A Reconfiguration Framework for Distributed Components

Boutheina Bennour, Ludovic Henrio, Marcela Rivera

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Reconfiguration Framework for Distributed Components

Boutheina Bennour — Ludovic Henrio — Marcela Rivera

N° 6911

April 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6911--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

A Reconfiguration Framework for Distributed Components

Boutheina Bennour , Ludovic Henrio , Marcela Rivera

Thème COM — Systèmes communicants
Équipe-Projet Oasis

Rapport de recherche n° 6911 — April 2009 — 17 pages

Abstract: Adaptability is a key feature of distributed systems. In component systems, adaptability can be realised by reconfiguration of the component assembly. The objective of this work is to increase the support for reconfiguration capabilities in distributed component models. This work extends an existing framework of reconfiguration language, FScript, by enabling remote interpretation of reconfiguration procedures. We provide an extension of the component model with a non-functional ability: the interpretation of reconfiguration scripts. This capability provides the basis for the non-centralised interpretation of reconfiguration scripts. This way, reconfiguration scripts can be evaluated in a distributed manner. We also provide an implementation of the extended script language and the interpreter.

Key-words: component, reconfiguration, distribution, fscript, adaptation

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Un environnement de reconfiguration pour composants distribuées

Résumé : L'adaptabilité est la caractéristique principale des systèmes distribués. Dans les systèmes de composants, l'adaptabilité peut être réalisée par la reconfiguration de l'assemblage de composants. L'objectif de ce travail est d'améliorer le support pour la reconfiguration dans les modèles à composants distribués. Ce travail étend un environnement de langage de reconfiguration existant, FScript, en permettant l'interprétation distante des procédures de reconfiguration. Nous avons étendu le modèle de composant avec une capacité non fonctionnelle : l'interprétation des scripts de reconfiguration. Ces caractéristiques constituent une base solide base pour l'interprétation non-centralisée des scripts de reconfiguration. De cette façon, des scripts de reconfiguration peuvent être évalués de façon distribuée. Egalement, nous fournissons une implémentation de l'extension du langage de script et de l'interprète.

Mots-clés : composant, reconfiguration, distribution, fscript, adaptation

Contents

1 Introduction

1.1 Context

This work is placed in the context of large-scale distribution, and distributed computing. Components have been considered over the recent years as a good abstraction to program distributed systems thanks to the encapsulation the components provide, and their clearly defined interfaces. The success of component models also comes from the high-level view of a component system (e.g. defined by an architecture description language), which is considered as the good level to design distribution of applications. Highly evolving distributed environments also require applications deployed on those environments to evolve. Dynamic reconfiguration of the component structure is particularly well suited to ensure this evolution. Dynamic reconfiguration allows applications not only to adapt to changes in their execution environments, but also to changes in their functional and non-functional requirements. Some approaches even consider component systems evolving in an autonomic way [1]. There, reconfiguration is handled automatically, in order to ensure a required quality of service.

1.2 Objective and Contribution

This article presents a distributed reconfiguration mechanism based on a scripting language. Our framework supports the implementation of reconfiguration procedures by providing to the programmer an adapted language. Using a scripting language, the programmer of the adaptation code can focus on the reconfiguration procedures.

More precisely, this article shows how distributed system can be reconfigured in a non-centralised manner. That means, the programmer can now write and execute distributed reconfiguration procedures. To reach this goal, we propose to extend existing component frameworks with:

- A non-functional port, localised in several (possibly all) components that is able to interpret reconfiguration orders.
- An extension of existing reconfiguration scripting languages with primitives for distributed interpretation.

This distributed evaluation allows the parallelisation of the reconfiguration. Also the reconfiguration interpreter can be moved close to the reconfigured entity improving efficiency of the adaptation process. Our contribution is applied to the scripting language FScript and the GCM (Grid Component Model) [2] component framework. We show the adequacy of our contribution by providing an implementation of those features in the context of the GCM, and of its reference implementation above the ProActive middleware.

