyncCallGlobal which then becomes idle, thus turning $cog(\mathtt{c}, o_\varepsilon)$ into $cog(\mathtt{c}, \mathtt{o})$ (in the initial configuration and in every subsequent one there is at most a non-idle object for every cog that is the one holding the cog lock). Since the only way to release the cog lock, so to produce $cog(\mathtt{c}, \varepsilon)$ needed by rule GrabCog, is to apply rule ReleaseCog to the object hodling the lock and such a rule, in turn, needs the object to be idle, we would need such an object to execute the await statement via rule AwaitFalse, which is the only way to turn an object into the idle state before reaching the end of the code of the active task. However s does not include neither await statements nor synchronous calls, hence rules AwaitFalse and SyncCallGlobal cannot originate reductions from $N = N_0$ so also for N_1 we have that $\mathtt{o}$ is not idle and it must include $cog(\mathtt{c}, \mathtt{o})$. By the same reasoning also $N_2, \ldots, N_i$ satisfy the same property. Now also notice that, according to the semantic rules the only way for waiting tasks of a non-idle object to become the active one is for the object to release the lock of its cog c and to grab it again via rule GrabCog or to perform a local synchronous call via rule SyncCallLocal. However, as we observed, since rule AwaitFalse cannot originate reductions from $N = N_0$ to N_i, the lock release cannot happen. Moreover, since s does not include synchronous calls also the rule SyncCallLocal cannot originate reductions in these configurations. Thus the task of o which is active in $N = N_0$ must be the same as that that is active in N_i, and since no rule can modify the value of the destiny of a task once produced, we have $\sigma_{t'}(\mathbf{destiny}) = \sigma_t(\mathbf{destiny}) = \mathtt{f}$. □

A direct consequence of Proposition 1 is that rebinding of several ports is executed atomically, provided that no await statements (or synchronous calls) are used inbetween. Formally, s including multiple **rebind** $e : x = e'$ statements, but no "**await** g" statement or "$e.\mathtt{m}(\overline{e})$" expression, is executed atomically.

Definition 7. *A configuration N' is reached by partially executing statement s in object $\mathtt{o}$ from a configuration $N \in conf_{init}$ if the following holds true. N is such that it includes an object $\mathtt{o}$ executing a task with remaining code $s; s'$, i.e. $ob(\mathtt{o}, \sigma, \{\, \sigma_t, s; s' \,\}, Q) \in N$ and suppose $\sigma_t(\mathbf{destiny}) = \mathtt{f}$. There exists a reduction sequence $N = N_0 \to N_1 \to \ldots \to N_k$ from N such that: there is no configuration N_j, with $0 \leq j \leq k$, where object $\mathtt{o}$ is executing a task with remaining code s', i.e. $ob(\mathtt{o}, \sigma', \{\, \sigma'_t, s' \,\}, Q) \in N_j$, and $\sigma'_t(\mathbf{destiny}) = \mathtt{f}$ (showing it to be the same task); and in N_k object $\mathtt{o}$ is executing a task with remaining code $s''; s'$, i.e. $ob(\mathtt{o}, \sigma'', \{\, \sigma''_t, s''; s' \,\}, Q) \in N_k$ and $\sigma''_t(\mathbf{destiny}) = \mathtt{f}$.*

In the following we will use, for simplicity, σ_o^N and ${\sigma'}_o^N$ to denote the field environment and the active task environment of object $\mathtt{o}$ in configuration N, i.e. $\sigma_o^N = \sigma$ and ${\sigma'}_o^N = \sigma'$ such that $ob(\mathtt{o}, \sigma, \{\, \sigma', s \,\}, Q) \in N$.

