

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

CONTRACTS FOR A HIERARCHICAL AND OPEN COMPONENT MODEL

Philippe Collet, Thierry Coupaye, Nicolas Rivierre, Roger Rousseau

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-31-FR

Octobre 2004

Contracts for a Hierarchical and Open Component Model

Philippe Collet

OCL project / I3S laboratory
UMR 6070 CNRS/UNSA
F-06390 Sophia Antipolis
France

philippe.collet@unice.fr

Thierry Coupaye

France Telecom R&D Division
28 Ch. du Vieux Chêne, BP 98
F-38242 Meylan
France

thierry.coupaye@francetelecom.com

Nicolas Rivierre

France Telecom R&D Division
38-40 rue du Général Leclerc
F-92794 Issy les Moulineaux
France

nicolas.rivierre@francetelecom.com

Roger Rousseau

OCL project / I3S laboratory
UMR 6070 CNRS/UNSA
F-06390 Sophia Antipolis
France

roger.rousseau@unice.fr

ABSTRACT

This paper describes the current status of *ConFract*, a contracting system for the *Fractal* hierarchical and open component platform. The *ConFract* system distinguishes different kinds of contracts, constraining both interface bindings and hierarchical compositions. We identify what verifications are currently done in both *Fractal* and the proposed contracting system, their shortcomings, and future work to alleviate them.

Keywords

Contract, Hierarchical Components, Software Architecture, Behavioral Specifications, Runtime Assertions Checking, Interface Contract, Composition Contract, Fractal, ConFract.

1. INTRODUCTION

The development of new specification and verification techniques for component-based systems, as well as the adaptation of existing ones are current challenges for the software community. In this research domain, we consider several requirements as essential in order to make the proposed techniques relevant. For modularity and encapsulation, components should only communicate through contractually specified interfaces and have explicit context dependencies only [30]; required (client) and provided (server) interfaces are then bound to allow communications between components. For homogeneity, the component model should be recursive: components can be built from other components, providing an uniform view of software systems at various levels of abstraction (self-similarity); to factorize components, e.g. for resource management purposes, sharing is then a desirable property. For an effective usage throughout the software lifecycle — including deployment and management of open systems — components have to be runtime entities, i.e. software architecture must exist and be controllable during execution, in terms of reified bindings between components interfaces and containment relations between components. Components platforms should also make dynamic reconfiguration possible. To facilitate extensions,

e.g. to integrate specification formalisms and associated verification techniques, component models should be open.

In this context, different kinds of verification need to be considered. Some are of structural (architectural) nature: they essentially concern binding and containment relationships between components of a system. Some other properties are more of behavioral nature: they are not only concerned with structure and need dedicated specification formalisms and verification techniques. Our long-term goal is to provide a framework general enough for the specification and verification of a large number of functional and non functional properties of component-based systems, while organizing all this in a contractual manner. Indeed we propose to adapt the design by contract principle [21] to software components with the following approach: i) the contracting system does not make any assumption on the functional or non functional nature of the handled properties ; ii) contracts can be defined both on interfaces and on components, taking also into account shared components ; iii) the contracting system builds contracts at configuration time from the specifications: the contract is then a configuration and runtime entity that ensures the properties, using an appropriate verification technique at an appropriate time ; iv) a metamodel reifies specifications so that different formalisms can be integrated in the future; it also associates with each specification construct a responsible component, which is in charge of reacting to a violation, doing dynamic reconfiguration if needed.

Our proposed contracting system, *ConFract*, currently relies on the *Fractal* component model [4, 5], which satisfies the requirements stated above. The *ConFract* model distinguishes three kinds of contract, for interface bindings, compositions and library units (called *base components* in the sequel). Although the *ConFract* and *Fractal* models are not specific to any particular language, the current implementation relies on the Java language and the only supported specification formalism is based on executable assertions.

