Enriching Java Enterprise Interfaces with Formal Sequential Contracts*

Marco Müller, Moritz Balz, Michael Goedicke
paluno - The Ruhr Institute for Software Technology
University of Duisburg-Essen, Campus Essen, Essen, Germany
{marco.mueller, moritz.balz, michael.goedicke}@paluno.uni-due.de

ABSTRACT

Many software systems are stateful or have stateful components, i.e. they manage and process data depending on certain steps performed before. For this reason, sequential contracts are of interest that describe component interfaces with respect to sequences of invocations leaving the component in a valid state. While sophisticated models for this purpose exist, component frameworks don't support sequential contracts, so that a validation of the state is performed algorithmically and mixed-up with application logic in method contents. In this contribution we present (1) meta data for object-oriented program code that allow to annotate component interfaces with sequential contracts and (2) a pattern and tools integrating sequential contracts in Java Enterprise components. The contracts can then be modeled as part of the program code at development time and be interpreted at run time to prevent invalid method invocations and provide tracing data in case of errors.

1. MOTIVATION

Complex software systems usually have components that are stateful – i.e., they have different states and act based on their current state. The states can be specified formally or informally and are often related to data that is managed by the component. Such stateful components do not exist standalone, but are part of larger systems and by this means coupled to other components. The coupling is realized with method invocations between components. Consequently, in stateful components, method invocations may be valid or invalid depending on the component's state. When this is specified formally, the component interfaces define valid *sequences of method invocations* which are called *sequential contracts*. As a simplified example consider a shop system with a shopping cart component that requires a login, then adding of products, and finally a checkout. This simple example is shown informally in figure 1.

*Part of the work reported herein was funded by the German Federal Ministry of Education and Research under the grant number 03FPB00320.

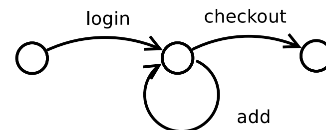

Figure 1: A simple example for a sequential contract: This shopping cart requries a login, then adding of products, and finally a checkout.

An exemplary environment for such components is that of enterprise applications, e.g. with the Java Enterprise Edition (JEE) [1]. Application logic components are part of larger systems consisting of different layers for user interfaces, remote communication, database access, etc. Such systems can also be subject to changes, adaptations, and reconfigurations of components, so that a robust description of component interfaces is desirable. The principle is shown in figure 2: Components with application logic, in this case the `ShoppingCartBean`, are coupled to other components with different purposes and depend on the actions taken by end users. While an interface exists that describes the application logic components, in this example `IShoppingCart`, sequential contracts cannot be modeled in current component frameworks.

For this reason, application logic and validation of the current state are usually mixed-up in method bodies, thus reducing the understandability and maintainability of such program code. For example, we can imagine that the method `add` of the shopping cart always validates if its class attribute `user` is not null since it expects the login method to set the user; by this means, the current state is inferred from the data. Although techniques exist for formal verification and compatibility checks of sequential contracts, the specifications are only informal when they are hidden in the implementation. In addition, observation and monitoring of method invocations and possible contract violations is handled locally in method bodies so that possible errors are not necessarily detected and tracked at run time. Considering these problems, we propose an approach to integrate sequential contracts in existing component frameworks, in this case the Java Enterprise Editions. The approach has the following goals:

1. Sequential contracts shall be specified in a formal way

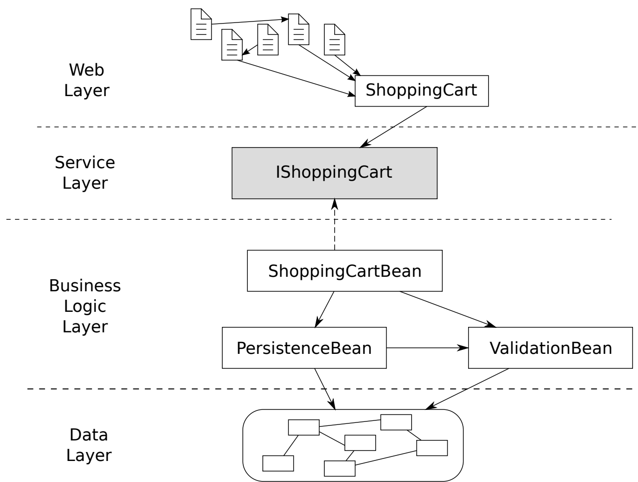

Figure 2: Components in the Java Enterprise Edition: The `ShoppingCartBean` contains stateful application logic and is described with the interface `IShoppingCart`, which is in turn accessed by other components, e.g. from the web layer.

for component interfaces of the JEE framework.

