Extensible Polyglot Programming Support in Existing Component Frameworks

Jaroslav Keznikl*†, Michal Malohlava*, Tomáš Bureš*†, Petr Hnětynka*
* Faculty of Mathematics and Physics, Charles University in Prague
Malostranské náměstí 25, 118 00 Prague 1, Czech Republic
†Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic
{keznikl,malohlava,bures,hnetynka}@d3s.mff.cuni.cz

***Abstract*—Utilization of various agile development practices brings demand of short development cycle with stress on early deployment and rapid delivery. Such practices require techniques permitting rapid prototyping of systems, corresponding tests and simulations. One of well-adopted rapid prototyping techniques is a *polyglot programming* combining multiple, mainly scripting, languages during development of systems. This permits the use of a specialized language for dedicated system concerns and also allows for continuous change (re-deployment) of implementation. Despite the advantages, rapid prototyping with help of polyglot programming is still not well integrated in the domain of component-based systems, which makes it difficult to quickly prototype and test these systems.**

To address this, the paper describes a general technique for transparent and extensible combining of multiple languages for purpose of rapid prototyping during development of component-based systems with help of advanced component frameworks.

***Index Terms*—components, rapid prototyping, polyglot programming, scripting languages, controllers, interface interceptors**

I. Introduction

Nowadays agile development methodologies often incorporate rapid prototyping as a method of early deployment and fast delivery of working solutions. Furthermore, it is often utilized for demonstrating new ideas, writing system tests, simulation or mocks. The cornerstone of rapid prototyping is an "elastic technology" which can be easily utilized in different contexts, for different concerns. Furthermore, such technology have to be inherently dynamic to allow on-the-fly changes of the implementation or re-deployment.

One of well-adopted directions of rapid prototyping is *Polyglot programming* [26], [27] incorporating multiple languages for building systems. Its idea is to utilize dedicated, especially scripting/dynamic, programming languages for particular parts of system representing different concerns. Furthermore, utilization of scripting languages permits fast development without long cycle between change and its deployment in a running system. A typical example is JavaScript [11] which is primarily used in web development to improve HTML based user interfaces and incorporate asynchrony into the stateless HTTP protocol; or the Groovy programming language which is largely utilized for writing system unit tests. All these languages have often mature tooling (e.g., IDE, debugger) and a large user community which makes the use of these languages easier.

Polyglot programming may be found for example in today's enterprise systems (e.g., combination of JavaScript for client side web-browser scripting, Python for server-side content generation, and Java for core framework tasks), as well as in platforms for service oriented architectures (e.g., ESB [24], SCA [1]).

Furthermore, Polyglot programming allows for integration of scripting languages, which bring the benefits of rapid prototyping and dynamic update of the implementation.

Despite the spread of polyglot programming, as far as advanced component-based frameworks like EJB [21], OSGi [23], Fractal [10], SOFA 2 [5], JBoss [9], COM [16] (i.e., those with existing runtime environment often called as *component container*) are concerned, the possibility of combining components implemented in different programming languages in a single application is still not usual. This fact not only makes it difficult to rapidly prototype new components, but also to write test- and mock-components for building testing environments. This is primarily caused by the fact that (i) either the component frameworks were not designed with such heterogeneity in mind (e.g., Fractal, EJB), or (ii) the component framework supports heterogeneity, but requires a dedicated component container for each language (e.g., CORBA and CCM [20] or SOFA 2), which turns out to be very heavyweight and difficult to manage and use.

The polyglot programming was partly solved by the introduction of virtual machines and common runtime (e.g., .NET CLR or JVM). This allows for binding on the byte-code level, thus making the combination of different languages invisible for the concerned component framework and components. However, the problem still persists when a particular language does not have a compiler for compilation to the byte-code – e.g., it is purely interpreted, possesses features which cannot be mapped to the byte-code or the compiler just does not exist.

These problems are especially true for JVM, for which only a few languages may be integrated on the byte-code level (currently Java, Groovy, Scala and Clojure[1]), but many other languages are already supported by interpreters implemented in Java (e.g., JavaScript, PHP, Python, Ruby, Erlang, Prolog)

[1]More languages are likely to gradually appear after the assumed release of Java 7 in 2011. However, the introduction of every new language will still require substantial effort in development of its compiler to the byte-code.

or via JNI binding (C, C++, R, Fortran, Ada, MATLAB). The result of it is that Java-based component frameworks still need some support for business code provided in different programming languages, which in turn means the necessity to partially re-implement concerned component frameworks.