After a review of related works on reconfiguration and component systems in Section 2, we define a Fractal controller for reconfiguration in Section 3. Section 4 extends the FScript language to support distribute reconfiguration, and ease the passing of parameters between reconfiguration scripts. Section 5 presents the implementation of our approach, and some experiments. Finally, Section 6 presents the conclusions and future works.

2 Related Works and Positioning

This section presents five component models and studies their reconfigurability. We focus here on reconfiguration and adaptation capabilities, first from the component model point of view, and then focusing on reconfiguration (scripting) languages. One of the most interesting features of component models for adaptive applications is the possibility to introspect and modify the component structure at runtime. Then, the component structure can be reconfigured in order to modify the system capabilities. Our approach is mainly based on this property.

CCM (CORBA Component Model) [13, 11] is a specification for business components which can be distributed, heterogeneous, and implemented over different programming languages or operating systems. CCM components communicate through ports that can be interconnected (also, the OMG D&C specification [12] supports hierarchical assemblies). All component instances are handled at runtime by their container. Distribution and container existence should make CCM a possible target of our approach, but unfortunately most of the CCM implementation does not support dynamic reconfiguration.

CCA (Corba Component Architecture) [8] aims at a minimal specification of component architecture for high-performance computing. The composition of component system is flat. A CCA component is defined as a set of ports. The components are assembled at runtime connecting ports together, using scripts that interact with the CCA framework. The CCA framework acts as a component container; it allows building, connecting and running components. Despite the presence of a container, applying our approach of reconfiguration to CCA components remains less interesting than considering it for hierarchical assemblies which should scale better.

Fractal [7] is a hierarchical and reflective component model. A Fractal component can be either *composite* (i.e. composed of sub-components), or *primitive* (a basic element encapsulating the business code). A Fractal component comprises a content (providing the functional code) and a membrane (a container managing the non-functional operations). The components have *client interfaces* and *server interfaces* which are connected by *bindings*. Fractal components can easily be reconfigured by invocations on their non-functional interfaces. The membrane takes care of the reconfiguration of the component structure. A fractal component architecture can be described using an architecture description language (ADL) [6].

SCA (Service Component Architecture) [3] provides a component-oriented programming model for building applications based on a Service Oriented Architecture. SCA provides a model for both the composition of services and the creation of service components, including the reuse of existing application function within SCA composites. SCA component structure is not specified at runtime. So, further work is required to extend the SCA model for reconfiguration. The SCOrWare runtime platform, called FraSCAti [4, 14] is an implementation of the SCA model built upon Fractal. It provides advanced functionalities such as dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture.

GCM [2] is a distributed extension of Fractal. Like Fractal, each component has a membrane and a content. Components are composed hierarchically, and interfaces are interconnected by bindings. Contrary to Fractal, GCM

specifies distribution aspects of the component model, and defines one-to-many and many-to-one communication, which are particularly efficient for distributed components. GCM model also refines the structure of the membrane, and defines some controllers for autonomic behaviour. GCM provides the same reconfiguration functionalities as Fractal.

We presented five prevalent component models namely: CCM, CCA, SCA, Fractal, and GCM. Reconfigurability is guaranteed for most models (like Fractal, GCM, CCM, and CCA) by the presence at runtime of a container or a membrane. As soon as component manipulation is allowed by some control structure, a reconfiguration language can be designed. If the component model is distributed, it is interesting to evaluate reconfiguration scripts in a decentralised manner.

Fractal is particularly well suited for component adaptability, because it comes with high reconfiguration capabilities, and a scripting language for reconfiguration: FScript. Our approach is designed for the GCM, a distributed extension of this approach can be adapted to other component models (for example, to extend GScript [10] and reconfigure CCA components). The approach relies on the possibility to embed a script interpreter in some components and to extend the scripting language with a couple of new primitives.

Let us now focus on languages for expressing dynamic evolution of component systems. The GScript [10] scripting language for CCA and GCM components is the closest approach to ours. It provides a scripting language for high-level orchestration and interaction with distributed components. GScript programs can trigger reconfiguration of GCM components by direct invocation on the adequate component interfaces.

