Recursive and Dynamic Software Composition with Sharing

E. Bruneton[†], T. Coupaye[†], and J. B. Stefani[‡]
{eric.bruneton, thierry.coupaye@rd.francetelecom.com},
{jean-bernard.stefani@inria.fr}

[†]France Télécom R&D (DTL/ASR Department)
Chemin du Vieux Chêne, BP 98, 38243 Meylan, France
[‡]INRIA (Sardes Project)
ZIRST, 655 avenue de l'Europe 38330 Montbonnot Saint Martin, France

1 Introduction

Component-based programming and component-based system construction have emerged as important topics in software engineering and distributed computer systems, as witnessed by the number of recent papers addressing these themes in the sub fields of software architecture (e.g. [1, 16]), software configuration management (see e.g. [12]), and configurable and reflective distributed systems (see e.g. [2, 4, 15]), not to mention standardized component models such as OMG's CCM, Sun's EJB, or Microsoft COM. *Components*, in these works, are variously understood as units of software configuration, units of dynamic system configuration, units of deployment, units of distribution and mobility, etc.

Despite these different works, we believe we are still missing an appropriate basis for the construction of highly flexible, highly dynamic, heterogeneous distributed environments. We attribute that to the failure of the proposed different models to meet several key requirements along several of the practical dimensions identified by Van Hoek et al [20] for the analysis of component-based system models. In particular, several key requirements along the composition dimension (which deals with facilities for modelling a system as a structured set of components) and the dynamism dimension (which deals with facilities for modelling the dynamic changes of a system) are currently insufficiently met.

In this paper, we introduce a component model which fares better along the two above dimensions than its predecessors. In particular, it includes a combination of sharing, hierarchy, and an open approach to component composition and binding which we believe to be novel. The paper is organised as follows. Section 2 gathers crucial requirements we identify on a general component model for distributed configurable systems. Section 3 introduces the Fractal model, our proposal for such a general component model. Section 4 introduces a software that we have developed to support the Fractal model and support the programming of Fractal components and systems in Java. Section 5 briefly reports on a first implementation of the Fractal framework. Section 6 concludes the paper.

2 Requirements for a general component model

We gather and motivate in this section seven key requirements, along the composition and dynamism dimensions, for a general component-based system model. The requirements are predicated upon the basic assumption that components are manifest during system execution, and that they can be construed as units of dynamic system configuration, two crucial assumptions for building highly dynamic and highly configurable distributed systems[1].

(1) Encapsulation and Identity. Components should exhibit encapsulation, abstraction and identity. Encapsulation, means that components should only interact with their environment through well-defined interactions, and that any modification of the state of a component initiated by its environment should be by means of such interactions. Abstraction means that a component should not reveal more details of its internal implementation than is necessary for entities in its environment to meaningfully interact with it. Identity means that components should have a well-defined identity, that unambiguously distinguishes one component from another so as to allow a component to be accessed and manipulated specifically. This requirement is very close to requiring a component to be an object, in the sense of object introduced in the Reference Model for Open Distributed Processing [9, 10].

(2) Composition. A general system model should allow the dynamic composition or assembly of components in order to form new, higher-level components. As far as possible, the model should not prescribe one particular way of composing components together, but should support different semantics of composition. This requirement is motivated in particular by reuse and dynamic configuration needs. A system administrator, a system integrator or an application developer may need to dynamically construct a system or service out of existing components, whether in response to failures, as part of the continuous evolution of a running system, or just to introduce new applications in a running system (a direct generalization of the dynamic binding used in standard distributed client-server applications). Workflow applications in a large scale distributed setting such as the World-Wide Web provide an illustration of the latter. In an e-commerce scenario, say, workflow applications need to dynamically combine already running subsystems and services, building new services in the process, which can in turn be used by others and further combined. As another example, consider a system administrator who needs to dynamically replace a failed component by a correct one in a given subsystem, while preserving the internal integrity of the subsystem and maintaining the continuity of the service it provides. To characterize integrity in this setting requires the identification of a subsystem as a component assembly and to be able to manifest, at run-time, the dependence of a subsystem on an internal component. Another way to phrase this requirement is to say that arbitrary containment relationships between components should be explicitly maintained, enforced and modifiable at run-time.

