A Type System for Components

Ornela Dardha, Elena Giachino, Michael Lienhardt

A Type System for Components*

Ornela Dardha[1] Elena Giachino[1] Michaël Lienhardt[2]

[1] INRIA Focus Team / University of Bologna, Italy
{dardha, giachino}@cs.unibo.it
[2] University of Paris Diderot, France
lienhar@inria.fr

Abstract. In modern distributed systems, dynamic reconfiguration, i.e., changing at runtime the communication pattern of a program, is challenging. Generally, it is difficult to guarantee that such modifications will not disrupt ongoing computations. In a previous paper, a solution to this problem was proposed by extending the object-oriented language ABS with a component model allowing the programmer to: *i*) perform updates on objects by means of *communication ports* and their *rebinding*; and *ii*) precisely specify when such updates can safely occur in an object by means of *critical sections*. However, improper rebind operations could still occur and lead to runtime errors. The present paper introduces a type system for this component model that extends the ABS type system with the notion of ports and a precise analysis that statically enforces that no object will attempt illegal rebinding.

1 Introduction

In modern complex distributed scenarios, unplanned dynamic reconfiguration, i.e., changing at runtime the communication pattern of a program, is challenging as it is difficult to ensure that such modifications will not disrupt ongoing computations. In [14] the authors propose to solve the problem by integrating notions coming from component models [2–4, 8] within the actor-based *Abstract Behavioral Specification* programming language (ABS) [13]. ABS is designed for distributed object-oriented systems and integrates concurrency and synchronization mechanisms to solve data races. Actors, called *cog*s or simply *group*s, are dynamic collections of collaborating objects. Cogs offer consistency by guaranteeing that at most one method per cog is executing at any time. Within a cog, objects collaborate using (*synchronous*) method calls and *collaborative concurrency* with the **suspend** and **await** operations which can suspend the execution of the current method, and thus allow another one to execute. Between cogs, collaboration is achieved by means of *asynchronous* method calls that return *future*, i.e., a placeholder where the result of the call is put when its computation finishes.

* This research is partly funded by the EU project FP7-231620 HATS and by the French National Research Agency (ANR), projects REVER ANR 11 INSE 007

On top of the ABS language, [14] adds the notions of *ports*, *bindings* and *safe state* to deal with dynamic reconfiguration. Ports define variability points in an object and can be *rebound* (i.e., modified) from outside the object (on the contrary, fields, which represent the inner state of the object, can only be modified by the object itself). To ensure consistency of the **rebind** operation, [14] enforces two constraints on its application: *i*) it is only possible to rebind an object's port when the object is in a *safe state*; and *ii*) it is only possible to rebind an object's port from *any* object within the *same* cog. Safe states are modeled by annotating methods as **critical**, specifying that while one or more **critical** methods are executing, the object is *not* in a safe state. The resulting language offers a consistent setting for dynamic reconfigurations, which means performing modifications on a program at runtime while still ensuring consistency of its execution. Consistency is based on two constraints: both synchronous method calls and rebinding operations must involve two objects in the same cog. These constraints are enforced at runtime; therefore, programs may encounter unexpected runtime errors during their execution.

In this paper, we define a type system for the component model that statically ensures the legality of both synchronous method calls and port rebindings, guaranteeing that well-typed programs will always be consistent. Our approach is based on a static tracking of group membership of the objects. The difficulty in retrieving this kind of information is that cogs as well as objects are dynamic entities. Since we want to trace group information statically, we need a way to identify and track every group in the program. To this aim, we define a technique that associates to each group creation a fresh *group name*. Then, we keep track of which cog an object is allocated to, by associating to each object a *group record*. The type system checks that objects indeed have the specified group record, and uses this information to ensure that synchronous calls and rebindings are always performed locally to a cog. The type system is proven to be sound with respect to the operational semantics. We use this result to show that well-typed programs do not violate consistency during execution.

Motivating example. In the following we present a running example that gives a better understanding of the ABS language and the component extension, and most importantly, motivates our type system. Consider the following typical distributed scenario: suppose that we have several clients working together in a specific workflow and using a central server for their communications. Updating the server is a difficult task, as it requires to update its reference in all clients at the same time in order to avoid communication failures.

First, in Fig. 1 we consider how this task is achieved in ABS. The programmer declares two interfaces `Server` and `Client` and a class `Controller`. Basically, the class `Controller` updates the server in all the clients $\texttt{c}_i$ by synchronously calling their setter method. All the clients are updated at the same time: since they are in the same cog as the controller they cannot execute until the execution of method `updateServer` has terminated.

```
interface Server { ... }
interface Client { Unit setServer(Server s); ... }

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

 Unit updateServer(Server s2) {
  c1.setServer(s2);
  c2.setServer(s2);
      ...
  cn.setServer(s2);
 }}
```

Fig. 1. Workflow in ABS.

However, this code does not ensure that the update is performed when the clients are in a safe state. This can lead to inconsistencies because clients that are using the server are not aware of the modification taking place. This problem can be solved by using the notions of **port** and **rebind** [14] as shown in Fig. 2. Here, the method `updateServer` first waits for all clients to be in a safe state (**await** statement performed on the conjunction of all clients) and then updates their reference one by one (**rebind** server `s` which is declared to be a **port**).