Theorem 2. *Let $s =$ **await** $\bigwedge_{i \in I} |e_i|; s'$ be a statement such that s' does not include any "**await** g" statement or "$e.\mathtt{m}(\overline{e})$" expression. Object $\mathtt{o}'$ is rebindable by object $\mathtt{o}$ in any o-rebinding-o' configuration reached by partially executing s in $\mathtt{o}$ from any $N \in conf_{init}$ such that, there exists $i \in I$ with $\sigma_o^N; {\sigma'}_o^N \vdash e_i \rightsquigarrow o'$ and object o' occurs in N and $\sigma_{o'}^N(\mathbf{cog}) = \sigma_o^N(\mathbf{cog})$.*

Proof. Since an o-rebinding-o' configuration N_k is reached from N, where the task executing s has already performed the **await** $\bigwedge_{i \in I} |e_i|$ statement, there exists N_i, with $1 \leq j \leq k$, such that N_i is the target of the reduction produced by the rule AwaitTrue. This implies that in N_i object $\mathtt{o}'$ is such that its special field $\mathbf{nb}_{cr}$ is 0 and that object $\mathtt{o}$ is executing a task with remaining code $s'; s''$ for some s'' . Since s' does not include any "**await** g" statement or "$e.\mathtt{m}(\overline{e})$" expression by Proposition 1 we have that it is executed atomically. Therefore in the configurations from N_i to N_k the task with destinity $\mathtt{f}$ is the only task of the cog of object $\mathtt{o}$ being executed. By Lemma 1 in N_i there no critical tasks under execution by object $\mathtt{o}'$, hence the same must hold true also in N_k and, again, by Lemma 1 we have that in N_k object $\mathtt{o}'$ is such that its special field $\mathbf{nb}_{cr}$ is 0. Therefore by rules Rebind-Local and Rebind-Global we have that $\mathtt{o}'$ is rebindable by object $\mathtt{o}$. □

We now present a direct corollary of Theorem 2 (and Proposition 1), where statement s' is taken to be a simple sequence of rebindings: they are shown to lead to a configuration where they all take effect.

Definition 8. *A configuration N' is reached by completely executing statement s in object $\mathtt{o}$ from a configuration $N \in conf_{init}$ if the same conditions as those in Definition 7 hold true, except that $0 \leq j < k$ replaces $0 \leq j \leq k$ and s' replaces $s''; s'$. A configuration N' is reached by completely executing statement s in object $\mathtt{o}$ from a configuration $N \in conf_{init}$ executing $s; s_1$ if we additionally take $s_1; s'$ to replace s'.*

Corollary 1. *Let* $s = \textbf{await } \bigwedge_{i \in I} |e_i|; s'$ *be a statement such that* $s' = \textbf{rebind } e'_0 : x_0 = e''_0; \textbf{rebind } e'_1 : x_1 = e''_1; .....; \textbf{rebind } e'_n : x_n = e''_n$*, for some* $n \geq 0$ *(*$s' = \textbf{rebind } e'_0 : x_0 = e''_0$ *in the case* $n = 0$*). Moreover suppose that a configuration* N' *is reached by completely executing statement* $\textbf{await } \bigwedge_{i \in I} |e_i|$ *in object* o *from a configuration* $N \in conf_{init}$ *executing* s *and that for all* j *with* $0 \leq j \leq n$*, considered* $\sigma_o^{N'}; {\sigma'}_o^{N'} \vdash e'_j \rightsquigarrow o_j$*, we have that:* N' *includes object* o_j *and* $\sigma_{o_j}^{N'}(\textbf{cog}) = \sigma_o^N(\textbf{cog})$*; there exists* $i \in I$ *such that* $\sigma_o^N; {\sigma'}_o^N \vdash e_i \rightsquigarrow o_j$*; and all pairs* (o_j, x_j) *are distinguished (we do not have multiple rebindings of the same port). Then there exists a configuration* N'' *reached by completely executing statement* s' *in object* o *from configuration* N'*, with* N'' *such that for all* j *with* $0 \leq j \leq n$ *we have that* $\sigma_{o_j}^{N''}(x_j) = v_j$ *with* $\sigma_o^{N'}; {\sigma'}_o^{N'} \vdash e''_j \rightsquigarrow v_j$*.*

4 Application to the Case Study

4.1 Implementation

ABS is a strict extension of the core ABS language with features like *Delta-Oriented Programming* [18] or *deployment components* [8] that we won't discuss in this paper. This language comes with a tool suite [22] that offers a compilation framework, a set of tools to analyse the code, an Eclipse IDE plugin and Emacs mode for the language. We implemented our extension of core ABS into the ABS compilation framework.