(3) Sharing. A general system model should allow the description and construction of system configurations (i.e. component assemblies) where at least multiple forms of component sharing are possible. The motivation behind this requirement lies with the necessity to deal with situations in (hardware and software) infrastructures involving sharing and multiplexing. This is the case, for instance, in resource management [3], whether dealing with low-level hardware and operating system resources, or with higher-level virtual resources such as servers, agents or virtual connections. In all these cases, dynamic system configuration requires the ability to make explicit and to manipulate resource dependencies (for instance, the sharing of the same processor components across multiple threads of activity and multiple protection domains, the multiplexing of a communication connection to support multiple higher level protocol sessions, the sharing of replicated data servers across multiple transactions, the construction of virtual network paths through the multiplexing of supporting router and transport resources). In other terms, a general component model should allow resource dependencies to be explicitly maintained, enforced and modifiable at run-time.

(4) Life-cycle. A general system model should support different forms and spans of component life-cycles, including e.g. different forms of bootstrapping, deployment, installation, initialization, suspension, termination, etc. This requirement stems from the need to support system management processes and the necessity to automate a large part of such processes in large distributed systems. This requirement is directly related to the requirement of control: a life-cycle model can be understood as a superimposition of life-cycle states on the functional behavior of a or collection of components.

(5) Activities. A general system model should allow activities to be made explicit and should support the manipulation of activities taking place in a system, possibly spanning or involving multiple components. This requirement is related to the preceding one on the ability to capture system configurations involving sharing and multiplexing: activities can be perceived as components in their own rights which rely on resources provided by the components involved. The case for dynamically managing activities and resource

[1]This assumption is at odds with certain definitions of components (see e.g. [19]). However, we believe these particular notions can be shown to be a special case of our more general component model.

dependencies in a distributed setting is further elaborated e.g. in [3]. This requirement is also related to the second requirement on component composition, since one would expect different forms of composition to apply to activities as well, a feature of obvious interest in workflow and (extended) transactional applications, for instance.

(6) Control. A general system model should allow the creation different forms of controller components, i.e. components that can provide various introspective capabilities for monitoring a collection of components and exercizing control over their execution. Forms of control which we can expect to find include for instance interception, whereby a controller component can intercept messages or invocations targeting a given component for e.g. adding pre and post-handlers to the normal component execution, and superimposition [7], whereby a controller component can alter the normal execution of a component in different ways (e.g. by delaying or preventing the handling of a message). Some interception capabilities are already manifest in component containers introduced in standard component models such as Enterprise Java Beans, and the CORBA Component Model. This requirement can be understood as a refinement of the second requirement above on composition.

(7) Mobility. A general system model should allow the description of arbitrary changes in configurations of components and component assemblies. In particular the model should allow the dynamic creation of new components and new component assemblies, as well as the migration of components or component assemblies from one component assembly to another. In other terms, containment and resource dependencies among components should be allowed to change and evolve over time, whether spontaneously or in reaction to interactions amongst components or between components and their external environment. Such a facility is crucial to avoid architectural erosion, whereby run-time system structures diverge from architecture descriptions, and to model complex reconfiguration processes.

3 The Fractal Component Model

The model is based around the concepts of *components*, *interfaces* and *names*. A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, which are under the control of the controller of the enclosing component. The component model is recursive and allows components to be nested (i.e. to appear in the content of enclosing components) at an arbitrary level (the recursion ends with components with empty content, i.e. multi-interface objects as per the ODP Reference Model).

Different components may have overlapping content, i.e. a component may be *shared* by (be a sub component of) several distinct enclosing components. A component that is shared among two or more distinct components is subject to the control of their respective controllers. The exact semantics of the resulting configuration (e.g. which control behaviour is enacted) is determined by an encompassing component that encloses all the relevant components in the configuration.