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

class Controller {
Client c1, c2, ... cn;
        ...
 Unit updateServer(Server s2) {
  await ‖c1‖ ∧ ‖c2‖ ∧ ... ∧ ‖cn‖;
  rebind c1.s = s2;
  rebind c2.s = s2;
        ...
  rebind cn.s = s2;
 }}
```

Fig. 2. Workflow using the Component Model.

However, runtime errors can still occur. For instance, if the clients and the controller are not in the same cog, the update will fail. Consider the code in Fig. 3. Method `main` instantiates classes `Client` and `Controller` –and possibly other classes, like `Server`, present in the program– by creating objects $\texttt{c}_1,\texttt{c}_2,\ldots,\texttt{c}_n,\texttt{c}$. These objects are created in the same cog by the **new** command,

except for client c_1, which is created and placed in a new cog by the **new cog** command. Now, suppose that the code in Fig. 2 is executed. At runtime, the program will check if the controller and the client belong to the same cog to respect the consistency constraints on rebinding. In case of c_1 this check will fail by leading to a runtime error.

The present paper addresses this problem in order to avoid these runtime errors and the overhead in dealing with them. We present a type system that tracks cog membership of objects thus permitting to typecheck only programs where rebinding is consistent. So, the code presented above would not typecheck, as shown in § 3, thus discarding the program at compile time instead of leading to a runtime error.