The ABS Compiler Figure 12 gives an overview of the current ABS compiler framework. The ABS compiler is informally divided in two parts: its *compiler front-end* translates textual ABS program into an internal representation and checks the models for syntax and semantic errors; its *compiler back-end* generates code from the models, targeting some suitable execution or simulation environment. Two different back-ends translate ABS programs into either Maude [4] or Java, which allow ABS models to be executed and analyzed on these platforms. The abstract syntax tree (AST) is the cornerstone of tool integration: it is the internal representation for ABS programs, and tools will typically reason about one or more such models or produce them. All tools will work on a common representation of the AST, which has the following benefits:

- *Reduced implementation costs*: individual tools need not know about concrete model syntax and type-checking.
- *Easier integration:* the output from one tool can be the input to another, or several tools can cooperate to produce results in the same format.

Modifying the ABS Compiler To implement our component model in ABS, we modified its compiler in several ways. We first modified its front-end in two ways: i) the AST of the language was extended with the new elements of the language: ports, critical methods, the new guard and the rebind statement;

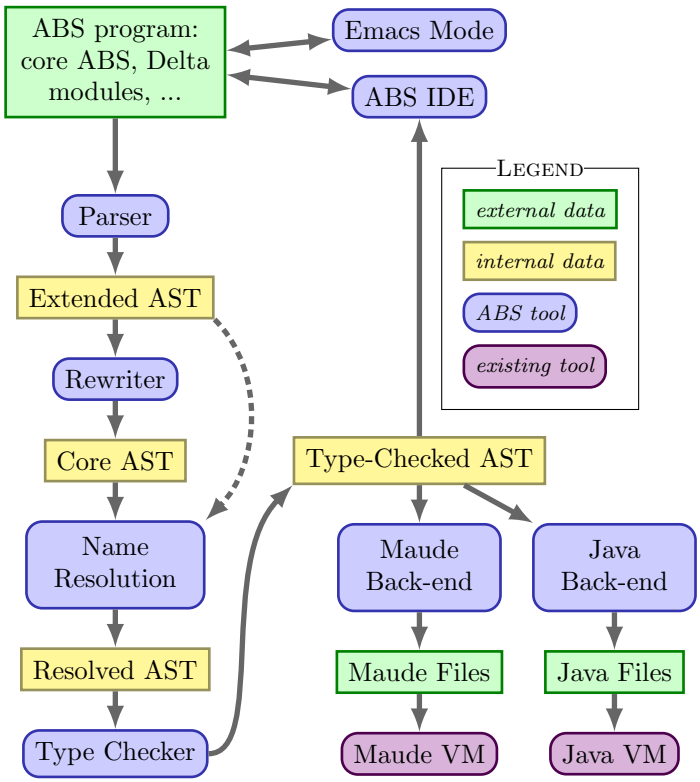

Figure 12: Overview of the ABS compiler framework.

ii) This extension of the AST was reflected in the parser with the addition to the language syntax of the different elements of our component model, as presented in Section 3.2.1. We also modified the maude back-end of the compiler, which was an exact translation of the core ABS semantics in maude. Our modification consisted of i) the addition of the rules for the new guard and for the rebind statement; ii) the modification of the 'bind' and 'atts' functions. The overall modifications to the compiler resulted in approximately 200 lines of java and maude code.

4.2 Switching Job Policies

In this section we show how to use the component model to model to switch between sequential and concurrent job policies during the runtime of FAS. We first record the following (safety) property (Definition 2): *The Replication System must be able to switch between sequential and concurrent replication jobs only when there is no more running session.*.

We first identify the following functional requirements:

1. record existing running sessions;

2. record the sessions that have been scheduled but has not been started;

3. deny further sessions from starting scheduled sessions.