2. The specifications shall be usable at development time for model checking.

3. The specifications shall be usable at run time for validation of actual method invocations and error tracking by a common framework.

Thus, we must find a way to integrate Java interfaces and sequential contract definitions. The resulting notation must not only be readable at development time, but also at run time. A technique must be developed that integrates validation and error tracking in the JEE framework. This paper will thus describe the following contributions:

1. We specify semantics for existing formal sequential contract models, *interface automata* [2], that are appropriate for describing Java interfaces.

2. We specify a notation for sequential contracts based on Java Annotations [3], i.e. type-safe compiled meta data that can be attached to Java interfaces and their methods.

3. We introduce a framework on top of the JEE that validates the contracts at run time, tracks errors, and allows for monitoring.

The paper is for this purpose organized as follows: In section 2 we consider existing approaches. In section 3 we describe interface automata and apply them to Java interfaces in section 4. The usage of the model at development time and run time is explained in section 5. Finally, we discuss the approach in section 6 and conclude in section 7.

2. RELATED WORK

The extension of interfaces with additional information is subject to many publications. However, most focus on implementing pre- and postconditions for method calls. It is possible to emulate call sequence contracts using behavioural descriptions with internal state fields that are checked in preconditions and set in postconditions of method invocations. This approach is discussed in [4] and argued to be error prone and hard to understand, because the sequential contract is just indirectly modeled in hat paper. Cheon and Perumandla present an extension to the Java Modeling Language (JML) [5] to describe call sequences in a regular expressions–like notation. Their notation is embedded into program code comments. The compilation process includes a transformation of the sequence definition to a structure of methods checking pre- and postconditions. The explicit sequence definition is thus only available at compile time, which contradicts our goal 3.

Heinlein [6] describes concurrency and sequence constraints for Java classes using interaction expressions. The constraints are checked at run time, postponing prohibited method calls. However, the Java language is extended in this approach to integrate the checks for the constraints. Thus this approach is not applicable for all JEE-based systems.

Pavel et al. [7] introduce a framework which allows to check and reject method calls to Enterprise Java Beans (EJB) [8] components, based on Symbolic Transition Systems. In their approach, the state checks have to be implemented manually or via a precompiler directly into the business methods. In contrast to our approach, the validation code is mixed-up with the business aspects in the code. Thus no explicit model is available at run time.

As another view, some architecture description languages (ADL) provide sequential or protocol information for component interfaces. As some of these languages have mappings to the Java programming language, an automatic mapping of sequential constraints to the interfaces expressed in the ADL could represent an extension of Java interfaces with a sequential contract. E.g., Reussner et al. [9] provide so-called Service Effect Specifications (SEFF) in their Palladio Component Model. However, they do not specify SEFFs any further. SEFF can e.g. be defined as Finite State Machine or any other language. No suggestion is given for an implementation of the SEFFs.

3. INTERFACE AUTOMATA

Beugnard et al. [10] define four levels of interface descriptions: (1) *syntactic* level, i.e. method signatures; (2) *behavioural* level, i.e. pre- and postconditions for method invocations; (3) *synchronization* level, which includes call synchronization and sequences; and (4) *quality of service* level, including performance and security information. Interface definition languages like IDL [11] or the interface syntax of current programming languages usually allow for defining first level interfaces. For defining second level interfaces, language extensions like JML can be used. Sequential contracts are defined in third level interfaces. However, this level of interfaces is not represented in current programming languages. Our objective is to use a model for synchronization level interfaces to define permitted call sequences for Java

interfaces. We use interface automata [2] to model these contracts.

Interface automata are essentially finite state machines with in- and output actions, where each input defines a received method call and each output defines an outgoing method call. This mechanism can be used to describe how a component that implements an interface can be called by its context and how it makes calls to external components.

As stated by de Alfaro and Henzinger [2], an interface automaton P is a six-tuple $P = \langle V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P \rangle$.

- V_P is a set of states.
- $V_P^{init} \subseteq V_P$ is a set of initial states, with at most one state. P is called *empty* if $V_P^{init} = \emptyset$.
- A_P^I, A_P^O and A_P^H are disjoint sets of input, output and internal (hidden) actions. $A_P = A_P^I \cap A_P^O \cap A_P^H$ is the set of all actions.
- $T_P \subseteq V_P \times A_P \times V_P$ is a set of steps, which move the automaton from one state to another when the action is performed.