However, there is an option to deal with the problem. A number of advanced component frameworks inherently support extension mechanism, which allows extending their runtime environments by introducing interceptors around the component's business code. The interceptors are responsible for relaying calls coming to and going out of an instantiated component. By reacting and possibly modifying the calls, interceptors may influence of the behavior of the component framework and enrich it of additional capabilities without having to change its core implementation.

A. Goals of the Paper

The paper contributes to the problem of rapid prototyping of component-based systems by introducing an extension to address the issue of polyglot programming with scripting languages in Java-based component frameworks that support user-defined interceptors. We provide a solution that allows seamless and transparent runtime integration of components written in different scripting programming languages inside one component container. Furthermore, the solution inherently provides a way of easy updates of component implementation at runtime.

The presented solution is generally applicable for Java-based component frameworks with the extension mechanism based on custom interface interceptors (e.g., Fractal, SOFA 2, JBoss, Spring, Castle). The solution is a pure extension, which does not require modification of component framework internals. Furthermore, the solution stresses separation of concerns by separating the support for different languages into component interceptors. Therefore, the component developer does not have to deal with any language integration and scripting frameworks – he or she just provides component implementation in a particular scripting language.

In addition to describing the solution concepts, we present results of a performance benchmark to assess the performance impact of our approach, and we discuss the lessons learned from implementing the solution for SOFA 2 component framework.

B. Paper structure

The rest of the paper is organized as follows. In the next section we define the component systems for which the presented approach is applicable. In Section III we analyze and generally describe the presented approach. Section IV presents a case implementation allowing scripting support in the SOFA 2 component system, evaluates the component developer experience and gives a brief performance analysis of the approach. In Section V we present our generally-applicable experience with implementation of the approach. In Section VI the related work is discussed. Finally, Section VII concludes the paper and outlines future work directives.

II. Prerequisites and definitions

Our approach allows introduction of components written in scripting languages into an existing component framework and their combination with the components native to the framework. This allows for rapid prototyping of selected parts of a component application.

We assume that the component container is extensible in the way that it provides the possibility of introducing custom *interceptors* on component interfaces. This has already become a de-facto standard way of extension, which is currently supported by many component frameworks (e.g., all Fractal-based component systems, SOFA 2, JBoss, Spring, Castle). Although, the terms used in each component framework differ to some extent, the essential feature of custom interceptors is the same — relaying of component interface calls. In this way, we can recognize a family of interceptor-based component frameworks, for which the solution presented in the paper is generally applicable.

To ease explanation, we use the terms featured in Fractal specification – we recognize a *component content*, which is the business logic of the component provided by the component developer, and a *component membrane*, which is the wrapper around the content that contains the interceptors (see Figure 1). The membrane thus displays the *component interfaces* on its outer boundary and relates them to the content. There are essentially three main kinds of interfaces: *provided business interfaces*, *required business interfaces*, and *provided control interfaces*. Business interfaces reflect the core business functionality of the component provided by the developer, thus they typically differ for each component. On the other hand, control interfaces (also termed *controllers*) reflect the management side of the component – e.g., lifecycle management, component bindings, interface queries (similar to Microsoft COM's IUnknown interface), etc. Controllers provide the component container and deployment and monitoring tools with an unified management interface for each component.

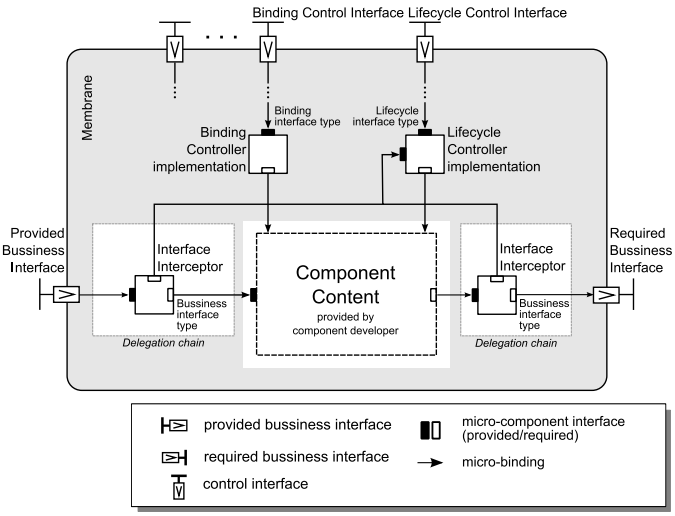

Fig. 1. Platform component overview

Interceptors in the membrane may intercept calls on the business interfaces as well as on the control interfaces. This interception is allowed to provide custom implementation of the interfaces being called. This is especially true in the case of control interfaces, where the interceptors implement the majority of functionality offered by the control interface, relaying only little (if anything) to the component content. The interceptors form together a kind of component architecture around the content. Thus we view them in our approach as a special kind of platform components, which are however at a different level of abstraction to the application components. To emphasize the component nature of interceptors, we term them *micro-components* (this term is used interchangeably with *interceptor*) and distinguish provided and required interfaces on micro-component boundaries.