We then encapsulate the behaviour to fulfil these requirements in the following interface `Policy`.

```
interface Policy {
  Unit scheduleAndMonitorJob(Schedule schedule);
  Unit runScheduled(List<Schedule> ss);
  List<Schedule> getScheduledJobs();
  Unit nextJob(Type tp);
  Unit setNext(Type tp);
  Unit prepareSwitch();
  Bool isShutdownRequested();
  Unit requestShutDown(); }
```

The methods exposed through this interface affect both scheduled and running sessions. The method `getScheduledJobs` then returns the list of already scheduled sessions, while `runScheduled` re-schedules all given scheduled sessions that have been stopped due to a policy switching. The method `prepareSwitch` alerts a policy switch has been requested. The method `Policy.scheduleAndMonitorJob` extends the original `SyncClient.scheduleJob` such that the method both creates a new ClientJob for the specified session, and monitors for the session to finish.

4.2.1 Sequential Job Policy

The following class `SequentialPolicy` implements the sequential job policy.

```
class SequentialPolicy(Network network, SyncClient client)
implements Policy {
  Bool next = True; Bool shutDown = False;
  Bool blocked = False; Bool finish = False;
  Unit scheduleAndMonitorJob(Schedule schedule) {
    await next || shutDown || blocked;
    if (blocked) { scheduled = Cons(schedule, scheduled);
    } else if (~shutDown && ~blocked) {
      this.setNext(stype(schedule)); client!makeJob(schedule);
      await finish; }}
  Unit nextJob(Type tp) { if (~shutDown) { next = True; finish = True; }}
  Unit setNext(Type tp) { next = False; finish = False; }
  Unit prepareSwitch() { blocked = True; }
  Unit runScheduled(List<Schedule> ss) { while (ss != Nil) { .. } }
  List<Pair<JobType, Schedule>> getScheduledJobs() { return scheduled; }
  Bool isShutdownRequested() { return shutDown; }
  Unit requestShutDown() { .. }}
```

This implementation has a flag `blocked`, which is set to `True` if a request has been made to switch job policy. The definition of `scheduleAndMonitorJob` is similar to `SyncClient.scheduleJob` shown earlier except the following: On invocation, the method not only first waits for either the next session may proceed (`next == True`), or the SyncClient has been requested to shut down (`shutDown == True`), it also waits until a request has been made to switch job policy (`blocked == True`). The method goes on recording the scheduled sessions if a request has been made to switch job policy. The method proceeds to create a ClientJob for the scheduled session if neither the SyncClient is shutting down *nor* a request has been made to switch job policy. After creating a ClientJob, this method *waits* for the session to finish (`finish == True`). This implementation ensures that no session is running if the method is not being executed.

4.2.2 Concurrent Job Policy

The following class `ConcurrentPolicy` implements the concurrent job policy.

```
class ConcurrentPolicy(Network network, SyncClient client)
implements Policy {
  Map<Type,Bool> nexts = EmptyMap; Map<Type,Bool> fs = EmptyMap;
  Bool shutDown = False; Bool blocked = False;
  Unit scheduleAndMonitorJob(Schedule schedule) {
    Type tp = stype(schedule);
    await (lookupDefault(nexts,tp,True) || shutDown || blocked);
    if (blocked) { scheduled = Cons(schedule, scheduled);
    } else if (~shutDown && ~blocked) {
      this.setNext(schedule); client!makeJob(schedule);
      await lookupDefault(fs,tp,False); }}
  Unit nextJob(Type tp) {
    if (~shutDown) {
      nexts = put(nexts,tp,True); fs = put(fs,tp,True); }}
  Unit setNext(Type tp) {
    nexts = put(nexts,tp,False); fs = put(fs,tp,False); }
  Unit prepareSwitch() { blocked = True; }
  Unit runScheduled(List<Schedule> ss) { while (ss != Nil) { .. } }
  List<Pair<JobType, Schedule>> getScheduledJobs() { return scheduled; }
  Bool isShutdownRequested() { return shutDown; }
  Unit requestShutDown() { .. }}
```

It records two maps `nexts` and `fs`. The first map records for which replication type a session may start, and the second map records for which replication type a session is running. The method `scheduleAndMonitorJob` then ensures to start a session if the session's replication type does not already have a running session (`lookupDefault(nexts,type,True) == True`. After creating a ClientJob, the method ensures to wait for the session is terminate before terminating the method `lookupUnsafe(fs,type) == True`. This implemenation then also ensures that no session is running if the method `scheduleAndMonitorJob` is not

being executed.

