

# Towards Contract-based Testing of Web Services

Reiko Heckel<sup>1</sup>

*Faculty of Computer Science, Electrical Engineering  
University of Paderborn  
33098 Paderborn, Germany*

Marc Lohmann<sup>2</sup>

*Faculty of Computer Science, Electrical Engineering  
University of Paderborn  
33098 Paderborn, Germany*

---

## Abstract

Web Services are composed by linking together service providers and requestors. To ensure interoperability, the requestor's requirements for a service have to be matched against a service description offered by the provider.

Besides data types and operation signatures, this requires service specifications to include behavioral information, like contracts specifying pre- and post-conditions of (required or provided) operations.

In this paper, we propose to visualize contracts by graph transformation rules which blend well with a UML-based notion of data models and signatures. The operational interpretation of rules could turn out to be useful for simulating the behavior of required components in unit testing.

*Key words:* Web Services, Design by Contract, graph transformation

---

## 1 Introduction

Modern distributed applications are built according to the service-oriented architecture (SOA), the conceptual model underlying platforms like Web Services [3] or Jini [17]. As shown in Fig. 1 (cf. [3]), service-oriented architectures involve three different kinds of participants: service providers, service

---

<sup>1</sup> Email: [reiko@uni-paderborn.de](mailto:reiko@uni-paderborn.de)

<sup>2</sup> Email: [mlohmann@uni-paderborn.de](mailto:mlohmann@uni-paderborn.de)

requestors and registry services. The service provider exposes some software functionality as a service to its clients. Such a service could, e.g., be a SOAP-based Web Service for electronic business collaborations over the Internet. In order to allow clients to access the service, the provider has to publish a description of the service. Since service provider and service requestor usually do not know each other in advance, the service descriptions are published via special registry services. After these registry services have registered and categorized the service descriptions, they can provide them in response to a query issued by one of the service requestors. As soon as the service requestor finds a suitable service description for its requirements at the registry, it can bind to the provided service and use it. Such service-oriented architectures are typically highly dynamic because the services are only loosely coupled and clients often bind to services at runtime.

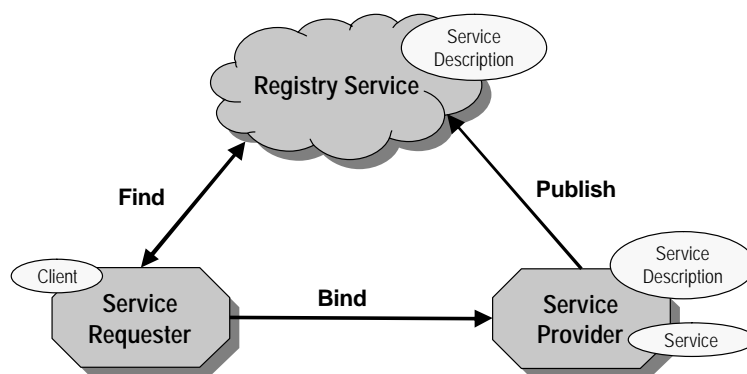


Fig. 1. Service Oriented Architecture

The mechanisms to achieve this vision of an open, dynamic architecture rely on two assumptions.

- (i) Provided services are functionally correct w.r.t. their descriptions.
- (ii) Matching descriptions with requirements is sufficient to establish the interoperability of provider and requestor.

The first assumption is obvious. It is necessary for all kinds of systems and it may be established by unit testing. The second assumption comes from the fact that loosely coupled distributed systems cannot be tested in integration. Instead, the specifications have to contain enough information about the provided and requested behavior that, once a match is established, requestor and provider can work together successfully.

A method addressing similar issues in object-oriented development is Design by Contract [12], where assertions are used to specify the relation between a component and its clients as a formal agreement, expressing each party's rights and obligations. Contracts are expressed in terms of pre- and post-conditions of operations as well as invariants and can range from very abstract specifications (given two positive integers, the result will never be zero) to complete specifications (this computes the sum of two integers). This

flexibility allows for providing just the right amount of information on the behavior required.

In this paper, we want to discuss the use of Design by Contract for Web Service descriptions. This includes the problem of testing Web Services against their descriptions.

The paper is organized as follows: In Section 2 we describe Design by Contract in more detail. In Section 3, we apply this concept to Web Services and introduce a visualization of contracts by graph transformation rules. Section 4 discusses the use of contracts for testing and Section 5 concludes the paper.

