SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **00**:1–7 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

The FRACTAL Component Model and Its Support in Java

Eric Bruneton[1], Thierry Coupaye[1], Matthieu Leclercq[2], Vivien Quéma[2], Jean-Bernard Stefani[2]

[1] *France Telecom R&D*
[2] *INRIA Rhône-Alpes*
{Eric.Bruneton,Thierry.Coupaye}@rd.francetelecom.com,
{Matthieu.Leclercq,Vivien.Quema,Jean-Bernard.Stefani}@inrialpes.fr

SUMMARY

This paper presents FRACTAL, a hierarchical and reflective component model with sharing. Components in this model can be endowed with arbitrary reflective capabilities, from plain black-box objects to components that allow a fine-grained manipulation of their internal structure. The paper describes JULIA, a Java implementation of the model, a small but efficient run-time framework, which relies on a combination of interceptors and mixins 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.

KEY WORDS: component-based programming, reflective component model, aspects and components, Java components

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 [36]. Indeed, these two principles form the basis for two essential properties: adaptability and manageability. Their role as units of software deployment and configuration in particular, are well understood: they allow for pre-run time adaptation in order to suit arbitrary deployment environments (construction of dedicated software infrastructures), evolutions in requirements and technical evolutions (maintenance), and organisational evolutions (integration, interoperation). When seen as run-time structures, components can serve as the basis for software reconfiguration. By fully delineating subsystem boundaries, they provide a natural scope for reconfiguration actions and a natural target for system instrumentation and supervision. Coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexity inherent in the handling of non-functional aspects in a software system, such as

distribution and fault-tolerance, as exemplified by the container concept in Enterprise Java Beans (EJB), CORBA Component Model (CCM), or Microsoft .Net [36].

Existing component-based frameworks and architecture description languages (see e.g. [27] for a recent survey of ADLs), however, provide only limited support for extension and adaptation, as witnessed by recent works on component aspectualization, e.g. [22, 30, 32]. The paper [30] argues at length, for instance, about the lack of tailorability of EJB containers, meaning that there is no mechanism to configure an EJB container or its infrastructural services, nor is it possible to add new services to it.

This limitation implies 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, including embedded systems. Even with reflective component models such as OpenCOM [20], i.e. models of components endowed with an explicit meta-object protocol to control the execution of components and introduce support for different non-functional aspects, we find it necessary to be able to finely tailor the reflective capabilities endowed in components in order to support the different trade-offs which are critical in low-level software infrastructure design, e.g. in operating system or middleware construction.

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, in the sense that their execution and their internal structure can be made explicit and controlled through well-defined interfaces. These reflective capabilities, however, 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 in Java by an extensible run-time framework, called JULIA. JULIA incorporates an innovative use of mixin classes to allow the definition and combination of arbitrary component controllers. This provides a JULIA programmer with effective and extensible means to deal with different cross-cutting aspects in controlling components.

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 in Java 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 used effectively 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 [18] for a detailed specification), is a general component model which is intended to implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model:

- Composite components (components that contain sub-components), in order to have a uniform view of applications at various levels of abstraction.
- Shared components (sub-components of multiple enclosing composite components), in order to model resources and resource sharing while maintaining component encapsulation.
- Introspection capabilities, in order to monitor and control the execution of a running system.
- Re-configuration capabilities, in order to deploy and dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, FRACTAL is defined as an extensible system. Control features of components are not predetermined in the model, rather the model allows for a continuum of reflective features or *levels of control*, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including e.g. access and manipulation of component contents, control over components life-cycle and behavior, etc).

2.1. Components and bindings

A FRACTAL component is a run-time entity that is encapsulated, has a distinct identity, and that supports one or more interfaces. An *interface* is an access point to a component (similar to a "port" in other component models), that implements an *interface type* (i.e. a type specifying the operations supported by the access point). 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.

A FRACTAL component (see Figure 1) can be understood generally as being composed of a *membrane*, which supports interfaces to introspect and reconfigure its internal features, 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 controllers. Typically, a membrane 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. Controllers can also play the role of interceptors. Interceptors are used to export the external interface of a subcomponent as an external interface of the parent component. They can intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of

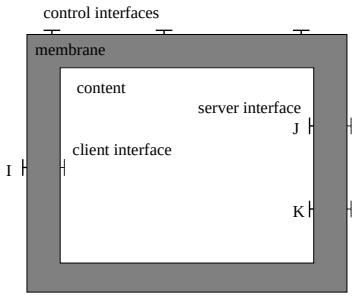

Figure 1. A Fractal component

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 can be understood as meta-objects or meta-groups as they appear in reflective languages and systems.

The FRACTAL model provides two mechanisms to define the architecture of an application: component nesting we have just described and bindings. 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 can be modeled as a component), 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. Importantly, bindings can embody remote communication paths between interfaces, and span different address spaces and different machines in a network. This allows the construction of distributed configuration of FRACTAL components.