FScript [9] is a DSL (Domain-specific Language) to program dynamic reconfigurations of Fractal architectures. FScript includes the FPath notation to navigate inside component architectures. FScript gives the possibility to define reconfiguration scripts to modify the architecture of a Fractal application. A key characteristic of FScript is that it is specifically tailored to the *reconfiguration* of components, in particular for Fractal components. Consequently, despite the high-level approach suggested by GScript, FScript proposes much higher-level primitives due this specialisation. In consequence, reconfiguration scripts are simpler in FScript. Of course, as a counterpart, GScript is much more expressive particularly for expressing orchestration and dataflow between components.

The expressivity of FScript is close to Fractal. In FScript for example it is not possible to add new interfaces to an existing component because this is not allowed by the Fractal model; however the component structure can be easily changed. In most implementations of Fractal it is necessary to stop the target components (but not the whole application) before reconfiguring them. Stopping a component can then be hierarchical (stopping all sub-components) or not, depending on the implementation. Reconfiguration scripts in FScript can invoke the life-cycle controller of some components to stop or start some of the components of the application in order to reconfigure them. Note that stopping a component does not stop the membrane and thus a stopped component can still interpret reconfiguration scripts and be reconfigured.

We focus on the distributed evaluation of FScript programs. The characteristics and restrictions of FScript mentioned above are unchanged. Like in FScript,

transactions are only guaranteed at the action level. Inter-action transactivity would be possible to ensure in FScript but is too costly to guarantee in our framework and would cancel most of the benefits of the distributed evaluation. The user can still run scripts in a centralised manner to benefit from FScript transactions. GScript language is also not distributed, and our approach can easily be adapted to this language. This would make GScript a rich distributed language for component management on the Grid; but we focused on FScript which is better tailored to GCM, and is easier to extend.

3 A Controller for Reconfiguration

To trigger decentralised reconfigurations on distributed components, we propose to incorporate a reconfiguration script interpreter into components. This way, interpretation of reconfiguration scripts can be distributed, and a script interpreter can remotely invoke another one. Therefore, several (possibly all) components must expose script interpretation capabilities in order to allow the system to perform distributed evaluation of reconfiguration scripts.

Exposing such a non-functional feature is possible in the Fractal model through the definition of a new controller. Fractal specification defines basic controllers ensuring component reflective abilities (introspection and intercession). We add a reconfiguration controller in the component membrane. This controller provides the interface allowing the invocation of a script interpreter. The reconfiguration controller improves component reconfigurability since it can interpret high level scripts. The programmer of adaptivity procedure can thus focus on a high level language, instead of direct invocation of basic reconfiguration primitives.

```
interface ReconfigurationController {
    void setInterpreter(String interpreterClassName)
    void loadScript(String scriptFileName)
    void executeAction(String actionName, Object... arguments)
}
```

Figure 1: Reconfiguration controller interface

Figure 1 shows the interface implemented by the reconfiguration controller. The method `setInterpreter` assigns an interpreter to the component. The method complies with the singleton pattern. If it does not exist, an interpreter instance is created. Actually, the interpreter is not instantiated when creating a component for performance efficiency. Only the components that must be able to interpret reconfiguration procedures encapsulate an interpreter.

Reconfiguration actions are defined in script files. The reconfiguration controller provides a method `loadScript` for the interpreter to recognise actions by parsing the script file. Once the script file is loaded, a reconfiguration action can be triggered on the component by calling method `executeAction` of the controller API. The action name and arguments are passed as parameters to the method. When a remote script interpretation is triggered on a component, the associated reconfiguration controller is created (see Section 4.1).

Thanks to the reconfiguration controller, the programmer can trigger reconfiguration actions on remotely accessible components. In Figure 2, direct invocations on the reconfiguration controller are performed by a Java program.