4.2.3 Switching policy at Runtime

The following class `PolicySyncClient` implements a `SyncClient` that allows policy switching at runtime.

```
class PolicySyncClient(Network network) implements SyncClient {
  port Policy policy = ..;
  Unit switch(Bool seq) {
    Policy op = policy;
    Policy np = null;
    if (seq) { np = new SequentialPolicy(network, this); }
    else { np = new ConcurrentPolicy(network, this); }
    await |this|;
    rebind this:policy = np;
    List<Schedule> scheduled = op.getScheduledJobs();
    policy.runScheduled(scheduled); }
  Bool isShutdownRequested() { return policy.isShutdownRequested(); }
  Unit nextJob(Schedule s) { policy.nextJob(s); }
  Unit requestShutDown() { policy.requestShutDown(); }
  critical Unit scheduleJob(JobType jb,Schedule s) {
    policy.scheduleAndMonitorJob(jb,s); }}
```

It defines the current policy as a **port**. All methods that affect scheduling and running of replication sessions are then redirected to synchronous method calls to the policy object. The method `switch` then implements the switching of job policy. The input argument of the method specifies whether the sequential or concurrent policy should be instantiated. On invocation the method waits for the policy object to be *safe*, that is, no **critical** methods of the `SyncClient` object are being executed (**await** `|this|`), by annotating the method `scheduleJob` as **critical**, this guarantees that the condition holds if no session is running. After the policy object is safe, the method *rebinds* the current policy to the new policy, acquires all existing scheduled but waiting sessions `policy.getScheduledJobs()` and schedules them `policy.runScheduled()`.

5 Towards Mobility

We have presented an approach to safe dynamic reconfiguration of components in an object oriented setting. Unlike objects, components are associated with location. In this section we introduce a notion of locations to our component model and provide a simple example to illustrate how such an addition can be used to express dynamic addition or removal of code, as well as distribution of a program over several computing resources. Locations themselves are structured into trees according to a sublocation relation, such that we always have a root location for an ABS program, and that the leaves of the tree are the object groups, considered as special cases of locations. Locations are introduced in the

syntax of our previous calculus with the following extension:

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

First, we add the possibility to create a new location with a command **new loc**; then we add the possibility to retrieve the group of an object with the command $\mathit{group}(e)$; and we add the possibility of modifying the father of a location with the command **move** e' **in** e which puts the location e inside the location e'. Technically, we also introduce a new data type for location values, called `location`.

5.1 Examples

In the following we consider a client that utilizes one or more services to execute a service workflow. We use locations to expressive the movement of the client from one location to another. The client has a set of output ports for connection to the services at the current client's location for executing the service workflow. As a result the client 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.

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

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

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

We represent the workflow provider 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 `ServiceFrontEnd` which: changes the ports in the `ServiceFrontEnd` object (possibly performing the service discovery enquiring the local service registry, see Example 2).

```
interface ServiceA { ... }
interface ServiceB { ... }

interface ServiceFrontEnd {
  port ServiceA a;
  port ServiceB b;
  critical Unit workflow();
}

class Client(Location l, ServiceFrontEnd s, ServiceA a, ServiceB b) {

  Unit changeLocation(Location l2, ServiceA a2, ServiceB b2) {
    await |s|;
    Location myGroup = group(this);
    move myGroup in l2;
    rebind s:a = a2;
    rebind s:b = b2;
  }

  Unit init() {
    this.changeLocation(l, a, b);
  }
}
```

Figure 13: Moving a Client

Example 2 In this example we also model the local registry component for each location, providing links to the services at that location, 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 services in the new location.

5.2 Semantics

Similarly to Section 3.3, our location model is added to the semantics of core ABS by both extending its runtime syntax, and its set of reduction rules.