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. Consider, for example, a menu and a toolbar components (see Figure 2), with an "undo" toolbar button corresponding to an "undo" menu item. It is natural to represent the menu items and toolbar buttons as sub components, encapsulated in the menu and toolbar components, respectively. But, without sharing, this solution does not work for

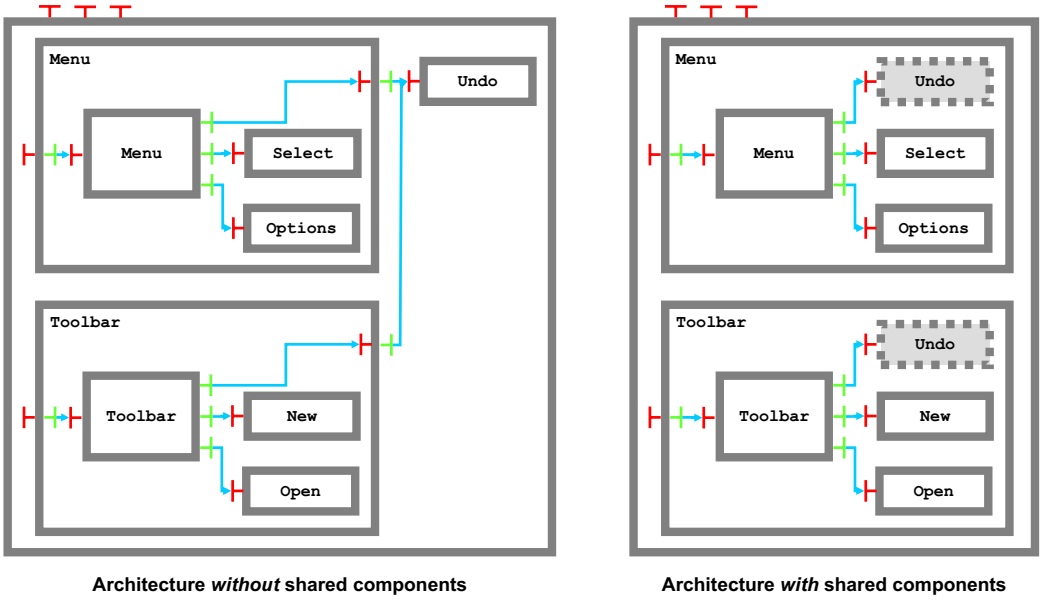

Figure 2. Component sharing in Fractal

the "undo" button and menu item, which must have the same state (enabled or disabled): these components, or an associated state component, must be put outside the menu and toolbar components. With component sharing, the state component can be shared between the menu and toolbar components, in order to preserve component encapsulation. Shared components are also useful to faithfully model access to low-level system resources (which are typically shared between applications), and to help separate "aspects" in component based applications (for instance it is possible to have components representing address spaces, i.e. a physical architecture, with sub components shared with other components representing a logical architecture)

2.2. Levels of control

The FRACTAL model does not enforce a fixed and pre-determined set of controllers in component membranes (hence the phrase "*open* component model"). It allows instead arbitrary forms of membranes, with different control and interception semantics. The FRACTAL specifiation, however, identifies specific forms of membranes and controllers, corresponding to different levels of control (or reflection capabilities) on components.

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 (although, even at the lowest level of control, the model allows components to have a varying number of interfaces during their lifetime). Their explicit inclusion in the model facilitates the integration of legacy software.

At the next level of control, 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 this level of control, components still do not provide any control function, but the Component interface provides elementary means for introspecting the external structure of a component. Also at this level of control, component interfaces may additionally support, via multiple interface inheritance, operations that allow to retrieve the Component interface of the supporting component. Such operations are gathered in the Interface interface type.

At upper levels of control, a FRACTAL component can expose elements of its internal structure, and provide increased introspection and intercession capabilities. The FRACTAL specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features:

- **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 support for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

2.3. Type system

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 describe the operations supported by an interface, the role of the interface (client or server), as well as its *contingency* and its *cardinality*. The contingency of an interface indicates if the functionality corresponding to this interface is guaranteed to be available or not, while the component is running:

- The operations of a *mandatory* interface are guaranteed to be available when the component is running. For a client interface, this means that the interface must be bound. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound.
- The operations of an *optional* interface are not guaranteed to be available. For a server interface, this can happen e.g. when the complementary internal interface of the

supporting component is not bound to a sub-component interface. For a client interface, this means that the component can execute without this interface being bound.

The cardinality of an interface type T specifies how many interfaces of type T a given commponent may have. A *singleton* cardinality means that a given component must have exactly one interface of type T. A *collection* cardinality means that a given component may have an arbitrary number of interfaces of type T. Such interfaces are typically created lazily, e.g. upon request of a bind operation through a BindingController interface.

Component types are just sets of component interface types. The type system is equipped with a subtyping relation which embodies constraints to ensure substitutability of components.

2.4. Instantiation

The FRACTAL model also defines *factory* components, i.e. components that can create new components. Again, the FRACTAL model does not constrain the form and nature of factory components, but the FRACTAL specification provides useful forms of such factories. In particular, it distinguishes between *generic component factories*, which can create several kinds of components, and *standard factories*, which can create only one kind of components, all with the same component type. A generic factory provides the GenericFactory interface, which allows a new component to be created, given its type, and an appropriate description of its membrane (controllers) and content. A *template* is a special standard factory that creates components that have the same internal structure as the template. Thus, a template component can have several templates as sub-components (sub-templates). A component created by such a template will have as many sub-components as sub-templates in the template, which will be bound together in the same way as the sub-templates are. Templates are useful to manifest at run-time a particular configuration.