This code installs an interpreter, loads a script, and executes it. Such script management is still low level. Each component equipped with a reconfiguration controller, can interpret reconfiguration scripts. In Section 4, we will show an easier way to manage component, relying only on the scripting language. The next section also shows how a reconfiguration language can be extended with a primitive for remotely invoking a reconfiguration controller.

```
/* retrieve a reference on component controller for reconfiguration    */
  ReconfigurationController rc;
  rc = comp.getFcInterface(RECONFIGURATION_CONTROLLER_NAME);

/* assigns an interpreter to the component                             */
  rc.setInterpreter(FSCRIPT_INTERPRETER);

/*loads the reconfiguration action specified in the script SCRIPT_NAME */
  rc.loadScript(SCRIPT_NAME);

/*triggers on the component the reconfiguration action ACTION_1 defined
in the script SCRIPT_NAME previously loaded                             */
  rc.executeAction(ACTION_1, arguments);
```

Figure 2: Code necessary for loading a script

4 An Extension to the FScript language

This section presents an extension of the reconfiguration language allowing remote invocation of reconfiguration scripts. Our approach relies on adding a new primitive triggering remote interpretation of reconfiguration script. An alternative approach would consist in detecting whether a component has a reconfiguration controller, and automatically triggering the invocation of a remote primitive if such a controller exists. Our choice is driven by the following reasons. First, it avoids many distributed tests that would slow down execution. Second, we want the script programmer to be able to control which reconfiguration procedures are executed locally or remotely. Third, we can add a target component as a parameter to the remote invocation procedure; this parameter specifies the component where the reconfiguration is to be executed. However, we consider that our framework could also be a good basis for the automatic distributed evaluation of reconfiguration scripts if necessary. In this section we also study the passing of parameters to remotely invoked script.

4.1 Remote Script Execution

If a component has a reconfiguration controller, it provides local control of reconfiguration since it is capable of performing its own script interpretation. Also, with the availability of a reconfiguration controller in its membrane, the component exposes interpretation features to its neighbours. A script that reconfigures a complex component system may be partially delegated to a subcomponent or to another neighbouring component. In order to delegate interpretation, we define a new primitive `remote_call`, shown in Figure 3.

The primitive `remote_call` triggers the execution of the reconfiguration action `action_1` by the interpreter associated with the component `target_component`. The target component is given by its node, specified as a FPath expression.

```
remote_call(target_component, action_1, arguments_list...)
```

Figure 3: Primitive Remote call

The second argument is a string that gives the reconfiguration action name; the name of the remotely invoked reconfiguration procedure. The arguments in `arguments_list` are passed as parameters of the remotely invoked action. Arguments are locally evaluated, then they are passed to the remote script interpreter. Upon remote script invocation, if no remote interpreter is available then one is automatically created by calling the `setInterpreter` method on the remote reconfiguration controller.

After this call, the target component becomes in charge of the interpretation of the reconfiguration. Unless the action is a primitive, the reconfiguration should be defined in the context of the target interpreter. Thus, if necessary, the action is automatically loaded in the remote reconfiguration controller (`loadScript` primitive).

The calling interpreter does not have to wait for the completion of the reconfiguration by the target component. Once the delegation of the reconfiguration action is complete, the calling interpreter continues the execution of its local script. Such a strategy allows reconfigurations to occur in parallel.

When the reconfiguration action finishes, no automatic notification mechanism is specified here. Consequently it is not possible to know directly whether a remotely invoked script succeeded or not. However, call-backs can be used to return the status of the remote script, and further synchronisation primitives could also be added to the language to synchronise the different reconfiguration controllers.

