An Object Group-Based Component Model*

Michaël Lienhardt, Mario Bravetti, and Davide Sangiorgi

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

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

1 Introduction

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

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

* Partly funded by the EU project FP7-231620 HATS.
it is difficult to express modifications of the component structure using objects 
(since these are rather supposed to describe the execution of the programs). For 
instance, models like Click \[13\] do not allow runtime modifications while OSGi 
\[1\] allows addition of new classes and objects, but no component deletions or 
binding modifications. In this respect, a more flexible model is Fractal \[3\], which 
reifies components and ports into objects. Using an API, in Fractal it is possible 
to modify bindings at runtime and to add new components; Fractal is however 
rather complex, and it is informally presented, without a well-defined model.

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

**Our Approach.** Most component models have a notion of component that is 
distinct from the objects used to represent the data and the main execution of 
the software. The resulting language is thus structured in two different layers, 
one using objects for the main execution of the program, one using components 
for the dynamic reconfiguration. Even though such separation seems natural, it 
makes difficult the integration of the different requests for reconfiguration into 
the program’s workflow. In contrast, in our approach we tried to have a uni-
form 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 ob-
jects and object groups with the basic elements of components (ports, bindings, 
consistency and hierarchy) and hence enable dynamic reconfigurations.

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