An Open Component Model and Its Support in Java

Eric Bruneton*, Thierry Coupaye*, Matthieu Leclerc**, Vivien Quema**,
Jean-Bernard Stefani**

*France Telecom R&D and **INRIA Rhône-Alpes

Abstract. This paper presents FRACTAL, a hierarchical and reflective component model with sharing. Components in this model can be endowed with arbitrary reflective capabilities, from black-boxes to components that allow a fine-grained manipulation of their internal structure. The paper describes a Java implementation of the model, by means of a small but efficient run-time framework, which relies on a combination of mixins and aspect weaving for the programming of reflective features of components. The paper presents a qualitative and quantitative evaluation of this implementation, showing that component-based programming in FRACTAL can be made very efficient.

1 Introduction

By enforcing a strict separation between interface and implementation and by making software architecture explicit, component-based programming can facilitate the implementation and maintenance of complex software systems [25]. Coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance, as exemplified e.g. by the container concept in Enterprise Java Beans (EJB), CORBA Component Model (CCM), or Microsoft .Net [25].

Existing component-based frameworks and architecture description languages, however, provide only limited support for extension and adaptation, as witnessed by recent works on component aspectualization,e.g. [12, 20, 22]. This limitation has several important drawbacks: it prevents the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects; it prevents application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption; and it can make difficult the use of these frameworks and languages in different environments, especially constrained ones such as embedded systems.

We present in this paper a component model, called FRACTAL, that alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in FRACTAL are reflective, and their reflective capabilities are not fixed in the model but can be extended and adapted to fit the programmer's constraints and objectives. Importantly, we also present in this paper how such an *open* component model can be efficiently supported by an extensible run-time framework.

The main contributions of the paper are as follows:

- We define a hierarchical component model with sharing, that supports an extensible set of component control capabilities.

- We show how this model can be effectively supported by means of an extensible software framework, that provides for both static and dynamic configurability.
- We show that our component model and run-time framework can be effectively leveraged to build highly configurable, yet efficient, distributed systems.

The paper is organized as follows. Section 2 presents the main features of the FRACTAL model. Section 3 describes JULIA, a Java framework that supports the FRACTAL model. Section 4 evaluates the model and its supporting framework. Section 5 discusses related work. Section 6 concludes the paper with some indications for future work.

2 The FRACTAL component model

The FRACTAL component model (see [9] for a detailed specification), is a general component model which is intended to implement, deploy and manage (i.e. monitor and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), shared components (to model resources and resource sharing while maintaining component encapsulation), introspection capabilities (to monitor a running system), and re-configuration capabilities (to deploy and dynamically configure a system). In order to allow programmers to tune the reflective features of components to the requirements of their applications, FRACTAL is defined as an extensible system of relations between selected concepts, where components can be endowed with different forms of *control* (reflective features).

A FRACTAL component is a run-time entity that is encapsulated, and that has a distinct identity. At the lowest level of control, a FRACTAL component is a black box, that does not provide any introspection or intercession capability. Such components, called *base components* are comparable to plain objects in an object-oriented programming language such as Java. Their explicit inclusion in the model facilitates the integration of legacy software.

An interface is an access point to a component (similar to a "port" in other component models), that supports a finite set of operations. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming operation invocations, and client interfaces, which correspond to access points supporting outgoing operation invocations. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). Server interfaces have a given multiplicity, which specifies an upper bound $(1, n, \infty)$ on the number of client interfaces they can be bound to. At the level of control immediately above the base level, a FRACTAL component provides a Component interface, similar to the IUnknown in the COM model, that allows one to discover all its external (client and server) interfaces. Each interface has a name that distinguishes it from other interfaces of the component.

At upper levels of control, a FRACTAL component exposes (part of) its internal structure. A FRACTAL component comprises two parts: a *membrane*, which provides external interfaces to introspect and reconfigure its internal features (called *control interfaces*), and a *content*, which consists in a finite set of other components (called

sub-components). The membrane of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are only accessible from the component's sub-components. The membrane of a component is typically composed of several controller and interceptor objects. A controller object implements control interfaces. Typically, a controller object can provide an explicit and causally connected representation of the component's sub-components and superpose a control behavior to the behavior of the component's sub-components, including suspending, checkpointing and resuming activities of these sub-components. Interceptor objects are used to export the external interface of a subcomponent as an external interface of the parent component. They intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of such invocations (e.g. pre and post-handlers). Each component membrane can thus be seen as implementing a particular semantics of composition for the component's sub-components. Controller and interceptors can be understood as meta-objects or meta-groups (as they appear in reflective languages and systems).

