Developing Adaptable Components using Dynamic Languages

Didier Donsez, Kiev Gama and Walter Rudametkin
Grenoble University
LIG Laboratory, ADELE Team
Grenoble, France
e-mail: {firstname}.{lastname}@imag.fr

Abstract— **The usage of dynamic languages is increasing among developers. As components are static entities, the usage of scripting languages, which are usually dynamically typed and interpreted, would bring flexibility in the development of components allowing component reconfiguration and adaptation at runtime without needing either to recompile the component code or to restart the application. This paper presents a general approach for creating adaptable components by using dynamic scripting languages combined with component models. This concept has been implemented and validated in two different contexts: in the OSGi platform and in a Fractal-like approach.**

Keywords-component; dynamic languages; component models; adaptable components; scripting

I. INTRODUCTION

Component Based Development [27] [35] promotes the reuse and integration of existing software components for assembling applications. Many components models, such as EJB [34], CCM [26] or CCA [9] are conceived for specific purposes, while others such as Fractal [6] are intended to be generalist. For these models, the life cycle of the application is handled by different actors following successive stages such as development, testing, assembly, configuration, deployment, execution and even monitoring (e.g. online diagnosis). The maintenance of component based applications may eventually involve operations of reconfiguration and adaptation of components. This process of reconfiguration consists of changing property values of components or changing the application architecture for redefining the connections between components.

The adaptation is usually the replacement of a component by another one that may provide and require different contracts than the replaced component. This adaptation can possibly require the disconnection of a component from the application to transfer its current state to the new component and then connect it to other application components. Usually, the adaptation operation requires stopping the application as well as resuming the life cycle from the assembly phase. Dynamic adaptation [7] [19] consists of introducing changes in the application during its execution without completely stopping the application. One of the delicate points of dynamic adaptation is the state transfer of a component to another when state comprises objects that are difficult to transfer such as a socket or a thread execution stack.

Adaptation is becoming a common requirement in the life cycle of applications for which their designers are looking for properties such as sensitivity to environmental changes (context-awareness) [13] or autonomic properties (self-*) [20]. The latter generally requires dynamic placement of probes in the software during operation or also changes in the dynamics of the self-management rules. The adaptation has to meet various constraints such as security, reliability, completeness and error recovery, which must take into account the possibility of turning back in the case where the adaptation can not be completed successfully.

Scripting languages [1] offer the strong capability of changing an application during its execution. They are used in various contexts like server side programming; dynamic generation of HTML pages; GUI programming; dynamic invocation of remote services; programming process flows and work flow scripts, and so on. The brevity and simplicity of writing programs makes scripting attractive [31] to the eyes of novice developers and business experts (who are not necessarily software engineers). A language paradigm index [36] based on search engines shows an increasing trend on dynamically typed languages. Other evidences of such trends are the revival of dynamic languages by academics [28] and also an industry shift towards such languages, as seen in the Java platform with the standardization of a scripting API (JSR 223) [18] which has been integrated into Java 6. A Virtual Machine project [11] focusing on extending the JVM for better efficiency of dynamic languages will introduce extensions in upcoming versions of the Java platform. In the .NET platform there is also an official effort called Dynamic Language Runtime [24] aiming to facilitate the integration of dynamic languages into the Common Language Runtime.

The property in which we are interested in this article is that these languages allow incremental programming during the execution of the developed application [32]. The developer can address simple aspects of application evolution without going through the whole process of the traditional iterative cycle development-testing-deployment-execution. However, we do not state here that such traditional models are bad or have to be replaced by a dynamic approach. We rather defend Meijer and Drayton's idea [25] of using such dynamic languages when necessary in a given context. In the context of dynamic adaptation, the incremental programming provided by scripting languages is of major interest. It allows the evolution of application code (functional or non-functional) and relieves the designer from the often *ad hoc* mechanisms for transferring state and restarting components whose behavior (functional code) has been changed. Moreover, it

does not call into question the linkages between components.

We argue that incremental programming offered by scripting languages is interesting in several points for the dynamic adaptation of components. Among other scenarios, it is particularly attractive for rapid prototyping; for developing programs of single use or restricted to limited runs (e.g. workflows, macros) by the script author; and for programs that use an ephemeral environment that is not known in advance. Through scripting languages, application components can evolve functionally and gradually as the context changes, without requiring redeployment. Component-oriented programming gives an opportunity to increase scripting language usage in large scale software development.