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.

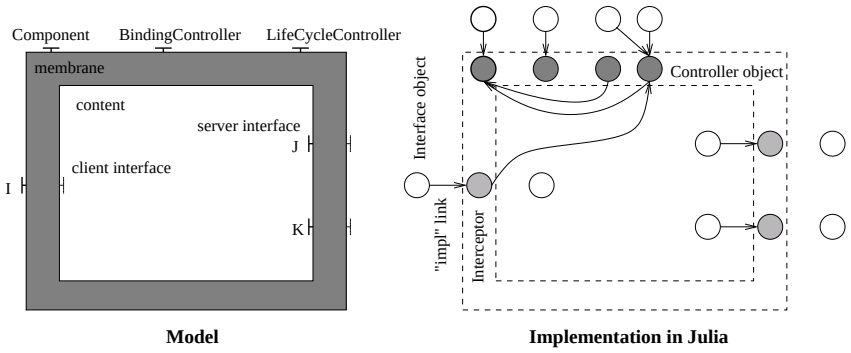

Figure 3. An abstract component and a possible implementation in JULIA

3.1. Main Data Structures

A FRACTAL component is generally represented by many Java objects, which can be separated into three groups (see Fig. 3):

- the objects that implement the component interfaces, in white in Fig. 3 (one object per component interface; each object has an impl reference to an object that really implements the Java interface, and to which all method calls are delegated; this reference is null for client interfaces; for server interfaces it can reference an interceptor or an object of the content part),
- 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 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. 3), 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 membrane 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 InitializationContextClassGenerator. 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

3.2.1. Motivations

The main design goal of JULIA is to implement a reusable and extensible *framework* to program component membranes. In particular, since everything in the Fractal specification is optional, JULIA must provide implementations of the Fractal API interfaces for any conformance level. For example, JULIA must provide a basic Component implementation, as well as an implementation for components whose type follows the basic type system (in the first case Component behaves like a read only hash map; in the second case, because of collection interface types, the getFcInterface method can lazily create new component interfaces - see section 2.3). Likewise, JULIA must provide a basic BindingController implementation, as well as an implementation for cases where the basic type system is used, where a life cycle controller is present, or where composite components are used (these implementations are needed to check type, life cycle or content related constraints on bindings). There must also be an

implementation for cases where both the basic type system and a life cycle controller are used, or where the basic type system, life cycle controllers, and composite components are used. And these implementations must be extensible, in order to take into account user defined controllers when needed.

In order to provide all these implementations, a first solution would be to use class inheritance. But this solution is not feasible, because it leads to a combinatorial explosion, and to a lot of code duplication. Consider for example the BindingController interface, and the "type system", "life cycle" and "composite" concerns. These three concerns give $2^3 = 8$ possible combinations. Therefore, in order to implement these three concerns, *eight* classes (and not just three) must be provided. Moreover these eight classes can not be provided without duplicating code, if multiple inheritance is not available.

Another solution to this problem would be to use an Aspect Oriented Programming (AOP) tool or language, such as Aspect/J [24], since the goal of these tools and languages is to solve the "crosscutting" problems. Aspect/J, for example, could effectively be used to solve the above problem: aspect classes could indeed be used instead of sub classes, which would solve the combinatory problems (an aspect can be applied to multiple classes and aspects). But using Aspect/J would introduce a new problem, due to the fact that, in Aspect/J, 1) aspects must be applied at compile time, and 2) this process requires the source code of the base classes*. It would then be impossible to distribute JULIA in compiled form, because then 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 extensibility 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 [29]. 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. Moreover, mixin classes used in JULIA can be mixed at load time, thanks to our bytecode generator ASM [1] (unlike in AspectJ and in most mixin based inheritance languages, where mixed classes are declared at compile time).

3.2.2. 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 [15] mixin class shown below (on the left) is written in pure Java as follows (on the right):

*This is no longer true with version 1.1 of AspectJ, but this was the case in 2002 when JULIA was developed.

```
mixin A {
  inherited public void m ();
  public int count;
  public void m () {
    ++count;
    super.m();
  }
}
```

```
abstract class A {
  abstract void _super_m ();
  public int count;
  public void m () {
    ++count;
    _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 overridden by the mixin class. Members that are required but not overridden are denoted with _this_:

```
abstract class M implements I {
  abstract void _super_m ();
  abstract void _this_n ();

  public int count;
  public void m () {
    ++count;
    _this_n();
    _super_m();
  }
}
```

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). For example, if N and O designate the following mixins:

```
abstract class N implements I {
  abstract void _super_m ();

  public void m () {
    System.out.println("m called");
    _super_m();
  }
}
```

```
abstract class O implements I {
  public void m () {
    System.out.println("m");
  }
  public void n () {
    System.out.println("n");
  }
}
```

then the mixed class O N M is equivalent to the following class (note that this class implements all the interfaces implemented by the mixin classes):