The FRACTAL model allows for arbitrary (including user defined) classes of controller and interceptor objects. This is the main reason behind the denomination "open component model". The FRACTAL specification, however, contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. The following are examples of controllers.

Attribute controller: An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.

Binding controller: A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.

Content controller: A component can provide the ContentController interface to list, add and remove subcomponents in its contents.

Life-cycle controller: A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in suppport for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

Communication between FRACTAL components is only possible if their interfaces are bound. FRACTAL supports both primitive bindings and composite bindings. A *primitive binding* is a binding between one client interface and one server interface in the same address space, which means that operation invocations emitted by the client interface should be accepted by the specified server interface. A primitive binding is called that way for it can be readily implemented by pointers or direct language references (e.g. Java references). A *composite binding* is a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc). A binding is a normal FRACTAL component whose role is to mediate communication between other components. The binding concept corresponds to the connector concept that is defined in other component models. Note that, except for primitive bindings, there is no predefined set of bindings in FRACTAL. In fact bindings can be built explicitly by composition, just as other components. The FRACTAL model thus provides two

mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of a group of components in a composite.

An original feature of the FRACTAL component model is that a given component can be included in several other components. Such a component is said to be *shared* between these components. Shared components are useful, paradoxically, to preserve component encapsulation: there is no need to expose interfaces in higher-level components to allow access to a shared component by a lower-level one. Shared components are useful in particular to faithfully model access to low-level system resources.

The FRACTAL model is endowed with an optional type system (some components such as base components need not adhere to the type system). Interface types are pairs (signature,role) . Component types reflect the different interface types that a component can bear. For lack of space, we do not detail the FRACTAL type system here.

3 The JULIA framework

The JULIA framework supports the construction of software systems with FRACTAL components written in Java. The main design goal for julia was to implement a framework to program FRACTAL component membranes. In particular, we wanted to provide an extensible set of control objects, from which the user can freely choose and assemble the controller and interceptor objects he or she wants, in order to build the membrane of a Fractal component. The second design goal was to provide a continuum from static configuration to dynamic reconfiguration, so that the user can make the speed/memory tradeoffs he or she wants. The last design goal was to implement a framework that can be used on any JVM and/or JDK, including very constrained ones such as the KVM, and the J2ME profile (where there is no ClassLoader class, no reflection API, no collection API, etc). In addition to the previous design goals, we also made two hypotheses in order to simplify the implementation: we suppose there is only one (re)configuration thread at a given time, and we also suppose that the component data structures do not need to be protected against malicious components.

3.1 Main Data Structures

Overview A Fractal component is generally represented by many Java objects, which can be separated into three groups (see Fig. 1):

- the objects that implement the component interfaces, in white in Fig. 1 (one object per component interface; each object has an impl link to an object that really implements the Java interface, and to which all method calls are delegated; this reference is null for client interfaces),
- the objects that implement the membrane of the component, in gray and light gray in the figure (a controller object can implement zero of more control interfaces),
- and the objects that implement the content part of the component (not shown in the figure).

The fact that each component interface is represented by its own Java object comes from the fact that component interfaces are typed (i.e., a component interface object

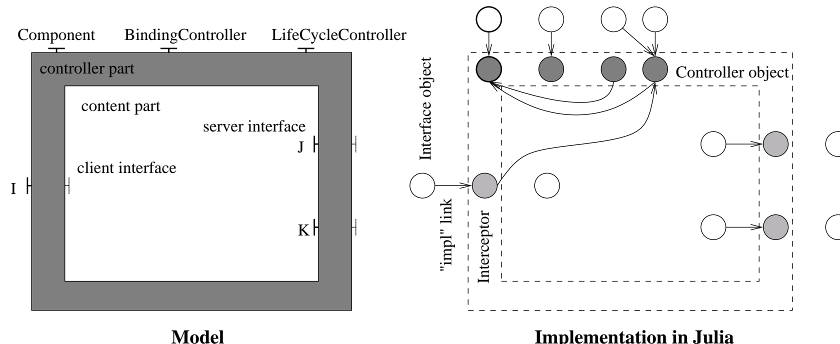

Fig. 1. an abstract component and a possible implementation in Julia

implements both Interface and the Java interface corresponding to this interface). It is not possible to do better, unless perhaps by using very complex bytecode manipulations that modify the signature of all the methods of all classes.