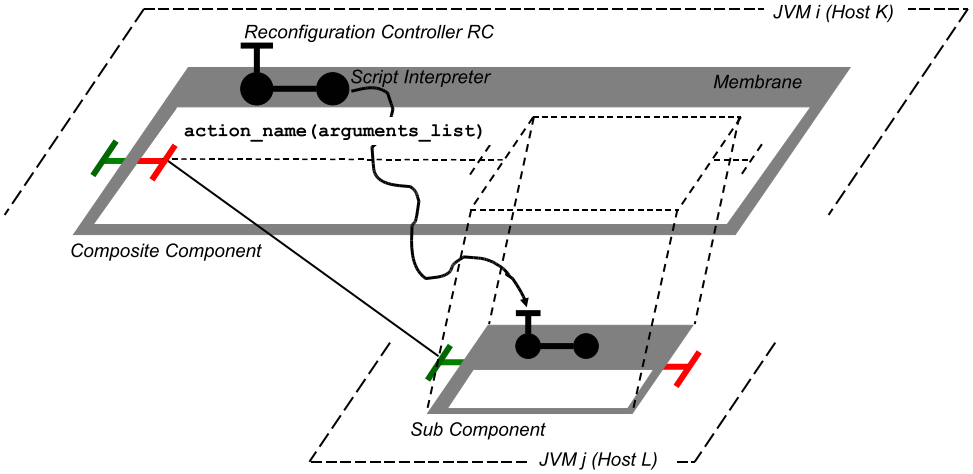

Figure 4: Distributed interpretation of the reconfiguration action *action_name*

Figure 4 illustrates the remote interpretation of the reconfiguration action `action_1` by a target component, in this case a subcomponent. As components are deployed on different hosts, they are distributed: the composite component is deployed on the host K, while de subcomponent is deployed on the host

L. In Figure 4, an FScript script reconfigures, in a decentralised manner, the subcomponent by executing the action `action_1`. This remote script invocation is shown in Figure 5.

```
remote_call($CompositeReference/child::subcomponent_name,'action_1');
```

Figure 5: An example of remote script invocation.

In this script, `$CompositeReference` is a variable defined in the calling interpreter. It contains a reference to the composite component executing the calling script. `subcomponent_name` is the name of the remote subcomponent where the action will be executed. In Figure 4, the delegation of the reconfiguration action is represented by a thick arrow from the calling interpreter to the target interpreter. To performs the action `remote_call`, the interpreter of the caller first retrieves a reference on the reconfiguration controller RC of the subcomponent; then, it calls the method `executeAction` exposed by RC. This triggers the remote interpretation of the reconfiguration action by the interpreter associated with the subcomponent.

4.2 Passing Parameters to a Remotely Invoked Script

Distributed script interpretation raises two issues in relation with the reconfiguration action parameters. First, it can be impossible to transmit some of those parameter along the network; and a possible second issue is the context of interpretation of these parameters. We present solutions to those two closely related problems in the following.

Let us first detail the problem of parameter communication. When the calling interpreter triggers the remote execution of the reconfiguration action, the parameters of this action are transferred from the calling execution context to the target execution context. So they are expected to be serialisable. Method parameters in FScript are either primitive types, which are serialisable, or nodes, which were made nodes serialisable objects. The object nodes are consequently serialised using the object serialisation mechanism of the Java platform. The passing of parameters to the target context is therefore done by value. Note that, a node is a reified reference to a part of the component structure. It is this reified reference that is transmitted when the node is serialisable. Declaring FScript nodes as *serialisable* is thus sufficient for allowing the transmission of action parameters to the remote reconfiguration controller.

Let us now focus on the other half of the problem, and in fact propose a solution solving both issues mentioned above.

In some applications, it is more adequate to evaluate an FPath expression at the receiving side, because a path is expressed in the context of the reconfiguration controller performing the action. We show here how to avoid the evaluation of the parameters by the calling interpreter. Instead of passing the parameters as serialisable nodes, the target interpreter receives them as string objects, which are serialisable. These string objects represent the FPath expressions that target FScript interpreter can evaluate. Such expressions must contain only global variables known by the interpreter.

At the receiver side, the string parameter must be evaluated explicitly; for this purpose, we enrich the FScript language with a primitive for the evaluation of strings representing FPath as shown in Figure 6.

```
node|string|boolean|... eval(fpath_expression_string)
```

Figure 6: Signature of the `eval` primitive

This primitive, shown in Figure 6, evaluates the string received as an argument and returns the result of evaluation. This may be a node or a primitive type (FScript is dynamically typed). The `eval` primitive allows the evaluation of FPath strings passed as arguments to the reconfiguration action. Parameters can then be interpreted by the interpreter that uses them instead of the one that invokes them. This behaviour is very helpful when using generic scripts taking a relative path as parameter.