Addressing the polyglot programming for components (i.e., the ability to develop components in dedicated languages), we essentially deal with two types of programming languages in our approach: (i) There is a language native for the component container. We call this programming language *container language*. (ii) By the introduction of polyglot programming, the component business functionality (embodied in the component content) may be provided in a language different to the container language. We call this programming language *content language*.

Referring back to the goals of the paper, we primarily focus on Java as the container language and either on Java or on a scripting language as the content language.

III. Analysis and Solution Design

The problem of integration of scripting languages into existing component frameworks (by the means of the polyglot programming) consists of two independent parts: (i) integration of the container language with the content languages, and (ii) integration with the component container.

The first part of the problem, the language-level integration, covers inherently two issues — data and control flow integration. With regard to polyglot support, the data flow integration comprises data type mapping, value conversion, integration of individual language concepts, access to objects in the other language, mapping of interfaces, etc. The control flow integration comprises integration of method conventions, handling of method return parameters (by value, by reference) and exception handling.

All these concerns are typically handled by an *adaptation framework*. For integration of scripting languages in Java-based containers, which we are primarily concerned with, such an adaptation framework is the Java Scripting API (JSA) [22]. It provides an unified interface for different scripting engines which are integrated in Java [15]. Each engine allows loading and executing code in its respective scripting language. Furthermore, it ensures data type mapping and instantiation and access to Java classes from the scripting language.

The second part of the problem of polyglot programming for components is how to integrate the adaptation framework (e.g., JSA) into the component so as it is used to execute the component code.

Since, the first part of the problem has been already sufficiently solved by adaptation frameworks (e.g., JSA in the case of Java), we focus on the second part of the problem only in the rest of the section.

A. Overall Design

Our approach introduces a specialized component runtime extension – further referred as a *polyglot extension* – which forms an adaptation layer between the component container and the component implementation in the content language. It creates a special environment inside a component membrane that allows executing code in the content language, which is different to the container language, using an adaptation framework. This environment is built using a set of membrane's controllers and interceptors and its general structure is shown in Figure 2. In more detail, the *polyglot extension* ensures redirection of the control flow to the adaptation framework and integration with related component controllers (e.g., life-cycle controller etc.).

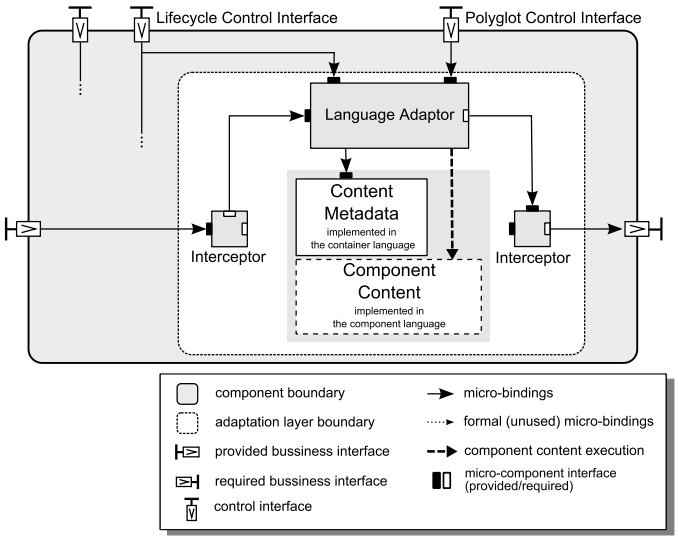

Fig. 2. Basic idea

The component content, i.e., the code in the content language, is not executable by the container directly. It is managed by the *language adaptor*, which loads the component content based on location provided by the *content metadata* and reroutes calls with the help of interceptors, which "surround" the content. The adaptor transforms calls into invocations of the adaptation framework that is part of the adaptor. The framework then executes the code in the content language and returns a result (or an exception), which is passed back via the interceptor. For required interfaces, the language adaptor similarly transforms and redirects calls from the component content to the corresponding interceptor. In the same manner, the polyglot extension allows propagation of control interfaces calls to the component content (from the low level view, control interface is just another provided interface).