To do this, we have to look at the impact of using incremental programming with components models. We revisit two components models recognized by relation to inputs herein described: Fractal [6] defined by the OW2 consortium, and Declarative Services (DS) [30] specified by the OSGi Alliance.

This article is structured as follows: Section 2 presents our proposal for extensions using scripting languages on component models. Section 3 presents FractScript[1], a Fractal-like implementation of a component model for the development of component primitives with scripting languages. Section 4 presents SOSOC[2] (Service Oriented and Script Oriented Components) which extends OSGi's service oriented component model (Declarative Services) by adding the support of scripting languages for programming services. Section 5 shows a performance evaluation of our scripting approach. Section 6 highlights related work and, finally, Section 7 concludes with some perspectives.

II. PROPOSED APPROACH

Existing component models are mostly based on the notion of type, on common design patterns and in principles like inversion of control. The type of a component is often linked to its facets (provided interfaces) and receptacles (required interfaces). The factory is responsible for building the instances of a component type while the typing ensures the appropriate binding between components and also their correct substitutions. The controllers allow the machine that executes the model (runtime) to control these instances. The notion of service contracts [2] reinforces the typing and allows the decoupling and the brokerage of components.

Our proposal looks to revisit these notions in the context of scripting languages to allow a more generic support of external and internal adaptation at runtime. The external adaptation allows the evolution and changing of facets and receptacles, while the internal adaptation allows the evolution and changes in the components' internal behavior (method updates) as well as adding or removing state variables without impacting the value of existing variables.

[1] http://svn.forge.objectweb.org/cgi-bin/viewcvs.cgi/fractal/sandbox/donsez/fractscript/

[2] http://svn.apache.org/viewvc/felix/sandbox/donsez/sosoc/

A. Controllers

The external and internal adaptations are driven by four controllers, which are depicted in Figure 1: `FacetController`, `ReceptacleController`, `ScriptController` and `PropertyController`. These controllers should be exposed by the target component model container. Thus they can be present either as a replacement or as an extension of existing container mechanisms.

The `FacetController` allows the addition and removal of facets to a component instance. The properties of a facet are used when the model supports brokering. The method `setFacet` can also change an existing facet by adding and removing interfaces in a facet signature. The handler of the `setFacet` method specifies an object for dealing with the call of the method when not implemented by the script or when an exception is thrown. For better illustrating, we take the example of the FractScript implementation seen in Figure 2. The `HttpServletHandler` instance (line 6 of code snippet) is responsible for handling such errors if a non implemented method is called, for example `Servlet.init()`, which is not present in the script definitions in Figure 2. In that example, only the `service(req,res)` method is defined for the component, as defined in the parameter script of line 8 and redefined in line 14. If no handler is available and a non implemented method is called, the exceptions are propagated to the scripting engine.

The `ReceptacleController` allows the addition and removal of receptacles in the component instance. Like the facet, the method `setReceptacle` allows the addition and removal of interfaces in a receptacle. The filter of a receptacle is used for facet searches when the model supports trading (like in OSGi Declarative Services) and specific property attributes help locating the appropriate service. The filter may be a boolean expression or an integer expression on the facet's properties.

The boolean expression eliminates facets failing to evaluate the expression while the integer expression performs a ranking among several candidate facets. Some component models targeting the OSGi platform, like ServiceBinder [8], Declarative Services [30], FROGi [12] and iPOJO [14], for example, indicate if the binding of a receptacle to a component facet is mandatory or optional. The binding makes the life cycle of the component dependent on the presence of that linkage, respectively having its state as active or passive depending on the presence or absence of the linkage. This behavior can be specified by defining the controller transitions based on customizable dependencies of a component with its environment.

The `ScriptController` allows to incrementally evaluate any script hat uses a scripting language supported by the execution environment (i.e. usage of the underlying scripting engine). These scripts or script fragments generally contain the definition of the functions corresponding to the operations of facet interfaces. In other words, it is possible to dynamically add code to existing components. The script can also contain declarations of variables which allow you to add state variables. The method `evaluateScriptText` can be invoked several times in succession to add the definition of new operations or to replace the current definitions of existing operations. The

reset method removes the definition of the operations, variables and their states.