In this paper, we show what structural verifications are made in the *Fractal* component model and its reference implementation *Julia*, and how their structuring in separate *controllers* makes them extensible. We also describe the *ConFract* model and its implementation in *Fractal* and *Julia* and we show what behavioral verifications can be done at interfaces and composition level. We discuss the current shortcomings regarding specification and verification in *Fractal* and *ConFract*, as well as future research directions. The remainder of this paper is orga-

nized as follows. Section 2 introduces the *Fractal* model and its implementation *Julia*. Section 3 describes the *ConFract* system and section 4 discusses current status and open issues. Related work are presented in section 5 and section 6 briefly concludes this paper.

2. FRACTAL

The *Fractal* initiative¹ aims at supporting component-based development, deployment and management of complex software systems, including in particular operating systems and middleware. The “Fractal ecosystem” includes the *Fractal* component model (detailed hereafter) and several implementations including *Julia* (introduced hereafter), *Kilim* and *ProActive* for Java components, *Think* for C components. It also include several extensions coming from R&D works for management (e.g. *Fractal* JMX), security [16], transactions support [28], auto-adaptation [8]. *Fractal* is also used for developing several middlewares such as *Speedo*, a Java Data Object implementation, *CLIF*, a load injection framework, etc.

2.1 The Fractal Component Model

Fractal (see [4] and [5] for more detailed description) is a general component model with the following main features: composite components (to have a uniform view of applications at various levels of abstraction), shared components (to model resources and resource sharing while maintaining component encapsulation), reflective capabilities (introspection capabilities to monitor a running system and re-configuration capabilities to deploy and dynamically configure a system) and openness (the model is minimal, almost everything is optional and can be extended).

Components, interfaces and bindings

A *Fractal component* is a run-time entity that is encapsulated, and that has a distinct identity. Communication between *Fractal* components is only possible if (some of) their interfaces are bound. An *interface* is an access point to a component (similar to a “port” in other component models), that supports a finite set of operations. Interfaces can be of two kinds: *server interfaces*, which correspond to access points accepting incoming operation invocations, and *client interfaces*, which correspond to access points supporting outgoing operation invocations. *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. A primitive binding is called that way for it can be readily implemented by pointers or direct language references (e.g. Java references, C pointers). 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 (“structurally speaking”) whose role is to mediate communication between other components.

Recursion with sharing, reflection and openness

¹Fractal and associated tools *Fractal* ADL, *Fractal* JMX, *Fractal* RMI, *Fractal* GUI are downloadable from the ObjectWeb open source consortium web site (<http://www.objectweb.org>). *Julia*, *Kilim*, *Think*, *ProActive*, *Speedo*, *CLIF* are downloadable from the same web site.

At the lowest level of control, a *Fractal* component is a black box, that does not provide any introspection or interception capability. Such components, called *base components* are comparable to plain objects in an object-oriented programming language such as Java. Their explicit inclusion in the model facilitates the integration of legacy software. At upper levels of control, a *Fractal* component exposes (part of) its internal structure (see upper part of figure 1). A *Fractal* component is composed of a *membrane*, which provides external interfaces to introspect and reconfigure its internal features (called *control interfaces*), and a *content*, which consists recursively in a finite set of other components (called *sub-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. Shared components are useful, paradoxically, to preserve component encapsulation: there is no need to expose interfaces in higher-level components to allow access to a shared component by a lower-level one. Shared components are useful in particular to faithfully model access to low-level system resources. The 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. Internal interfaces can only exist as “mirrors” of external interfaces: they are used to *export* and *import* interfaces. The membrane of a component is typically composed of controllers and interceptors. Controllers can export cor-