The objects that represent the membrane of a component can be separated into two groups: the objects that implement the control interfaces (in gray in Fig. 1), and (optional) *interceptor* objects (in light gray) that intercept incoming and/or outgoing method calls on non-control interfaces. These objects implement respectively the Controller and the Interceptor interfaces. Each controller and interceptor object can contain references to other controller / interceptor objects (since the control aspects are generally not independent - or "orthogonal" - they must generally communicate between each other).

Instantiation JULIA components can be created manually or automatically. The manual method can be used to create any kind of components, while the automatic one is restricted to components whose type follows the basic type system defined in the Fractal specification, which provide a Component interface, and which provide interface introspection functions. In both methods, a component must be created as follows:

- creation of the component interface objects (if the component must provide interface introspection), of the controller objects, of the interceptor objects, and of the component's content (for container components).
- initialization of the impl references between the component interfaces objects and the content, controller and interceptor objects.
- creation of an InitializationContext, and set up of this context, with references to the previous objects.
- initialization of the controller and interceptor objects by calling their initFcController method, with the previous InitializationContext object as parameter (this step allows the controller and interceptor objects to initialize themselves, i.e. to set up the references between all these objects).

In the automatic method, i.e. when components are created through the GenericFactory interface, the operations that must be done at the previous steps are deduced from the component's type, and from its controller and content descriptor. Once these

descriptors have been analyzed and checked, and once the previous operations have been determined, a sub class of the InitilizationContext class that implements these operations is generated, directly in bytecode form, with the InitializationContextClass-Generator. Finally the component is created by using this generated class (in other words, the controller and content descriptors are compiled on the fly, once and for all, instead of being interpreted and checked each time a component must be created).

3.2 Mixin classes

Motivations The main design goal of Julia is to implement a *framework* to program component membranes. To do so, Julia provides a collection of pre-defined controller and interceptor classes and a class mixin mechanism. Mixin classes are used to build controller classes that combine several aspects.

In order to provide these framework, one could have thought of using class inheritance. But this solution is not feasible, because it leads to a combinatorial explosion, and to a lot of code duplication.

Another solution could have been to use an Aspect Oriented Programming (AOP) tool or language, such as AspectJ [14]. But using AspectJ would have introduced a new problem, due to the fact that, in AspectJ, aspects must be applied at compile time[1], and that this process requires the source code of the "base" classes. It would then be impossible to distribute Julia in compiled form, in a jar file, because then the users would not be able to apply new aspects to the existing Julia classes (in order to add new control aspects that crosscut existing ones).

What is needed to really solve our modularity and extensiblity problem is therefore a kind of AOP tool or language that can be used at load time or at runtime, without needing the source code of the base classes, such as JAC [19]. For performance reasons the current Julia version does not use JAC or other similar systems: it uses instead some kind of *mixin* classes. A mixin class is a class whose super class is specified in an abstract way, by specifying the minimum set of fields and methods it should have. A mixin class can therefore be *applied* (i.e. override and add methods) to any super class that defines at least these fields and methods. This property solves the above combinatory problem. The AspectJ problem is solved by the fact that the mixin classes used in Julia can be mixed at runtime, thanks to our bytecode generator [1] (unlike in most mixin based inheritance languages, where mixed classes are declared at compile time).

Implementation Instead of using a Java extension to program the mixin classes, which would require an extended Java compiler or a pre processor, mixin classes in Julia are programmed by using patterns. For example the JAM [6] mixin class shown below (on the left) is written in pure Java as follows (on the right):

```
mixin A {                              abstract class A {
  inherited public void m ();            abstract void _super_m ();
  public int count;                      public int count;
  public void m () {                     public void m () {
    ++count;                               ++count;
```

[1] This is no longer true with version 1.1 of AspectJ, however this was the case in 2002 when Julia was developed

```
    super.m();                          _super_m();
  }                                   }
}                                   }
```

In other words, the _super_ prefix is used to denote the inherited members in JAM, i.e. the members that are required in a base class, for the mixin class to be applicable to it. More precisely, the _super_ prefix is used to denote methods that are overriden by the mixin class. Members that are required but not overriden are denoted with _this_ (a mixin class cannot contain a _super_m method if it does not have a corresponding m method. Likewise, a mixin class cannot have both a _this_m method and a corresponding m method).