```
public class C55d992cb_0 implements I, Generated {
  // from 0
  private void m$1 () {
    System.out.println("m");
  }
  public void n () {
    System.out.println("n");
  }
  // from N
  private void m$0 () {
    System.out.println("m called");
    m$1();
  }
  // from M
  public int count;
  public void m () {
    ++count;
    n();
    m$0();
  }
}
```

The mixed classes are generated dynamically, directly in bytecode form with ASM [1], by the MixinClassGenerator class. In order to ease debugging, the class generator keeps the line numbers of the mixin classes in the mixed class. More precisely, a line number l of the mixin class at index i (in the list of mixin classes, and starting from 1) is transformed into $1000 * i + l$. For example, if a new Exception().printStackTrace() were added in the N.m method, the stack trace would contain a line at C55d992cb_0.m$0(ONM:2005), meaning that the exception was created in method m$0 of the C55d992cb_0 class, whose source is the ONM mixed class, at line 5 of mixin 2, i.e. at line 5 of the N mixin class.

3.3. Interceptors

Some control aspects, such as the control of bindings, can be completely implemented in a generic way, in a single controller object. But most control aspects must be implemented in two parts: a generic part, and a non generic "hook" part that must be weaved into the user code. In JULIA, this non generic "hook" part is made of the interceptor objects (see Section 3.1), and the weaving is done by inserting these interceptor objects between user objects.

The interceptor classes, since they are not generic (they must implement one or more user interfaces), cannot be written by hand, unlike controller classes, and must therefore be generated automatically. As in some Meta Object Protocol (MOP) implementations, JULIA generates these classes directly in compiled form, by using the ASM library. This is much faster than generating these classes in source code form, which allows JULIA to generate

them dynamically, during the application's execution (but JULIA also offers the possibility to generate them statically, before launching an application).

However, unlike in most MOP implementations, the generator that generates the interceptor classes is open and extensible. This flexibility was introduced for efficiency reasons. Indeed this open generator can be used not only to generate interception code that reifies all method calls, as in MOPs, but also to generate much more efficient code, specialized for a given set of aspects. This open generator is described in the rest of this section.

The interceptor class 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 weaver(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 weavers.

Each aspect code weaver is an object that can manipulate the *bytecode* of each application specific method *arbitrarily*. For example, an aspect code weaver A can modify an empty interception method `void m () { return delegate.m() }` into:

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

where the pre and post code blocks can be adapted to the precise arguments and return types of `m`, while another aspect code generator B will modify this method into:

```
void m () {
  // pre code B
  delegate.m();
}
```

When an interceptor class is generated by using several aspect code weavers, the transformations performed by these weavers 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 (depending on the order in which A and B are composed):

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

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

```
    }                          }
  }                          }
```

Note that, thanks to this elementary automatic weaving, which is very similar to what can be found in Aspect/J, 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 provides two specific aspect code weavers (to manage the life cycle and trace aspects), and two generic code weavers that reify method calls (one that just reifies method names, and one that also reifies the arguments). Users can of course provide their own code weavers, but writing such a weaver requires a good knowledge of the Java bytecode instructions.

3.4. Optimizations

3.4.1. Intra component optimizations

In order to save memory, JULIA provides some optimization options to merge some or all the objects that make up a component into a single Java object. These optimizations are based on a tool provided by JULIA, which uses the ASM library to merge several controller classes into a single class. This tool is based on the following assumptions:

- 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 configuration, 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 bindings between objects in a given configuration are established automatically in a two steps process: a) each controller object of the configuration is registered into a "naming service" (in practice, this naming service is the InitializationContext interface), and b) each controller object initializes itself by using the previous "naming service" to retrieve the interface it requires.

To be more precise, lets suppose we have four control interfaces I, J, K and L, and three controller classes IImpl, JImpl and KImpl. These classes should look like this:

```
public class IImpl implements Controller, I {
  public J weaveableJ;    // = required interface of type J
  public L weaveableOptL; // = optional required interface of type L
  public int foo;         // normal field
```

```
    // implementation of the Controller interface
    public void initFcController (InitializationContext ic) {
      weaveableJ = (J)ic.getInterface("j");
      weaveableOptL = (L)ic.getOptionalInterface("l");
    }
    // other methods
    public void foo (String name) {
      weaveableJ.bar(weaveableOptL, foo, weaveableC.getFcInterface(name));
    }
  }

  public class JImpl implements Controller, J {
    public K weaveableK;
    public void initFcController (InitializationContext ic) {
      weaveableK = (K)ic.getInterface("k");
    }
    // other methods ...
  }

  public class KImpl implements Controller, K {
    // other methods ...
  }
```

In the non optimized case, a component with these three controller objects is instantiated in the following steps. First an instance of IImpl, JImpl and KImpl is created, then the resulting objects are put in an InitializationContext object, and finally the initFcController method is called on each controller object with this context as argument. In the optimized case, the class obtained by "merging" (see below) the IImpl, JImpl and KImpl is dynamically generated (or loaded from the classpath if it has been statically generated before launching the application, or just returned if it has already been generated or loaded) and then an instance of this class is created.

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 with this, and those whose name begins with weaveableOpt are replaced either with this, if a class that implements the corresponding type is present in the list of the classes to be merged, or null otherwise. Finally, the initFcController methods from the Controller interface are merged into a single initFcController method. The result is the following class:

```
public class Cb234f2 implements Controller, I, J, K, ..., Generated {
  // fields and methods copied from IImpl:
  public int foo;
  public void foo (String name) {
    bar(null, foo, this.getFcInterface(name));
  }
```

```
    private void initFcController$0 (InitializationContext ic) {
      (J)ic.getInterface("j");
      (L)ic.getOptionalInterface("l");
    }
    // fields and methods copied from JImpl (not shown) ...
    // fields and methods copied from KImpl (not shown) ...
    // merged initFcController method:
    public void initFcController (InitializationContext ic) {
      initFcController$0(ic);
      initFcController$1(ic);
      initFcController$2(ic);
    }
  }
```

As explained in section 3.1, the membrane of a component is made of controller objects and of interceptor objects. The above optimization only applies to controller objects. Therefore, even with this optimization, the membrane of a component is still made, in general, of several Java objects. However, if the interceptor objects all delegate to the same object, and if they do not have conflicting interfaces, it is possible to really have only one Java object for the whole membrane of the component. In this case, which happens for most primitive components, the instantiation process is the following:

- a class that merges the controller classes is generated or loaded as before,
- a sub class of this class that implements the interception code for each method of each functional interface is generated or loaded,
- this sub class is instantiated.

Even if the controllers and interceptors are merged into a single object, the content of the component is still made of a separate object. It is however possible to instantiate a whole component (i.e. the controllers, the interceptors and the content part) as a single Java object. In order to do this, the user component class is used as a super class to generate the merged controller class, which is itself used a super class to generate the interceptor class (as described above).

3.4.2. Inter component optimisations

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 in section 3.1, 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 another server interface.

More precisely, this is the case with the CompositeBindingMixin. With the OptimizedCompositeBindingMixin, the impl references are optimized when possible. For example, in Fig.

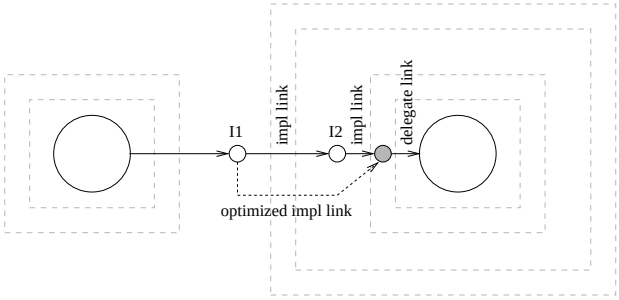

Figure 4. Shortcut bindings

4, 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. The OptimizedCompositeBindingMixin automatically manages these shortcuts. In particular, this mixin invalidates and recomputes the necessary shortcuts when a binding is modified (indeed, modifying a binding somewhere may invalidate existing shortcuts, and/or create new shortcuts).

3.5. 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 the JULIA runtime (35kB, plus 10kB for the FRACTAL API), which is the only part of JULIA (175 kB as a whole) 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.

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 [33], auto-adaptability [21], 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 [34]. As discussed below, we have also built with JULIA a component library, called DREAM, for building message-oriented middleware (MOM) and reengineered an existing MOM using this library. DREAM components exhibit specific control aspects, dealing with on-line deployment and reconfiguration. In all these experiences, the different mechanisms in JULIA have proved sufficient to build the required control aspects.

Accessibility Besides JULIA, several tools are available to easily implement, assemble, deploy and manage Fractal components in Java: Fractlet provides annotations to generate several artifacts from a single source file (like XDoclet), Fractal ADL can be used to describe and deploy Fractal architectures, Fractal GUI can be used to graphically edit Fractal ADL XML files, and Fractal Explorer and Fractal JMX can be used to introspect and manage running Fractal applications.

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 overridden 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 I. 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 [†] (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 I, for different optimization options. The measurements were 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 a component interface 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).

[†]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.

4.3. Quantitative evaluation II: the DREAM communication framework

In this section we present DREAM, a framework for the construction of asynchronous middleware, which relies on JULIA. We first briefly describe the framework. Then we show how it has been used to re-engineer JORAM [3], an open-source JMS-compliant middleware (Java Messaging Service [2]).

4.3.1. A component-based framework for asynchronous middleware

Motivations The use of asynchronous middleware (MOM for *Message-Oriented Middleware*) is recognized as a means of achieving scalability in applications made of loosely coupled autonomous components that communicate on large-scale networks [16]. Several MOMs have been developed in the past ten years [3, 19, 35, 37]. The research work has primarily focused on the support of various non functional properties like message ordering, reliability, security, etc. Less emphasis has been placed on the MOM configurability. Indeed, existing middleware are not very configurable, both at the functional and non-functional level. From the functional point of view, they implement a fixed programming interface (API), thus providing a fixed subset of asynchronous communication models (publish/subscribe, event/reaction, message queues, etc.). From the non-functional point of view, existing middleware often provide the same non-functional properties for all event disseminations. This reduces their performance and makes them difficult or impossible to use with devices having limited computational resources.