Both the caller and the callee of the component, as well as the rest of the component controllers, do not note any difference between a common component and the one with the polyglot extension (except for possible type incompatibilities of interface call arguments and exception handling, which should be resolved by the author of the prototype implementation).

In the rest of the section, we describe each of the polyglot extension elements in more detail.

B. Provided Interface Interceptors

The interceptor on a provided interface redirects calls to the language adaptor. To do this, the interceptor passes the method identification and values of arguments of the called method. During the method call return, it returns the return value or an exception.

Each interceptor has three interfaces. Two of them – one provided and one required – have the same type as the intercepted interface. Using these interfaces the interceptor is connected in the membrane to the chain of interceptors for the particular interface. In normal case, the interceptor receives the call on the provided interface, performs its functionality, and forwards call on the required interface. However in our case, the interceptor stops the call and redirects it, i.e., the required interface is never used and in component systems that allow interceptors with a single interface only it can be omitted.

As the interceptor is based on the type of the intercepted interface, i.e., it is unique for each interface type, its implementation has to be either reflection-based or dynamically generated at deployment or instance time.

The remaining interface of the interceptor is the required interface via which the method call is redirected to the language adaptor. The type of this interface is the same for all provided interface interceptors. Basically, it contains one method which has two arguments, one for the intercepted method name and one for the original argument array, and which returns a generic return value (e.g., an `Object` reference). The interceptor implementation handles required type conversions.

C. Required Interface Interceptors

The required interface calls are (similarly to provided interfaces) redirected to the interceptor corresponding to the particular required interface.

As in the case of the provided interfaces interceptor, there are three interfaces on each required interface interceptor. Two of them – one provided and one required – are used for connecting the interceptor to the interceptors chain (again, the provided one in this case can be omitted if the component model allows it) and they have the type of the intercepted interface. The remaining interface has also the same type as the intercepted interface and is used for call redirections from the language adaptor.

Again, as the interceptor is unique for each interface type, its implementation has to be reflection-based or dynamically generated.

D. Control Interfaces

Certain controllers – like the life-cycle management one – may require access to the component content, i.e., call methods on it. For example in the case of the life-cycle control interface, it has methods for notifying the content about starting/stopping the component and the content can react for example by starting/stopping its internal threads.

As from the low level view, the control interfaces are the same as provided interfaces, the situation is almost the same as in the case of the business provided interfaces. The difference is that both the control interface interceptors and the language adaptor have to understand these control interfaces in order to invoke corresponding functions on the content (in case of the business interfaces, the calls are forwarded only based on their names and types of arguments).

E. Component Content

The presented approach allows achieving maximal transparency and simplicity of component implementation. The code implementing the particular component is represented by a regular source file containing purely code in the content language. The developer only provides this file and in the component description (e.g., in an ADL description based on the particular component model) he/she defines the identification of the used language.

For languages without interfaces and/or object support, the methods of the provided interfaces have to be implemented as global functions. The language adaptor can find the particular function using for example prescribed name convention.

The required interfaces are available via dependency injection which is managed by the language adaptor. For example for scripting languages, the required interfaces can be available as predefined variables, which are filled during the component instantiation by the language adaptor.

F. Content Metadata

Content metadata is a unit implemented in the container language. The role of content metadata is twofold: 1) it provides the content class required by the component container during component instantiation, and 2) it wraps component-specific information needed by the language adaptor and by itself forms a unit that is recognized by the component container as a component.

Thus, from the container's point of view, the content metadata has the role of component content and in fact the content metadata is typically a class implemented in the container language with all interfaces prescribed to the content by the particular component framework. From the point of view of the language adaptor, the content metadata holds information about the real component content, i.e., at least the identification of the content language and location of the content. The information is used by the language adaptor to properly configure the adaptation framework and to load the particular business code.

Additionally the existence of separate content metadata allows keeping the language adaptor independent of a particular component implementation. Additionally since the content

metadata does not directly deal directly with execution of the code in the content language (it only points to its location), the content metadata class is independent of the particular content language.

The content metadata may be generated automatically by an interceptor attached to component instantiation if the component container allows such an extension or by a dedicated tool during component development.

IV. Evaluation

A. SOFA 2 Case–Study

To validate the presented approach, we have implemented it as an extension of the Java-based SOFA 2 runtime (in SOFA 2 terminology, such an extension is called a component aspect), further referred as the *script aspect*, allowing component implementation in scripting languages using JSA as the adaptation framework.