## 2 Design by Contract

Design by Contract is an object oriented design technique that shares some similarities with establishing a legal contract [12]: viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. With this precise definition of a class claims and responsibilities at code level we can add specifications to the source code and attain a significant degree of trust in large software systems [12].

Design by Contract can also be used at the level of components. As a developer using Design by Contract, you specify component contracts as part of the components offered interfaces. The contract then describes what that component expects of its clients and what clients can expect of it. Typically contracts for a component are then written from the server's perspective: "If a client promises to provide x when sending a message, then the server promises always to return y in response to such a message."

A contract states what both parties must do, independent of how it is accomplished. A contract is typically defined by assertions and associated concepts. An assertion is a Boolean expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution. A typical assertion might express that a certain integer has a positive value or that a certain reference is not void. At runtime, you can evaluate these assertions during the system's execution. In a valid software system, all assertions evaluate to true. In other words, if any assertion evaluates to false, we consider the software system invalid or broken according to its specification.

In Design by contract we identify three different kinds of assertions [2]:

**Pre-conditions:** Pre-conditions specify conditions that must hold before a operation can execute. As such, they are evaluated just before a method executes. Pre-conditions involve the system state and the arguments passed into the method.

**Post-conditions:** In contrast, post-conditions specify conditions that must hold after a operation completes. Consequently, post-conditions are evaluated after a method completes.

**Invariants:** An invariant specifies a condition that must hold anytime a client could invoke an object's method. Invariants are defined as part of a class definition. In practice, invariants are evaluated anytime before and after a method on any class instance executes. A violation of an invariant may indicate a bug in either the client or the software component.

### 3 Design by Contract for Web Services

In the following, we describe different levels the concept of Design by Contract is needed for Web Services. Further, we introduce the notions of provided and required contracts to allow for using Design by Contract on provider as well as on requestor side.

#### 3.1 Specification levels of DbC for Web Services

A Web Service is a component that is network-accessible via standardized XML-messaging mechanisms. The vision of Web Services is that they connect applications with each other using the internet to exchange data and combine data in new ways [7]. To inform a service requestor about its rights and obligations, contracts have to be transferred from the service provider to the service requestor. Thus, using Design by Contract for Web Services we face different representations of contracts beyond the classical implementation-level. Altogether, we face representation of contracts during the development process of a Web Service on three different levels:

**Implementation-level:** Here, a contract consists of a Boolean expression of the implementation language, using all the functional features (access to fields, method calls, basic types and operations) with some specific operators to refer to pre-conditions (e.g. *pre*) and to post-conditions (e.g. *post*).

Current Web Services are in the majority of cases based on object-oriented programming languages. Most of these programming languages only support syntactic contracts, but some languages, as for example Eiffel [11], inherently incorporate behavioral contracts with pre-, post-conditions, and invariants. However, there are different approaches to extend existing languages with behavioral contracts, like iContract [10], JML [5], or JContract [15] for Java. An example of JContract looks as follows:

```
/**
 * @pre ((b!=null) && (a!=null) && (c!=null))
 * @post \$.result.getBill() != null
 */
public Order orderBook(Book b, DeliveryAddress a,
                        CreditCard c) {
    ...
    return order;
    ...}

```

**XML-level:** Web-Services are components that are used over the Internet using XML-based protocols. The Web Service Description Language (WSDL) [6], for example, describes the interface offered by a Web Service as a collection of network endpoints (ports) in terms of operation signatures. This description can be read by a service requestor to access the interfaces offered by a service provider (see Fig. 2). However, WSDL falls short of providing behavioral information on operations. An extension with operation contracts would solve this problem.

