

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

CONTRACT-BASED TESTING: FROM OBJECTS TO COMPONENTS

Philippe Collet, Daniel Deveaux, Roger Rousseau, Yves Le Traon

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-30-FR

Octobre 2004

Contract-based Testing: from Objects to Components

Philippe Collet
Assistant Professor
I3S Laboratory - CNRS
Univ. of Nice S.A.
Nice - FRANCE
philippe.collet@unice.fr

Daniel Deveaux
Assistant Professor
Lab. Valoria
Univ. of South Brittany
Vannes - FRANCE
daniel.deveaux@univ-ubs.fr

Roger Rousseau
Assistant Professor
I3S Laboratory - CNRS
Univ. of Nice S.A.
Nice - FRANCE
rr@unice.fr

Yves Le Traon
Assistant Professor
Triskell Project - IRISA
Rennes-1 University
Rennes - FRANCE
yletraon@irisa.fr

ABSTRACT

Contracts on object-oriented classes have been first developed as a software design approach. They were also quickly used for supporting class testing, providing a form of design for testability. In this paper, we identify the tracks to extend the contract-based built-in test technique to hierarchical components. To do that, we build on our previous work on STclass, a framework supporting Design by Contract and built-in test for Java, and on ConFract, a contracting system for the Fractal component platform. Tests are embedded in the components and are generated with respect to a category of contract (library, interface, composition). In this paper, we study how this approach, firstly dedicated to objects, can be valuable for components testability. As a result of the test process, the embedded contracts are more robust and offer an original way to improve the observability of the component-based system. Contracts make it aware of its execution, and thus able to detect erroneous behaviors at runtime.

Keywords

Design by contract, built-in test, hierarchical components, design for testability, design for trustability, STclass, ConFract, Fractal.

1 Introduction

Since 1987, B. Meyer [20] proposed a software design approach based on runtime checked contracts embedded in the source code. The first role of the contracts associated with the methods definition is to provide a semi-formal documentation of the specification. They were also quickly used for supporting class testing. The preconditions define the test field limits

and checking the postconditions make it possible to evaluate the methods good execution. Supported by our past experience in using and teaching the Eiffel language [7, 6, 5, 12, 11], we were convinced of the contracts effectiveness in code documentation and errors detection; we then have conducted several research to extend and improve their application.

For a few years, Valoria laboratory at UBS and Triskell project at IRISA have developed for object oriented languages a built-in test technique based on the "Design by Contract" approach. It works fine and a test framework, comparable to Junit [2], was developed for the Java language and is distributed under GPL license (STclass-Java¹). This technique is also supported by a classes development and improvement process (called "Design for Trustability") and by another technique to control the quality of contracts and tests by "mutation testing" [13].

However, its broad scale use in the software industry is limited, as it is restricted to functional contracts, assertion languages are not expressive (lack of efficiently checkable quantifications) and precise (lack of precision in intention and importance of assertions) enough. Moreover, the support in current programming languages is weak.

Emerging models for designing component-based system (UML 2.0, Fractal) introduce new key-challenges for obtaining testable systems. Indeed, in the component-based approach, components offer mechanisms that may allow to express rigorously and precisely interface protocols, and embed specification artifacts (documentation etc.).

The use of the software components makes it possible to consider the contract-based test with a new point of view, not only for objects but also for component. Indeed, the contract and the clear definition of the bindings between components is extremely relevant to the software component concept. Indeed, the contract and the clear definition of the bindings between components is extremely relevant to the software component concept, as complexity is shifted

I3S Research report N° ISRN I3S/RR-2004-30-FR, published in 1st International Workshop on Testability Assessment (IWoTA) in ISSRE 2004.

¹<http://www.stclass.org/>

from the entire system to these binding points. Moreover, hierarchical components make it possible to define new kinds of contracts between a composite components and its content. New problems also appear as required and provided properties can only be compared during components assembly, when the contract is built, and contracts must be negotiated or dynamically renegotiated between the components as they must take into account non-functional properties.