The aspect architecture is shown in Figure 3. The aspect consists of several micro-components implementing the language adaptor controller (Call Transceiver, Script Invoker and various Proxy micro-components). These micro-components are responsible for integration with other component aspects, as well as for implementation of the language adaptor logic. The aspect also comprises designated micro-components representing the interface interceptors (which are referred as *script interceptors* in scope of polyglot support). The content metadata has form of a generic component content created and initialized by the component container for all components implemented in programming languages other than Java.

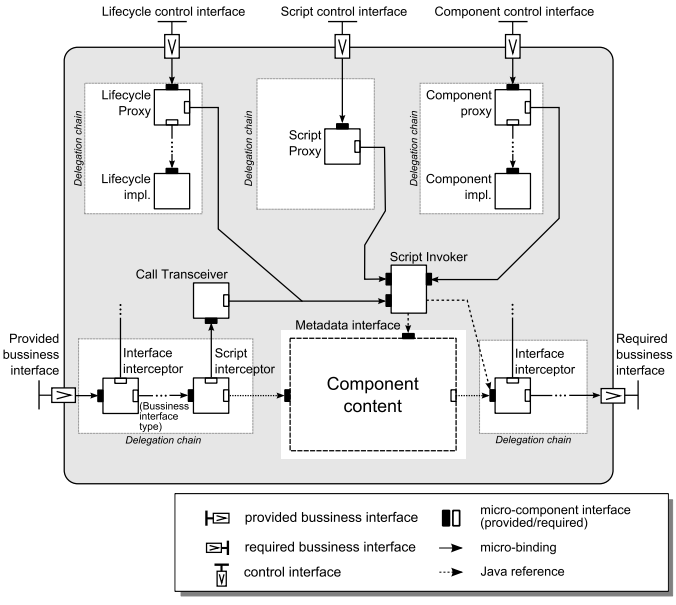

Fig. 3. SOFA 2 Script aspect architecture

The language adaptor is implemented as follows. The *Script Invoker* micro-component contains and manages the JSA scripting engine and transforms the incoming calls of the provided interfaces into the scripting engine invocations. The *Call Transceiver* micro-component manages all the interceptors of the associated component and redirects their notifications to the Script Invoker.

```
public interface Script {
    void evaluateScriptText(String text) throws SOFAException;
    void reset() throws SOFAException;
    void setValue(String name, Object value) throws SOFAException;
    Object getValue(String name) throws SOFAException;
}
```

Fig. 4. Script control interface (Java)

In SOFA 2 case, the interceptor implementation is dynamically generated using a designated interceptor generator (with help of the ASM [3] bytecode manipulation library).

For interaction with other component aspects (e.g., accessing the component content or reacting to the life cycle changes of the component), the script aspect intercepts the `Component` and the `Lifecycle` control interfaces. The former allows tracking changes of the component content in case of dynamic updates. The latter allows tracking lifecycle changes of the component (and therefore notifying the script code).

The script aspect introduces a new `Script` control interface (see Figure 4) which allows manipulation of the encapsulated scripting engine and the associated component implementation. Moreover, the Script interface is remotely accessible via Java RMI.

Together with a designated tool which we had developed in order to access this interface from command line (and thus to enable changes of implementation of a running component), we have successfully enabled the rapid prototyping of component implementation using scripting languages in our Java-based component system.

The script aspect implementation can be found on the official SOFA 2 website[2].

B. Component Developer Experience

To demonstrate the achieved language integration, we present a prototype implementation of a component from the CoCoMe [12] application in the SOFA 2 system. We focus on the *ReportingLogic* component. Its main goal is to provide reports (using its provided interface, see Figure 5) which are computed from data in a shared data storage (accessed using required component interfaces). When this component is being developed, its interfaces are already defined and thus it is possible to use the proposed polyglot support for rapid-prototyping of its implementation.

```
public interface ReportingIf {
    ReportTO getStockReportForStore(StoreTO sTO);
    ReportTO getStockReportForEnterprise(EnterpriseTO eTO);
    ReportTO getMeanTimeToDeliveryReport(EnterpriseTO eTO);
}
```

Fig. 5. Reporting interface definition (Java)

On Figure 6 we present a fragment of the ReportingLogic component prototype implementation using Python which

[2]http://sofa.ow2.org/extensions/index.html#dynamic

demonstrates the basic concepts of scripting language integration in SOFA 2. In scope of component implementation, it is possible to use Java classes using regular Python import statements (served by designated classloader). It is also possible to use additional Python modules and classes.