To overcome these limitations, we have developed DREAM (*Dynamic REflective Asynchronous Middleware*), a software framework dedicated to the construction of asynchronous middleware. DREAM provides a component library and a set of tools to build, configure and deploy middleware implementing various asynchronous communication paradigms: message passing, event-reaction, publish-subscribe, etc.

Architecture of a DREAM *component* DREAM components are standard Fractal components with two characteristic features: the presence of input/output interfaces and the ability to manipulate DREAM resources (messages and activities).

Input/Output interfaces allow DREAM components to exchange messages. Messages are always sent from outputs to inputs (Figure 5 (a)). Output and input interfaces come in pairs corresponding to two kinds of connections, *push* and *pull*. As shown in Figure 5 (b) and (c), "input" and "output" are roles played by normal client and server interfaces (the input and output roles are played by server and client interfaces, respectively, for a push connection; and vice versa for a pull connection).

Message managers Messages are managed by dedicated shared components, called *message manager*. They allow DREAM components to create, duplicate or delete messages. Messages are particular Fractal composites that encapsulate *chunks*. A chunk is a unit of data allocation. Each chunk provides a server interface exported by the message it belongs to. As an example, messages that need to be causally ordered have a chunk that provides a `Causal` server interface. This interface defines methods to set and get a matrix clock. A message may encapsulate other messages and is uniquely identified by an interface called `Message` that gives access to the message's chunks and encapsulated messages.

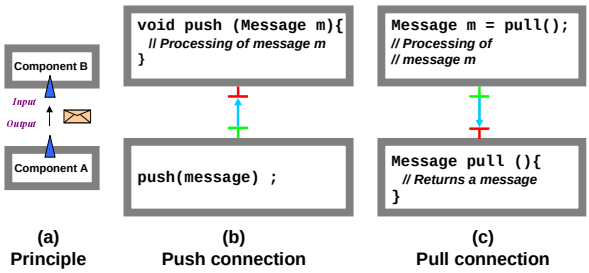

Figure 5. Connection between input/output interfaces

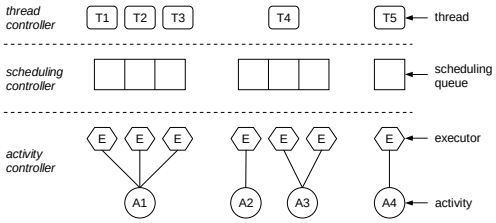

Figure 6. Activity management

Activity management A DREAM component can either be passive or active. An active component has its own *activities*; a passive component doesn't, i.e. calls to other component interfaces can only be made in the activity of a calling component. An activity is a Java object implementing a method `run`. This method is executed as long as it returns a positive integer.

Active components have three controllers depicted in Figure 6, which we now describe. The **activity controller** allows the component to register, unregister, start, and stop activities. Activities are wrapped by *executors* that are in charge of the lifecycle of the activities they wrap. In particular, when a component needs to be stopped, the executors guarantee a safe interruption of the activies of the component. The number of executors wrapping a given activity is specified as a parameter of the activity's registration. Executors are executed by threads managed by a **thread controller**. Each thread is associated to a *scheduling queue*, where the **scheduling controller** places the next activities to be executed. This architecture allows fine-grained control over threads executing in the system, which is a required feature to build scalable asynchronous middleware as illustrated by the SEDA framework [38].

The DREAM *library and tools* By lack of space we only describe the core components of the DREAM library, i.e. the components encapsulating functions and behaviors commonly found in an asynchronous middleware. Note that the library also contains specific components developed

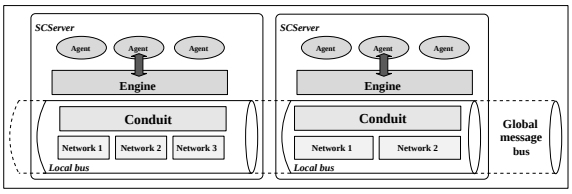

Figure 7. Two interconnected agent servers

for particular middleware: for instance, components implementing event-reaction processing. Examples of such components are given with the example presented in the next section.

Message queues are used to store messages. Queues differ by the way messages are sorted (FIFO, LIFO, causal order, etc.), and the behavior of the queue when the capacity is exceeded (blocks vs. removes messages), when the queue is empty, etc.

Transformers have one input to receive messages and one output to deliver transformed messages. Typical transformers include stampers.

Routers have one input and several outputs (also called "routes"), and route messages received on their input to one or several routes.

Filters have one input and one output. Messages received on the input are either delivered on the output, or deleted.

Aggregators have one or several inputs to receive the messages to be aggregated, and one output to deliver the aggregated message.

De-aggregators implement aggregators' reverse behavior, i.e. they take an aggregated message and generate appropriate individual messages from it.

Channels allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate, at least, two components: a *ChannelOut* — which aims at sending messages to another address space —, and a *ChannelIn* — which can receive messages sent by the ChannelOut.

4.3.2. Re-engineering JORAM

This section presents how DREAM has been used to re-engineer JORAM. We first briefly present JORAM. Then we detail its implementation using DREAM. Finally, we compare both implementations in terms of configurability and performance.

A brief introduction to JORAM JORAM comprises two parts: the ScalAgent message-oriented middleware (MOM) [17], and a software layer on top of it to support the JMS API.

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

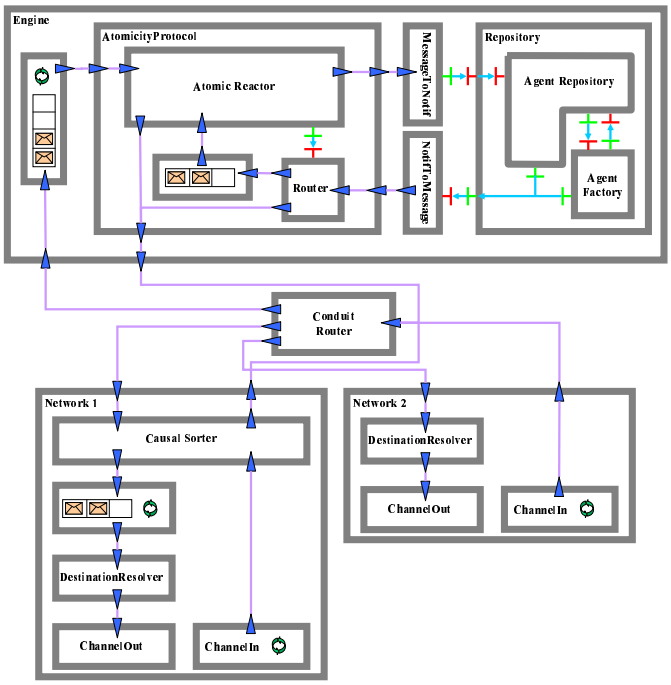

Figure 8. Architecture of an agent server

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.