The I3S laboratory, in collaboration with France-Telecom-R&D, developed a model supporting contracts for software components; the *ConFract* system, which implements this model, is based on the *Fractal* hierarchical component model². In *ConFract*, two new kinds of contracts are defined. An interface contract is located on each binding between a required and a provided interface of two components. A composition contract is located on each component in order to manage constraints on several of its interfaces or with its subcomponents.

In order to provide a "Design for Testability" approach for hierarchical components, it is from now on possible to consider the definition of a contract-based built-in test environment for the software components that reuses and extends our knowledge on the class level. By comparing the object vs. component paradigms, the main contribution of this paper is thus three-fold :

- definition of new categories of contract specific to component-based software,
- introduction of the notions of built-in tests / self-testability at component level,
- use of the triangle view of a component (see Figure 3), as well as a test process, to make the component-based software more observable at runtime, thanks to the use of executable contracts as embedded oracle functions.

The remainder of this paper is organized as follows. Section 2 describes the built-in test technique based on contracts on classes, *STclass*. In section 3, the *ConFract* system is presented. Section 4 identifies the tracks to extend the built-in test technique to hierarchical components. Section 5 briefly concludes this paper.

2 Applying contracts based builtin-test at class level

In a recent paper [24], Dave Thomas complains on "*the deplorable state of class libraries*" and explains that the classes must be specified, designed, implemented and tested using best practices. He suggests to use different techniques such as Design by Contract [21], interface specifications and test driven development. To support the design of such libraries, we

propose to consider a class file as a *class document* that contains all the necessary information: standard code and technical comments, but also links to design documents, contracts and test material. This is the "*Self-Testable Class*" concept that we have defined in 1999 and is distributed under the name of *STclass*³. This development support is a part of a so-called "*Design for Trustability*" (*DfT*) [10] process, a pragmatic approach that gives a partial but immediate answer to the kind of questions presented above.

Self-Testable classes in Java – Because there is no standard support for contracts in java, we have chosen, like several authors, to embed the contracts in class and methods javadoc comments (see [18] and [23] for a comparison of different contract supports for java and also the Java Modeling Language proposal [19]). We have also chosen to embed test units into java comments (this is an original feature). In this way the classes remain independent from any specific tool, although they include contracts and tests: it is still possible to compile the classes with any standard java compiler. Nevertheless, if we want contracts to be verified at runtime, we need a preprocessor that will extract the contracts and convert them into java code. The *javacst* preprocessor generates an instrumented class, which in turn may be compiled with any standard java compiler. The test units (also written as java comments) are recognized by the preprocessor which transforms them into java methods. Additionally, the preprocessor generates a *main()* method so as to make the class *really* self-testable: running the class will launch its test units.

How to write contracts – As in R. Kramer's *ICContract* [17], pre- and postconditions are defined with javadoc *@pre* and *@post* tags; a third tag, *@invariant*, can be used in class javadoc comment to define a class invariant. The three tags have the same syntax: an assertion in java syntax and an explanatory comment, as in the next example.

```
@post size() == size()@pre+1 // size increased
```

The *@pre* tag used after an expression (as in the above example) has the same semantic as in UML OCL : it implies the evaluation of postfix expression *before* the method execution. As in OCL, we have also added an *implies* operator and set expressions (*forall* and *exists*) that improve the contracts expressiveness.

How to write test units – As in Junit proposed by Beck and Gamma [2], tests are organized in small test-units and a testing environment is provided by a library (*STclass.jar*). Test units are written into a simple comment at the end of the class file, using some more *STclass*-specific tags as shown on figure 1.

Three *STclass*-specific tags are used for writing test

²<http://fractal.objectweb.org>

³<http://www.stclass.org/>

```

/* Test definition
 * @tcreate SetOfIntegers()
 * @tunit TST\_add1() : adding elements
 * Add two elements to an empty set
 * @tunitcode {
 * Integer one, two; // elements used for this unit
 * one = new Integer(1); two = new Integer(2);
 *
 * add(one);
 * testMsg ("Integer '1' added");
 * add(two);
 * testCheck("string image equals to {1,2}" +
 *           toString().equals("{1,2}"));
 * }
 * ...