The `PropertyController` allows the addition and removal of properties in the component instance. These properties are manipulated in the form of variables in the script. The controller also queries and positions the value of a property. The changing of a property value can cause type conversion errors if the language structure does not support the type of the new value or if the scripting language does not support variable type changes.

```
interface FacetController {
 String[] getFacetNames();
 void setFacet (String name,
                String[] interfaces,
                Map properties,
                Handler handler);
 String[] getFacetInterfaces(String name);
 Map getFacetProperties(String name);
 void removeFacet(String name);
}

interface ReceptacleController {
 String[] getReceptacleNames ();
 void setReceptacle (String name,
                String[] interfaces,
                String filter,
                boolean multiplicity);
 String[] getReceptacleInterface(
                String name);
 String getReceptacleFilter(String name);
 void removeReceptacle (String name);
}

interface PropertyController {
 String getPropertyNames();
 Object getProperty(String name);
 void addProperty(String name,Object val);
 void removeProperty(String name);
}

interface ScriptController {
 void setLanguage(String language);
 void evaluateScriptText(String text);
 String getScriptText();
 void reset();
 void reset(String language);
}
```

Figure 1. Controller interfaces

B. *Types*

The type of a component is usually associated with its list of facets and its list of receptacles. In service oriented components, this is limited to the list of facets. In our context, both `ReceptacleController` and `FacetController` allow evolving (adding or removing) the list of facets and receptacles of a component. The type of a component is no longer considered immutable as in many component model implementations where the component type remains the same throughout its lifetime.

A facet consists in a set of interfaces provided by the component, which can be added or removed dynamically. Like so, a receptacle consists in a set of interfaces which are used by the component, and can be dynamically changed by adding new interfaces or removing existing ones. As long as the underlying script behind the component provides the corresponding methods of the component receptacles, the calls on the receptacle type will work properly.

One consequence if this dynamic changing is the possible introduction of errors in the architecture when using of these operations. The architecture must be checked dynamically after each mutation in order to prevent that.

C. *Prototypes*

Generally, component models use instantiation principles from class languages: the instances of a component are created by a factory associated with the type and the implementation component. Instances of a component have the same type and same functional behavior throughout their lifetime. For our approach of evolving components, from the viewpoint of type and behavior, a component factory can not keep this role. It is replaced by a factory prototype. In our context, a prototype is a pattern of components having an initial type (its facets and receptacles), an initial behavior (i.e. script run at instantiation) and an initial list of controllers. The type and behavior of each instance can mutate independently by the action of the controllers on the instance. The controllers can also be added or removed throughout the life of a component instance.

Open questions that we have not addressed yet are the evolution of the factory prototype and component instantiation by cloning existing instances. The former would allow evolving all instances of a component which have been created from a given prototype component together. While the latter would allow duplicating the internal state and behavior of an already established component, while using in the new instance the same facet and receptacle bindings that exist in the component to be cloned.

III. FRACTSCRIPT

FractScript is an implementation of our proposition that uses a Fractal-like approach. Fractal [6] defines a framework for building applications using a component model that supports the creation of hierarchical compositions. The Fractal design framework provides an API allowing the definition of types of components, the fabrication of instances from these types, the configuration of these instances and the connection between them. A Fractal component provides and requires functional interfaces, which are related to the application's logic and are in the form of server interfaces (i.e. facets) and client interfaces (i.e. receptacles). The component also provides a set of control interfaces called controllers.

The list of control interfaces includes, among other things, an interface dedicated to the life-cycle management (`LifecycleController`), the linking of client and server interfaces between instances of components (`BindingController`) and control of content (`ContentController`) in the case where the component is a composite. An interface dedicated to the configuration of attributes (`AttributeController`) is equipped with accessors and modifiers in the form of setXX / getXX where XX is the name of the attribute.