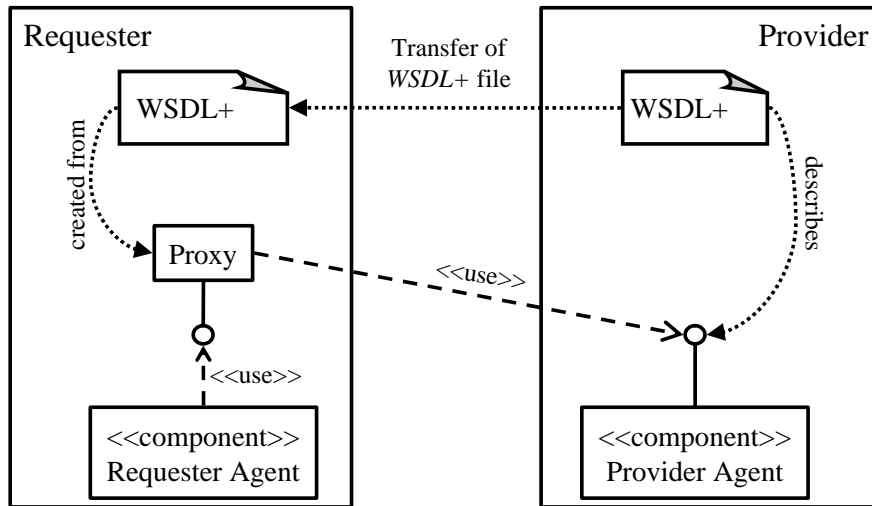


Fig. 2. Transfer of interface descriptions to requestor agent

**Model-level:** The main drawback of the first two levels is that they are hard to read and write for humans. A visualization for contracts that enables documentation and communication between developers would be preferable. In the following, we will concentrate on the model-level specification of contracts to explain our ideas.

The goal is to derive implementation- and model-level contracts automatically from the model-level specification. A developer then only has to create and reason about contracts on model-level.

### 3.2 Model-based specification of contracts

A model for a Web Service should have at least the following parts.

- the offered interface including its operations;
- a model of the externally visible data of the service, i.e., all data types needed for communication with its clients;
- contracts describing the operations offered by the service.

For describing an interface, we use UML interfaces [13] as shown in Fig. 3.

We use class diagrams to model the data model of a problem domain. These class diagrams describe the data structures that are of interest for



Fig. 3. Interface of a simple online shop

providers and requestors of a specific domain (e.g. business areas or technical domains). This enables requestor and provider to share a common language to be sure that they use the same keywords for describing identical concepts. Fig. 4 provides a small data model for the domain for bookselling. It provides the basic terminology and is depicted as a UML class diagram. In this domain, an order is composed of a *DeliveryAddress*, the ordered *Book*, and a *Bill*. A generalization relationship indicates that *CreditCard* and *BankTransfer* are specializations of the basic concept *PaymentMethod*. This means, that a bill can be paid either by credit card or by a bank transfer.

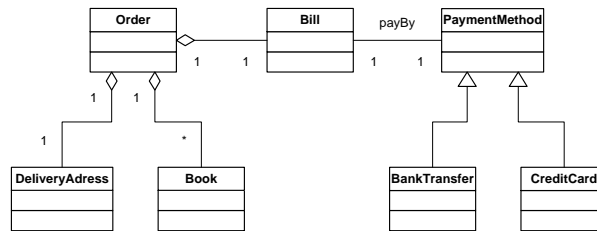


Fig. 4. Data model for bookselling (attributes omitted)

A service provider now can express by a contract that his service is able to handle orders for books which are payable by credit card. We therefore developed a technique, which uses the mechanism of graph transformation rules to facilitate an intuitive visual formulation of contracts. Before we introduce the technical details let us look at the example rule of a service provider in Fig. 5. The rule expresses that the Web Service needs data on a book to order, the personal data of the buyer and his credit card and will effect a new order for the specified book and a bill (which is paid by credit card) to be created for this customer. We can see that the service behaves as we intuitively expected from its name. Using the mechanism of graph transformation it becomes easy to describe all kinds of different services in this area. One might e.g. formulate descriptions for Services, which add items to existing orders or delete items or which offer books without any payment information. A simple translation of the graph transformation rule in Fig. 5 into JContract is shown in Section 2.

Formally, a graph transformation rule consists of two graphs (left and right hand side), which are in our case visualized by using UML object diagrams. Each of the graphs is typed over a class diagram. Refer to [16,1] for the technical details. The basic intuition is that every object, which is only present in the right hand side of the rule, is newly created and every object, which is present only in the left hand side of the rule, is being deleted. Objects which are present on both sides are unaffected by the rule. If only one object of a type exists, it can remain anonymous, if a distinction between different objects

of one type is necessary, then they carry names, separated from their type by a colon. If an even closer resemblance to standard UML concepts is called for, it is also possible to encode graph transformations in UML Collaboration Diagrams [9].