```
Unit main () {      ...
        Client c1 = new cog Client (s);
        Client c2 = new Client (s);
                    ...
        Client cn = new Client (s);
        Controller c = new Controller (c1, c2, ... cn);
}
```

Fig. 3. Client and Controller objects creation.

Roadmap. The rest of the paper is structured as follows: § 2 introduces the calculus, types and terms; § 3 presents our type system and its properties; and § 4 concludes the paper and discusses future and related works.

2 The calculus

In this section we present the calculus underlying our approach, which is a component extension of the ABS language[3]. We present formally only the syntax of the calculus which is necessary for specifying the type system. We already gave some intuitions about the operational semantics of the calculus in the introduction and through the example, whereas for the formal definition we refer to the original paper [14] and the extended version of this paper [9].

The syntax of the calculus is given in Fig. 4 and corresponds to the original one, except for types, which are here extended in order to store also group information. This syntax is based on several categories of names: `I` and `C` range over interface and class names; `V` ranges over type variables for polymorphism; `G`

[3] For the sake of readability, the calculus we consider is a subset of [14]. The notion of *location* has been dropped, since it is orthogonal to ports and rebinding. The validity of our approach and of our type system still holds for the full calculus.

P	$::= \overline{Dl}\ \{\ s\ \}$	Program
Dl	$::= D \mid F \mid I \mid C$	Declarations
T	$::= \mathtt{V} \mid \mathtt{D}[\langle\overline{T}\rangle] \mid (\mathtt{I}, \mathtt{r})$	Type
$\mathtt{r}$	$::= \bot \mid \mathtt{G}[\overline{f : T}] \mid \alpha \mid \mu\alpha.\mathtt{r}$	Record
D	$::= \textbf{data}\ \mathtt{D}[\langle\overline{T}\rangle] = \mathtt{Co}[(\overline{T})] \vert \overline{\mathtt{Co}[(\overline{T})]};$	Data Type
F	$::= \textbf{def}\ T\ \mathtt{fun}[\langle\overline{T}\rangle](\overline{T\ x}) = e;$	Function
I	$::= \textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}$	Interface
C	$::= \textbf{class}\ \mathtt{C}\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \overline{M}\ \}$	Class
Fl	$::= [\textbf{port}]\ T\ x$	Field Declaration
S	$::= [\textbf{critical}]\ (\mathcal{G}, \mathtt{r})\ T\ \mathtt{m}(\overline{T\ x})$	Method Header
M	$::= S\ \{\ s\ \}$	Method Definition
s	$::= \textbf{skip} \mid s;s \mid T\ x \mid x = z \mid \textbf{await}\ g$	Statement
	$\mid \textbf{if}\ e\ \{\ s\ \}\ \textbf{else}\ \{\ s\ \} \mid \textbf{while}\ e\ \{\ s\ \} \mid \textbf{return}\ e$	
	$\mid \textbf{rebind}\ e.p = z$	
z	$::= e \mid \textbf{new}\ [\textbf{cog}]\ \mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid \textbf{get}(e)$	Expression with Side Effects
e	$::= v \mid x \mid \mathtt{fun}(\overline{e}) \mid \textbf{case}\ e\ \{\overline{p \Rightarrow e_p}\}$	Expression
v	$::= \textbf{null} \mid \mathtt{Co}[(\overline{v})]$	Value
p	$::= \_ \mid x \mid \textbf{null} \mid \mathtt{Co}[(\overline{p})]$	Pattern
g	$::= x \mid x? \mid \|x\| \mid g \wedge g$	Guard

Fig. 4. Core ABS Language and Component Extension.

ranges over cog names, which will be explained thoroughly in § 3; `D`, `Co` and `fun` range respectively over data type, constructor and function names; `m`, f and p range respectively over method, field and port names (in order to have a uniform presentation, we will often use f for both fields and ports); and x ranges over variables, with the addition of the special variable **this** indicating the current object. For the sake of readability, we use the following notations: an overlined element corresponds to any finite, possibly empty, sequence of such element; and an element between square brackets is optional.

A program P consists of a sequence of declarations ended by a main block, namely a statement s to be executed. Declarations include data type declarations D, function declarations F, interface declarations I and class declarations C. A type T can be: a type variable `V`; a data type D like `Bool` or futures $\mathtt{Fut}\langle T\rangle$, used to type data structures; or a pair consisting of an interface name `I` and a *record* $\mathtt{r}$ to type objects. Note that the ABS type system only uses interface names to type objects, but here we add records to track in which cog an object is located. Records can be: $\bot$, meaning that the structure of the object is unknown; $\mathtt{G}[\overline{f : T}]$, meaning that the object is in the cog `G` and its fields $\overline{f}$ are typed with $\overline{T}$; or regular terms, using the standard combination of variables α and the μ-binder. Data types D have at least one constructor, with name `Co`, and possibly a list of type parameters $\overline{T}$. Functions F are declared with a return type T, a name `fun`, a list of parameters $\overline{T\ x}$ and a code e. Interfaces I declare methods and ports that can be modified at runtime. Classes C implement interfaces; they have a list of fields and ports Fl and implement all declared