```

Figure 1: A *test-unit Example*

units: the `@tcreate` explains how to build the object under test; each test-unit is structured with `@tunit` and `@tunitcode` tags. Test units are expanded as regular methods: the code inside test units is standard java code and the object under test methods are called as local methods. Some useful routines are available to manage messages (`testMsg()`) and to define "oracles" (`testCheck()`). In the previous example, very few "oracle checking" are made during test; indeed, contract checking is activated and post-conditions are checked at end of each method execution: it is not necessary to repeat these checks by an explicit oracle in the test unit. Also most test units contain only simple method calls: if they are complete, the contracts provide all the necessary oracles.

To launch the test, one only has to launch the instrumented class itself. Test support provides many options, especially statistics on methods calls to evaluate test coverage at method call level, and the possibility to launch only one or several test units.

Contracts, tests and inheritance – Contracts have an interesting feature: they make tests very easy to write! Once the protocol has been specified in terms of contracts, testing it is equivalent to using it with contracts activated. This approach is not only for individual class testing. For integration tests, activate contracts for the component to be integrated, place the component in its environment, and use the service. A contract violation may have several meanings: either a fault remains in the protocol itself (erroneous contract), or there is a fault in the implementation (post-condition violation means that a bug has to be fixed by the provider of the class), or the client of the protocol is not conforming to the preconditions. Anyway the interesting feature of the *STclass* environment is its ability to underline the fault responsibility: you instantly get the fault location, the reason why the fault happened, and the appropriate action to be taken. This saves a lot of time while improving the trustability of components.

The *STclass* environment for java includes support for contracts and test inheritance, which again will help class designers and developers. Defining a java interface is the same as defining a protocol between a service provider (implementation) and a service client; contracts and tests come to enrich this interface. They will be automatically inherited within implementations, which ensures that implementations are fully compliant with the protocol. The same goes for class inheritance: the contracts that are defined in the super class are inherited by its children. This ensures coherency within the class hierarchy and avoids contract duplication. With the test inheritance, an interface can provide a validation test for all its implementations: the developer activity is improved in two ways, a saving of time and a guarantee of implementation validity.

The *DfT* and the *STclass* environment give a real support to improve the trustability of object-oriented software. Simplicity and usability have been emphasised, program sources remain compatible with standard compilers and JVM: the environment is well accepted by programmers that dispose of a structured framework to create and maintain classes and packages. The pragmatic way that we have used is not opposite to formal and model transformation approaches that are under research; it has the great advantage to give immediately a usable result and to prepare current programmers to more structured working methods.

The tools that support *DfT* are freely available under GPL license and are now usable for real life developments. In the frame of SCoT project [9], we are carrying out an experiment on a 100 classes middleware library.

3 Contracts definition and evaluation in a component model

The Fractal component platform – ConFract [8] is a contracting model that is based on the *Fractal* component platform [4, 14]. This open platform supports the hierarchical structuring of components (a component can be made of other components), which is now necessary to organize complex systems. It also provides metamanipulation facilities in order to manage message interceptions between components, introspection of relevant runtime entities, dynamic assemblies and negotiations.

A *Fractal* component is thus a runtime entity that is composed of (i) a content, which can be simple or composite, and that is delimited by a sort of membrane (ii) a controller, which can intercept all messages passing through the membrane, and (iii) internal and external interfaces located on the membrane. Interfaces can be required (client) or provided (server), single or collection-based, mandatory or optional. The internal structure of a composite compo-

ment is hierarchic with some sharing possibilities.

Interfaces are described using an IDL, which is derived from Java, and a static type system enables to compare interfaces or components according to their compatibility. The encapsulation of components is respected as only external interfaces are accessible from outside of a component and its internal interfaces are only visible from internal components of the first level. During assembly, all required and mandatory interfaces of each component must be bound, in a type-safe way, to some provided interfaces.