Mixin classes can be mixed, resulting in normal classes. More precisely, the result of mixing several mixin classes M_1, ... M_n, *in this order*, is a normal class that is equivalent to a class M_n extending the M_{n-1} class, itself extending the M_{n-2} class, ... itself extending the M_1 class (constructors are ignored; an empty public constructor is generated for the mixed classes). Several mixin classes can be mixed only if each method and field required by a mixin class M_i is provided by a mixin class M_j, with $j < i$ (each required method and field may be provided by a different mixin).

3.3 Interceptor classes

This section gives some details about the generator used to generate interceptor classes. This bytecode generator takes as parameters the name of a super class, the name(s) of one or more application specific interface(s), and one or more aspect code generator(s). It generates a sub class of the given super class that implements all the given application specific interfaces and that, for each application specific method, implements all the aspects corresponding to the given aspect code generators.

Each aspect code generator can modify the bytecode of each application specific method arbitralily. For example, two aspect code generators A and B can modify the method void m () { impl.m() } into:

```
void m () { // pre code A      void m () { // pre code B
  try {
    impl.m();                  impl.m();
  } finally {
    // post code A             // post code B
  }
}                              }
```

When an interceptor class is generated by using several aspect bytecode generators, the transformations performed by these generators are automatically composed together. For example, if A and B are used to generate an interceptor class, the result for the previous m method is the following (there are two possibilities, depending on the order in which A and B are used):

```
void m () { // pre code A     void m () { // pre code B
  try {                      // pre code A
    // pre code B            try {
    impl.m();                  impl.m();
    // post code B           } finally {
  } finally {                  // post code A }
```

```
    // post code A }            // post code B
}                           }
```

Note that, thanks to this (elementary) automatic weaving, which is very similar to what can be found in AspectJ or in Composition Filters [8], several aspects can be managed by a single interceptor object: there is no need to have chains of interceptor objects, each object corresponding to an aspect.

Like the controller objects, the aspects managed by the interceptor objects of a given component can all be specified by the user when the component is created. The user can therefore not only choose the control interfaces he or she wants, but also the interceptor objects he or she wants. JULIA only provides two aspect code generators: one to manage the lifecycle of components, the other to trace incoming and/or outgoing method calls. JULIA also provides two abstract code generators, named SimpleCodeGenerator and MetaCodeGenerator, that can be easily specialized in order to implement custom code generators.

Note: each aspect bytecode generator is given a Method object corresponding to the method for which it must generate interception code. Thanks to this argument, a generator can generate bytecode that adapts to the specific signature of each method. It can also generate bytecode to reify the method's parameters if desired (although this is less efficient than the first method).

3.4 Support for Constrained Environments

One of the goals of JULIA is to be usable even with very constrained JVMs and JDKs, such as the KVM and the J2ME libraries (CLDC profile). This goal is achieved thanks to the following properties:

- the size of JULIA runtime (35KB, plus 10KB for the Fractal API), which is the only part of JULIA that is needed at runtime, is compatible with the capabilities of most constrained environments;
- JULIA can be used in environments that do not provide the Java Reflection API or the ClassLoader class, which are needed to dynamically generate the JULIA application specific classes, since these classes can also be generated statically, in a less constrained environment;
- the JULIA classes that are needed at runtime, or whose code can be copied into application specific runtime classes, use only the J2ME, CLDC profile APIs, with only two exceptions for collections and serialization. For collections a subset of the JDK 1.2 collection API is used. This API is not available in the CLDC profile, but a bytecode modification tool is provided with JULIA to convert classes that use this subset into classes that use the CLDC APIs instead. This tool also removes all serialization related code in JULIA. In other words the JULIA jars cannot be used directly with CLDC, but can be transformed automatically in new jars that are compatible with this API.

3.5 Optimizations

Intra component optimizations In order to save memory, JULIA provides optimization options to merge some or most of the objects that constitute a component into a

single object. This merging is done thanks to a bytecode class generator that can merge several controller classes into a single class provided that these classes respect the following rules:

- Each controller object can provide and require zero or more Java interfaces. The provided interfaces must be implemented by the object, and there must be one field per required interface, whose name must begin with weaveable for a mandatory interface, or weaveableOpt for an optional interface (see below). Each controller class that requires at least one interface must also implement the Controller interface (see below).
- In a given component, a given interface cannot be provided by more than one object (except for the Controller interface). Otherwise it would be impossible to merge these objects (an object cannot implement a given interface in several ways).