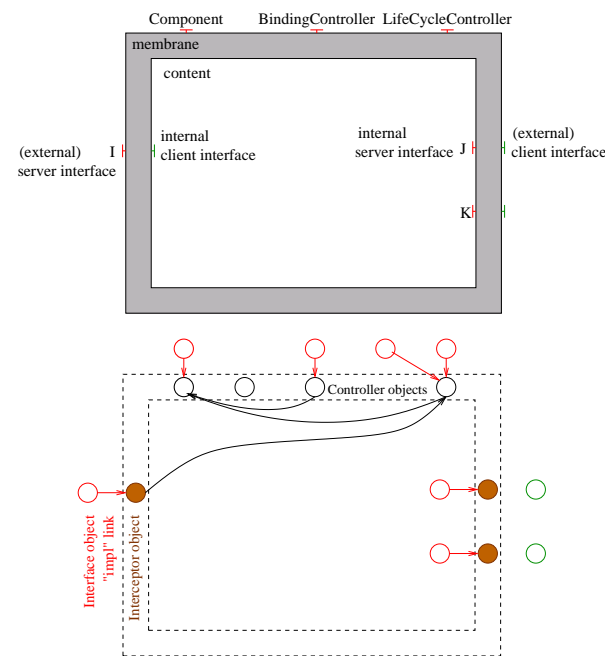


Figure 1: An (abstract) Fractal component and a possible implementation in Julia.

responding control interfaces or not. A membrane can typically 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. It can intercept the incoming and outgoing operation in-

Julia is an execution support for *Fractal* components written in Java. It is a full-fledged implementation of *Fractal* that supports the highest conformance level. In particular and interestingly with respect to contracts on components, Julia supports the *Fractal* basic type system which defines interface types including cardinality and contingency.

More generally, all interfaces defined in the *Fractal* specification are optional, they can be extended and new interfaces can be added at will². The *Fractal* specification, however, contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. The following are examples of controllers. A component can provide an *AttributeController* interface to expose getter and setter methods for its attributes (configurable properties). A component can also provide the *BindingController* interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings. A *ContentController* interface can be provided to list, add and remove subcomponents in the content of a component. A component can provide the *SuperController* interface to list its super component(s) (a shared component has several super components). The *LifeCycleController* interface allows explicit control over the main behavioral phases of a component as a support for dynamic reconfiguration. Basic lifecycle methods supported by a *LifeCycleController* interface include methods to start and stop the execution of the component.

Typing

The *Fractal* model is endowed with a *basic type system*. Note that some components such as base components do not need to adhere to this type system: as control capabilities, typing is optional and arbitrary in *Fractal*. The type of an interface describes its *signature* (the operations its supports), *role* (*client* or *server*), *contingency* (*optional* or *mandatory*) and *cardinality* (*singleton*, *collection*). The operations of a mandatory interface are guaranteed to be available when the component is running. For a client interface, this implies that the interface must be bound. As a consequence, a component with mandatory client interfaces cannot be started until all these interfaces are bound. All other combinations (e.g. an optional client interface not bound, a mandatory server interface of a composite not bound to a sub-component supporting the interface) are not considered as erroneous. The cardinality of an interface type T specifies how many interfaces of type T a given component may have. A singleton cardinality specifies that the component must have exactly one interface of type T. A collection cardinality specifies that the component may have an arbitrary number of interfaces of type T. Such interfaces are typically created lazily upon request of a bind operation (through the *BindingController* interface) and are organized as lists so as to be accessed by position later on. Components types are simply sets of interface types. The minimal type system defines a sub-typing relation which embodies constraints to ensure substitutability of components.

2.2 The Julia Implementation

² *Conformance levels*, not detailed here, defined "levels of control" and can be thought of as "helpers" to grab the openness of the model.

Julia is an execution support for *Fractal* components written in Java. It is a full-fledged implementation of *Fractal* that supports the highest conformance level. In particular and interestingly with respect to contracts on components, Julia supports the *Fractal* basic type system which defines interface types including cardinality and contingency.

More fundamentally, Julia is a software framework dedicated to components membrane programming. It is a small run-time library together with bytecode generators that relies on an AOP-like mechanism based on *mixins* and *interceptors*. A component membrane in Julia is basically a set of controller and interceptors objects (see figure 1). A mixin mechanism based on lexicographical conventions is used to compose controller classes. Julia comes with a library of mixins and interceptors classes the components programmer can compose and/or extend. Julia relies heavily on Java bytecode generation thanks to the ASM bytecode transformation framework³.