The controller part of a component is organized in several open service interfaces, for naming, binding, life cycle or content management. The *Fractal* platform is reflexive and bootstrapped, so that instantiation facilities are provided through specific components and introspection is possible on any entity. Using all these facilities, each component can dynamically control types, assemble components, intercept messages, introspect components, stop, replace or activate its sub-components, etc.

The current reference implementation of *Fractal*, named *Julia*, is based on the Java language. The assembly and configuration of components can be done through an XML-based ADL, but also entirely dynamically using the *Julia* API.

From a structural point of view, *Fractal* is close to the ArchJava system [15], but this system uses language integration to enhance Java with constructs to handle hierarchical components, whereas *Fractal* components are defined using an dedicated API or through an ADL, and *Fractal* makes it possible to manage components while separating concerns with controller objects.

ConFract's essential – The *ConFract* model aims at specifying and verifying some functional or non functional properties of components using contracts. Although it is dedicated to the *Fractal* platform, the model can be applied to similar platforms that provide required or provided interfaces, supporting hierarchical decomposition or not. The current *ConFract* prototype is bound to the *Julia* implementation and the Java language is used to describe component's interfaces and implementations.

Verifying some component properties may use varied formalisms and techniques, concerning functional or temporal aspects, as well as quality of services. Some checking can be done before component assembly like static type checking, proof or some certification testing. But admission testing is still needed as all properties cannot be known in advance. *ConFract* thus uses testing as its main verification technique.

In this context, a general contracting system must be able to use different formalisms to describe ex-

pected properties and determine rights and duties of the parties at the most relevant times. That is why the *ConFract* model clearly distinguishes the description of properties as **specifications** in appropriate formalisms and **contracts**, which are built in a reified form by the contracting system when required and provided parties are compared. As a result, contracts take the common sense of "*document negotiated between several parties, in which responsibilities are clearly established for each provision*". Provisions of a same contract can be expressed in different formalisms according to needs and checking capabilities.

However the current *ConFract* prototype only uses one specification formalism, *CCL-J* ("*Component Constraint Language in Java*"), which is partly derived from OCL [22]. Specifications are then quantified assertions organized in clauses such as preconditions, postconditions, invariants, etc. Although its specifications are described separately from the components, *CCL-J* is very close from the *STclass* language, with the addition of constructs dedicated to hierarchical components.

Kinds of contracts – The comparison of required and provided properties and some responsibility rules make it possible to distinguish different kinds of contracts (cf. figure 2):

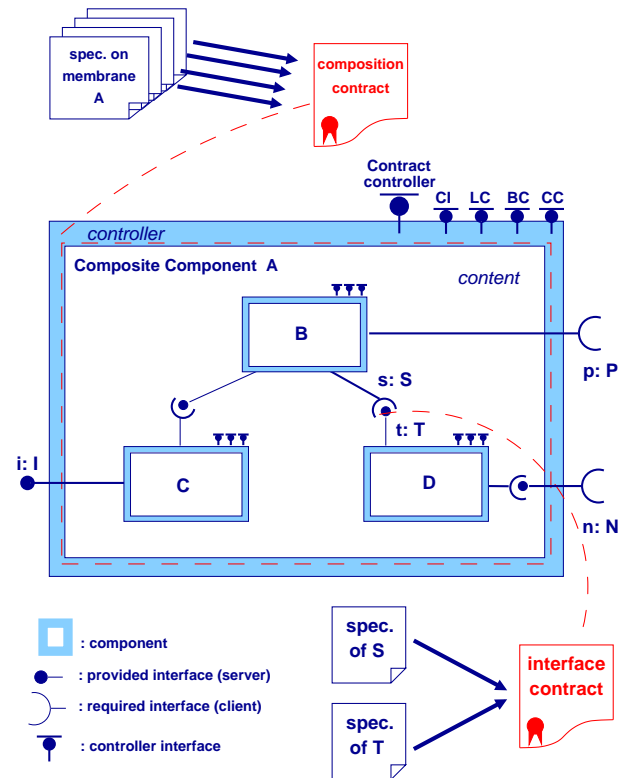


Figure 2: Interface and composition contracts in *ConFract*.