methods. Method headers S are used to declare methods with their classic type annotation, and i) the possible annotation **critical** that ensures that no rebinding will be performed on that object during the execution of that method; and ii) a *method signature* $(\mathcal{G}, \mathbb{r})$ which will be described and used in our type system section. Method declarations M consist of a header and a body, the latter being a sequential composition of local variables and commands. Statements s are standard except for **await** g, which suspends the execution of the method until the guard g is `true`, and **rebind** $e.p = z$, which rebinds the port p of the object e to the value stored in z. Expressions z include: expressions without side effects e; **new** `C` $(\overline{e})$ and **new cog** `C` $(\overline{e})$ that instantiate a class `C` and place the object in the current cog and in a new cog, respectively; synchronous $e.\mathtt{m}(\overline{e})$ and asynchronous $e!\mathtt{m}(\overline{e})$ method calls, the latter returning a future that will hold the result of the method call when it will be computed; and **get**(e) which gives the value stored in the future e, or actively waits for it if it is not computed yet. Pure expressions e include values v, variables x, function call $\mathtt{fun}(\overline{e})$ and pattern matching **case** e $\{\overline{p \Rightarrow e_p}\}$ that tests e and execute e_p if it matches p. Patterns p are standard: _ matches everything, x matches everything and binds it to x, **null** matches a null object and $\mathtt{Co}(\overline{p})$ matches a value $\mathtt{Co}(\overline{e_p})$ where p matches e_p. Finally, a guard g can be: a variable x; $x?$ which is `true` when the future x is completed, `false` otherwise; $\|x\|$ which is `true` when the object x is in a safe state, i.e., it is not executing any **critical** method, `false` otherwise; and the conjunction of two guards $g \wedge g$ has the usual meaning.

3 Type System

The goal of our type system is to statically check whether synchronous method calls and rebindings are performed locally to a cog. Since cogs and objects are entities created at runtime, we cannot know statically their identity. We address this issue by using a *linear* type system approach on names of cogs $\mathtt{G}, \mathtt{G}', \mathtt{G}'' \ldots$ that abstracts the runtime identity of cogs. This type system associates to every cog creation a unique cog name, which makes it possible to check if two objects are in the same cog or not. Precisely, we associate objects to their cogs using records $\mathbb{r}$, having the form $\mathtt{G}[\overline{f : T}]$, where `G` denotes the cog in which the object is located and $[\overline{f : T}]$ maps any object's fields $\overline{f}$ to its type $\overline{T}$. In order to correctly track cog membership of each expression, we also need to keep information about the cog of the object's fields in a record. This is needed, for instance, when an object stored in a field is accessed within the method body and then returned by the method; in this case one needs a way to bind the cog of the accessed field to the cog of the returned value. Let us now explain the method signature $(\mathcal{G}, \mathbb{r})$ annotating a method header. The record $\mathbb{r}$ is used as the record of **this** during the typing of the method, i.e., $\mathbb{r}$ is the binder for the cog of the object **this** in the scope of the method body, as we will see in the typing rules in the following. The set of cog names $\mathcal{G}$ is used to keep track of the fresh cogs that the method creates. In particular, when we deal with recursive method calls, the set $\mathcal{G}$ gathers the fresh cogs of every call, which is then returned to the main

$$\textsc{S:Data}\ \frac{\forall i \quad T_i \leq T_i'}{D\langle \overline{T} \rangle \leq D\langle \overline{T}' \rangle} \qquad \textsc{S:Bot}\ (L, \mathtt{r}) \leq (L, \bot) \qquad \textsc{S:Fields}\ \frac{\forall i \quad T_i \leq T_i' \qquad f \notin \mathit{ports}(L)}{(L, \mathtt{G}[f : T; \overline{f : T}]) \leq (L, \mathtt{G}[\overline{f : T'}])}$$

$$\textsc{S:Ports}\ \frac{\forall i \quad T_i \leq T_i' \qquad f \in \mathit{ports}(L)}{(L, \mathtt{G}[\overline{f : T}]) \leq (L, \mathtt{G}[f : T; \overline{f : T'}])} \qquad \textsc{S:Type}\ \frac{L \leq L' \in CT}{(L, \mathtt{r}) \leq (L', \mathtt{r})}$$

Fig. 5. Subtyping Relation[4]

execution. Moreover, when it is not necessary to keep track of cog information about an object, because the object is not going to take part in any synchronous method call or any rebind operation, it is possible to associate to this object the *unknown* record $\bot$. This special record does not keep any information about the cog where the object or its fields are located, and it is to be considered different from any other cog, thus to ensure the soundness of our type system. Finally, note that data types also can contain records: for instance, a list of objects is typed with $\mathrm{List}\langle T \rangle$ where T is the type of the objects in the list and it includes also the record of the objects.

A *typing environment* Γ is a partial function from names to typings, which assigns types T to variables, a pair $(\mathtt{C}, \mathtt{r})$ to **this**, and arrow types $\overline{T} \to T'$ to function symbols like `Co` or `fun`.

3.1 Subtyping Relation

The subtyping relation $\leq$ on types is a preorder and is presented in Fig. 5. Rule S:Data states that data types are covariant in their type parameters. Rule S:Bot states that every record $\mathtt{r}$ is a subtype of the unknown record $\bot$. Rules S:Fields and S:Ports use *structural* subtyping on records. Fields, like methods, are what the object provides, hence it is sound to forget about the existence of a field in an object. This is why the rule S:Fields allows to remove fields from records. Ports on the other hand, model the *dependencies* the objects have on their environment, hence it is sound to consider that an object may have more dependencies than it actually has during execution. This is why the rule S:Ports allows to add ports to records. Notice that in the standard object-oriented setting this rule would not be sound, since trying to access a non-existing attribute would lead to a null pointer exception. Therefore, to support our vision of port behavior, we add a Rebind-None reduction rule to the component calculus semantics which simply permits the rebind to succeed without modifying anything if the port is not available. Finally, rule S:Type adopts *nominal* subtyping between classes and interfaces.

[4] For readability, we let L be either a class name `C` or an interface name `I`.