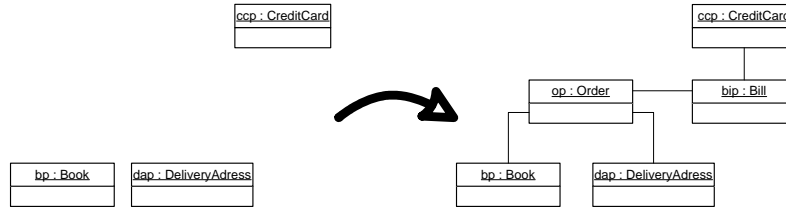


Fig. 5. Contract describing the behavior of the operation *orderBook*

### 3.3 Required and provided contracts

Service-oriented architectures are highly dynamical and flexible. A service requestor can replace a service provider dynamically at runtime or development time. This flexible architecture requires new mechanisms to ensure that service requestor and service provider can work together. On this account we use graph transformation rules for both: a description of an offered service and as the formulation of a request. A service provider details its offered interface by a *provided contract* and a service requestor details its required interfaces by a *required contract*. From a providers point of view the left hand side of the rule specifies the pre-condition of his service, i.e. the situation that must be present or the information that must be available for the service to perform its task. The right hand side of the rule depicts the post-condition, i.e. it characterizes the situation after the successful execution of the Web Service. From a requestors point of view the left hand side of the rule represents the information he is willing to provide to the service and the right hand side of the rule represents the situation he wants to achieve by using the service. Such a rule from a requestors point of view can be found in Fig. 6. This rule expresses that the client is able to provide information on a book, a credit card and his delivery address and is looking for a provider, which is able to construct an order for him based on these information. This means he is looking for a book-seller and intuitively the provider rule from Fig. 5 should be a suitable candidate for this request, because in this rule an order is created. Notice that the rule in Fig. 6 is a minimal demand of the service requestor to order a book. Unlike the contract in Fig. 5 the bill is not part of the post-condition as the requestor is primarily interested in ordering a book (paying for it is a kind of unwelcome side effect from the requestor's point of view). To allow changing the service provider we have defined in [8] criteria's whether a provided contract of a service provider fulfills the required contract of a service requestor.

In summary, a Web Service offers interfaces that are detailed by provided

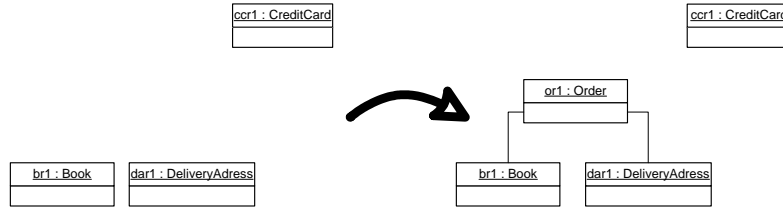


Fig. 6. Required contract of a service requestor describing the behavior of the operation *orderBook*

contracts and can self use other services. Since the latter are only known at run-time, at design-time behavior specifications of all required interfaces are created which contain operation contracts (Fig. 7).

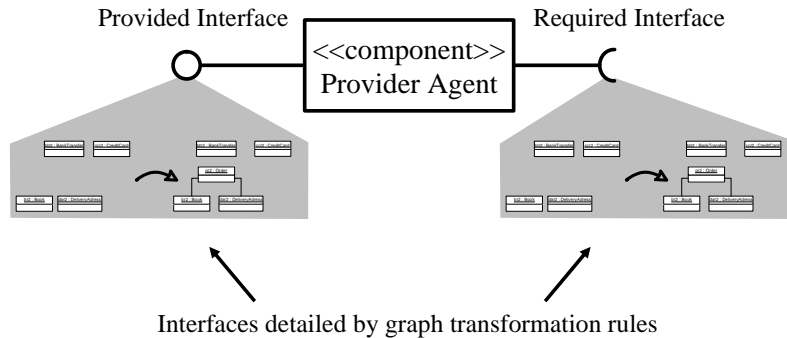


Fig. 7. Service Provider specified by provided and required contracts

## 4 Testing Web Service contracts

Design by Contract is a formal way to specify the behavior of operations. Using Design by Contract at implementation level is a way to incorporate specification information into the code itself. However, we still have to test the implementation against its specification. The question is how Design by Contract can be used to support testing, i.e., to create, select, and execute test cases.