The merging process is the following. Basically, all the methods and fields of each class are copied into a new class (the resulting class does not depend on the order into which the classes are copied). However the fields whose name begins with weaveable are replaced by this, and those whose name begins with weaveableOpt are replaced either by this, if a class that implements the corresponding type is present in the list of the classes to be merged, or null otherwise.

Inter component optimizations In addition to the previous intra component optimizations, which are mainly used to save memory, JULIA also provides an inter component optimization, namely an algorithm to create and update *shortcut* bindings between components, and whose role is to improve time performances. As explained above, each interface of a component contains an impl reference to an object that really implements the component interface. In the case of a server interface s, this field generally references an interceptor object, which itself references the server interface to which the complementary interface of s is bound (see Fig. 2).

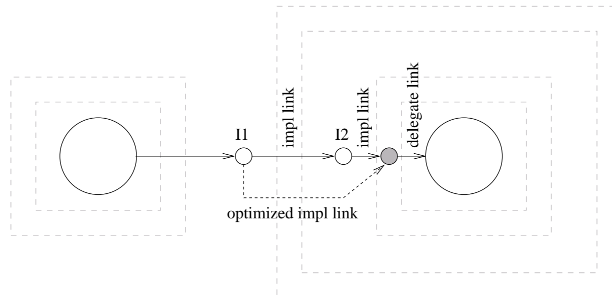

Fig. 2. shortcut bindings

More precisely, this is the case with the CompositeBindingMixin. With the OptimizedCompositeBindingMixin, the impl references are optimized when possible. For example, in Fig. 2, since I1 does not have an associated interceptor object, and since

component interface objects such as I2 just forward any incoming method calls to the object referenced by their impl field, I1 can, and effectively references directly the interceptor associated to I2.

4 Evaluation

We provide in this section an evaluation of our model and its implementation. We first provide a qualitative assessment of our component framework. We then provide a more quantitative evaluation with micro-benchmarks and with an application benchmark based on a reengineered message-oriented middleware.

4.1 Qualitative assessment

Modularity JULIA provides several mixins for the binding controller interface, two implementations of the life cycle controller interface, and one implementation of the content controller interface. It also provides support to control component attributes, and to associate names to components. All these aspect implementations, which make different flexibility/performance tradeoffs, are well separated from each other thanks to mixins, and can therefore be combined freely. Together with the optimization mechanisms used in JULIA, this flexibility provides what we call a *continuum* from static to dynamic configurations, i.e., from unreconfigurable but very efficient configurations, to fully dynamically reconfigurable but less efficient configurations (it is even possible to use different flexibility/performance tradeoffs for different parts of a single application).

Extensibility Several users of JULIA have extended it to implement new control aspects, such as transactions [23], auto-adaptability [11], or checking of the component's behavior, compared to a formal behavior, expressed for example with assertions (pre/post conditions and invariants), or with more elaborate formalisms, such as temporal logic [24]. As discussed below, we have also built with JULIA a component library, called DREAM, for building message-oriented middleware (MOM) and reengineered and existing MOM using this library. DREAM components exhibit specific control aspects, dealing with on-line deployment and re-configuration. In all these experiences, the different mechanisms in JULIA have proved sufficient to build the required control aspects.

Limitations There are however some limitations to JULIA's modularity and extensibility. For example, when we implemented JULIA, it was sometimes necessary to refactor an existing method into two or more methods, so that one of this new methods could be overriden by a new mixin, without overriding the others. In other words, the mixin mechanism is not sufficient by itself: the classes must also provide the appropriate "hooks" to apply the mixins. And it is not easy, if not impossible, to guess the hooks that will be necessary for future aspects (but this problem is not specific to mixins, it also occurs in AspectJ, for example).

options	memory overhead (bytes)	time overhead (μs)
lifecycle, no optimization	592	0.110
lifecycle, merge controllers	528	0.110
lifecyle, merge all	504	0.092
no lifecycle, no optimization	496	0.011
no lifecycle, merge controllers	440	0.011
no lifecycle, merge all	432	0.011

Table 1. Julia performances

4.2 Quantitative evaluation I: Micro-benchmarks

In order to measure the memory and time overhead of components in Julia, compared to objects, we measured the memory size of an object, and the duration of an empty method call on this object, and we compared these results to the memory size of a component [2] (with a binding controller and a life cycle controller) encapsulating this object, and to the duration of an empty method call on this component. The results are given in Table 1, for different optimization options. The measurements where made on a Pentium III 1GHz, with the JDK1.3, HotSpotVM, on top of Linux. In these conditions the size of an empty object is 8 bytes, and an empty method call on an interface lasts 0.014 μs.