A component can interact with its environment through *operations* at identified access points, called *interfaces*. The visibility of the interfaces of a sub-component, in and out of the enclosing component, is determined by the controller of the enclosing component. In particular, a component controller may hide an interface of a sub-component, or make it visible from its environment. Interfaces of a component visible from its environment comprise interfaces from its sub-components made visible from its controller, as well as external interfaces from its controller. Note that a component controller may have internal interfaces, which are not visible from the component environment but which are visible from its sub-components.

Operations are basic interactions in the Fractal model. They can be of two sorts: one-way and two-way. A one-way operation consists only in an operation invocation. A two-way operation consists in an operation invocation, followed by the return of a result. Operation invocations and operation returns carry arguments which can be names (esp. interface references), values or passivated forms of components.

Interfaces can be of two sorts: client interfaces and server interfaces. A server interface can receive operation invocations (and return operation results of two-way operations). A client interface can emit operation invocations (and receive operation results of two-way operations). Fractal components can have a variable number of interfaces during their life-time.

A *binding* is a connection between two or several components. The Fractal model comprises primitive bindings and composite bindings. A primitive binding is a directed connection between a *client interface* and a *server interface*. Primitive bindings are typically implemented as language-level bindings in a realization of the Fractal model (see below). A primitive binding can be established between a client and a server interface only if the server interface can accept at least all the operation invocations that the client interface can emit and if the client interface can accept (at least) all the returns from previously invoked operations on the server interface. In other words, the type of the server interface must be a sub-type of the type of the client interface. Composite bindings are realized through a combination of primitive bindings and ordinary components, i.e. composite bindings are themselves Fractal components. This notion of binding generalizes the notion of connectors in architecture definition languages and maps directly on notions of bindings in flexible distributed systems such as the Jonathan ORB framework [6].

A component controller embodies the control behaviour associated with a particular component. In particular, a component controller can intercept oncoming and outgoing operation invocations and operation returns targeting or originating from the component's content it controls; it can provide an explicit and causally connected representation of the component's forming the content it controls; and it can superimpose a control behaviour to the behaviour of the components in its content, including suspending (passivating) and resuming activities of these components. Each controller can thus be seen as implementing a particular composition operator for the components in its content. The Fractal model does not place a priori restrictions on the forms of control and composition a component controller can realize: it can be mainly interception-based as in industrial component frameworks containers for instance ; it can be limited to providing a common execution context for the components in its content; or it can realize intrusive forms of superimposition [7].

Fractal components can create other components, within or outside of their content. A component may also passivate itself, i.e. serialize its entire state into some defined data structure. A fractal component can be distributed, i.e. the components in its content may actually execute in different address spaces, located in different machines. This is typically the case for composite bindings that encompass network connections.

Names are symbols for denoting in the model. A name that refers to a component interface is called an interface reference. The model does not place constraints on the form and structure of names. Instead, names are understood to be relative to a given naming context, i.e. names perform their referral function unambiguously only within a given naming context. Naming contexts can be nested and overlapping, allowing names to be valid in different naming contexts. A component controller constitutes a primitive naming context.

Components, interfaces and method arguments are typed: each of these entities is characterized by a (statically verifiable) predicate constraining its nature and behavior. The model places few constraints on the type system, other than it being organized as a lattice. Also, the type of a component must comprise at least (i.e., as a predicate, implies at least) the union of the types of its interfaces. The type of an interface comprises at least the following elements: an interface signature, which consists in a set of operations signature (name of the operation plus types of the arguments and of the possible operation return arguments); the role of the interface (client or server); the contingency of the interface, which may take two values: mandatory or optional. A mandatory interface is a usual (client or server) interface. An optional interface is an interface that may be bound or not to a component at runtime. Optional interfaces are used to specified optional components in a given configuration.

4 The Fractal Software Framework

The Fractal framework is a Java software framework that supports component-based programming according to the Fractal model. The Fractal framework is an open framework that comprises a *core* and several *increments*. The core defines the minimal concepts and APIs necessary for Fractal-based component programming. Increments define additional concepts and APIs which extend the core framework to allow for different forms of component composition, configuration and administration. Most increments are actually Fractal components themselves built with the core framework.