$$\mathrm{tmatch}(T, T) = id \qquad \mathrm{tmatch}(\mathtt{r}, \mathtt{r}) = id \qquad \mathrm{tmatch}(V, T) \triangleq [V \mapsto T]$$

$$\frac{\forall i.\ \mathrm{tmatch}(T_i, T_i') = \sigma_i \qquad \forall i, j,\ \sigma_{i|\mathrm{dom}(\sigma_j)} = \sigma_{j|\mathrm{dom}(\sigma_i)}}{\mathrm{tmatch}(D\langle\overline{T}\rangle, D\langle\overline{T'}\rangle) \triangleq \bigcup_i \sigma_i} \qquad \frac{\mathrm{tmatch}(\mathtt{r}, \mathtt{r}') = \sigma}{\mathrm{tmatch}((\mathtt{I}, \mathtt{r}), (\mathtt{I}, \mathtt{r}')) \triangleq \sigma}$$

$$\frac{\forall i.\ \mathrm{tmatch}(T_i, T_i') = \sigma_i \qquad \forall i, j,\ \sigma_{i|\mathrm{dom}(\sigma_j)} = \sigma_{j|\mathrm{dom}(\sigma_i)} \qquad \forall i,\ \sigma(\mathtt{G}) \in \{\mathtt{G}, \mathtt{G}'\}}{\mathrm{tmatch}(\mathtt{G}[\overline{f : T}], \mathtt{G}'[\overline{f : T'}]) \triangleq [\mathtt{G} \mapsto \mathtt{G}'] \bigcup_i \sigma_i}$$

$$\mathrm{pmatch}(\_, T) \triangleq \emptyset \qquad \mathrm{pmatch}(x, T) \triangleq \emptyset; x : T \qquad \mathrm{pmatch}(\mathbf{null}, (\mathtt{I}, \mathtt{r})) \triangleq \emptyset$$

$$\frac{\Gamma(\mathtt{Co}) = \overline{T} \to T' \qquad \mathrm{tmatch}(T', T'') = \sigma \qquad \forall i.\ \mathrm{pmatch}(p_i, \sigma(T_i)) = \Gamma_i}{\mathrm{pmatch}(\mathtt{Co}(\overline{p}), T'') \triangleq \biguplus_i \Gamma_i}$$

$$\frac{\mathtt{C} \leq \mathtt{I} \in CT \qquad \mathrm{dom}(\sigma') \cap \mathrm{dom}(\sigma) = \emptyset \qquad \mathit{fields}(\mathtt{C}) = (\overline{f : (\mathtt{I}, \mathtt{r})}; \overline{f' : \mathtt{D}(\ldots)})}{(\mathtt{I}, \mathtt{G}[\sigma \uplus \sigma'(\overline{f : (\mathtt{I}, \mathtt{r})})]) \in \mathrm{crec}(\mathtt{G}, \mathtt{C}, \sigma)}$$

$$\frac{\mathrm{equals}(\mathtt{G}, \mathtt{G}')}{\mathrm{coloc}(\mathtt{G}[\ldots], (\mathtt{C}, \mathtt{G}'[\ldots]))}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathtt{C}) \subseteq \mathit{ports}(\mathtt{I}) \text{ and } \forall p \in \mathit{ports}(\mathtt{C}).\ \mathit{ptype}(p, \mathtt{C}) \leq \mathit{ptype}(p, \mathtt{I}) \\ \mathit{heads}(\mathtt{I}) \subseteq \mathit{heads}(\mathtt{C}) \text{ and } \forall \mathtt{m} \in \mathtt{I}.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathtt{C})\end{array}}{\mathrm{implements}(\mathtt{C},\ \mathtt{I})}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathtt{I}) \subseteq \mathit{ports}(\mathtt{I}') \text{ and } \forall p \in \mathit{ports}(\mathtt{I}).\ \mathit{ptype}(p, \mathtt{I}) \leq \mathit{ptype}(p, \mathtt{I}') \\ \mathit{heads}(\mathtt{I}') \subseteq \mathit{heads}(\mathtt{I}) \text{ and } \forall \mathtt{m} \in \mathtt{I}'.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathtt{I}')\end{array}}{\mathrm{extends}(\mathtt{I},\ \mathtt{I'})}$$

Fig. 6. Auxiliary functions and predicates.

3.2 Functions and Predicates

The type system makes use of several auxiliary functions and predicates presented in Fig. 6 [5]. Function tmatch returns a substitution σ of the formal parameters to the actual ones. It is defined both on types and on records. The matching of a type T to itself, or of a record $\mathtt{r}$ to itself, returns the identity substitution id; the matching of a type variable V to a type T returns a substitution of V to T; the matching of data type D parameterized on formal types $\overline{T}$ and on actual types $\overline{T'}$ returns the union of substitutions that correspond to the matching of each type T_i with T_i' in such a way that substitutions coincide when applied to the same formal types; the matching of records follows the same idea as that of data types. Finally, tmatch applied on types $(\mathtt{I}, \mathtt{r}), (\mathtt{I}, \mathtt{r}')$ returns the same substitution obtained by matching $\mathtt{r}$ with $\mathtt{r}'$. Function pmatch, performs matchings on patterns and types by returning a typing environment Γ. In particular, pmatch returns an empty set when the pattern is _ or **null**, or $x : T$ when applied on a variable x and a type T. Otherwise, if applied to a constructor expression $\mathtt{Co}(\overline{p})$ and a type T'' it returns the union of typing environments corresponding to patterns in $\overline{p}$. Function crec asserts that $(\mathtt{I}, \mathtt{G}[\sigma \uplus \sigma'(\overline{f : (\mathtt{I}, \mathtt{r})})])$ is a member of $\mathrm{crec}(\mathtt{G}, \mathtt{C}, \sigma)$ if class $\mathtt{C}$ implements interface $\mathtt{I}$ and σ' and σ are substitutions defined on disjoint sets of names. Function $\mathit{fields}(\mathtt{C})$ returns the

[5] For readability reasons, the lookup functions like *ports, fields, ptype, mtype, heads* are written in italics, whether the auxiliary functions and predicates are not.

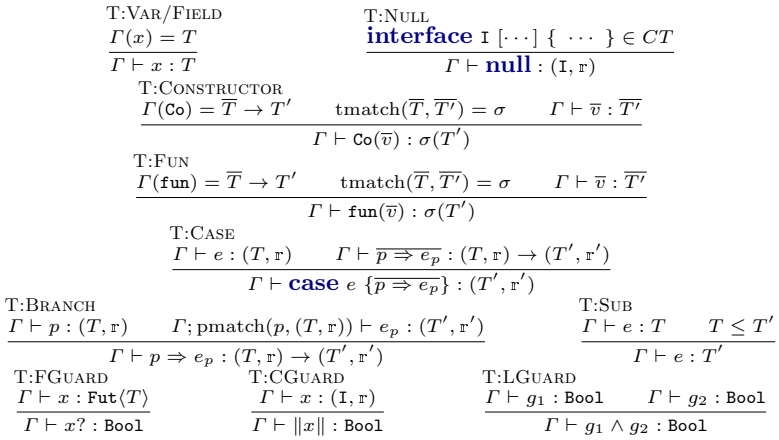

Fig. 7. Typing Pure Expressions and Guards.

typed fields and ports of a class C. Function *port* instead, returns only the typed ports. Predicate coloc states the equality of two cog names. Predicates implements and extends check when a class implements an interface and an interface extends another one properly. A class C implements an interface I if the ports of C are at *most* the ones of I. This follows the intuition: since ports indicate services then an object has at most the services declared in its interface. Then, any port in C has a subtype of the respective port in I. Instead, for methods, C may define at *least* the methods declared in I having the same signature. The extends predicate states when an interface I properly extends another interface I′ and it is defined similarly to the implements predicate.

3.3 Typing Rules

In this section we present the typing rules. Typing judgments use a typing environment Γ and possibly a set $\mathcal{G}$ which indicates the set of new cogs created by the term being typed. They have the following forms: $\Gamma \vdash g : \texttt{Bool}$ for guards, $\Gamma \vdash e : T$ for pure expressions, $\Gamma, \mathcal{G} \vdash z : T$ for expressions with side effects and $\Gamma, \mathcal{G} \vdash s$ for statements. Finally, typing judgments for method, class and interface declarations are $\Gamma \vdash M$, $\Gamma \vdash C$ and $\emptyset \vdash I$, respectively.