3. CONFRACT

The *ConFract* model aims at specifying and verifying some functional or non functional properties of hierarchical components using contracts. In order to match our requirements of dynamicity and openness, a general contracting system must be able to describe expected properties and determine rights and duties of the parties at the most relevant times. Consequently the *ConFract* model distinguishes the description of properties as **specifications** in appropriate formalisms, which play the role of provisions, and **contracts**, which are built in a reified form by the contracting system at configuration time. As a result, contracts take the common sense of "document negotiated between several parties, in which responsibilities are clearly established for each provision".

In *ConFract* different formalisms are intended to be used in the same contract. However the current *ConFract* prototype only uses an executable assertions language, *CCL-J* ("*Component Constraint Language for Java*"), which is partly derived from OCL [22]. Specifications are then classical assertions such as precondition, postcondition, invariant, rely, guarantee, as currently, *CCL-J* is first a validation element of the contracting model.

3.1 Kinds of contracts

In *ConFract*, three kinds of contracts are distinguished in order to constrain both the component interfaces and the hierarchical structure of composite components while taking into account classical contracts:

Interface contracts are established on each binding between a client interface (e.g. *s* of Java type *S* in figure 2) and a server one (*t* of type *T*). Specifications (**spec. of S** and **spec. of T**) can then only refer to entities that are accessible from the interface and its methods and are combined to check hierarchy errors as defined in [11]. For example, the resulting preconditions are checked according to the following formula: $Pre_{cli} \Rightarrow Pre_{serv}$. The interface contract in figure 2 is built from specifications like the following:

³ASM is downloadable from the ObjectWeb web site.

```
pre: m2(a) > b
...
```

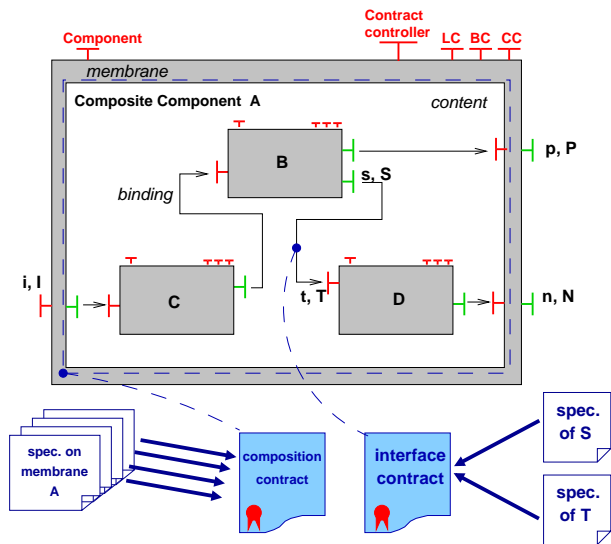


Figure 2: Interface and composition contracts in ConFract.

Composition contracts are associated with the membrane of a component. They are built from specifications that are defined on this membrane (*spec. on membrane A* in figure 2) and refer to entities of several of its external interfaces, and if the component is composite, they can also refer to entities of the external interfaces of its subcomponents. We distinguish two kinds of composition contracts: (i) external composition contracts, which only refer to external interfaces of the component, playing the role of *use contracts*, (ii) internal composition contracts, which refer to at least one interface in the content of a component and play the role of *assembly contracts*. In figure 2, the external composition contract of A would be built from specifications like:

```
on <A> // on <componentName>
  context void i.mI(...)
  post: n.mN() < p.mP()
```

where *i*, *n* and *p* denotes *Fractal* interface of the component A. Similarly, the following specification would be part of the internal composition contract of A:

```
on <A>
  context void i.mI(...)
  post: <D>.t.mT() < p.mP()
```

as it refers to the internal interface *t* of sub-component D.

Library contracts are the contracts that are defined on reusable units of the underlying language. The rest of the paper only discusses interface and composition contracts as they are the original contributions of our work.