5.2.1 Runtime syntax

Figure 15 presents our extension of the runtime syntax. An hierarchy statement $(\mathsf{c}_\perp, \mathsf{c})$ states that the location c is a child of the location $\mathsf{c}_\perp$ ($\perp$ being the name

```
interface ServiceA { ... }
interface ServiceB { ... }

interface Register {
  ServiceA discoverA();
  ServiceB discoverB();
}

interface RootRegister {
  Register discoverR(location l);
}

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

class Client(Location l, ServiceFrontEnd s, RootRegister rr) {

  Unit changeLocation(Location l2) {
    Fut<Register> fr = rr!discoverR(l2); await fr?; Register r = fr.get;
    await |s|;
    Location myGroup = group(this);
    move myGroup in l2;
    Fut<ServiceA> fa = r!discoverA();
    rebind s:a = fa.get;
    Fut<ServiceB> fb = r!discoverB();
    rebind s:b = fb.get;
  }

  Unit init() {
    this.changeLocation(l);
  }

}
```

Figure 14: Moving a Client and rebinding to local services

of the top level location).

5.2.2 Reduction relation

We extend that reduction relation $\rightarrow$ over configurations, $N \rightarrow N'$ defined in Section 3 in Figure 16. First, we extend the reduction definition with three

$$
\begin{array}{rcl}
v & ::= & \ldots \mid \mathtt{c}_\perp \\
N & ::= & \ldots \mid (\mathtt{c}_\perp, \mathtt{c}) \\
\mathtt{c}_\perp & ::= & \perp \mid \mathtt{c}
\end{array}
$$

Figure 15: Runtime Syntax Extension; here $\mathtt{c}$ are location names

LocMove
$$(\mathtt{c}_\perp, \mathtt{c})\ ob(\mathtt{o}, \sigma, \{\, \sigma', \mathbf{move}\ \mathtt{c}\ \mathbf{in}\ \mathtt{c}'_\perp; s \,\}, Q) \rightarrow (\mathtt{c}'_\perp, \mathtt{c})\ ob(\mathtt{o}, \sigma, \{\, \sigma', s \,\}, Q)$$

NewLocation
$$\frac{\mathtt{c}\ \text{fresh}}{ob(\mathtt{o}, \sigma, \{\, \sigma', x = \mathbf{new}\ \mathbf{loc}; s \,\}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\, \sigma', x = \mathtt{c}; s \,\}, Q)\ (\perp, \mathtt{c})}$$

GetGroupLocal
$$\frac{\sigma; \sigma' \vdash e \rightsquigarrow \mathtt{o}}{ob(\mathtt{o}, \sigma, \{\, \sigma', x = group(e); s \,\}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\, \sigma', x = \sigma(\mathbf{cog}); s \,\}, Q)}$$

GetGroupGlobal
$$\frac{\sigma; \sigma' \vdash e \rightsquigarrow \mathtt{o}'}{ob(\mathtt{o}, \sigma, \{\, \sigma', x = group(e); s \,\}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, K_{\mathbf{idle}}, Q_{\mathtt{o}'}) \rightarrow ob(\mathtt{o}, \sigma, \{\, \sigma', x = \sigma_{\mathtt{o}'}(\mathbf{cog}); s \,\}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, K_{\mathbf{idle}}, Q_{\mathtt{o}'})}$$

Figure 16: Operational Semantics Extension

reduction rules that define the semantics of the **subloc** operator. Rule LocMove moves a location $\mathtt{c}$ (initially put inside the location $\mathtt{c}_\perp$) inside another location $\mathtt{c}'_\perp$. Rule NewLocation creates a new location with a fresh name $\mathtt{c}$, and put it in the top level location $\perp$. Finally, rules GetGroupLocal and GetGroupGlobal gives the semantics of the *group* function, which basically look into the store of its object in parameter, and extract from it the name of the object's cog.