As can be seen the class merging options can reduce the memory overhead of components (merging several objects into a single one saves many object headers, as well as fields that were used for references between these objects). The time overhead without interceptor is of the order of one empty method call: it corresponds to the indirection through an interface reference object. With a life cycle interceptor, this overhead is much greater: it is mainly due to the execution time of two `synchronized` blocks, which are used to increment and decrement a counter before and after the method's execution. This overhead is reduced in the "merge all" case, because an indirection is saved in this case. In any cases, this overhead is much smaller than the overhead that is measured when using a generic interceptor that completely reifies all method calls (4.6 μs for an empty method, and 9 μs for an `int inc (int i)` method), which shows the advantages of using an open and extensible interceptor code generator.

The time needed to instantiate a component encapsulating an empty object is of the order of 0.3 ms, without counting the dynamic class generation time, while the time to needed instantiate an empty object is of the order of 0.3 μs (instantiating a component requires to instantiate several objects, and many checks are performed before instantiating a component).

4.3 Quantitative evaluation II: Re-engineering JORAM

Joram [3] is an open-source JMS-compliant middleware (Java Messaging Service [2]). Joram comprises two parts: the ScalAgent message-oriented middleware (MOM) [7], and a software layer on top of it to support the JMS API.

[2] the size of the objects that represent the component's type, which is shared between all components of the same type, is not taken into account here. This size is of the order of 1500 bytes for a component with 6 interfaces.

MOM	Number of rounds 0 KB	1 KB	Memory footprint (KB)
ScalAgent	325	255	4 × 1447
Dream (non-reconf.)	329	260	4 × 1580
Dream (reconf.)	318	250	4 × 1587

Table 2. Performance of DREAM implementations vs ScalAgent implementation

The Scalagent MOM is a fault-tolerant platform, written in Java, that combines asynchronous message communication with a distributed programming model based on autonomous software entities called *agents*. Agents behave according to an "event → reaction" model. They are persistent and each reaction is instantiated as a transaction, allowing recovery in case of node failure. The ScalAgent MOM comprises a set of agent servers. Each agent server is made up of three entities. The *Engine* is responsible for the creation and execution of agents; it ensures their persistency and atomic reaction. The *Conduit* routes messages from the engine to the networks. The *Networks* ensure reliable message delivery and a causal ordering of messages between servers.

Using JULIA, we have built a component-based library, called DREAM, which is dedicated to the construction of asynchronous middleware. For lack of space we do not present the DREAM library here. Let us just note that DREAM components are relatively fine-grained, comprising e.g. components such as messages, message queues, message aggregators and de-aggregators, message routers, and distributed channels. Using DREAM, we have reengineered the ScalAgent MOM. Its main structures (networks, engine and conduit) have been preserved to facilitate the functional comparison between the ScalAgent MOM and its DREAM re-implementation: networks, engine and conduits are implemented as composites that control the execution of DREAM subcomponents. The software layer that implements the JMS API in Joram runs unmodified on the DREAM implementation.

Performance comparisons Measurements have been performed to compare the efficiency of the same benchmark application running on the ScalAgent MOM and on its DREAM implementation. The application involves four agent servers; each one hosts one agent. Agents in the application are organized in a virtual ring. One agent is an initiator of rounds. Each round consists in forwarding the message originated by the initiator around the ring. Each agent behaves very simply as follows: each message received by an agent is forwarded to the next agent on the ring until the message has gone full circle. We did two series of tests: messages without payload and messages embedding a 1kB payload. Experiments have been done on four PC Bi-Xeon 1,8 GHz with 1Go, connected by a Gigabit Ethernet adapter, running Linux kernel 2.4.20.

Table 2 shows the average number of rounds per second, and the memory footprint. We have compared two implementations based on DREAM with the ScalAgent implementation. Note that the DREAM implementations do not make use of the Julia memory optimizations. The first implementation using DREAM is not dynamically reconfigurable. As we can see, the number of rounds is slightly better ($\approx$ 1,2 to 2%) than in the ScalAgent implementation: this marginal improvement comes from a better de-

MOM	Number of rounds 0 KB	1 KB	Memory footprint (KB)
ScalAgent	182	150	4 × 1447
Dream (4 agent servers)	188	153	4 × 1580
Dream (2 agent servers)	222	181	2 × 1687
Dream (1 agent server)	6597	6445	1 × 1900