3.2 Responsibilities

The *ConFract* metamodel contains description of contracts and specifications, so that the responsibilities of each specification can be clarified among its participants: an unique guarantor, a set of beneficiaries and some possible contributors. If a specification is not satisfied, the guarantor is not considered as guilty, but only responsible of the actions to be triggered. Due to lack of space,

we only describe here the responsibilities of the composition contracts through the *pre* and *post* specification constructs. Referring to figure 2, the responsibilities of the external composition contract of B are given by the following table:

Interface kind	Construct	Guarantor	Beneficiaries
server	pre	A	B
server	post	B	C, A
client	pre	B	D, A
client	post	A	B

For example on the component B, for a postcondition of a method on its server interface, the guarantor is itself as it implements the method and provides the interface, and the beneficiaries are both A, as it surrounds B, and C as it is connected to the interface. As the external composition contract represents a contract of use at the component level, it is for example natural to entrust the responsibilities to the surrounding component A, because it is the component in charge of connecting all subcomponents (B, C and D) together. Conversely, the responsibilities of the internal composition contract of the component A are straightforward. As this contract represents a form of assembly contract between A and its subcomponents, A is both the guarantor and the only beneficiary in all cases.

3.3 The Life Cycle of Contracts

One composition contract is created along with each composite component. From each specification related to this component (see next section), a specification template is then created and attached to the composition contract. Each template is waiting for all its contributing components to become an operational specification. As a new subcomponent is added into the composite, all templates in which it contributes are completed. When all contributors of a template are known, its become a specification. When all specifications of the composition contract are established, the contract is said to be closed, as all beneficiaries and guarantors are identified.

As for an interface contract, there is only two participants in the contract, that is the client and the server. Consequently, the life cycle of this kind of contract is very simple. It is created along the binding between the client and server interfaces, and is automatically closed as all participants are known. Conversely the contract is lifted when the interfaces are unbound.

3.4 Integration in Fractal

All contracts are managed by *Contract controllers* placed on each composite component. Each contract controller is in charge of: (i) the internal composition contract of the composite, which constrains the assembly; (ii) the external composition contract of each of its subcomponents, which constrains their use in the composite; (iii) each interface contract in the content of the composite, which constrains each binding. The Contract controller then manages the life cycle of contracts and their evaluation at the appropriate times.

Contract Construction. In the current implementation of *ConFract*, which is based on *Julia*, the Contract controllers use mixins on existing controllers. As the *Binding*

controllers of subcomponents drive the creation, or destruction, of bindings between subcomponents, a mixin directly creates, resp. destroys, the interface contract along with each binding. A mixin is placed on the *Content controller* of the membrane, in order to be notified of the addition or withdrawal of subcomponents. When a subcomponent is added, resp. removed, the related complete specifications are added, resp. removed, to the internal composition contract and the external composition contract of the subcomponent is created, resp. destroyed. A mixin on the *Life-cycle controller* of the composite ensures that no component is started if its composition contract is not closed. As a result, the contract management is dispatched and stratified along the hierarchy of components. This makes possible to manage hierarchically the actions to trigger in case of violation.

Contract Checking. The specifications of composition contracts that define invariant properties of components are checked at configuration time. Due to the openness of the *Fractal* model, the only time one can be sure that a component is completely configured is just before the *start* method is called on its life-cycle controller. Consequently, the contract controller checks such contracts at this time. As for pre and postconditions of all contracts, they are checked at runtime. When a method is called on a *Fractal* interface, the following contract specifications are then checked before. Preconditions from the interface contract are first checked. As they are created from the client and server specifications, they also check hierarchy errors [11]. Preconditions from the external composition contract are then checked, ensuring the environment of the component is as expected. Preconditions from the internal composition contract are then checked. It should be noted that preconditions from the three different contracts are simply checked sequentially. No specific rule is needed to ensure substitutability as the interface contract already defined it, and that the other preconditions are not sharing the same scope and responsibilities. A similar checking is done with postconditions and method invariants after the call.

