On Contract Monitoring for the Verification of Component-Based Systems

Philippe Collet
Objects and Software Components Group

Laboratoire I3S - CNRS - Université de Nice - Sophia Antipolis
Les Algorithmes- Bât. Euclide B, 2000 route des Lucioles
BP 121, F-06903 Sophia Antipolis Cedex, France

Philippe.Collet@unice.fr

ABSTRACT

This position paper focuses on contract monitoring for component interfaces, considering the verification of functional and non-functional properties in the contracts. We investigate what properties are needed on behavioral and *Quality of Service* contracts. We also define what are the requirements on a monitoring environment to handle properly those contracts. We briefly transpose those requirements to a meta-level architecture.

1. INTRODUCTION

The development of component-based systems intends to deliver the beneficial effects that the object-oriented approach failed to completely provide: reuse of out-sourced pieces of software and thus increased productivity. The definition of component devised during the 1996 Workshop on Component-Oriented Programming [1] is the following: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

In this definition, the specification process is clearly related to *contractually specified interfaces*. This position paper focuses on the general notion of contract for components, that is a component must expose functionalities, through its functional contract, and its *performances*, using some non-functional contract. More precisely, Beugnard et al [3] categorize contracts in four levels:

1. Syntactic contracts, that is signatures of data types.
2. Behavioral contracts, that is some semantic description of data types,
3. Synchronization contracts, which deal with concurrency issues.
4. Quality of Service (QoS) contracts, which encompass all non-functional requirements and guarantees.

We consider it crucial to dispose of such contracts if we want software components to behave like components from other engineering domains. Moreover, software components can certainly be the right software units to justify the additional cost of using more formal approaches, as their life cycle and their marketing strategies might be driven by quality. To create a real market of software components, application developers must be capable of browsing, comparing and choosing components [13] according to all their exposed properties: an expression of services and quality of these services is then obviously necessary. Those *specifications* must then be *verified* one way or another.

Consequently we believe that the specification and verification of component-based systems must take into account those four levels of contract from the start, to provide a broad and consistent framework to handle those different kinds of properties. As we also believe that expressive formalisms are needed to support the contractual approach, we base our work on the hypothesis that it is not possible to fully verify statically that such contracts are never violated. The runtime enforcement of those contracts together leads to specific monitoring problems as a straightforward combination of separate contracts monitoring would interfere with each other. A specific monitoring framework is needed to handle all kinds of contract and we consider that an appropriate meta-level architecture must be defined to provide such a framework.

In this paper, we investigate what properties are needed on contracts, by considering two specific levels, behavioral and QoS. We define the global contract which consists of the combination of the four levels. We describe what are the requirements on a monitoring environment to handle properly those contracts. We briefly transpose those requirements to a meta-level architecture, both in its design and in its implementation.

2. SPECIFICATION OF BEHAVIORAL AND QOS CONTRACTS

In this paper, we only focus on the behavioral and QoS contracts. The first level of contract corresponds to type signatures, and type checking is usually performed statically. The synchronization aspects of contracts still need

to be studied in a general enough component framework, as concurrency issues are often reduced to the means of communication of a given connector of a component.

2.1 Behavioral Contract

Several specification techniques can be envisaged to specify the behavioral semantics of component interfaces. Currently the contractual approach based on preconditions, postconditions and invariants is promoted as a pragmatic but valuable specification technique for components. Current component frameworks are based on, or at least promote, OO programming languages. The majority of those OO languages provide only the first level of contract, but there are languages, such as Eiffel [12], that inherently incorporate behavioral contracts with preconditions, postconditions and invariants. Many extensions to existing languages with behavioral contracts, such as iContract [10] or JML [11] for Java, have also been designed.

Szyperski [14] showed that behavioral contracts based on pre and postconditions have several drawbacks related to call-backs and re-entrance. Nevertheless, we believe it still constitutes the best trade-off between expressiveness and ease of comprehension by an average developer.

However, there are two different specification approaches for those behavioral contracts:

- The first one can be qualified as *language-based specification*, as the contract expression is a boolean expression of the annotated language, usually using all the functional features (access to fields, method calls, basic types and operations) with some specific operators to refer to previous states (`old, @pre`), to the result of the annotated function (`result`), etc. The main reference is the Eiffel language [12], even if more expressive assertion language have been proposed [10], with the addition of quantification operators for example.