Several instances of a Fractal component can be created from a factory associated to a type of component. The Fractal specification has been the subject of

implementations in Java and other languages. Examples of these are Julia, AOKell, Fractal / Proactive, Fractalk and Cecilia.

```
// Example 1: A FractScript component
1 HttpDispatcher hd = ...
  ...
  // Component is instantiated
  // by the prototype factory
2 Object comp=Prototype.newInstance(
    new String[]{
    ScriptController.class.getName(),
    FacetController.class.getName(),
    BindingController.class.getName(),
    LifeCycleController.class.getName(),
    },false
    );
3 FacetController fc=(FacetController)comp;
4 ScriptController sc =
          (ScriptController)comp;
5 LifeCycleController lc =
         (LifeCycleController)comp;

   // Facet is added after instantiation
6  fc.setFacet("s", new String[]{
         HttpServlet.class.getName()},
         null, new HttpServletHandler());
   ...
   //behavior added after instantiation
7  sc.setLanguage("JavaScript");
8  sc.evaluateScriptText(
    "function service(req,res){"+
    " var out=res.getOutputStream();"+
    "out.println(\"<html><body>Hello"+
    " World </body></html>\");}"
    , false);

   // Facet is bound to other component
9  ((BindingController)hd).bindFc(
   "servlet",
   ((Component)comp).getFcInterface("s"));
10  lc.startFc();
11  ((LifeCycleController)hd).startFc();
   ...

   // A variable and a function are added
   // to the already existing script
12 lc.stopFc();
13 sc.evaluateFc(
    "var counter=0;"+
    "function incr(){ return ++counter; }"
    , false);
   // service(req,res) function is replaced
14 sc.evaluateFc(
    "function service(req,res){"+
    "var name=req.getParameter(\"name\");"+
    " var out=res.getOutputStream();"+
    " out.println(\"<html><body>"+
    "Hello \"+name+\""+
    " \" (\"+incr()+\")</body></html>\");"+
    "}" , false);
15 lc.startFc();
```

Figure 2. Fractscript example

One of the primary objectives of our proposition, FractScript, is the usage of primitive components developed with scripting languages in Java implementations like Julia and AOKell. FractScript completes the base membrane of scriptable component-based controllers with the four controllers described in the previous section: `FacetController`, `ReceptacleController`, `ScriptController` and `PropertyController`. These controllers may also interact with other Fractal controllers (e.g. `AttributeController`, `LifecycleController`, `BindingController`,).

The script of each component contains a variable that references the FractScript context giving access to the 4 controllers, as shown in the example code of Figure 2. The schema presented in Figure 3 provides an illustratation of FractScript's binding steps that take place in the code snippet from Figure 2 where the statements and corresponding line numbers can be seen in each step.

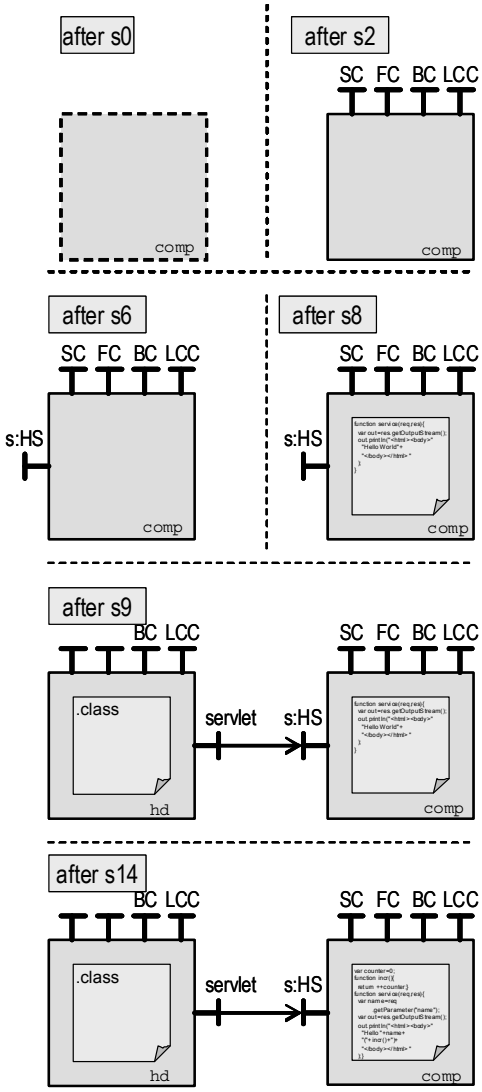

Figure 3. Dynamic evolution of a FractScript container from example 1 of figure 2

The current implementation of FractScript is limited to the scripting languages with engines compliant to the Java Platform scripting API previously mentioned. Script interpreters, also called engines, are accessible from Java context allowing it to execute scripts. This solution is not limited to the scripting API which is available in Java 6. Code using previous Java platform versions could also execute scripts by using a Java Native Interface [2](JNI) that provides scripting engines like Mozilla JS DLL for JavaScript in Windows.