First we focus on provided interface implementation. In SOFA 2 case, the provided interfaces of scripted components have to be implemented as global script functions. This makes the implementation shorter than using Java (see `getStockReportForStore`).

```
from java.lang import Thread
from org...inventory.application.reporting import *
# Downloading additional python modules using SOFA importing API
SOFAPythonImporter.loadCodeBundle('org...application.reportinglib')
from org...application.reportinglib import HTMLReportPrinter
...
def getStockReportForStore(storeTO):
    result = ReportTO()
    pctx = persistmanager.getPersistenceContext()
    ...
    store = storequery.queryStoreById(storeTO.getId(), pctx)S
    items = storequery.queryAllStockItems(storeTO.getId(), pctx)
    store_name = store.getName()
    printer = HTMLReportPrinter()
    ...
    for si in stockitems:
        printer.addRow(...)
    ...
    result.setReportText(printer.getText())
    return result
...
class PeriodicReportGenerator(Thread):
    def run(self):
        while not stop_reporting:
            report = getStockReportForStore(find_store())
            periodic_publisher.publish_report(report)
            Thread.sleep(REPORT_PERIOD)
...
def start():
    thread = PeriodicReportGenerator()
    thread.start()
```

Fig. 6. Prototype component implementation (Python)

Next we focus on required interface access. The scripted component implementation uses automatic dependency injection where variables holding the provided interface references are created and assigned automatically during component initialization (using names in the associated component definition, see `storequery`, `persistmanager` and `periodic_publisher`).

To manage internal threads, a component in SOFA 2 has to implement the `SOFALifecycle` interface. This interface provides automatic notifications when the component is started or stopped. This interface also has to be implemented using global script functions (see `start` function managing the `PeriodicReportGenerator` thread).

To summarize, the use of scripted components is transparent, without any significant adaptation code needed. It is also generally shorter than Java implementation, since the scripting languages are typically more expressive and there is no service code needed for typical tasks such as implementation of provided interfaces, acquisition of required interfaces, lifecycle management, etc. Therefore, the implemented polyglot support is suitable for the rapid prototyping task.

C. Performance Evaluation

The proposed polyglot support introduces a performance overhead caused by scripting controller infrastructure and by execution of the script code in the scripting engine. To asses this overhead, we have created a simple benchmark which uses a JavaScript component to forward interface calls between two Java components (see Figure 7). We have repetitively measured the execution time of 10000 forwarded interface calls (before that, we have used 5000 calls to eliminate initialization and JIT overhead). Then we have calculated the average execution time of one invocation for each of the 40 measurements.

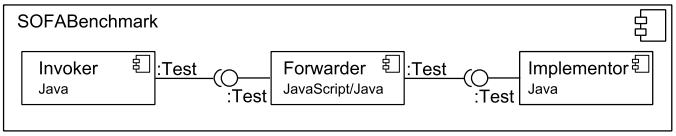

Fig. 7. SOFA benchmark architecture

We have compared the results with measurements of the same scenario which uses a standard Java component instead of the scripted one to forward the calls. For each measurement, we computed the difference between the corresponding average execution time of script forwarder and median of all Java-only measurements. The computed overhead, which is approximately 0.147 ms per one call, comprises both scripting engine overhead and the scripting extension infrastructure overhead.

To get an overhead approximation of the scripting controller infrastructure only, we have created a simple regular Java application implementing the previous scenario (both Java-only and Java/JavaScript version) using simple objects in place of components. In this case, the measured overhead comprises only the scripting framework overhead. For JSA using Mozilla Rhino version 1.6 release 2 it is approximately 0.059 ms per one call.

By subtracting this value from the average component measurement we get the average overhead of the scripting controller infrastructure – approximately 0.088ms per one call. This is 40% of the default SOFA controller overhead. To summarize, a prototyped scripted component has 40% bigger overhead than regular Java components, which is still acceptable in the development environment.

The measurements were performed on the following configuration: Intel(R) Core(TM)2 Duo P8600, 2.4GHz, 4GB DDR2 RAM, Windows 7 Professional OS 32bit, JVM 1.6.0_22-b04, SOFA 2 revision 1194.

V. Lessons Learned

This section presents our experience with implementation of component runtime extensions for purpose of scripting language integration using user-defined controllers, interface interceptors and JSA.

Interceptor ordering. The presented approach relies on the fact, that the polyglot interceptor is the closest one to the component content. If it was not, the following interceptors would not be notified about the intercepted interface call since the call is redirected in the polyglot interceptor. In SOFA 2 and Fractal containers (e.g., Julia, AOKell), the ordering of interceptors is determined by the sequence of their instantiation, JBoss allows explicit ordering definition. However, there is no mechanism to enforce or check a particular ordering based on definition of a particular interceptor. Instead, the correctness of the interceptor ordering relies on the application developer. To mitigate this problem, the current mechanism can be improved by introducing a set of attributes for each interceptor, which will represent the behavior and characteristics of the interceptor. Each interceptor would define conditions upon these attributes which have to be met by other interceptors in the same chain of delegation. In natural language, these conditions could look like: *"This interceptor has to be executed before all other interceptors which may change the arguments of the call"*. This way every interceptor ordering can be expressed declaratively and therefore a constraint solving approach can be used to find the target order. This approach has been already used in a similar manner in SOFA 2 connector generator to find the proper connector implementation [4], [6].