Interface contracts are established on each binding

between a required interface (e.g. s of Java type S on figure 2) and a provided 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.

Composition contracts are associated to the membrane of a component. They are built from specifications that are defined on the membrane ($spec.$ on membrane A on 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, and (ii) internal composition contracts, which refer to at least one internal interface.

Library contracts are the contracts that are defined at level of object classes and interfaces. Contracts that are developed with the *STclass* environment are examples of such library contracts.

The responsibilities of the contract guarantor differs according to the kind of contract. For example, the guarantor of a composition contract is the composite component that assembles the components. If a provision is not satisfied, the guarantor is not *guilty*, but only responsible of the actions to be set about in order to solve the problem, by negotiation, replacement of a subcomponent, etc.

Life Cycle of the Contract – A contract is incrementally built. When it is complete, i.e. all its parties are identified and all provisions are complete by the identification of all responsibilities, the contract is said to be **closed**. A closed contract is never modified, but it can be terminated and replaced by another one. During its construction, a contract is then open. Two contracts can be compared according to their substitutability. The construction of a contract can be done automatically when the compatibility of the required and provided specifications can be determined. If not, a negotiation is started ; the design of this process is currently an undergoing work.

The *ConFract* metamodel contains description of contracts and specifications, so that the responsibilities of each provision of a closed contract can be clarified among its participants: an unique guarantor, a set of beneficiaries and some possible contributors. For example, if a precondition of a method m provided by a component A states " $B.p() < C.max$ ", the guarantor is the component D that calls m , the beneficiary is A , and B and C are contributors. D must check that precondition is satisfied before calling m ; if not, a possible scenario is that D asks the contributors if they can provide results that satisfy the condition by decreasing p or increasing max .

4 Contract-based built-in tests for software components

Testing contracted components: the triangle view

The methodology is based on an integrated design and test approach that has been first proposed for OO software components. It is particularly adapted to a design-by-contract approach, where the specification is systematically derived into executable assertions (invariant properties, pre/postconditions of methods). Test suites are defined as being an "organic" part of software OO component. Indeed, a component is composed of its specification (documentation, methods signature, invariant properties, pre/postconditions), one implementation and the test cases needed for testing it. This view of an OO component is illustrated under the triangle representation (cf. Figure 3). To a component specified functionality is added a new feature that enables it to test itself: the component is made self-testable, in a way that is analogous to hardware components. Self-testable components have the ability to launch their own tests as detailed in [16].

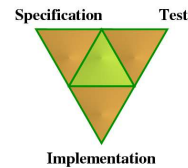


Figure 3: *The triangle view of component*

From a methodological point of view, we argue that the trust we have in a component depends on the consistency between the specification (refined in executable contracts), the implementation and the test cases. The confrontation between these three facets leads to the improvement of each one. Before definitely embedding a test suite, the efficiency of test cases must be checked and estimated against implementation and specification, especially contracts. Tests are built from the specification of the component. They are composed from two independent conceptual parts: test cases and oracles. Test cases execute the functions of the component. Oracles - *predicates for the fault detection verdict* - can either be provided by assertions included into the test cases or by executable contracts. In a design-by-contract approach, our experience is that most of the verdicts are provided by the contracts that are derived from the specification. The fact that contracts of the components are inefficient to detect a fault exercised by the test cases reveals a lack of precision in the specification. The specification should be refined and new contracts added. The trust in the component is thus related to the test cases efficiency and the contracts "*completeness*". We can trust the implementation

since we have tested it with a good test cases set, and we trust the specification because it is precise enough to derive efficient contracts as oracle functions.

Applying the triangle approach to *ConFract*

This section describes how the self-testable approach can be applied to a component model enhanced with executable contracts expressed within the *ConFract* model. We address the issue of how to test a component and an assembly, using its contracts as oracles. To simplify the terminology, we consider that "*testing contracts*" refers to the task of testing the component implementation with respect to a particular facet of its specification expressed either by library, interface or composition contracts. The questions we tackle are three-fold and may be expressed as follows in our specific context: what, when and how to test.