The membrane component of FractScript is based on a Java dynamic proxy. It implements several synchronization policies between alternative controllers.

The factory prototype is currently implemented. Each scripted component instance can also be created from a blank prototype (having just the `FacetController`). The current implementation handles the controllers as facets for simplicity and unification. Each instance of a FractScript component has its own Type object that can evolve depending on the handling of the `FacetController` and `ReceptacleController` controllers.

IV. SOSOC

A second implementation of our approach, called SOSOC (Service Oriented and Script Oriented Components), was built targeting the OSGi Services Platform [30]. The OSGi specification provides a framework for the implementation and dynamic deployment of components and services. Initially conceived for building home automation gateways, this specification is also used for the development of modular Java applications such as Eclipse plugins and the core of several Java EE application servers (e.g. JOnAS 5, Weblogic, Glassfish 3). In OSGi's vocabulary, components are called *bundles*, which are regular jar files, like any ordinary Java application. The main difference between a regular jar file and a jar that represents an OSGi bundle lies in the jar manifest file, which holds attributes specific to the OSGi framework like required packages (type dependencies), provided packages, bundle vendor, bundle description and so on.

Application development with OSGi follows the principles of service-oriented programming [3], which allows the construction of applications from weakly coupled software entities and possibly using third party code. These entities, called services, offer functionality that is contractually described. The OSGi service contract is a set of Java interfaces qualified by key/value pairs enabling service filtering. The contract allows the selection of services relevant to the application. Brokering and binding are performed just at the time of use (late binding) and any service instance can be replaced by another implementation of the same contract. In OSGi, the code in the application components (which can themselves provide services) refers directly (i.e. without a proxy) to the objects implementing the services, thus a service method invocation is done without any extra overhead.

Services are considered as independent because the initiator of the application using services has no control over their administration. In addition, the life cycle of services is not necessarily synchronized with the application's lifecycle. In these conditions, dynamic service-oriented programming presumes that new services can appear while the application is running and that services that are being used may temporarily or permanently disappear, before the completion of the application.

The handling of such dynamicity is performed by the developer and is a task that frequently leads to errors. The OSGi specification, release 4, has introduced a model called Declarative Services (DS a.k.a SCR for Service Component Runtime) [16], largely based on ServiceBinder [8]. DS helps to address the management of dynamic connections to required services and the life cycle which is based on the presence or absence of the required mandatory services. In this model, the provided and required services are immutable and implemented in the Java language (as any other standard OSGi service).

SOSOC uses our proposed approach adapted to the concepts of OSGi's Declarative Services, enabling the development of dynamic service-oriented components using scripting languages. Component instances are created from the XML description file for a prototype component, as seen in the example depicted in Figure 4.

```
<!-- Example 2: a SOSOC component -->
<component name="Sample.SOSOC.Component"
  xmlns="http://www.osgi.org/xmlns/scr/v1.0.0"
   xmlns:sosoc=
   "http://felix.apache.org/xmlns/sandbox/soso
c/v1.0.0">
   <sosoc:implementation
language="javascript">
   <![CDATA[
     // a Java reference to the bundle context
     var bdlcontext;
     // the Java reference to the HelloService
     var service;
     // Command functions
     function getName(){
       return "hello";
     }
     function getUsage(){
       return "hello <msg>";
     }
     function getShortDescription(){
       return "invoke a HelloService";
     }
     function execute(commandLine,out,err){
       if(service!=undefined){
         msg=service.sayHello(commandLine)
         out.println(msg);
       }
     }
     // Binding/unbinding functions
     function bindHelloService(hs){
       service=hs;
     }
     function unbindHelloService(hs){
       service=undefined;
       // undefined is null in JavaScript
     }
     // Lifecycle functions
     function activate(componentCtx){
       bdlcontext=
componentCtx.getBundleContext();
       java.lang.System.out
         .println("call activate() function");
     }
     function deactivate(componentContext){
       java.lang.System.out
         .println("call deactivate function");
     }
   ]]></ sosoc:implementation>
       <provides
    service="org.apache.felix.shell.Command"/>
       <property value="service.pid"
                 name="sample.command.hello"/>
       <property value="category"
                 name="command"/>
     <!-- Required services -->
     <reference name="HELLO"
       interface="sample.hello.HelloService"
       cardinality="1..1"
       policy="dynamic"
       bind="bindHelloService"
       unbind="unbindHelloService"
     />
   </component>
```

Figure 4. SOSOC example

The component instance registers a service into the OSGi platform. These interfaces can be added or removed from the service by manipulating the FacetController. The four controllers are available from the component (functional code) through the context object.