6 Conclusion

Components are an intuitive tool to achieve unplanned dynamic reconfigurations. In this paper we proposed a novel object-oriented component model for safe dynamic reconfigurations of components. We introduced three new primitives to the core ABS language: instance output **port** references that are distinct from instance fields and correspond to the objects' dependencies to the environment that can be modified only when the object is in a *safe* state; methods may be **critical**, denoting the fact that the object is not in a safe state while one or more of these methods are executing; and a new **await** statement on objects with critical methods such that it becomes possible to wait for all executions of critical methods to finish execution, before rebinding an port to a new object. We have provided our component model with a formal operational semantics in terms of reduction relations and formalized the concept of safe and atomic rebinding of ports. 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's 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).

- The separation of output ports and fields 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 3.2

- The separation of output ports and fields could be profitable in formal reasoning, in particular when developing static analysis techniques.

- The presence of output ports may be useful in the deployment phase of a system, facilitating, for instance, the connection to local communication resources.

We have demonstrated the applicability of the component model via an *industrial* case study in which we were able to model how the policy of executing replication jobs can be changed at runtime. The complete ABS model of the Replication System can be found at `http://www.hats-project.eu/sites/default/files/replication-system-dynamic-policy.zip`.

Furthermore, we have introduced a model of a hierarchy of locations to the component model to describe the movement of components. 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).

Mobility in Virtualized Environments One example in which the ability to safely move components between locations becomes important is when components require access to varying amounts of resources. This is particularly apparent in a virtualized environment such as the cloud. Specifcally virtualization gives access to elastic amounts of resources to services on the application-level; for example, the processing capacity allocated to a service may be changed according to demand. In our case study, FAS is today deployed as a service (SaaS) over virtualized resources that provide the necessary elasticity in order to cater for the increasing demands on more flexible SLA. To be able to model the behaviour of FAS running on virtualized resources such as the cloud, we require the model of locations. Using this model, we hope to not only allow the Replication System to safely switch job policies at runtime but also to safely *move* jobs between virtualized instances in order to acquire the right amount of resources to execute them.

Future Work The component model has been implemented in ABS and can be executed in the Maude back-end of the ABS compiler. As part of future work, we would like to extend the Java back-end of the ABS compiler to support our

component model. We would consider the recent work on MetaABS [16] as a Java implementation vehicle to dynamically rebind output ports. MetaABS is a flexible meta-programming facility for the ABS language, and is supported by the Java back-end. It allows introspection and manipulation of running code.

Acknowledgements

We wish to thank Behrooz Nobakht, Andreas Kohn and Stephan Schroevers for proof reading this paper.

References

[1] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[2] 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.

[3] G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal calculus. *Inf. Comput.*, 201(1), 2005.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[5] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. OpenCOM v2: A Component Model for Building Systems Software. In *Proceedings of IASTED Software Engineering and Applications (SEA '04)*, 2004.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[7] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.

[8] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2011.

[9] S. Lenglet, A. Schmitt, and J.-B. Stefani. Howe's Method for Calculi with Passivation. In M. Bravetti and G. Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, volume 5710 of *LNCS*, pages 448–462. Springer-Verlag, 2009.

[10] F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM. Trans. Prog. Languages and Systems, vol. 25, no 1*, 2003.

[11] M. Lienhardt, I. Lanese, M. Bravetti, D. Sangiorgi, G. Zavattaro, Y. Welsch, J. Schäfer, , and A. Poetzsch-Heffter. A component model for the abs language. In *Formal Methods for Components and Objects (FMCO) 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 165–185. Springer Berlin / Heidelberg, 2010.

[12] M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/K: A kernel language for component-based open programming. In *GPCE'07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

[13] H. Miranda, A. S. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*. IEEE Computer Society, 2001.

[14] F. Montesi and D. Sangiorgi. A model of evolvable components. In M. Wirsing, M. Hofmann, and A. Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 2010.

[15] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.

[16] R. Muschevici, J. Proença, and D. Clarke. MetaABS and dynamic model updates. Submitted for publication.

[17] OSGi Alliance. *Osgi Service Platform, Release 3*, 2003.

[18] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, Sept. 2010.

[19] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

[20] 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.

[21] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements, 2006.

[22] P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.