  This approach suffers from its reuse of the annotated language, as it is often hard to provide a complete formal semantics for the assertion language, as the underlying programming language does not provide one either. Despite this problem, this approach is open to partial specification, fits well with subtyping and is well understood by developers.

  Moreover, recent work [7] provides a sound framework for behavioral contracts based on pre and postconditions. This theory [6] brings soundness regarding inheritance and interfaces in the context of the Java programming language. But this theory is applicable to many other cases.

- The other approach relies on model-based specifications [4], that is the annotations that make up the contract are stated in terms of a mathematical model of the states of objects. This kind of specification usually enables static analysis and theorem proving, as they are based on an algebraic *style*. However, as the expressiveness of the language increases, proofs are no longer possible. Moreover, this kind of specification is hard to understand by developers as they are not trained or used to think in such a way.

  A somewhat hybrid approach is now developed : JML (Java Modeling Language) [11] restricts the use of models to model fields and reuses as much as possible the Java syntax and semantics for basic operations in the assertion language (access to fields, method calls, basic operators, etc.). JML also provides very interesting features with the separate specification of normal and exceptional behaviors, quantification operators and refinement of the specification models[1]. The runtime enforcement of some parts of the contract is also possible. As a result, JML provides a well-founded basis from the start, trying to be closer to developers.

Both approaches have advantages and drawbacks, and are trying to eliminate their respective disadvantages: theoretical work is done in the language-based approach, practical issues motivate work in the model-based approach. One can hope that the approaches will merge or that one approach will reuse and adapt all beneficial aspects of the other one.

In the meantime, it must be noted that both JML and language-based contracts systems lead to the same kind of runtime monitoring systems, which will be shown to interfere with other contract levels.

2.2 QoS Contract

Regarding QoS, contracts have been investigated in the world of distributed objects and components systems. Software components, in the broad sense, must be able to expose many different non functional properties, such as performance, reliability, policies related to persistence, transactions or security. Apart from the properties that are usually provided by the container — or context —, designing a QoS contract system expressing time and space performance in function of some resources usage seems quite challenging, as many aspects must be taken into account.

The complexity of algorithms can be easily related to an order using the O notation on both the average and worst cases. Both cases are likely to be relevant for a software component. But this notation expresses complexity bounds, independent of any deployment platform, so these formulas need to be related to absolute bounds [14], showing some real figures. In addition to the issue of comparing performance, contracting QoS leads to the problem of handling negotiation and renegotiation.

To our knowledge, no QoS contract language or system expresses and verifies performance issues based on input parameters and resources usage, but QML (QoS Modeling Language) [8] looks like the most advanced QoS specification language. In QML, the QoS specification is made of three mechanisms: contract type, contract and profile. Contract type are QoS aspects, such as performance or reliability. A contract is an instance of a contract type and a profile associates a QML contract with an interface. The QoS aspects that can be represented in QML are quite powerful, with different domains of value, constraints and even statistics on measured values over a period of time. However QML does not provide any means to express a QoS contract according to some parameters that would come from the component interface, e.g. to specify a time constraint in relation with the size of an input data structure. Moreover resources consumption cannot be specified. QML contracts are made of constraints on domains of values, and a contract can *refine*

[1] JML also provides a `when` clause, which can be seen as part of a level 3 contract on synchronization: if a method is called and its preconditions hold, the call will wait until the `when` clause holds as well.

another one by adding constraints or putting stronger constraints on an already constrained domain. Each kind of constraint that can be defined in QML must specify a total order among its values. A conformance relation is then defined between the contracts.

Monitoring QoS is not considered in QML [8], but similar QoS oriented approaches monitor some properties at runtime by configuring the middleware, or by using meta-level mechanisms [2]. As some categories of QoS can involve pervasive monitoring, like security in Java [5], interferences between the separate QoS monitoring already proposed would certainly occur. As general-purpose QoS specification formalisms are likely to be proposed, a contract monitoring environment must be carefully designed to enable to express the correct combination of each corresponding monitoring process. It must be also kept open enough to take into account the possible new features. The environment must also handle the case of partial conformance between QoS contracts or during monitoring, e.g. a time constraint is respected but a space constraint is not. Different policies are then applicable: termination, renegotiation, etc. The same problem arises on the global contract, as described in the next section.