Implementing JORAM *using* DREAM We have implemented the ScalAgent MOM using DREAM (see Figure 8). Its main structures (networks, engine and conduit) have been preserved to facilitate the functional comparison between the ScalAgent MOM and its DREAM re-implementation.

The **engine** comprises two main components: the `AtomicityProtocol` composite that ensures the atomic execution of agents; the `Repository` composite, which is in charge of creating and executing agents. Two typical **networks** are depicted. Both are composite components encapsulating a `TCPChannelIn`, a `TCPChannelOut` and a `DestinationResolver` component. The latter is a *transformer* that adds the information required by the `TCPChannelOut` component (i.e. IP address, and port number). The `Network 2` composite contains two more components: the `CausalSorter` causally orders messages; the message queue

decouples the workflows of the engine and the network. The **conduit** is implemented by a router.

4.3.3. Configurability assessment

A first benefit of the DREAM implementation comes from the ability to easily change provided non-functional properties. For instance, it is straightforward to remove causal ordering, or to remove the atomic protocol ensuring transactional execution of agents. Both modifications can be programmatically done at runtime. On the other hand, removing these properties from the ScalAgent MOM requires modifying and recompiling its source code. Moreover, by implementing the conduit as a router, an agent server can have multiple engines, which is not the case in the ScalAgent implementation. This is interesting for two reasons: it allows the parallelization of agent executions (within an agent server, agent executions are serialized [17]) and different non-functional properties can be simultaneously enforced (persistency, atomicity).

Another benefit brought by implementing the MOM with DREAM is that it is easy to change the number of active components encapsulated within the agent server. The architecture we have presented in Figure 8 involves three active components for an agent server with one network. A mono-threaded architecture can be obtained by removing the message queues encapsulated within the engine and the network.

A last experiment we have done, is to build an agent server for mobile equipments. These equipments may be temporarily disconnected from the network and have limited storage capacity. To overcome these limitations, we have built an engine whose message queue is replaced by a `TCPChannelIn` component, and which encapsulates a `TCPChannelOut` component to send messages. Another device acts as a proxy and message storage unit for this engine. This architecture preserves the MOM functionnality, while saving memory: it is mono-threaded; messages are pulled instead of pushed; it has no `CausalSorter` and `DestinationResolver` components.

4.3.4. Performance comparisons

Measurements have been performed to compare the efficiency of the same 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. 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 II shows the average number of rounds per second, and the memory footprint. We have compared two implementations using DREAM with the ScalAgent implementation. 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. Concerning the memory footprint, the DREAM implementation requires 9% more memory, which can be explained by some of the structure needed by Fractal ($\approx$ 70kB) and the fact that each component has several controller objects. This memory overhead is not significant for standard

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 II. Performance of DREAM implementations vs ScalAgent implementation

MOM	Number of rounds 0 kB	1 kB	Memory footprint (kB)
Dream (3 threads)	329	260	4 × 1580
Dream (2 threads)	346	268	4 × 1516
Dream (1 thread)	370	279	4 × 1452

Table III. Impact of the concurrency level

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 IV. Impact of the number of engines by agent server

PC. The second implementation is dynamically reconfigurable (in particular, each composite component supports a life-cycle controller and a content controller). This implementation is slower than the ScalAgent one (≈ 2,2 to 2%) and only requires 7kB more than the non-reconfigurable implementation made using DREAM.

Table III reports on experiments we have done to assess the impact of the concurrency level on the performances of the ScalAgent MOM. We compare three architectures built using DREAM that differ by the number of active components they involve. In the 2-thread architecture the message queue encapsulated in the network has been removed. In the mono-threaded architecture, both active message queues have been removed (Engine and Network). We see that, in this particular case, reducing the number of active components improves the number of rounds (+ 5 to 3% for the 2-thread architecture, and + 12 to 7% for the mono-threaded architecture). This can be explained by the fact that agents are organized in a virtual ring, thus each agent server only processes one message at a time. As a consequence, only one thread is necessary.

We have also evaluated the gain brought by changing the configuration in a multi-engine agent server. 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. Moreover, in the latter case, agents are placed so that two consecutive agents in the virtual ring are hosted by different agent servers. Table IV 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 by the causal sorter 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 to be sent with messages, and tested before delivering them. Table IV 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 directly sends the message to the appropriate engine.