The component container is realized by means of a Java dynamic proxy available as a service registered in the OSGi platform. It shares part of its code with the implementation of FractScript and also relies on the Java 6 scripting API. Changes on the required service and controller are notified to the other application components via the service catalog of the platform. The addition of a new interface requires that the current service proxy be unregistered from the OSGi platform so the service consumers can release their references to the previous proxy, a new dynamic proxy is generated with the additional interfaces and then re-registered in the platform. The previous proxy is invalidated (i.e. new calls to its methods will fail since it is no longer valid), allowing to find stale references [16], if any.

An interface withdrawal does not require proxy re-generation, since modifying the objectClass property associated with the service registration in OSGi is enough. This brings the advantage of not losing the service's internal state, which is not the case in a regular OSGi service update. An invoked method which is not present in the given service registration would raise a `NoSuchMethodException`.

V. PERFORMANCE EVALUATION

The usage of scripting languages has surely a performace penalty due to its interpreted nature. For that reason we wanted to evaluate that in one of the two propose solutions by comparing it to similar component approaches in the Java platform that are not built on top of scripting languages. In order to do so, we have perfomed a microbenchmark (run on an Intel Centrino Duo T2400 1.83 GHz, 2GB of RAM, WinXP Operating System, Java HotSpot Client Virtual Machine 11.0-b16, mixed mode, sharing), which is available online (svn://scm.gforge.inria.fr/svn/aofractal/JACbench), to evaluate the performance of calling a set of methods with different signatures in FractScript server components, on regular Fractal components, direct Java method calls, calls on static proxies, and calls on dynamic proxies. For these last three (presented in lines 1, 2 and 6 of Table I, respectively) there was no component model involved. Table I shows the measurements of the same microbenchmark that was perform in the different scenarios previously mentioned.

TABLE I. MICROBENCHMARK RESULTS

	Type of method call	Time (ns/call)
1	Java direct	125
2	Java static proxy	203
3	AOKell Spoon (Fractal)	234
4	AOKell AspectJ (Fractal)	781
5	Fractal (Julia)	1250
6	Java dynamic proxy	1640
7	FractScript/Javascript	8188

FractScript's flexibility has a performance tradeoff, as shown in the microbenchmark. The individual usage of Java's dynamic proxies already shows a significant overhead, as seen in line 6. Since FractScript combines that approach with script interpretation, this naturally leads to a result (line 7) with much more overhead and being significantly slower than lines 3 to 5, which measured Fractal implementations (http://fractal.objectweb.org/).

VI. RELATED WORK

The idea of using scripting languages in CBSD is not new. Nierstrasz et al [29] use a scripting model for binding the relationships between components. In that work, the scripts themselves are also considered as components encapsulating functionality that can be used in other scripts. Similarly, different scripting languages can be used as glue code in the Bean Markup Language (BML) [10], a declarative language for the composition of JavaBeans.

FScript [15] is a scripting language dedicated to the specification and consistent reconfiguration of Fractal component assemblies. SAFRAN (Self-Adaptive FRActal compoNents) is used as a language support to achieve self-adaptive aspects on Fractal components. The scripts can be modified dynamically from the console. However, the FScript language remains dedicated exclusively to the adaptation of components.

XPDL [38] (XML Processing Description Language) and BPEL [4] (Business Process Execution Language) are workflow targeted languages which use the flexibility provided by scripting languages. External scripts (e.g. written in JavaScript) can be defined in XPDL as part of a process and be invoked. BPEL has its own scripting language (BPELScript) based on an XML schema, allowing to write or to generate activities.

Microsoft's Windows Script [23], formerly called Scriptlets, allows to easily instantiate reusable COM components by using scripting languages, such as JScript or VBScript, supported by the Windows Script Host. Component properties and its script implementation are defined as an XML file which can be registered as a component.