Runtime extension dependencies. It is often necessary for a runtime extension to use features of other already-existing extensions. This requires to deal with various extension dependencies. For example we have developed an extension for scripting support in our Java-based component container. Then we have developed an extension which allows managing changes of the scripted component code at runtime, which depends on the previous one. However, dependency resolving among component runtime extensions in not addressed in general. Instead, it is again up to the application developer.

We have used the JSA to execute the script code of components. Therefore, we had to solve many problems connected to JSA and we have identified a number of disadvantages and pitfalls. In the following paragraphs we summarize our experience with JSA.

Classloading support. In the scope of the scripting support implementation, the most important limitation was the absence of classloading support. Although the individual engines use the standard classloading mechanism to locate Java classes, the JSA does not provide any global mechanism to set a classloader to an engine. Since custom classloading is heavily used in the SOFA 2 case-study for purposes of versioning and dependency resolving, another layer of abstraction API has been introduced to provide the missing classloading support.

Script module importing. Another disadvantage of the JSA is the absence of script module importing support (in means of module retrieval, etc.). This is a highly language-dependent task. Therefore, we have introduced a generic script importing mechanism which allows evaluating a script file in the current execution context and which is accessible from the script code. Moreover, to support the language-specific module importing (e.g., Python), a similar mechanism was introduced for each supported language (For example in case of the Python `import` directive in the Jython engine, the JAR file containing the target module has to be added to the system path of the engine before the module can be actually imported. This can be done by the above-mentioned import mechanism).

Exception handling. A common use-case when implementing component applications is declaration of exceptions on interface methods. However, all exceptions thrown in the script code are translated to a `ScriptException`. The information contained in the `ScriptException` is highly scripting-engine-dependent. The original exception is either not accessible at all or it is wrapped with an engine-dependent exception implementation. Therefore, it is not possible to use declared exceptions in the scripted component implementation. This disturbs the language-independence of the polyglot support using JSA.

VI. Related Work

The state-of-the-art programming language integration styles reflecting the concerns of component-based systems can be divided according to the layer of the system which they are using: (a) an operating system layer, (b) shared runtime platform layer, and (c) an adaptation layer on a shared runtime.

The system layer approach, the case (a), requires to implement a technological bridge between two subsystems implemented in two different languages. The CORBA Component Model [20] ensures the interoperability by using the CORBA middleware in the component container implementation. In the Service Component Architecture specification (SCA) [1], the integration is ensured by a predefined set of communication protocols which have to be implemented by each subsystem. The FraSCAti [25] SCA implementation supports implementation of components in Java, Java supported scripting languages (Groovy, JavaScript, etc.) and Scala. The common pitfall of these system-layer approaches is that they require designated deployment platform implementation for each supported language thus yielding a heterogeneous distributed system.

The shared runtime platform approach (JVM, CLR etc.), i.e., the case (b), allows all programming languages supported by the platform to be transparently used for component implementation (e.g., C# and Visual Basic for .Net CLR or Java and Scala for JVM). There are also platform extensions which allow easy integration of scripting languages (DLR [17] for .Net CLR and Da Vinci Machine Project [7] for JVM). The COM [16] component model can be also considered a representative of this approach. In this case, the binary-defined data types and interfaces form the shared platform. Integration of scripting languages (Microsoft VBScript or Microsoft JScript) into COM is ensured by Windows Script Components [18]. The disadvantage of this approach is the limited number of supported languages on one platform where for each new language a platform compiler has to be provided.

The third approach (c) utilizes an adaptation layer to integrate the new language into the shared runtime. The adaptation layer

is typically provided in the form of a library encapsulating an interpreter of the new language implemented on the base platform (Jython [14] in Java or IronPython [13] in .Net). Since this approach typically comprises scripting language integration, it is also the most suitable approach for rapid prototyping. In [8], this approach was used to allow component implementation, as well as interface and bindings management using scripting languages in scope of Fractal [10] component model and OSGi Services Platform [23]. Both cases rely on the Java Scripting API. The new Eclipse 4 [2] (E4) provides a support for Eclipse components (OSGi bundles) written in JavaScript. The E4 improves the underlying modularity framework based on OSGi [23] by integration of JavaScript bundles in standard Java-based OSGi runtime (including support for namespaces, dependencies, etc.). The XPCOM [19] is a cross platform interface-based component object model supporting implementation of components in C++, JavaScript, Python, Perl and Ruby. The components communicate between different environments using automatically generated wrapper objects. The typical problem of the mentioned approaches is that their language support is hard-coded into the platform implementation.