What to test – Depending on the kind of properties that will be tested on a component (or an assembly), test cases are built in function of the kind of contracts:

Library contracts: The facet of a component which relates to library contracts can be tested using the *STclass* approach,

Interface contracts: The facet of a component which relates to closed contracts is tested using the *STclass* approach. For opened contracts, some interfaces of the component under test are not bound to the services of the required component. In this case, interface stubs must be introduced for required interfaces. Provided interfaces are self-testable, using the *STclass* approach.

Composition contracts: This is the most difficult part, since composition contracts use the dependencies between components/sub-components. The *STclass* approach cannot be directly applied since the test must focus on these dependencies. Testing composition contracts is close to integration testing, and the selftests associated to a composition must implement a test protocol. Testing external composition contracts is thus related to integration, and selftests have to be based on the component hierarchy, that has to be browsed. A future work may consist of extracting this hierarchy and then building automatically the test protocol. Component interactions are exercised incrementally: a basic strategy consists of exercising recursively the components hierarchy from the leaves to the top (the assembly component), as for a classical bottom-up integration. More complex strategies may be used [3]. Components and their related contracts are thus exercised from the simple ones to the most complex (the assembly) by test cases, each test case being dedicated to each of the various assembly contexts. As for contracts, the test cases are associated to component membranes. The issue of

opened contracts has to be taken into account by creating stubs for non available components.

When to test – Contracts and testing are meaningful before assembling, during the integration of components into an assembly and during the execution (at runtime). Before assembling, components are unitary tested. Then the test protocol is built for a chosen assembly, and the test cases are launched with composition contracts as oracle function. During these two stages, both library/interface and assembly contracts are checked and may be improved. Indeed, if the contracts used as embedded oracles fail in detecting a faulty state, it means that they must be improved. So, this is not only the components implementation and assembly that is improved by the test process but also the capability of the contracts to detect a faulty state at component's execution. This leads us to the third stage, that is the use of contracts at runtime as an automatic mean for controlling that the system behaves correctly. At this stage, we benefit from the previous stages which made contracts more robust [1] and able to detect internal faulty states or external misuses. So, the system is made "*aware*" of its execution and has a reasonable probability to detect errors before the failure occurs. A new issue has to be addressed that is the strategy to activate only the contracts that are needed at a given time, since it is not realistic to check all the contracts at runtime without a computational expense. Moreover, at this stage, the relevant properties are most often non functional (resources consumption, quality of services), and specific formalisms and contracts may be designed for these non functional properties. However, it is no more a "*testing stage*", since we do not execute test cases on-line, but a way to make the system more controllable/observable at runtime.

How to test – The issue we address under the question of how to test a component and an assembly is the one of the test adequacy criterion. What is the good criterion to guide the test, that is adapted to each of the contracts facet of a component-based system. Indeed, the trust we will obtain in such a system will depend on the relevance of the chosen criteria. Depending on the testing stage, the test criterion may vary. The question is thus to be able to estimate the consistency of a component/assembly based on the triangle view. This quality estimate quantifies the trust one can have in a component. The chosen quality criteria proposed by Triskell/Valoria teams is the proportion of injected faults the self-test detects when faults are systematically injected into the component implementation. This estimate is, in fact, derived from the mutation testing technique. However, the main classical limitation for mutation analysis is the combinatory expense, and makes it difficult to apply directly on a complex system.

5 Conclusion

In this paper, we have identified the tracks to extend the contract-based built-in test technique to hierarchical components in order to improve components testability. We have built on our previous work on *STclass*, a framework supporting Design by Contract and built-in test for Java, and on *ConFract*, a contracting system for the *Fractal* component platform. In our proposal, tests are embedded in the components and are generated with respect to a category of contract (library, interface, composition). As a result of the test process, the embedded contracts are more robust and offer an original way to improve the observability of the component-based system. Contracts make it aware of its execution, and thus able to detect erroneous behaviors at runtime.