In this paper, we are interested in functional tests, i.e., black-box tests derived from contracts. Such tests ensure that the delivered product meets its specification [4]. With this goal in mind, it makes sense to use provided contracts both for the creation of test cases (sample input data) and test oracles (conformance checks for the output).

However, as discussed in the introduction, open distributed applications limited our scope to unit testing. Indeed, classical integration testing is difficult within the field of Web Services because it has to connect different services to determine if they are interoperable. This is a problem because (i) the Web Services called by the Web Service under test may not be known at design-time and (ii) it may impossible or expensive to call them just for testing purposes. For example, a Web Service for booking a holiday could require



other services for booking flights, hotels, etc. Actually booking all facilities with each test case would soon exceed the testing budget. Therefore, we need to simulate required services. We propose to use required contracts to drive this simulation.

#### 4.1 Creating test suites from provided contracts

For testing purposes, we have to execute a Web Service using combinations of input parameters to demonstrate the conformance of the service to its provided contracts. A test driver applies different test cases or test suites to a Web Service using its provided interface (see Fig. 8). Within this paper, a test case specifies input values for an operation of an interface of a Web Service. A test suite is a collection of test cases, typically related by a testing goal [2]. In our case, a test suite could be a collection of test cases to test all individual operations offered by an interface of a Web Service. Using Design by Contract, the input values making up a test case can be derived from the pre-condition of a provided contract. As described above, pre-conditions specify conditions that must hold before a method can execute. As such, they describe allowed arguments for an operation call.

Rather than deriving test cases manually, we consider to use the Jtest tool by Parasoft [14] for this purpose. Jtest is a Java-based unit testing tool. Its aim is to support the testing of components before they are integrated into an environment. Jtest works together with JContract whose implementation-level contracts are analyzed to derive test cases and oracles. Thus, Jtest enables automated unit-level functional testing.

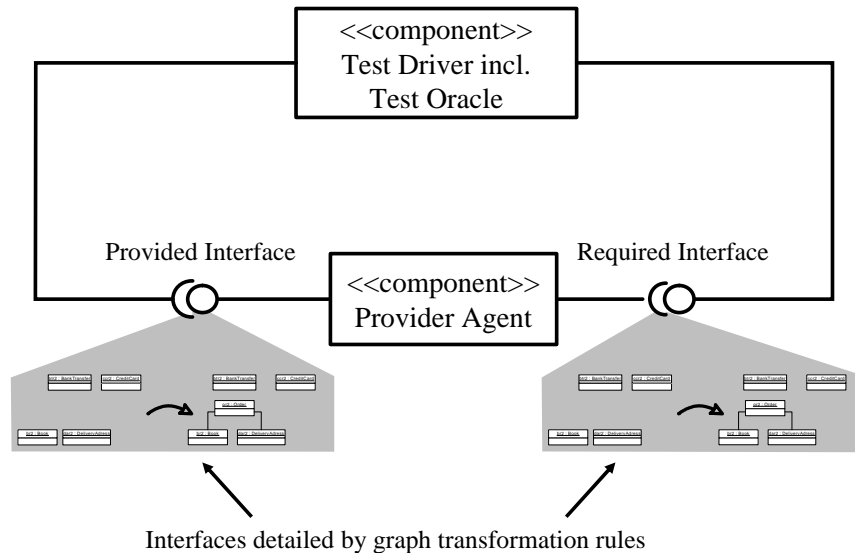


Fig. 8. Design by Contract for testing Web Services

#### 4.2 Required contracts for component simulation

For the reasons mentioned above, tests should be restricted to single Web Services. To simulate the behavior of the services required by the service under test, we propose to use the corresponding required contracts. The idea is that the contracts describe the interaction between different Web Services. A test driver can use these contracts to simulate the Web Service called by the Web Service under test (see Fig. 8). Each time the Web Service under test uses the required interface the test driver is called and creates a return value by *applying* the contract to the input data provided by the client service. That means contracts are given an operational semantics.

Such operational interpretation is not obvious for logic-based descriptions, but naturally contained in the rule-based view of graph transformations. In this sense, a required service is simulated by a typed graph transformation system having all required contracts as rules and working on object diagrams over the class diagram of the service.