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 [36, 27, 25], 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 [28]), 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 [20]. 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). With respect to industrial standards such as EJB and CCM, FRACTAL constitutes a more flexible and open component model (with hierarchical composites and sharing) which does not embed predetermined non functional services. It is however perfectly possible to implement such services in FRACTAL, as demonstrated e.g. by the development of transactional controllers in [33]. Note also that FRACTAL is targeted at system engineering, for which EJB or CCM would be inadequate.

Software architecture in Java Several component models for Java have been devised in the last ten years. Apart from "standardized" models such as Java Beans, Enterprise Java Beans (EJB) or OSGI [4], we find open source initiatives such as Avalon [5] which is a general component model, Kilim [9], Pico [10] and Hivemind [7] which are targeted towards software configuration, Spring [12], Carbon [6], and Plexus [11] which are targeted towards component containers (in the line of EJB). These models suffer generally from the lack of extensibility and tailorability mentioned in the introduction. Carbon is probably the closest from FRACTAL as it provides extensibility and dynamicity through a mechanism based on *decorators* and *interceptors* and a JMX-based supervision. Two recent proposals for Java-based component programming include Jiazzi [26] and ArchJava [13]. Unlike these works, our

approach to component-based programming in Java does not rely on language extensions for configuration purpose: JULIA is a small run-time library, complemented with simple 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, ArchJava, works cited above and most existing architecture description languages (ADLs). 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 [23], 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 [14], 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. [22, 30, 32, 8, 12]. 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 [31] provides dynamic aspect weaving (whereas JULIA currently supports only load-time controller generation), with performance which appears to be comparable to that of JULIA. Prose, however, relies on a modified JVM, which makes it impractical 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 load time 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 Byte-Code 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://joram.objectweb.org/`.
4. OSGi Service Platform, Release 3, 2003. `http://www.osgi.org/`.
5. The Apache Avalon project, 2004. `http://avalon.apache.org/`.
6. The Carbon project, 2004. `http://carbon.sourceforge.net/`.
7. The Hivemind project, 2004. `http://jakarta.apache.org/hivemind`.
8. The JBoss Aspect Oriented Programming project, 2004. `http://www.jboss.org/`.
9. The Kilim project, 2004. Objectweb, `http://kilim.objectweb.org/`.
10. The PicoContainer project, 2004. `http://www.picocontainer.org/`.
11. The Plexus project, 2004. `http://plexus.codehaus.org/`.
12. The Spring framework, 2004. `http://www.springframework.org/`.
13. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th ECOOP*, 2002.
14. J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings 17th ECOOP*, 2003.
15. D. Ancona, G. Lagorio, and E. Zucca. A Smooth Extension of Java with Mixins. In *ECOOP'00, LNCS 1850*, 2000.
16. G. Banavar, T. Chandra, R. Strom, and D. Sturman. A Case for Message Oriented Middleware. In *Lecture Notes in Computer Science*, volume 1693, pages 1–18, Bratislava, Slovak Republic, September 1999. 13th International Symposium on Distributed Computing. ISBN 3-540-66531-5.
17. 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.
18. E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model. Technical report, Specification v2, ObjectWeb Consortium, 2003.
19. A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
20. 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 Middleware Conference*, 2001.
21. P. David and T. Ledoux. Towards a Framework for Self-adaptive Component-Based Applications. In *DAIS 2003, LNCS 2893*, 2003.
22. F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *AOSD02*, 2002.

23. 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.
24. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W Griswold. An Overview of AspectJ. In *ECOOP 2001, LNCS 2072*, 2001.
25. G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
26. S. McDirmid, . Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings OOPSLA '01, ACM Press*, 2001.
27. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Soft. Eng., vol. 26, no. 1*, 2000.
28. G. Outhred and J. Potter. A Model for Component Composition with Sharing. In *Proceedings ECOOP Workshop WCOP '98*, 1998.
29. 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.
30. R. Pichler, K. Ostermann, and M. Mezini. On Aspectualizing Component Models. *Software – Practice and Experience*, 2003.
31. A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for Java. In *AOSD03*, 2003.
32. A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *17th ECOOP*, 2003.
33. M. Prochazka. Jironde: A Flexible Framework for Making Components Transactional. In *DAIS 2003, LNCS 2893*, 2003.
34. N. Rivierre and T. Coupaye. Observing component behaviors with temporal logic. In *8th ECOOP Workshop on Correctness of Model-Based Software Composition (CMC '03)*, 2003.
35. Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *International Symposium on Software Reliability Engineering (ISSRE'98), fast abstract*, Paderborn, Germany, November 1998.
36. C. Szyperski. *Component Software, 2nd edition*. Addison-Wesley, 2002.
37. R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), 2003.
38. M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, 2001.