Table 3. Impact of the number of engines by agent server

sign of some low-level components. Concerning the memory footprint, the DREAM implementation requires 9% more memory, which can be explained by some of the structure needed by Julia and the fact that each component has several controller objects. This memory overhead is not significant on standard PCs. The second implementation is dynamically reconfigurable (in particular, each composite component supports a life-cycle controller and a content controller). This implementation is only slightly slower than the ScalAgent one ($\approx$ 2,2 to 2%) and only requires 7KB more than the non-reconfigurable implementation made using DREAM. Note that DREAM performances are proportionally better with the 1kB messages than with the empty ones. This is easily explained by the fact that less messages are handled (more time is spent in message transmissions), thus limiting the impact of interceptors.

We have also evaluated the gain brought by changing the configuration in a multi-engine agent server. Such a server can be very useful to parallelize the execution of agents and to provide different non-functional properties to different sets of agents. In contrast to the Scalagent MOM, such a change of configuration is trivial to make in a DREAM implementation (it can be done dynamically in a reconfigurable implementation). We have compared four different architectures: the ScalAgent one, an equivalent DREAM configuration with four mono-engine agent servers, a DREAM configuration with two 2-engine agent servers, and a DREAM configuration with one 4-engine agent server. Contrary to the previous experiment, agent servers are hosted by the same PC. Also, agents are placed so that two consecutive agents in the virtual ring are hosted by different agent servers. Table 3 shows that using two 2-engine agent servers improves the number of rounds by 18% and reduces the memory footprint by 47%. The increase of the number of rounds can be explained by the fact that matrix clocks used to enforce the causal ordering have a n^2 size, n being the number of agent servers. Thus, limiting the number of agent servers reduces the size of the matrix which is sent with messages. Table 3 also shows that using a 4-engine agent servers is 29 (35 for 1kB messages) times faster than using four mono-engine agent servers. This result may seem surprising, but can be easily explained by the fact that inter agent communication do not transit via the network components. Instead, the router component, that implements the conduit in the dream implementation, directly sends the message to the appropriate engine. Again, such a change is trivial to make using DREAM, but would require important changes in the Scalagent code.

5 Related work

Component models The FRACTAL model occupies an original position in the vast amount of work dealing with component-based programming and software architecture [25, 17, 15], because of its combination of features: hierarchical components with sharing , support for arbitrary binding semantics between components, components with selective reflection. Aside from the fact that sharing is rarely present in component models (an exception is [18]), most component models provide little support for reflection (apart from elementary introspection, as exemplified by the second level of control in the FRACTAL model discussed in Section 2). A component model that provides extensive reflection capabilities is OpenCOM [10]. Unlike FRACTAL, however, OpenCOM defines a fixed meta-object protocol for components (in FRACTAL terms, each OpenCOM component comes equipped with a fixed and predetermined set of controller objects).

Software architecture in Java Several component models for Java have been devised in the last five years. Two recent representatives include Jiazzi [16] and ArchJava [4]. Unlike these works, our approach to component-based programming in Java does not rely on language extensions: JULIA is a small run-time library, complemented with simple code and byte-code generators. This coupled, with the reflective character of the FRACTAL model, provides for a more dynamic and extensible basis for component-based programming than Jiazzi or Archjava. Note that FRACTAL and JULIA directly support arbitrary connector abstractions, through the notion of bindings. We have, for instance, implemented synchronous distributed bindings with an RMI-like semantics just by wrapping the communication subsystem of the Jonathan Java ORB [13], and asynchronous distributed bindings with message queuing and publish/subscribe semantics by similarly wrapping message channels from the DREAM library introduced in the previous section. ArchJava also supports arbitrary connector abstractions [5], but provides little support for component reflection as in FRACTAL and JULIA. Unlike JULIA, however, ArchJava supports sophisticated type checking that guarantees communication integrity (i.e. that components only communicate along declared connections between ports - in FRACTAL, that components only communicate along established bindings between interfaces).

Combining aspects and components The techniques used in JULIA to support the programming of controller and interceptor objects in a FRACTAL component membrane are related to several recent works on the aspectualization of components or component containers, such as e.g. [12, 20, 22]. The mixin and aspect code generators in JULIA provide a lightweight, flexible yet efficient means to aspectualize components. In line with its design goals, JULIA does not seek to provide extensive language support as AOP tools such as AspectJ or JAC provide. However such language support can certainly be build on top of JULIA. Prose [21] provides dynamic aspect weaving, whose performance appear to be comparable to that of JULIA. However, Prose relies on a modified JVM, which makes it inpractical for production use. In contrast, JULIA can make use of standard JVMs, including JVMs for constrained environments.