2.3 Putting Contracts Together

Considering all four levels together (type, behavior, synchronization, QoS), a proper combination can be determined in order to provide a global contract. The general specification can simply be done separately in each contract formalism and the conformance rule of this global contract is the conjunction of all conformance rules. However, it is also important to consider the case where some partial conformance is achieved, which typically leads to contract renegotiation in QoS-aware systems. Different actions regarding the contract can be started:

- Termination if the QoS contract is considered as too important to be renegotiated.
- Renegotiation of the QoS contract with weaker constraints (e.g. a 3D component cannot provide a 30 frames/s rate and the new QoS contract asks for 25).
- Withdrawal of the QoS contract, getting back to a purely functional *best-effort* approach.
- Renegotiation of the functional contract and possibly of the QoS contract (e.g. the same 3D component is asked to lower its resolution and may be asked to maintain the 30 frames/s rate).

Consequently the combination of all contracts must be provided with the addition of dynamic negotiation capabilities, which can be taken for example from the QoS formalism.

2.4 Monitoring Issues

Monitoring at runtime needs a proper support so that a specific contract monitoring does not affect another monitoring process at a different level. For example, behavioral and QoS monitoring can interfere if the monitoring code that evaluates assertions create new objects when the QoS is monitoring space occupancy. In the same way, the time spent in monitoring must not be taken into account in profiling time, unless explicitly specified. Consequently, monitoring behavioral contracts must be done in a framework that will not interfere on any other contract level:

- by not adding new types in the type hierarchy;
- by not modifying the program behavior in relation to synchronization;
- and finally by not consuming any time or space!

Even if the first property can be achieved by modifying all the methods that give access to type information, it is not feasible to completely achieve the second and third properties. Nonetheless, the monitoring environment must strive for minimizing the effect of the observer on the observed phenomena.

3. REQUIREMENTS FOR MONITORING CONTRACTS

In order to provide an appropriate framework to monitor all kinds of contract, we propose exposing the necessary concepts that are manipulated by behavioral contract systems. In the same way, we expect to describe an open enough framework for QoS monitoring, so that the monitoring processes can be manipulated and composed at the global level.

3.1 Behavioral Contract Monitoring

The monitoring technique for such contracts consists in checking the appropriate preconditions at the entry of a method, the postconditions and invariants at the exit. Defining what are the appropriate assertions, i.e. the semantically correct ones, to be monitored on a given object at runtime, according to inheritance, subtyping and implemented interfaces [6] is considered as out of the scope of the monitoring process. We present a list of requirements on the monitoring system:

- The integration of the contract enforcement code with the normal code must not create new *visible* classes or methods — wrapping asserted methods is a common way to integrate assertions —. Even if programmers can be told not to use these, any tool that uses the modified class will consider them as normal unless correctly hidden or specified. Avoiding a pervasive integration is also important for the deployment footprint, which could be constrained in a QoS contract.
- Specific data structures and code are usually necessary to manage the integration, to avoid non-termination of assertion checking due to recursion, to provide assertion triggering at a fine-grained level (class or object) and to make the checker thread-safe!
- All accesses to instance fields and all method calls that are made to evaluate an assertion are recorded as such, i.e. not counted in time measurement.
- All created objects during any evaluation are excluded from space measurement.
- The synchronization policies and behaviors normally defined for the component should not be modified. How this can be achieved, totally or partially, remains an open question. However, the sketched framework is expected to be able to design and experiment proper solutions.

The assertion languages always provide enhancements to boolean expressions in order to increase expressiveness. The most common ones are studied in relation to the monitoring issues:

- Quantification operations ($\forall$, $\exists$), or more generally higher-level functions, need to be translated to the underlying language, thus generating extra code and new functions. That boils down to the first side-effects presented above.
- Access to the result of the annotated function usually generates side-effects because of methods wrapping or any other techniques used to provide this feature.
- Reference to the previous state of objects in the specification of procedures (`old`, `@pre`) is usually done by generating local variables that keep references or values of the concerned variables by computation at the method entry. They are later referenced at method exit. This additional code generates side-effect. This is also the same for the `let` construct, which is used to avoid repetitions in assertions.

All the prototyped approaches that have been proposed so far generates all, or almost all, side-effects listed above. That includes approaches based on source to source processing, bytecode adaptation, compile-time or runtime reflection and aspect-based processing.