4.1 Core Framework

The core framework defines a set of Java interfaces which map the concepts defined by the Fractal component model into the Java programming language. The base interfaces are `ComponentIdentity`, `Name`, `NamingContext`, `InterfaceReference` and `Type`. A *component identity* unambiguously identifies a component and provides access to its external interfaces (both server and client) and to the component's type. All Fractal components must support this interface. An *interface reference* is a name that references a component interface. It further contains a reference to the component identity interface, the interface identifier, and a property which specifies if the interface is internal or external to the component. An interface reference can reference a remote interface and may be sent over a network. An interface reference is typed and its type is subtype of the union of type `Name` and of the type of the interface it references.

Interfaces `InterfaceType` and `ComponentType` define a simple type system conforming to the Fractal model type system. The `TypeFactory` interface defines type factories which are used to create interface and component types.

The `Template` interface supports the Fractal instantiation mechanism. *Templates* are used to actually instantiate components by *creation* (from a Java class for primitive components) or by *introduction* (from already existing component references). Templates are related to configurations in Architecture Descriptions Languages (ADLs), i.e. un-instantiated component assemblies. Template factories, conforming to the `TemplateFactory` interface are used to create templates.

The core framework further comprises several controller interfaces, that define basic component controller behaviors. Interface `ContentController` provides the ability to manage the content of a component. It allows for addition and removal of components into or from the component's content. This allows a component content and component configurations to change over time. Interface `BindingController` provides the ability to control the bindings between components. Interface `LifeCycleController` provides the ability to manage a component's life-cycle. The Fractal framework only specifies a minimal component life-cycle, which allows for starting and stopping the execution of a component. More complex component life-cycles can be obtained by specializing this basic one. Interface `AttributeController` provides the ability to manage a component's attributes (properties). Fractal component controllers may support none, some or all of these interfaces. Other controller interfaces can be defined, either by specializing the core controller interfaces or by defining new ones, according to the needs of the programmer.

4.2 Framework Increments

The core framework defines a minimal set of concepts and corresponding interfaces and components. The Fractal framework is open to extension by means of framework increments. The current framework includes several such increments, including (template) components and interfaces for bootstrapping and component instantiation.

We plan to add several other increments to deal with component distribution, configuration, protection, mobility, and different forms of component composition and control. Distribution can be added in the framework by leveraging distributed bindings provided by the Jonathan ORB framework [6]. Configuration increments will allow for the instantiation of component configurations described in ADL-like formalisms. Provided they are themselves built from Fractal components, configuration tools implementing the configuration increments would be naturally extensible to supervise, dynamically reconfigure and administrate Fractal systems. Protection increments will deal with basic resource management and protection mechanisms necessary to ensure isolation between components and as a basis for the construction of more advanced security mechanisms. The ability of the Fractal framework to explicitly handle component structures with sharing is crucial in that respect.

Fractal is targeted toward software composition (cf. requirement on composition above). The core framework deals with *structural composition*, where structure can be expressed in terms of *containment relationships* and *bindings* between components. However, some higher level composition semantics are needed to express and assess properties and constraints on components structure. We are currently investigating the use of assertions, synchronous languages and temporal logic for what may be called contractual forms of composition, i.e. forms of component composition that rely on explicit behavioral contracts between components and their environment.

5 The Fractal Framework Prototype

A Java prototype of the Fractal framework has been implemented. This first prototype only provides an instantiation of the core framework and some programming increments to ease component programming, composition and instantiation. The other framework increments cited above (distribution, protection, configuration, etc) have been deferred to future versions. The Java implementation maps operations on Java method calls, and components to various Java object configurations. The prototype increments include: a generic container component, a generic template component, and a bootstrap component. The generic container component facilitates the construction of components conforming to the Fractal model (programmers need only be concerned with the proper functions of the components and Fractal specifics such as the component identity and controller interfaces are generated). The generic template component can be used to build composite templates which contain other template components, which can be bound to “virtual interfaces” corresponding to interfaces of the components they create. The instantiation of a such a template yields a component configuration whose structure at instantiation mirrors that of the composite template. This facilitates the instantiation of complex component configurations. The bootstrap component provides a `TypeFactory` and a `TemplateFactory` interface to create generic containers and generic templates.