Pure Expressions. Typing rules for pure expressions are given in Fig. 7. Rule T:Var/Field states that a variable is of type the one assumed in the typing environment. Rule T:Null states that **null** is of type any interface I declared in the CT and any record $\mathbb{r}$. Rule T:Constructor states that constructor Co applied to a list of values $\overline{v}$ is of type $\sigma(T')$ where the constructor is of a functional type $\overline{T} \to T'$ and the values are of type $\overline{T'}$ obtained by the auxiliary function tmatch. Rule T:Fun for function expressions is the same as the previous one for constructor expressions. Rule T:Case states that if all branches are well-typed and have the same type, then the case expression is also well-typed. Rule

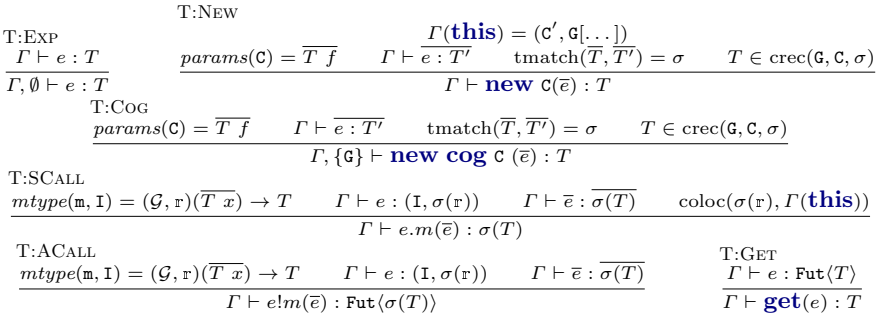

Fig. 8. Typing Expressions.

T:Branch states that a branch $p \Rightarrow e_p$ is well-typed if the pattern p is well-typed and the expression e_p is well-typed in the extension of Γ with typing assertions for the pattern. Rule T:Sub is the standard subsumption rule.

Guards. Typing rules for guards are given in Fig. 7. Rule T:FGuard states that if a variable x has type $\mathtt{Fut}\langle T\rangle$, the guard $x?$ has type `Bool`. Rule T:CGuard states that $\|x\|$ has type `Bool` if x is an object. Rule T:LGuard states that if each g_i has type `Bool` for $i = 1, 2$ then the conjunction $g_1 \wedge g_2$ has type `Bool`.

Expressions. The typing rules for expressions with side effects are given in Fig. 8. These are different w.r.t. the previous ones as they keep track of the new cogs created. Rule T:Exp is a weakening rule which asserts that a pure expression e is well-typed in a typing context Γ and an empty set of cogs, if it is well-typed in Γ. Rule T:New assigns type T to the object **new** $\mathtt{C}(\overline{e})$ if the actual parameters have types compatible with the formal ones, by applying function tmatch, the cogs of the object and **this** coincide and the type T is in the crec predicate. Rule T:Cog is similar to the previous one, except for the creation of a new cog `G` where the new object is placed. Rules T:SCall and T:ACall type synchronous and asynchronous method invocations, respectively. Both rules use *mtype* to obtain the method signature as well as the method's typed parameters and the return type, i.e., $(\mathcal{G}, \mathtt{r})(\overline{T\ x}) \to T$. The group record $\mathtt{r}$, the parameters types and the return type of the method are the "formal" ones. In order to obtain the "actual" ones, we use σ that maps formal cog names to actual cog names. Consequently, the callee e has type $(\mathtt{I}, \sigma(\mathtt{r}))$ and the actual parameters $\overline{e}$ have types $\overline{\sigma(T)}$. Finally, the invocations are typed in the substitution $\sigma(T)$. The rules differ in that the former also checks whether the group of **this** and the group of the callee coincide, by using the auxiliary function coloc, and also the types of the returned value are $\sigma(T)$ and $\mathtt{Fut}\langle\sigma(T)\rangle$, respectively. Rule T:Get states that **get**(e) is of type T, if expression e is of type $\mathtt{Fut}\langle T\rangle$.

Statements. The typing rules for statements are presented in Fig. 9. Rule T:Skip states that **skip** is always well-typed. Rule T:Decl states that $T\ x$ is

$$\textsc{T:Skip}\ \frac{}{\Gamma, \emptyset \vdash \textbf{skip}} \qquad \textsc{T:Decl}\ \frac{\Gamma(x) = T}{\Gamma, \emptyset \vdash T\ x} \qquad \textsc{S:Semi}\ \frac{\Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash s_1; s_2} \qquad \textsc{S:Assign}\ \frac{\Gamma(x) = T \qquad \Gamma, \mathcal{G} \vdash z : T}{\Gamma, \mathcal{G} \vdash x = z}$$

$$\textsc{S:Await}\ \frac{\Gamma \vdash g : \texttt{Bool}}{\Gamma, \emptyset \vdash \textbf{await}\ g} \qquad \textsc{S:Cond}\ \frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash \textbf{if}\ e\ \{\ s_1\ \}\ \textbf{else}\ \{\ s_2\ \}} \qquad \textsc{S:While}\ \frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma, \mathcal{G} \vdash s}{\Gamma, \mathcal{G} \vdash \textbf{while}\ e\ \{\ s\ \}}$$

$$\textsc{S:Return}\ \frac{\Gamma \vdash e : T \qquad \Gamma(\textbf{destiny}) = T}{\Gamma, \emptyset \vdash \textbf{return}\ e}$$

$$\textsc{Rebind}\ \frac{\Gamma(\textbf{this}) = (\texttt{C}, \texttt{G}[\ldots]) \qquad T\ p \in \mathit{ports}(\texttt{I}) \qquad \Gamma \vdash e : (\texttt{I}, \texttt{r}) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad \mathrm{coloc}(\texttt{r}, \Gamma(\textbf{this}))}{\Gamma, \mathcal{G} \vdash \textbf{rebind}\ e.p = z}$$

Fig. 9. Typing Statements.

$$\textsc{T:Method}\ \frac{\Gamma, \overline{x} : \overline{T}, \textbf{destiny} : \texttt{Fut}\langle T \rangle, \textbf{this} : (\texttt{C}, \texttt{r}), \mathcal{G} \vdash s}{\Gamma \vdash [\textbf{critical}]\ (\mathcal{G}, \texttt{r})\ T\ \texttt{m}(\overline{T\ x})\{\ s\ \}\ \mathit{in}\ \texttt{C}}$$

$$\textsc{T:Class}\ \frac{\forall \texttt{I} \in \overline{\texttt{I}}.\ \mathit{implements}(\texttt{C}, \texttt{I}) \qquad \Gamma, \overline{x} : \overline{T} \vdash \overline{M}\ \mathit{in}\ \texttt{C}}{\Gamma \vdash \textbf{class}\ \texttt{C}\ (\overline{T\ x})\ \textbf{implements}\ \overline{\texttt{I}}\ \{\ \overline{Fl}\ \overline{M}\ \}} \qquad \textsc{T:Interface}\ \frac{\forall \texttt{I}' \in \overline{\texttt{I}}.\ \mathit{extends}(\texttt{I}, \texttt{I}')}{\vdash \textbf{interface}\ \texttt{I}\ \textbf{extends}\ \overline{\texttt{I}}\ \{\ \overline{\textbf{port}\ T\ x; S}\ \}}$$

Fig. 10. Typing Declarations.