LOOP (Lua Object Oriented Programming Model) [22] provides a component model for Lua [17], an extensible dynamic language. The composition in LOOP is also based on Facets and Receptacles, which are its basic set of ports. Templates specify the set of ports provided by a component, which is instantiated by a Factory that is constructed based on the component's template.

The CoSi component model [5] is another recent component model approach built on top of Java that similarly to our work takes advantage of a dynamic language (Groovy) for addressing component substitutability.

SCA [33] (Service Component Architecture) allows modeling components and applications that use a Service-Oriented Architecture. SCA deals with heterogeneous components which may even be implemented in a scripting language. Such scripted components can be defined in an extension of SCA's Service Component Definition Language (SCDL). The Apache Tuscany [37] SCA implementation allows components implemented in any language supported by Java's Scripting API. That approach remains static in terms of component typing (required and provided services) and behavior.

VII. Conclusion and Perspectives

Dynamic language usage is increasing in the software industry. Advantages brought by these languages such as easier hot swapping of code and rapid prototyping can be used for dynamic adaptation in software component models. In this article, we have looked at the impact of the usage of scripting languages on concepts for control and type instantiation attached to sample components. We revisited two recognized component models in relation to proposed extensions: Fractal, defined by the OW2 consortium, and Declarative Services, specified by the OSGi Alliance. This has resulted in two implementations: FractScript and SOSOC. Although such an approach brings significant flexibility for dynamic adaptation in component based applications, the utilization of scripting languages has performance costs, which have been analyzed and shown in the results of the microbenchmarks that we have performed.

Next directions to this work would be evaluating the usability of components with scripted applications: the autonomic managers and the mediation architectures in areas such as Machine-to-Machine (M2M) [21]. In both areas, components mainly target event-oriented filtering and pattern recognition of periodic events. The approach we propose is planned to be evaluated in the context of the OW2 AspireRfid middleware (http://aspire.ow2.org/) where the components using this approach would provide reconfigurable and dynamically deployable rules based on scripting for handling application events triggered by tag scans and sensor measurements (e.g. a given tag scanned when the temperature was above a certain temperature would trigger an event which would be delegated to a script).

Another perspective of this work is the use of scripting components to support dynamic architectures. The idea behind this is to describe different possible changes and to conduct operations means of dynamic languages, allowing the modification of the rules governing the architecture running an application without being forced to restart it. Other implications (e.g. system stability, bugs) as a consequence of supporting such dynamic evolution during execution was kept out of the scope of this paper, but are of major importance and should be taken into account in future work.

Acknowledgment

The authors would like to thank the anonymous reviewers for their invaluable comments which helped to improve this paper. Part of this work has been carried out in the scope of the ASPIRE project (http://www.fp7-aspire.eu) which is co-funded by the European Commission in the scope of the FP7 programme under contract number 215417. The authors acknowledge help and contributions from all partners of the project.

References

[1] D.W. Barron, The World of Scripting Languages, John Wiley & Sons, 2000.

[2] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. “Making Components Contract Aware”. Computer 32-7, IEEE Computer Society Press, July 1999, pp 38-45, doi: 10.1109/2.774917

[3] G. Bieber and J. Carpenter, "Introduction to Service-Oriented Programming", Online Whitepaper, http://www.openwings.org/donwload.cfm

[4] Business Process Execution Language (BPEL). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[5] P. Brada. “The CoSi Component Model: Reviving the Black-Box Nature of Components”. 11th International Symposium on Component-Based Software Engineering (CBSE 08), LNCS 5282, Springer-Verlag, Berlin-Heidelberg, 2008, doi: 10.1007/978-3-540-87891-9_22

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.-B. Stefani. "An open component model and its support in Java". 7th International Symposium on Component-Based Software Engineering (CBSE), LNCS 3054, pp 7-22, May 2004.

[7] J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel, "Towards a Taxonomy of Software Change", Journal of Software Maintenance and Evolution: Research and Practice, JohnWiley & Sons, 2003, doi:10.1002/smr.319

[8] H. Cervantes and R. Hall, "A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences", 7th Symposium on Component-Based Software Engineering (CBSE), Scotland, 2004

[9] Common Component Architecture (CCA) Forum, http ://www.cca-forum.org/

[10] F. Curbera, S. Weerawarana, and M. J. Duftler. “On component composition languages”. 5th International Workshop on Component–Oriented Programming (WCOP), May 2000.