5.1 Implementation

The Fractal prototype was implemented so as to minimize the space and time overheads of enclosing components on sub-components, during their normal execution (the performance of reconfiguration phases was not a priority). However, since these overheads can never be completely null, another goal was to provide a range of possibilities between static and dynamic configuration, and thus to provide a way to trade dynamic reconfiguration capabilities against performance.

In order to save memory, each composite component is implemented by a single Java object, and primitive bindings are represented by direct Java references between these objects. The component interfaces are therefore completely implicit, and interface references are very cheap, which saves a lot of memory (compared to an approach were interfaces and interface references are explicitly represented by Java objects). The drawback of this approach is that temporary objects representing interface and interface references must be created each time a method that returns such objects is called, which is not very efficient. But these methods are often called only during reconfiguration phases, and since the performances of these phases was not a priority, this drawback is not really important.

In order to be able to start and stop a component, each container component intercepts incoming method calls, and increments and decrements a counter before and after each call (in two `synchronized` blocks). Indeed, it is then very easy to stop a component: the container just needs to wait for this counter to become null and, atomically, to set a flag so that any new method call will be blocked on a `wait`. But this interception has a cost and, in order to minimize it, we use "shortcut" links so that this cost is paid only once per inter component method call, independently of the nesting level of components.

To support different configuration possibilities, from static to fully dynamic configuration, the prototype supports three different methods to instantiate a given component configuration. The first method just instantiate component configurations from the corresponding Java classes, yielding completely reconfigurable components. The second method uses a single object to implement containers and their encapsulated components: this allows for some method inlining, and therefore speed improvements, but makes it more difficult to dynamically change the container of a component (this approach uses bytecode manipulation tools to merge the container class into the user component classes). Finally the third method yields completely unreconfigurable components by compiling away the interception mechanism of container components (in fact, in this case, there is absolutely *no* time overhead compared to a "manual", static configuration).

5.2 Performances

In order to measure the space and time overheads due to container components, we measured the size of an empty object, and the duration of an empty method call on this object, with or without a container encapsulating this object. The results are given in the table below. The measurements were made on a Pentium III 1GHz, with Linux and the JDK1.3 HotspotVM, for which the size of an empty object is 2 words and an empty method call takes 0.014 μs. The table reports the (constant) overhead added by the interception mechanism on an empty method call with the three different instantiation methods discussed above ("dynamic" refers to the first method, "byte code manipulation" to the second, and "static" refers to the third one). About 40% of the cost is due to the cost of the `synchronized` blocks, and about 25% of the cost can be attributed to the use of the `finally` clause in exception handling. Since this overhead is constant, its cost decreases, in relative terms, with more complex method calls. To have an idea of a more realistic overhead on a working application with fine grained components, we re-engineered the Jonathan ORB with components built according to the first method, thus yielding a fully reconfigurable ORB. The reengineered remote method call, when measured on the same machine between two distinct Java virtual machines (a scenario where network costs are minimized), suffers a mere 7% time overhead compared to the original Jonathan implementation, confirming that our component framework, even with our non-optimized implementation, need not hamper performance.

	Space overhead (words)	Time overhead (μs)
fully dynamic	12	0.11
bytecode manipulation	10	0.085
static	0	0

6 Conclusion

We have presented the Fractal model, a dynamic and recursive component model which allows for sharing of sub-components between components. The Fractal model allows for a clean definition of structural composition based on containment and binding relationships between components. We believe this makes the Fractal model very expressive compared to the different component and system models that underlie ADLs and component-based languages such as C2 [16], Wright [1], Rapide [13], Acme [8], Darwin [15] or Piccola [14]. The only other component model with sharing we know of is Ernie [18]. Ernie is a component composition language that includes syntactic constructs to declare a component as *private* or *shared*. Compared to the Fractal model, however, the degree of dynamicity in Ernie seems to be far less since it does not allow for composite components to change their structure over time. Controlling sharing is an active area of research in object-based programming (see e.g. [5, 17]), where control of sharing is primarily sought by means of type systems. In the Fractal model, controlling sharing is achieved directly, within the model itself, via the control components can exercize on their content. The use of general forms of bindings between components in Fractal, in lieu of more restricted connectors, is similar to that of the OpenCOM model that underlies the OpenORB reflective middleware [4]. OpenORB, however, makes use of pre-defined meta-spaces to deal with certain forms of sharing (resource sharing). The Fractal model seems to provide a more direct and more powerful model for component sharing.