## 5 Summary

In this paper, we proposed some ideas to enable testing of Web Services by using Design by Contract to add behavioral information to the specification of a Web Service. As an important new aspect, we introduced provided and required contracts. Provided contracts describe the behavior offered and required contracts the behavior needed by a Web Service. We have used graph transformation rules to describe the contracts at the level of models. This allows a simulation of the required Web Service by accomplishing the required contracts during the execution of a test case.

To conclude, we list the open issues of the approach (hoping that some of them might be resolved by building on the work of other people).

- An XML-based language for contracts has to be defined and integrated into existing standards like WSDL.
- A UML-based notation for contracts has to be fixed using concepts of graph transformation rules. A CASE tool has to be found or adapted to support the editing of these models.
- A mapping of UML-based contracts to JContract (or other implementation-level contract languages), as well as to a XML-based contract language is required. Based on this mapping we want to deduce the general contract semantics that can be expressed by means of graph transformation rules. Also, these mappings have to be supported by tools.
- A simulation engine for required services has to be provided, e.g., by wrapping a graph transformation tool like AGG or Fujaba into a Web Service interface.

## References

- [1] Baresi, L. and R. Heckel, *Tutorial introduction to graph transformation: A software engineering perspective*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, Lecture Notes in Computer Science **2505** (2002), pp. 402–429, [http://www.upb.de/cs/ag-engels/ag\\_engl/People/Heckel/talks/Agtive2003Tutorial/](http://www.upb.de/cs/ag-engels/ag_engl/People/Heckel/talks/Agtive2003Tutorial/).
- [2] Binder, R. V., “Testing Object-Oriented Systems: Models, Patterns, and Tools,” Object Technology Series, Addison Wesley, 1999.
- [3] Booth, D., H. Haas, F. McCabe, E. Newcomer, I. M. Champion, C. Ferris and D. Orchard, “Web Services Architecture, W3C Working Group Note,” (2004), <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>.
- [4] Carmichael, A. and D. Haywood, “Better Software Faster,” Prentice Hall PTR, 2002.
- [5] Cheon, Y. and G. T. Leavens, *A runtime assertion checker for the Java Modeling Language (JML)*, in: H. R. Arabnia and Y. Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002* (2002), pp. 322–328, <ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf>.
- [6] Chinnici, R., M. Gudgin, J.-J. Moreau and S. Weerawarana, “Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language,” (2002), <http://www.w3.org/TR/wsdl12/>.
- [7] Fensel, D. and C. Bussler, *The web service modeling framework wsmf*, Technical report, Vrije Universiteit Amsterdam (VU) (2002), <http://www.csd.uch.gr/~hy565/Papers/wsmf.pdf>.
- [8] Hausmann, J. H., R. Heckel and M. Lohmann, *Towards automatic selection of web services using graph transformation rules*, in: R. Tolksdorf and R. Eckstein, editors, *Berliner XML Tage* (2003), pp. 286–291, <http://www.upb.de/cs/ag-engels/Papers/2003/BXMT-AutomaticSelectionOfWS.pdf>.
- [9] Heckel, R. and S. Sauer, *Strengthening uml collaboration diagrams by state transformations*, in: H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, LNCS **2029** (2001), pp. 109–123.
- [10] Kramer, R., *iContract – the Java design by contract tool*, TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California (1998), pp. 295–307.
- [11] Meyer, B., “Eiffel: The Language,” Object-Oriented Series, Prentice Hall, New York, NY, 1992.

- [12] Meyer, B., “Object-Oriented Software Construction, 2nd Ed,” Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1997, second edition, xxvii + 1254 pp.  
URL [http://www.prenhall.com/allbooks/ptr\\_0136291554.html](http://www.prenhall.com/allbooks/ptr_0136291554.html)
- [13] Object Management Group, *UML specification version 1.4* (2001), <http://www.celigent.com/omg/umlrtf/>.
- [14] Parasoft, “Automated Java Unit Testing Coding Standard Compliance, and Team-Wide Error Prevention,” (2003), <http://www.parasoft.com/>.
- [15] Parasoft, “Using Design by Contract™ to Automate Java Software and Component Testing,” (2003), <http://www.parasoft.com/>.
- [16] Rozenberg, G., editor, “Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations,” World Scientific, 1997.
- [17] Sun microsystems, “Jini Architectural Overview - Technical White Paper,” (1999), <http://www.sun.com/software/jini/whitepapers/architecture.pdf>.