REFERENCES

- [1] B. Baudry, Y. LeTraon, and J.-M. Jézéquel. Robustness and diagnosability of oo systems designed by contracts. In *Proceedings of Metrics'01*, London, UK, April 2001.
- [2] K. Beck and E. Gamma. Test-infected: Programmers love writing tests. *Java Report*, pages 37–50, July 1998.
- [3] L.C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(6), 2003.
- [4] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal component model. Specification Draft 2.0-2, The ObjectWeb Consortium, September 2003.
- [5] Philippe Collet and Roger Rousseau. Classification et réification des assertions — application au langage Eiffel. In *3ème Conférence intern. sur les Langages et Modèles à Objets (LMO'96)*, pages 10–27. Leysin (Suisse), October 1996.
- [6] Philippe Collet and Roger Rousseau. Efficient implementation techniques for advanced assertion languages. *L'objet*, 5(3-4):417–442, Déc. 1999.
- [7] Philippe Collet and Roger Rousseau. Contrôle d'admission de composants avec des contrats comportementaux. In *LMO'2003 (Langages et Modèles à Objets)*, pages 31–44. Hermes Science Publications, *RSTI L'objet*, volume 9, numéro 1-2/2003, January 2003.
- [8] Philippe Collet and Roger Rousseau. Contracting hierarchical components. Technical report, I3S Laboratory - Sophia Antipolis, March 2004.
- [9] D. Deveaux. Self-Testable Classes Requirements. Technical report, SCoT Project: "ITR Bretagne Program", March 2002. Published in the STclass-java distribution.
- [10] Daniel Deveaux. The design for trustability approach. In *Trusted Components Workshop*, Prato Italy, 9 January 2003. ETH Zürich, Monash University, TOOLS Conferences.
- [11] Daniel Deveaux, Régis Fleurquin, and Patrice Frison. Software Engineering Teaching: a 'Docware' Approach. In ACM, editor, *ITiCSE'99*, Cracow, June 1999. ACM - ITiCSE'99 Symposium.
- [12] Daniel Deveaux and Patrice Frison. Teaching software engineering working methods : an experimental environment. In *ISSEU97*, March 1997.
- [13] Daniel Deveaux and Yves Le Traon. XML to Manage Source Code Engineering in Object-Oriented Development: an Example. In Cecilia Mascolo, Wolfgang Emmerich, and Anthony Finkelstein, editors, *XML Technologies and Software Engineering*, pages 28–31, Toronto, Canada, May 2001. XSE01 workshop at ICSE'2001.
- [14] France-Telecom-R&D. Fractal web site, 2002. <http://fractal.objectweb.org>.
- [15] C.Chambers J.Aldrich and D.Notkin. Architectural reasoning with ArchJava. In *ECOOOP'2002*. Malaga, Spain, June 2002.
- [16] Jean-Marc Jézéquel, Daniel Deveaux, and Yves Le Traon. Reliable Objects: Lightweight Testing for OO Languages. *IEEE-Software*, 18(4):76–83, jul-aug 2001.
- [17] Reto Kramer. iContract – the Java(tm) design by contract(tm) tool. In *26th Conference on Technology of Object-Oriented Systems (TOOLS USA '98)*, August 1998.
- [18] Martin Lackner, Andreas Krall, and Franz Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76, 2002. Special issue: TOOLS USA 2002 proceedings.
- [19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. <http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html>.
- [20] Bertrand Meyer. Programming as contracting. Report tr-ei-12/co, Interactive Software Engineering, 1987.
- [21] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
- [22] Inc Object Management Group. Object constraint language specification. Technical Report version 1.1, ad/97-08-08, IBM www.software.ibm.com/ad/ocl, September 1997.
- [23] Reinhold Plösch. Evaluation of assertion support for the Java programming language. *Journal of Object Technology*, 1(3):5–17, 2002. Special issue: TOOLS USA 2002 proceedings.
- [24] Dave Thomas. The deplorable state of class libraries. *Journal of Object Technology*, 1(1):21–27, May 2002.