We have also presented the Fractal framework, which is a projection of the Fractal model in the Java programming language, together with a first prototype implementation. The prototype offers three different forms of configuration, from static to dynamic configuration: all binding and containment relationships are reconfigurable in the first case, some containment and binding relationships are not reconfigurable in the second case (which allows for some code inlining), and none are reconfigurable in the third case (full inlining). The first form adds some constant overhead to inter component method calls, which does not depend on the nesting levels of components, and resulting in a 7% in average in realistic applications. For the time being, the decision to create static, dynamic or partially dynamic components has to be made at instantiation time. Additional work is needed toward a true continuum between static and dynamic configuration. Additional work is also needed to more accurately assess and, where necessary, to improve the performance of the implementation of the Fractal framework.

More generally, we plan in the near future to work on the different software increments mentioned in Section 4 so as to turn Fractal into a comprehensive and efficient component-based distributed programming framework.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, vol. 6, no 3*, 1997.

[2] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in hadas. *IEEE Transactions on Software Engineering vol. 27 no 9*, 2001.

[3] G. Blair, F Costa, G. Coulson, F. Delpiano, H. Duran, B. Dumant, F. Horn, N. Parlavantzas, and J.B. Stefani. The Design of a Resource-Aware Reflective Middleware Architecture. In *Proceedings Reflection '99, Saint Malo, France*, 1999.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The Design and Implementation of OpenORB v2. *IEEE Distributed Systems Online, vol. 2 no 6, Special Issue on Reflective Middleware*, 2001.

[5] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *Proceedings ECOOP*, 2001.

[6] 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.

[7] N. Francez and I. Forman. *Interacting Processes : A multiparty approach to coordinated distributed programming*. Addison-Wesley, 1996.

[8] D. Garlan, R. Monroe, and D. Wile. *Acme: Architectural Description of Component-Based Systems*, chapter 3. In [11], 2000.

[9] ITU-T ISO. *ITU-T Recommendation X.902 – ISO/IEC International Standard 10746-2: "ODP Reference Model: Foundations"*. 1995.

[10] ITU-T ISO. *ITU-T Recommendation X.903 – ISO/IEC International Standard 10746-3: "ODP Reference Model: Architecture"*. 1995.

[11] G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[12] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings 18th ACM International Conference on Software Engineering*, 1996.

[13] D. Luckham and J. Vera. An event-based architecture definition Language. *IEEE Transactions on Software Engineering, vol. 21, no 9*, 1995.

[14] M. Lumpe, F. Achermann, and O. Nierstrasz. *A Formal Language for Composition*, chapter 4. In [11], 2000.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings 5th European Software Engineering Conference, Lecture Notes in Computer Science 989, Springer-Verlag*, 1995.

[16] N. Medvidovic, D. Rosenblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st ACM International Conference on Software Engineering*, 1999.

[17] P. Muller and A. Poetzsch-Heffter. *Modular Specification and Verification Techniques for Object-Oriented Software Components*, chapter 7. In [11], 2000.

[18] G. Outhred and J. Potter. A Model for Component Composition with Sharing. In *Proceedings Workshop on Component Oriented Programming (WCOP), ECOOP Workshop Reader*, 1998.

[19] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[20] A. van der Hoek, D. Heimbigner, and A. L. Wolf. Software architecture, configuration management, and configurable distributed systems: A ménage a trois. Technical report, CU-CS-849-98, Department of Computer Science, University of Colorado, Boulder, Colorado, USA, 1998.