4. DISCUSSION

4.1 Current Verifications

Fractal, as expressed by its APIs, basically deals with the *structure* (or *topology*) of software systems based on two mechanisms: *binding* and *containment relationships*. Verifications on these mechanisms include classical type checking based on interfaces signatures (programming language types) - including verifications on constraints on cardinality and contingency when the basic type system is used (cf. section 2). Now, the component-based approach allow for more verifications of constraints on component assemblies such as: "*there must be no cycle in the containment relations*" (a component cannot contain itself directly or indirectly). These verifications are undertaken by the *Fractal* ADL parser and Julia at instantiation time and after a reconfiguration has taken place, i.e. when bindings between components are established or changed.

The *ConFract* system complements the structural verifications done in *Fractal* by checking behavioral properties associated to the two same mechanisms, binding and containment, as shown by the distinction between the different kinds of contract. The interface contract ensures only

properties of the binding between a client interface and a server one. The internal composition contract states and verifies properties of the assembly of components inside a surrounding one, thus being close to an *implementation* contract in object-oriented systems. The external composition contract ensures properties between the external interfaces of a component. It thus plays the role of an *use* contract as expressed properties often link required and provided sides, stating a form of input/output relationship. Composition contracts that state invariants on a component itself defines structural properties that complements the architecture definition and are checkable at configuration time. Other kinds of *CCL-J* specifications lead to runtime checking.

4.2 Open Issues

Structure. *Fractal* specifies that bindings can only exist between sibling components or between components and their direct sub-components (import/export) - which resembles *communication integrity* in ArchJava [1]. The enforcement of this constraint is not guaranteed by current *Fractal* platforms but relies on a fair conformance to the *Fractal* programming model. Due to the openness of the model, this constraint is expensive to enforce and might appear as too coercive. More work is definitely needed - starting with establishing if/when it should be enforced or could be relaxed. In *ConFract*, the interface contract defines what is contractually necessary between the participants of a binding, the client and server interfaces. Similarly, the composition contract is placed on the membrane and manages the containment relationship. Nevertheless, there is no compositional reasoning based on this kind of contract, as no deduction is made. As taking into account non functional properties and resources will need to define compositional models of static and dynamic properties, it should be possible to rely on these to enable some deductions. Finally, as the composition contract respects the boundary of the membrane, it is not possible to organize properties that go through several levels of architecture.

Formalisms. The *ConFract* system currently supports the *CCL-J* language but different formalisms must be integrated to validate the abstraction of the *ConFract* metamodel. First, *CCL-J* must be extended to support more expressive constructs in a compositional context, such as model-based variables or some temporal operators. We also plan to integrate efficient evaluation techniques for assertions that has been previously developed for O-O systems [6]. The integration of other formalisms is likely to make the *ConFract* metamodel evolve, in order to take into account a general category of formalisms. As for specification languages, we intend to adapt QML (QoS modeling language [13]) to the hierarchical nature of *Fractal*, and to investigate formalisms combining static and dynamic verifications, such as regular expressions [26] or temporal logic [29].

Dynamicity. *Fractal* offers basic mechanisms to perform dynamic reconfiguration by means of structure and life-cycle control. However, these mechanisms do not guarantee the consistency of reconfigurations for state capture, and restoration (a.k.a checkpointing) is under definition in the current *Fractal*. This is a complex problem that might require coercion functions when substituting a component by another one with a different state structure or

when substituting a component by several components... Works have started on the subject in the context of the Julia and Think *Fractal* implementations. Using the current version of *ConFract*, it is only possible to rely on all kinds of contract to verify the result of the reconfiguration process, as the specifications of contracts are added or removed along with each reconfiguration step. But when assembling or reconfiguring components with specifications on non functional properties, unsatisfied constraints are likely to be detected in composition contracts and handling such incompatibilities with current mechanisms would be quite cumbersome. We thus intend to provide a negotiation mechanism for contracts so that more autonomy would be available during component assembly while establishing and perserving agreed properties. Work on this negotiation mechanism has already begun, taking existing negotiation protocols in multi-agent systems [12] as starting point.