The presented approach falls to category (c) (adaptation-layer-based systems). It uses the component container as the integration platform. The main difference compared to the related work is that the presented polyglot support is content-language-independent. It forms a highly extensible generic framework for component runtime extensions which only themselves support a particular programming language. Moreover, the presented approach is transparent for the component container (whereas other approaches are usually hard-coded into the runtime implementation).

VII. Conclusion

We have presented a method allowing polyglot programming for rapid prototyping in interceptor-based component systems. Our approach defines an architecture of interceptors and their responsibilities, thus allowing for the use of component business code in a programming language different to the language of the component container. Although the presented approach is limited to languages which are accessible from the container language through an adaptation layer (such as scripting languages in Java), it is clean and non-intrusive due to the component interceptor mechanism. Moreover, the extension is transparent to the component developer and introduces only lesser differences in component implementation using the individual programming languages and thus makes the polyglot paradigm easier to use.

The presented approach is applicable not only for programming language integration but generally for integration of distinct technologies into the base component runtime (for example integration of different component system implementations). Therefore, as the future work we plan to exploit the presented approach in implementation of support for new component technologies into the selected component runtime, for example support for implementation of components using OSGi bundles in SOFA or Fractal.

References

[1] OSOA (2007). Service component architecture - assembly model specification. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[2] J. Arthorne. White Paper: e4 Technical Overview, July 2009. http://www.eclipse.org/e4/resources/e4-whitepaper.php.

[3] ASM, Java bytecode manipulation library, v. 3.1. http://asm.ow2.org.

[4] Tomas Bures. Generating connectors for homogeneous and heterogeneous deployment. In *Ph.D. Thesis*, 2006.

[5] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proc. of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48, USA, 2006. IEEE Computer Society.

[6] Tomas Bures and Frantisek Plasil. Communication style driven connector configurations. In C. V. Ramamoorthy, Roger Lee, and Kyung Whan Lee, editors, *Software Engineering Research and Applications*, volume 3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin / Heidelberg, 2004.

[7] The Da Vinci Machine Project. http://openjdk.java.net/projects/mlvm/.

[8] Didier Donsez, Kiev Gama, and Walter Rudametkin. Developing adaptable components using dynamic languages. In *Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '09, pages 396–403, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Marc Fleury and Francisco Reverbel. The jboss extensible server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 344–373, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[10] Fractal component model. http://fractal.objectweb.org.

[11] Danny Goodman, Michael Morrison, and Brendan Eich. *Javascript® bible, sixth edition*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[12] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. Cocome - the common component modeling example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer Berlin / Heidelberg, 2008. http://dx.doi.org/10.1007/978-3-540-85289-6_3.

[13] Iron Python. http://ironpython.net/.

[14] The Jython Project. http://www.jython.org/.

[15] Thomas Knneth. Making Scripting Languages JSR-223-Aware, September 2006. http://sofa.ow2.org/.

[16] Microsoft. Component Object Model Technologies. http://www.microsoft.com/com/.

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

[18] Microsoft. Windows Script Components. http://msdn.microsoft.com/en-us/library/asxw6z3c(v=VS.85).aspx.

[19] Mozilla. XPCOM. https://developer.mozilla.org/en/XPCOM.

[20] OMG. CORBA Component Model. http://www.omg.org/technology/documents/formal/components.htm.

[21] Oracle. Enterprise Java Beans (EJB). http://www.oracle.com/technetwork/java/index-jsp-140203.html.

[22] Oracle. Java Scripting API. http://java.sun.com/javase/6/docs/technotes/guides/scripting/.

[23] OSGi Service Platform. http://www.osgi.org.

[24] Marc-Thomas Schmidt, Beth Hutchison, Peter Lambros, and Robert W. Phippen. The enterprise service bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–798, 2005.

[25] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. *Services Computing, IEEE International Conference on*, 0:268–275, 2009.

[26] Dean Wampler. Polyglot Programming. http://www.polyglotprogramming.com/.

[27] Dean Wampler and Tony Clark. Guest editors' introduction: Multiparadigm programming. *Software, IEEE*, 27(5):20 –24, 2010.