3.2 QoS Contract Monitoring

As a proper general QoS specification language is not available, we infer some principles on how to monitor QoS. Taking QML as an example, the monitoring would be based on time measurement and appropriate recording to compute necessary values and statistics. QoS in general would be based on measurement of many parameters, to compute results ranging from simple constraints to complex function of several parameters, such as time and space requirements together with dependencies on available resources, size of input data, etc.

Consequently, the computation of these results needs to be semantically correct, that is:

- *without any interference* from other contracts, and actually, without any interference coming from the component surrounding, such as the services provided by the container.
- *at the appropriate times.* Considering the method call as the essential point of contracting, the monitoring environment can be kept open by reusing the fine-grained model of aspect-oriented programming [9] to consider the following events: before the method call (client side), at the method entry (provider side), before the exit (provider side), just after the method call (client side). A distinction can also be made between the method call and the effective method execution (late binding).

3.3 Requirements for an appropriate meta-level

Monitoring behavioral contracts can be seen as an aspect in the sense of aspect-oriented programming [9]. Monitoring some QoS properties in middleware has been done through message reflection [2]. Consequently we consider that the monitoring of all forms of contracts must be done in a appropriate meta-level framework supporting message interception, as contracts are mainly monitored on method calls. However this meta-level must satisfy strong constraints, as it must provide a clear separation between the normal behavior and other aspects, so that monitoring of contracts can be as **transparent** as possible to the semantics and to QoS for the client.

We believe that this transparency can be achieved by defining a minimal set of interactions between the two levels, taking into account low-level issues such as object allocation. At the meta level, the careful implementation of the specific contractual constructs we have described is expected to enforce transparency as well. The main issue of such a framework will certainly be performance, as both levels will need to act as almost separate execution environments. The implementation of a prototype to experiment these ideas has begun. It uses a language-based specification language for Java, OCL-J, which adapts the Object Constraint Language of UML to the Java programming language. The developed prototype is intended to use the Java Platform Debugging Architecture (JPDA), as this architecture provides a framework that is close in some points to our requirements. We expect these experiments to provide feedback on how to properly design the interactions between the base and the meta levels, as well as new insights in the area of contract monitoring.

4. CONCLUSION

In order to provide quality software components, the specification and verification of component-based systems must take into account both the functional and non-functional contracting of interfaces. By considering a global contract merging all kinds of contracts, we showed that renegotiation of QoS contracts must be supported and that different monitoring codes must be aware of each other and must not interfere. Consequently we argue that contract monitoring must be handled globally inside a meta-level that clearly separates the base level and the meta-level in all functional and non-functional aspects.

5. ACKNOWLEDGMENTS

Thanks to Jacques Malenfant for discussions on non-functional contracts and for pinpointing the QoS specification language QML.

6. REFERENCES

[1] *International Workshop on Component-Oriented Programming (WCOP'96)*, 1998.

[2] M. Aksit, A. Noutash, M. van Sinderen, and L. Bergmans. QoS Provisioning in Corba by Introducing a Reflective Aspect-Oriented Transport Layer. In *1st ECOOP Workshop on Quality of Service in Distributed Object Systems (QoSDOS 2000)*, 2000.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7), July 1999.

[4] Y. Cheon and G. T. Leavens. A Quick Overview of Larch/C++. *Journal of Object Oriented Programming*, 7(8):39–49, Oct. 1994.

[5] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[6] R. B. Findler and M. Felleisen. Contract Soundness for Object-Oriented Languages. In *Proceedings of OOPSLA'2001*, 2001.

[7] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral Contracts and Behavioral Subtyping. In *Proceedings of Foundations of Software Engineering (FSE'2001)*, 2001.

[8] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems Design. *Distributed System Engineering*, December 1998.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'2001)*, Lecture Notes in Computer Science. Springer Verlag (Berlin), 2001.

[10] R. Kramer. iContract - the Java Design by Contract Tool. In M. Singh, B. Meyer, J. Gil, and R. Mitchell, editors, *International Conference on Technology of Object-Oriented Languages and Systems (Tools 26, USA'98)*, IEEE Computer Society Press (New York), 1998.

[11] 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.

[12] B. Meyer. *Object-Oriented Software Construction*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 2nd edition, 1997.

[13] H. L. Nielsen and R. Elmstrom. Proposal for Tools Supporting Component Based Programming. In *Fourth International Workshop on Component-Oriented Programming (WCOP'99)*, 1999.

[14] C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley Publishing Co. (Reading, MA), 1998.