The following example illustrates local versus remote parameter evaluation; Figure 7 shows the transmission of an evaluated FPath expression, leading to a node that has been turned into a serialisable object. In Figure 8, we use the `eval` primitive, and thus rely on the transmission of the FPath as a string parameter.

```
action reconfigureComposite(){
    argument1 =  $CompositeReference/interface::interface_name;
    remote_call($CompositeReference/child::subcomponent_name,
                                          'action_1',argument1);
}
action action_1(argument){
    interface = argument;
    ...
}
```

Figure 7: Parameter evaluation by the calling component interpreter

```
action reconfigureComposite(){
    argument2 = '$SubComponentReference/parent::compositecomponent_name/
                                          interface::interface_name';
    remote_call($CompositeReference/child::subcomponent_name,
                                          'action_1',argument2);
}
action action_1(argument){
    interface = eval(argument);
    ...
}
```

Figure 8: Parameter evaluation by the target component interpreter

Consider the distributed interpretation of the reconfiguration action `action_1` in the 2 figures above. This action receives as argument a functional interface `interface_name`: the functional interface of the calling component. In both cases, the effect of the reconfiguration action is the same. However, in the first case the target interpreter receives a reference to the interface. In the second case, the target interpreter calculates the reference by evaluating the FPath expression using the primitive `eval` (`argument1` is dynamically typed as string and `argument2` is of type interface node). For the second approach, the FScript interpreter does not evaluate FPath expressions as serialisable object

nodes. The target interpreter receives a FPath expression string and performs its own evaluation of the parameters, returning a node.

5 Prototype and Experiments

We have tested our approach on GCM/ProActive components. In this section, we discuss the implementation of the distributed reconfiguration mechanism over the middleware ProActive and the FScript language. We describe then an experiment on an application, made up of hierarchical and distributed component, which demonstrates the feasibility of our approach.

5.1 An Implementation in GCM/ProActive

The ProActive Middleware provides an implementation of the GCM model. The components are thus hierarchical and distributed. Each component encapsulates an active object. Components communicate through asynchronous method calls. Predefined controllers and custom controllers compose the membrane (non-functional manager) of a GCM/ProActive component.

We have extended the component framework of GCM/ProActive to support distributed reconfiguration. We added the reconfiguration controller as a custom controller in the membrane. Therefore, the application programmer can specify entry points for reconfigurations: this is possible by configuring only some components to include reconfiguration controllers. In the GCM/ProActive framework, some/all components are now able to host script interpreters and expose minimal script interpretation capabilities for reconfiguration. To optimise the performance of the application, it is crucial to determine the granularity of reconfiguration, together with the distributed reconfiguration mechanism.

We used the FScript interpreter designed for Fractal components, extended by the primitives described above. As the GCM Model is an extension of the Fractal Model, it is also reconfigurable by FScript; the FScript interpreter can be encapsulated, into GCM components to reconfigure them. The FScript language contains predefined actions corresponding to the Fractal API such as adding a component to a composite component `add`, binding two compatible component interfaces `bind`, etc. We have also implemented the remote call primitive as a predefined action `remote_call`. In addition, the FScript expression evaluation `eval` has been implemented to improve the passing of parameters.

5.2 Experiment: Distributed reconfiguration of a component system

Our approach has been tested on the model of a Turntable Production Cell, described [5]. The Turntable Production Cell is illustrated in Figure 9. It consists of a rotary disc with four product slots. A product is *loaded* into a slot at position 0, and is then rotated to position 1 where it is *drilled*. After that, it is rotated into position 2 where it is *tested*, and finally it is rotated at position 3 where it is *unloaded*. All slots of the rotary disc may be occupied at the same time, and the products are processed in parallel. We assume that the system works without faults and there is no product loss, because those issues are unrelated with our contribution.