6 Conclusion

We have presented the FRACTAL component model and its Java implementation, JULIA. FRACTAL is *open* in the sense that FRACTAL components are endowed with an extensible set of reflective capabilities (controller and interceptor objects), ranging from no reflective feature at all (black boxes or plain objects) to user-defined controllers and interceptors, with arbitrary introspection and intercession capabilities. JULIA consists in a small run-time library, together with bytecode generators, that relies on mixins and dynamic aspect weaving to allow the creation and combination of controller and interceptor classes. We have evaluated the effectiveness of the model and its Java implementation, in particular through the re-engineering of an existing open source message-oriented middleware. The simple application benchmark we have used indicates that the performance of complex component-based systems built with JULIA compares favorably with standard Java implementations of functionally equivalent systems. In fact, as our performance evaluation shows, the gains in static and dynamic configurability can also provide significant gains in performance by adapting system configurations to the application context.

FRACTAL and JULIA have already been, and are being used for several developments, by the authors and others. We hope to benefit from these developments to further develop the FRACTAL component technology. Among the ongoing and future work we can mention: the development of a dynamic ADL, the exploitation of containment types and related type systems to enforce architectural integrity constraints such as communication integrity, the investigation of dynamic aspect weaving techniques to augment or complement the JULIA toolset, and the formal specification of the FRACTAL model with a view to assess its correctness and to connect it with formal verification tools.

Availability JULIA is freely available under an LGPL license at the following URL: `http://fractal.objectweb.org`.

References

1. ASM, A Java Bytecode Manipulation Framework, 2002. Objectweb, `http://www.objectweb.org/asm/`.
2. Java Message Service Specification Final Release 1.1, Mars 2002. Sun Microsystems, http://java.sun.com/products/jms/docs.html.
3. JORAM: Java Open Reliable Asynchronous Messaging, 2002. Objectweb, `http://www.objectweb.org/joram/`.
4. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conference on Object-Oriented Programming (ECOOP)*, 2002.
5. J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.
6. D. ancona, G. Lagorio, and E. Zucca. A Smooth Extension of Java with Mixins. In *ECOOP'00, LNCS 1850*, 2000.
7. L. Bellissard, N. de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Plateform for Reliable Asynchronous Distributed Programming. In *Symposium on Reliable Distributed Systems (SRDS'99)*, Lausanne, Switzerland, October 1999.

8. L. Bergmans and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM, 44(10)*, 2001.
9. E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model. Technical report, Specification v2, ObjectWeb Consortium, http://www.objectweb.org/fractal, 2003.
10. M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, pages 160–178, Heidelberg, Germany, November 2001.
11. P. David and T. Ledoux. Towards a Framework for Self-adaptive Component-Based Applications. In *DAIS 2003, LNCS 2893*, 2003.
12. F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *AOSD02*, 2002.
13. B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani. Jonathan: an open distributed platform in Java. *Distributed Systems Engineering Journal, vol.6*, 1999.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W Griswold. An Overview of AspectJ. In *ECOOP 2001, LNCS 2072*, 2001.
15. G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems.* Cambridge University Press, 2000.
16. S. McDirmid, . Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings OOPSLA '01, ACM Press*, 2001.
17. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering, vol. 26, no. 1*, 2000.
18. G. Outhred and J. Potter. A Model for Component Composition with Sharing. In *Proceedings 3rd ECOOP Int. Workshop on Component-Oriented Programming (WCOP '98)*, 1998.
19. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001, LNCS 2192*, 2001.
20. R. Pichler, K. Ostermann, and M. Mezini. On Aspectualizing Component Models. *Software Practice and Experience*, 2003.
21. A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for Java. In *AOSD03*, 2003.
22. A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *17th ECOOP*, 2003.
23. M. Prochazka. Jironde: A Flexible Framework for Making Components Transactional. In *DAIS 2003, LNCS 2893*, 2003.
24. N. Rivierre and T. Coupaye. Observing Component Behaviors with Temporal Logic. In *Proceedings of the 8th ECOOP International Workshop on Correctness of Model-Based Software Composition (CMC '03)*, 2003.
25. C. Szyperski. *Component Software, 2nd edition.* Addison-Wesley, 2002.