[11] The Da Vince Machine Project. http://openjdk.java.net/projects/mlvm/

[12] M. Désertot, H. Cervantes and D. Donsez, "FROGi: Fractal components deployment over OSGi", 5th International Symposium on Software Composition (SC 2006), LNCS 4089, Austria, 2006, doi:10.1007/11821946_18

[13] Dey A., Providing Architectural Support for Building Context-Aware Applications. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[14] C. Escoffier, R. S. Hall and P. Lalanda. “iPOJO: An extensible service-oriented component framework”. IEEE International Conference on Service Computing. Salt Lake City, USA, 2007, pp. 474-481.

[15] FPath and FScript. http://fractal.objectweb.org/fscript/

[16] K. Gama and D. Donsez. “A Practical Approach for Finding Stale References in a Dynamic Service Platform”. 11th International Symposium on Component-Based Software Engineering (CBSE), LNCS 5282, Springer-Verlag, Berlin-Heidelberg, 2008, doi: 10.1007/978-3-540-87891-9_16

[17] R. Ierusalimschy, L. H. de Figueiredo and W. C . Filho. “Lua—an extensible extension language”. Softw. Pract. Exper. 26, 6, 1996, pp. 635-652, doi: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

[18] JSR 223: Scripting for the JavaTM Platform. http://jcp.org/en/jsr/detail?id=223

[19] M. Ketfi, N. Belkhatir, P.Y. Cunin, "Automatic Adaptation of Component-based Software: Issues and Experiences", PDPTA, Las Vegas, Nevada, USA, 2002

[20] J.O. Kephart, D.M. Chess, "The Vision of Autonomic Computing", Computer 36, IEEE Computer Society Press, Jan. 2003, pp. 41-50, doi: 10.1109/MC.2003.1160055

[21] G. Lawton, "Machine-to-machine technology gears up for growth," Computer, vol.37, no.9, Sept. 2004, pp. 12-15, doi:10.1109/MC.2004.137

[22] LOOP: Lua Object Programming Language. http://loop.luaforge.net/

[23] Microsoft Developer Network, Script Components, http://msdn.microsoft.com/library/asxw6z3c(VS.85).aspx

[24] Microsoft Dynamic Language Runtime. http://dlr.codeplex.com/

[25] E. Meijer, and P. Drayton, “Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages”, OOPSLA Workshop On The Revival Of Dynamic Languages, Vancouver, Canada, 2004
[26] OMG, "CORBA Component Model Specification, Version 4.0", http ://www.omg.org/docs/formal/06-04-01.pdf
[27] O. Nierstrasz, S.J. Gibbs, D. Tsichritzis, "Component-Oriented Software Development", CACM, Vol. 35, Number 9, September 1992, pp. 160-165
[28] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gaelli, and R. Wuyts, "On the Revival of Dynamic Languages", 4th International Symposium on Software Composition (SC05), LNCS 3628, 2005, pp. 1-13
[29] O. Nierstrasz., D. Tsichritzis, V. de Mey, M. Stadelmann, "Objects + Scripts= Applications", Esprit 1991 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1991, pp. 534-552
[30] OSGi Alliance. http://www.osgi.org
[31] L. D. Paulson, “Developers Shift to Dynamic Programming Languages”. Computer 40-2, IEEE Computer Society Press, Feb. 2007, pp. 12-15, doi: 10.1109/MC.2007.53
[32] V. Savikko. “Generative and Incremental Approach to Scripting Support Implementation”. International Conference on Software Engineering Research and Practice, SERP '03, CSREA Press, Las Vegas, Nevada, USA, 2003.
[33] Service Component Architecture (SCA). http://osoa.org/display/main/service+component+architecture+home
[34] Sun Microsystems. Enterprise Java Beans. http://java.sun.com/products/ejb/
[35] C. Szyperski, D. Gruntz, and S. Murer. Component software: beyond object-oriented programming, Addison-Wesley, 2nd edition, 1998.
[36] Tiobe Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/
[37] Apache Tuscany. http://tuscany.apache.org/
[38] XML Process Definition Language (XPDL). http://www.wfmc.org/xpdl.html