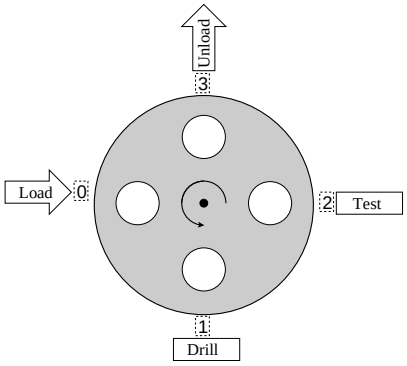

Figure 9: Schematic diagram of a Turntable system

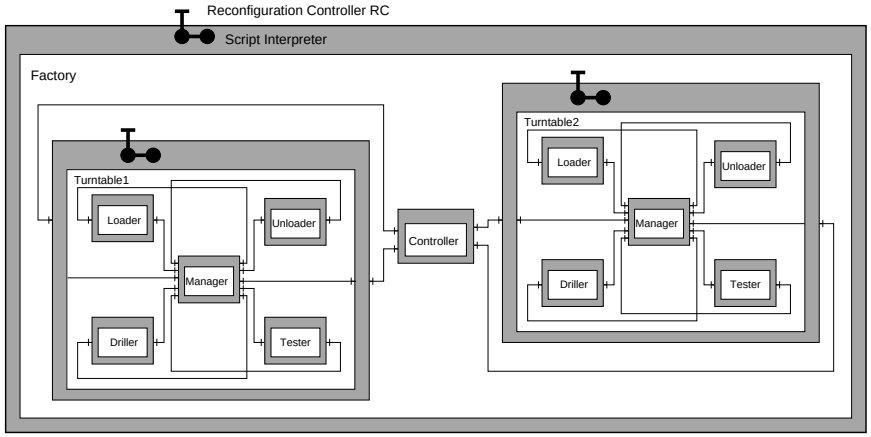

Figure 10: A component system for experiment

We consider a *Factory* that has two Turnable Systems; the component system representing the Factory is shown in Figure 10. We can identify three subcomponent of Factory: `Controller` (for synchronising the two other components), `Turntable1` and `Turntable2`. Each of the components `Turntable1` and `Turntable2` has five subcomponents: `Manager`, `Loader`, `Driller`, `Tester` and `Unloader`.

The execution of each turntable operation requires a certain amount of time, because the duration of the turntable operations has not been defined. For that, the component `Manager` must synchronise and coordinate the other four components. Note that the components are distributed since they are deployed on different hosts.

The script of Figure 11 defines the reconfiguration action `change_driller` which creates a component `Driller2`. The action replaces the component `Driller` of a given composite component by the new component. Also the script defines the reconfiguration action `reconfigure_factory`; this action locally performs the action `change_driller`, and triggers the distributed interpretation of the change driller action by the component `Turntable`. Three instances of reconfiguration controller (RC) are installed: one on the factory, and one on each turntable.

The experiment consists of two remote script invocations. The `reconfigure_factory` action should be triggered on the RC of the factory. It

```
action change_driller(factory){
   driller = $factory/child::Driller;
   intf-tester-r = $factory/child::Tester/interface::r;
   unbind($driller/interface::s);
   unbind($intf-tester-r);
   driller2 = new('DrillerImpl);
   set-name($driller2,"Driller2");
   add($factory,$driller2);
   intf-driller2-r = $driller2/interface::r;
   intf-driller2-s = $driller2/interface::s;
   bind($intf-driller2-s,$factory/child::Manager/interface::s);
   bind($intf-tester-r,$intf-driller2-r);
 }

action reconfigure_factory(factory){
   remote_call($factory/child::Turntable1,'change_driller',$factory/child::Turntable1);
   remote_call($factory/child::Turntable2,'change_driller',$factory/child::Turntable2);
 }
```

Figure 11: An experimental reconfiguration script

delegates the action `change_driller` to the RC of component `Turntable1`. This action is loaded and executed in the interpreter of `Turntable1`. The arguments of this action are evaluated locally in the factory. The second action is executed similarly in component `Turntable2`. After the reconfiguration, the components `Factory`, `Turntable1` and `Turntable2` have been successfully reconfigured and the application features the behaviour of the new driller.