well-typed if variable x is of type T in Γ. Rule T:Semi types the composition of statements, if s_1 and s_2 are well-typed in the same typing environment and, like in linear type systems, they use distinct cog names. Hence, their composition uses the disjoint union $\uplus$ of the corresponding sets. Rule T:Assign asserts the well-typedness of the assignment $x = z$ if both x and z have the same type T. Rule T:Await asserts that **await** g is well-typed whenever the guard g has type `Bool`. Rules T:Cond and T:While are quite standard, except for the presence of the linear set of cog names. Rule T:Return asserts that **return** e is well-typed if expression e has the same type as the variable **destiny**. Finally, rule T:Rebind types statement **rebind** $e.p = z$ by checking that: *i*) p is a port of the right type, and *ii*) z is in the same group as **this**.

Method, class and interface declarations. The typing rules are presented in Fig. 10. Rule T:Method states that method `m` is well-typed in class `C` if the method's body s is well-typed in a typing environment augmented with the method's typed parameters; type information about **destiny** and the current object **this**; and cog names as specified by the method signature. Rule T:Class states that a class `C` is well-typed when it implements all the interfaces $\overline{\texttt{I}}$ and all its methods are well-typed. Rule T:Interface states that an interface `I` is well-typed if it extends all interfaces in $\overline{\texttt{I}}$.

Remark. The typing rule for assignment requires the group of the variable and the group of the expression being assigned to be the same. This restriction applies to rule for rebinding, as well. To see why this is needed let us consider

$$\frac{\Gamma(\textbf{this}) = (\texttt{Controller}, \texttt{G}[\dots]) \qquad (\texttt{Server}, \mathbb{r})\ s \in \mathit{ports}(\texttt{Client}) \quad \forall i = 2, ..., n \quad \Gamma \vdash \texttt{c}_i : (\texttt{Client}, \texttt{G}[\dots, s : (\texttt{Server}, \mathbb{r})]) \quad \Gamma, \emptyset \vdash s2 : (\texttt{Server}, \mathbb{r}) \qquad \mathrm{coloc}(\texttt{G}[\dots, s : (\texttt{Server}, \mathbb{r})], \Gamma(\textbf{this}))}{\forall i\ \Gamma, \emptyset \vdash \textbf{rebind}\ \texttt{c}_i.s = s2}$$

Fig. 11. REBIND derivation.

a sequence of two asynchronous method invocations `x!m();x!n()`, both called on the same object and both modifying the same field. Say `m` does **this**.`f` = $\texttt{z}_1$ and `n` does **this**.`f` = $\texttt{z}_2$. Because of asynchronicity, there is no way to know the order in which the updates will take place at runtime. A similar example may be produced for the case of rebinding. Working statically, we can either force the two expressions $\texttt{z}_1$ and $\texttt{z}_2$ to have the same group as `f`, or keep track of all the different possibilities, thus the type system must assume for an expression a set of possible objects it can reduce to. In this paper we adopt the former solution, we let the exploration of the latter as a future work. We plan to relax this restriction following a similar idea to the one proposed in [11].

Example Revisited. We now recall the example of the workflow given in Fig. 2 and Fig. 3. We show how the type system works on this example: by applying the typing rule for **rebind** we have the derivation in Fig. 11 for any clients from $\texttt{c}_2$ to $\texttt{c}_n$. For client $\texttt{c}_1$, if we try to typecheck the rebinding, we would have the following typing judgments in the premise of REBIND:

$$\Gamma(\textbf{this}) = (\texttt{Controller}, \texttt{G}[...]) \qquad \Gamma, \emptyset \vdash \texttt{c}_1 : (\texttt{Client}, \texttt{G}'[\dots, s : (\texttt{Server}, \mathbb{r})])$$

But then, the predicate $\mathrm{coloc}(\texttt{G}'[\dots, s : (\texttt{Server}, \mathbb{r})], \Gamma(\textbf{this}))$ is false, since $\mathrm{equals}(\texttt{G}, \texttt{G}')$ is false. Then one cannot apply the typing rule REBIND, by thus not typechecking **rebind** $\texttt{c}_1.\texttt{s} = \texttt{s2}$.

3.4 Properties of the type system

In this section we briefly overview the properties of the type system and we outline the runtime system devised in order to provide the proofs of those properties. The full technical treatment with proofs can be found in [9]. Before stating the properties that our type system enjoys, we first introduce the following notions:

Runtime typing environments Δ are obtained by augmenting typing environments Γ with runtime information about objects and futures, namely $\texttt{o} : (\texttt{C}, \mathbb{r})$ and $\texttt{f} : \texttt{Fut}\langle T \rangle$ where `o` and `f` are object and future variables, respectively.

Runtime configurations N extend the language with constructs used during execution, mainly with objects. An object $ob(\texttt{o}, \sigma, K_{\textbf{idle}}, Q)$ has a name `o`; a substitution σ mapping the object's fields, ports and special variables like **this**, **destiny**, to values; a running process $K_{\textbf{idle}}$, that is **idle** if the object is idle; and

a queue of *suspended processes* Q. A process K is $\{\ \sigma \mid s\ \}$ where σ maps the local variables to their values and s is a list of statements.

Reduction relation $N \to N'$ is defined over runtime configurations and follows the definition of such relation in [13, 14].

Runtime judgments are of the form $\Delta, \mathcal{G} \vdash_R N$ meaning that the configuration N is well-typed in the typing context Δ by using a set $\mathcal{G}$ of new cogs.

Our type system enjoys the classical properties of subject reduction and type correction stated in the following.

Theorem 1 (Subject Reduction). *If* $\Delta, \mathcal{G} \vdash_R N$ *and* $N \to N'$ *then* $\exists\ \Delta',\ \mathcal{G}'$ *such that* $\Delta' \supseteq \Delta$, $\mathcal{G}' \subseteq \mathcal{G}$ *and* $\Delta', \mathcal{G}' \vdash_R N'$.

Proof. The proof is done by induction over the operational semantics rules. □