5. RELATED WORK

Assertion Languages and Contracts. Our work builds on previous works on assertion languages [15] and *design by contract* [21]. In Eiffel [20], the executable assertions are completely integrated in the programming language. However contracts are only an interpretation of the assertions to assign blame to developers and they are not runtime entities. This interpretation, has been recently clarified for class and interface hierarchies [11]. Besides, the runtime checking of *CCL-J* interface contracts in *ConFract* follows these rules. Numerous assertion languages have been proposed for Java and the reader may refer to [27] for an overview. JML (Java Modeling Language) [17] is currently the richest one as it combines executable assertions with some abstract program features, enabling to build an executable model with an abstraction function on the specified class. Assertions have also been integrated in the UML notation with the Object Constraint Language (OCL) [22]. The *CCL-J* language adapts the OCL syntax to Java and extends it to take into account the specification of hierarchical components. *CCL-J* is simpler than JML in terms of available constructs, but its current version should be seen as a minimal but practical set for the validation of executable assertions integrated in the *ConFract* model.

Contracts on Components. Works on programming by contracts have been adapted to components. Contracts on .NET assemblies have been proposed in [3], using AsmL as specification language, but they only concern component interfaces. The composition contract produced by *ConFract* can be compared to collaboration contracts on objects as proposed in [14]. The notion of views in the collaboration are similar to the role played by contract participants in *ConFract*. Nevertheless, *ConFract* makes a component carry the contract to take into account the hierarchy of components. Several works have proposed some forms of contract for UML components. In [25], contracts between service providers and users are formulated, based on abstractions of action and operation behaviour using the pre and postcondition technique. A refinement relation is provided among contracts but they only concerns peer to peer composition in this approach. The forthcoming version 2 of the UML notation [23] supports a form of hierarchical components but contrary to *Fractal*, it focuses more on component connectors than on composite structures. Moreover, version 2 of OCL

does not provide any extension to express compositional constraints. Consequently, it is likely to become quite cumbersome to express *CCL-J* like composition constraints with OCL. A recent work [9] defines UML 2 extensions to describe QoS contracts and to infer end-to-end QoS information from individual component contracts. However, the expressed properties are only attached to single interfaces and their expressiveness is rather poor as they are limited to interval definition.

Formal ADLs. A number of architecture description languages (ADLs) have been proposed for modelling software architectures in terms of components and their overall interconnection structure [19]. Many of these languages support formal notations to specify components and connectors behaviors. For example, Wright [2] and Darwin [7] use CSP-based notations, Rapide [10] uses partially ordered sets of events and supports simulation of reactive architectures. These formalisms allow to verify correctness of component assemblies, checking properties such as deadlock freedom. Some ADLs support implementation issues, typically by generating code to connect component implementation, however most of the work on applying formal verifications to component interactions has focused on design time. A notable exception is the SOFA component model and its behavior protocol formalism [26], based on regular-like expressions, that permit the designer to verify the adherence of a component specification at runtime. In the same line, the work presented here targets both the design of component models equipped with contracts as well as the runtime checking of contracts on actual components in execution. The assertion-based language *CCL-J* currently used is less powerful than process algebra or temporal logic but is expressive and executable. *ConFract* is likely to be integrated with other formalisms.

Software Architecture in Java. Two recent proposals for Java-based component programming include Jiazzi [18] and ArchJava [1]. ArchJava integrates component structure and implementation in one language. This ensures good traceability between architecture and Java code but also restricts architecture management and dynamic evolution since architecture and implementation are strongly-coupled. ArchJava also guarantees a property called *communication integrity* during execution. Intuitively, it means that a component instance A may not call the methods of another component instance B unless B is subcomponent of A, or A and B are sibling sub-components of a common component instance that declares a connection or connection pattern between them. Communication integrity and *Fractal* architectural constraints have been discussed in Section 4.2. Unlike ArchJava and Jiazzi, *Fractal* does not rely on language extensions of the Java language.

6. CONCLUSION

This paper described the current status of *ConFract*, a contracting system for the *Fractal* hierarchical and open component platform. The *ConFract* system distinguishes different kinds of contract, constraining both interface bindings and hierarchical compositions. We identified what verifications are currently done in both *Fractal* and the proposed contracting system, their shortcomings, and future works to alleviate them. We plan to integrate new specification formalisms in the *ConFract* system, to provide compositional model to take into account non functional properties and resources management in *Fractal*,

Aknowledgements

This work has been partially funded by France Telecom under the collaboration contract n°422721832I3S. The authors wish to thank E. Bruneton, R. Lenglet and A. Ozanne for their participation in this work.