The interpreter does not wait for the completion of the first `remote_call`. Once the reconfiguration action is delegated, the interpreter continues with the next `remote_call`; therefore, the reconfiguration in `Turntable1` and the reconfiguration in `Turntable2` occur in parallel. The experiment shows that the reconfiguration operations are processed in a distributed and parallel manner relying on the distributed interpretation of the reconfiguration script.

6 Conclusion

In this work we proposed an extension of a framework for distributed reconfiguration of components. The main objective is to increase the support for reconfiguration capabilities in distributed component models.

We extended an existing scripting language (FScript) for reconfiguring a component system in a distributed manner. Our extension consists first of a new component controller able to interpret scripts. Second, we defined new primitives which allow the remote invocation of reconfiguration scripts. This way, we enhanced the support for independent reconfiguration procedures. We showed that the introduction of a remote invocation primitive leads to two problems: the evaluation context of arguments, and the possibility to send script arguments. A prototype of the distributed FScript interpreter has been implemented and experiments involving distributed reconfiguration have been performed.

The originality of our approach is that it requires a minimal extension to an existing component framework and its associated reconfiguration language. Additionally, it has been specifically designed to provide the key operations for the programmer of reconfiguration procedures.

Our work should impact the design of adaptation procedures, especially in the context of autonomic adaptation. Indeed autonomic adaptation of distributed components require components to self-adapt in a non-centralised man-

ner. Without contributing to the design of self-adaptation procedures themselves, this work allows their fast and straightforward implementation.

In the future, we plan on designing primitives for directly triggering synchronisation inside the reconfiguration scripts.

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. In *PDP*, pages 54–63. IEEE Computer Society, 2008.

[2] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications*, accepted for publication, 2008.

[3] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, and Oisin Hurley. SCA service component architecture, assembly model specification. Technical report, March 2007. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[4] Gaël Blondelle, Philippe Merle, Vivien Quema, Samir Tata, Daniel Hagimont, and et al. SCOrWare Project, SCA Platform Specifications - version 1.0. Technical report, EBM WebSourcing, INRIA Adam team, INRIA Sardes team, INT, IRIT, September 2007. `http://www.scorware.org/projects/en/deliverables`.

[5] E. Bortnik, N. Trcka, A.J. Wijs, B. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a [chi] model of a turntable system using spin, cadp and uppaal. *Journal of Logic and Algebraic Programming*, 65(2), 2005.

[6] Eric Bruneton. Fractal ADL tutorial. France Telecom, 2004. `http://fractal.objectweb.org/tutorials/adl/index.html`.

[7] Eric Bruneton, Thierry Coupaye, and Jean Bernard Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[8] CCA-Forum. The Common Component Architecture (CCA) Forum home page, 2005. http://www.cca-forum.org/.

[9] Philippe David and Thomas Ledoux. Safe dynamic reconfigurations of Fractal Architectures with FScript. In *Proceeding of Fractal CBSE Workshop*, Nantes, France, 2006.

[10] Maciej Malawski, Tomasz Gubala, Marek Kasztelnik, Tomasz Bartynski, Marian Bubak, Francoise Baude, and Ludovic Henrio. High-level scripting approach for building component-based applications on the grid. In *CoreGRID Workshop on Grid Programming Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments*, Heraklion, Crete, June 2007. Springer.

[11] Object Management Group, Inc. (OMG). *CORBA Component Model Specification*, omg headquarters edition, April 2006. `http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf`.

[12] OMG. Deployment and configuration of component-based distributed applications, v4.0. Document formal/2006-04-02 Edition, Apr. 2006.

[13] omg.org team. CORBA Component Model, V3.0. `http://www.omg.org/technology/documents/formal/components.htm`, 2005.

[14] OW2.Consortium. FraSCAti, Open SCA middleware platform. `https://wiki.objectweb.org/frascati/Wiki.jsp?page=FraSCAti`, 2009.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399