Theorem 2 (Correction). *If* $\Delta, \mathcal{G} \vdash_R N$, *then for all objects* $ob(\mathtt{o}, \sigma, \{\ \sigma_k \mid s\ \}, Q) \in N$ *with either* $s = \textbf{rebind}\ x.f_i = x'; s'$ *or* $s = x.\mathtt{m}(\overline{x}); s'$, *there exists an object* $ob(\mathtt{o}', \sigma', K_{\textbf{idle}}, Q') \in N$ *such that* $\sigma \circ \sigma_k(x) = \mathtt{o}'$ *and* $\sigma(\textbf{cog}) = \sigma'(\textbf{cog})$. *Where* $\circ$ *defines the composition of substitutions.*

Proof. The proof is done by induction over the structure of N. □

As a consequence of the previous results, rebinding and synchronous method calls are always performed between objects of the same cog:

Corollary 1. *Well-typed programs do not perform i) an illegal rebinding or ii) a synchronous method call outside the cog.*

4 Conclusions, Future and Related Works

This paper presents a type system for a component-based calculus [14], an extension of ABS [13] with **port**s and **rebind** operations. Ports denote the access point to the functionalities provided by the environment and can be modified by performing a rebind operation. There are two consistency issues involving ports: *i*) ports cannot be modified while in use; this problem is solved in [14] by combining the notions of ports and critical section; *ii*) it is forbidden to modify a port of an object outside the cog; this problem is solved in the present paper by designing a type system that guarantees the above requirement. The type system tracks an object's membership to a certain cog by adopting group records. Rebind statement is well-typed if there is compatibility of groups between objects involved in the operation.

Regarding future work, we want to investigate several directions. First, as discussed in Section 3 our current approach imposes a restriction on assignments, namely, it is possible to assign to a variable/field/port only an object belonging to the same cog. We plan to relax this restriction following an idea similar to the one proposed in [11], where instead of having just one group associated to a variable, it is possible to have a set of groups. Second, we want to deal with runtime misbehavior. For instance, deadlocks are intrinsically related to the semantic model, which requires a component to be in a safe state when rebinded,

thus introducing synchronization points between the rebinder and the rebindee. For this reason deadlocks may arise. How to detect and avoid this kind of misbehavior is left as future work, possibly following [10]. Moreover, in this paper we showed how to use our technique for a very specific safety problem in the context of a component-based language, but we believe the tracking of object/-group identities/memberships is useful for other problems (deadlock detection, race detection, resource consumption) and other settings (business processes and web-services languages). We plan to investigate this direction further.

Related Works. Most component models [2–4, 8] have a notion of component distinct from that of object. The resulting language is structured in two separate layers, one using objects for the main execution of the program and the other using components for the dynamic reconfiguration. This separation makes it harder for the reconfiguration requests to be integrated in the program's workflow. The component model used in the present paper has a unified description of objects and components by exploiting the similarities between them. This bring several benefits w.r.t. previous component models: *i*) the integration of components and objects strongly simplifies the reconfiguration requests handling, *ii*) the separation of concepts (component and object, port and field) makes it easier to reason about them, for example, in static analysis, and *iii*) ports are useful in the deployment phase of a system by facilitating, for example, the connection to local communication. Various type systems have been designed for components. The type system in [15] categorizes references to be either Near (i.e., in the same cog), Far (i.e., in another cog) or Somewhere (i.e., we don't know). The goal is to automatically detect the distribution pattern of a system by using the inference algorithm, and also control the usage of synchronous method calls. It is more flexible than our type system since the assignment of values of different cogs is allowed, but it is less precise than our analysis: consider two objects o_1 and o_2 in a cog c_1, and another one o_3 in c_2; if o_1 calls a method of o_3 which returns o_2, the type system will not be able to detect that the reference is Near. In [1] the authors present a tool to statically analyze concurrency in ABS. Typically, it analyses the concurrency structure, namely the cogs, but also the synchronization between method calls. The goal is to get tools that analyze concurrency for actor-like concurrency model, instead of the traditional thread-based concurrency model. On the other hand, our type system has some similarities with the type system in [5] which is designed for a process calculus with *ambients* [6], the latter roughly corresponding to the notion of components in a distributed scenario. The type system is based on the notion of group which tracks communication between ambients as well as their movement. However, groups in [5] are a "flat" structure whether in our framework we use group records defined recursively; in addition, the underlying language is a process calculus, whether ours is a concurrent object-oriented one. As object-oriented languages are concerned, another similar work to ours is the one on *ownership types* [7], where basically, a type consists of a class name and a context representing object ownership: each object owns a context and is owned by the context it resides in. The goal of the

type system is to provide alias control and invariance of aliasing properties, like role separation, restricted visibility etc. [12].