7. REFERENCES

- [1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *16th ECOOP*, 2002.
- [2] R. J. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Soft. Eng. and Methodology*, 6, July 1997.
- [3] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.
- [4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *ICSE 2004 - CBSE7*, volume 3054 of *LNCS*. Springer Verlag, May 2004.
- [5] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model. Specification, Technical Report v1, v2, The ObjectWeb Consortium, 2002,2003. <http://fractal.objectweb.org>.
- [6] P. Collet and R. Rousseau. Efficient implementation techniques for advanced assertion languages. *L'objet*, 5(3-4):417–442, Dec. 1999.
- [7] J. K. D. Giannakopoulou and S. Cheung. Analysing the behaviour of distributed systems using Tracta. *Journal of Automated Soft. Eng., special issue on Automated Analysis of Software*, 6(1), Jan. 1999.
- [8] P.-C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In *DAIS 2003*, volume 2893 of *LNCS*, Paris, Nov. 2003. Springer Verlag.
- [9] O. Defour, J.-M. Jézéquel, and N. Plouzeau. Extra-functional contract support in components. In *Intern. Symposium on Component-based Software Engineering (CBSE7)*, May 2004.
- [10] D. C. L. et al. Specification and analysis of system architecture using Rapide. *IEEE Trans. on Soft. Eng.*, 24(4):336–355, Apr. 1995.
- [11] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of OOPSLA '2001*, 2001.
- [12] FIPA TC Communication. Fipa contract net interaction protocol specification. Technical report, FIPA organization, 2002.
- [13] S. Frølund and J. Koistinen. Quality of service in distributed object systems design. In *4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, Apr. 1998.
- [14] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In N. Meyrowitz, editor, *OOPSLA/ECOOP'90*, pages 169–180, Ottawa, Canada, Oct. 1990.
- [15] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.
- [16] T. Jarboui, J.-P. Fassino, and M. Lacoste. Applying components to access control design: Towards a framework for OS kernels. In *DSN 2004*, Florence, July 2004.
- [17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
- [18] S. McDirmid, . Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *OOPSLA 2001*, *ACM Press*, 2001.
- [19] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. on Soft. Eng.*, 26(1), 2000.
- [20] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
- [21] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, Oct. 1992.
- [22] I. Object Management Group. Object constraint language specification. Technical Report version 1.1, ad/97-08-08, IBM www.software.ibm.com/ad/oc1, Sept. 1997.
- [23] OMG. UML 2 OCL final adopted specification. Technical Report ptc/03-10-14, Object Management Group, Oct. 2003.
- [24] OMG. UML 2 superstructure final adopted specification. Technical Report ptc/03-08-02, Object Management Group, Aug. 2003.
- [25] C. Pahl. Components, contracts and connectors for the unified modelling language UML. In S. Verlag, editor, *FME2001 - Formal Methods Europe*, volume 2021 of *LNCS*, pages 259–277, 2001.
- [26] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. on Soft. Eng.*, 28, Nov. 2002.
- [27] R. Plösch. Evaluation of assertion support for the Java programming language. In *Journal of Object Technology, Special issue: TOOLS USA 2002 proceedings*, volume 1,3, pages 5–17, 2002.
- [28] M. Prochazka. Jironde: a flexible framework for to make components transactional. In *DAIS 2003*, volume 2893 of *LNCS*, Paris, Nov. 2003. Springer Verlag.
- [29] N. Rivierre and T. Coupaye. Observing component behaviours with temporal logic. In *ECOOP Intern. Workshop on Correctness of Model-based Software Composition (CMC'03, ECOOP 2003)*, Darmstadt, July 2003.
- [30] C. Szyperski and C. Pfister. Report of the intern. workshop on component-oriented programming (wcop'96/ecoop'96). University of Linz, 1996.