The formal foundation of interface automata allows for verification of e.g. the compatibility of interface automata. The compatibility of interface automata should be verified when a component is to be replaced by another implementation in a component assembly, where a set of components is interconnected to an application. This ensures that the new sequential contract is compatible with the prior one. Algorithms for computing the compatibility of interface automata are given in [2]. In addition, interface automata can be statically checked for a set of properties, including deadlock-freedom.

As an example, figure 3 shows the graphical representation of an interface automaton *Shopping Cart*, which is a refinement of the contract shown in figure 1. This interface automaton defines three input actions `login(...)`, `add(Product)`, and `checkout()`, as well as two output actions `success` and `LoginFailedException`. The automaton consists of three states `start`, `start'`, and `ready`, of which `start` is the initial state. When the input action `login(...)` is triggered, the automaton moves to the state `start'`, which will result in one of the output actions. If the login was successful, products can be added or a checkout can be processed.

4. INTERFACE AUTOMATA IN JAVA

A notation for describing the interface automaton for a specific interface is needed for using interface automata in Java. In this section we explain our notation for automata using meta data on interfaces.

For attaching interface automata to Java interfaces, we use a notation that is integrated with the interface definition. In this notation interface automata are defined by information included in the Java interface description, and additional data attached to the interface. This additional data is added using Java's own concept for meta data in source code, called annotations [3]. The interface class and its methods are

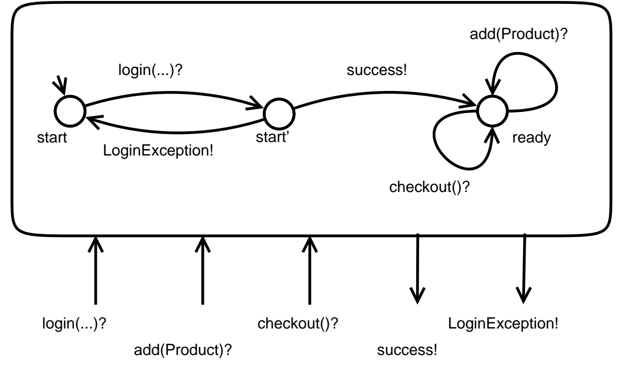

Figure 3: A simple interface automaton

extended with special annotations defining automaton elements that cannot be found in the interface definition otherwise. Thus the notation is integrated with the interface description. This integration technique avoids redundantly defined information. Listing 1 shows an example of the notation. The automaton described in this listing is the same as shown in figure 3.

For using Java interface definitions within the notation, elements of the interface description are mapped to elements of the interface automaton definition: The methods declared in the interface are defined to be the set of input actions A_P^I of the automaton. If a method throws any exception, the successful return of a method and each explicitly thrown exception are defined as output actions in A_P^O. This information is available directly from the interface description. In addition, the interface automaton needs the set of states V_P, the initial state V_P^{init}, and the set of steps T_P.

The annotation `@InterfaceAutomaton` declares an interface to be enriched with an interface automaton. The initial state V_P^{init} is as an attribute of this annotation. The steps of the automaton are defined by the annotations `@IAStep` on the interface methods. This annotation type includes the name of the automaton state that enables the method as input action, the state of the automaton if the method call was successful, and optionally the state to get into for each possibly thrown exception. Thus an `@IAStep` annotation can declare more than one step. The set of states V_P is implicitly defined within the in- and output parameters of `@IAStep` annotations. Each state named in any of these annotations is a state in V_P. The set of internal actions A_P^H is not used in this model.

In order to represent such descriptions of Java interfaces, the interface automaton design has to follow a special pattern: (1) A method without exceptions is a step between two states with an input action, which is the method call. (2) In addition to that, a method with n declared exceptions describes $n + 1$ output actions: the successful method execution (named `success` by convention of this pattern) and each declared exception. An additional state is defined for handling these output actions. The pattern is shown in figure 4. In this example, a connection from state `A` to state `B` exists if the method call `x` is successful. The additional state `A'` as introduced by the pattern handles possible exceptions.

```
@InterfaceAutomaton(initialState = "start")
public interface IShoppingCart {
  @IAStep(from = "start", to = "ready", exceptions = {
      @IAException(at = LoginFailedException.class, to = "start")
     })
  public boolean login(String user, String password) throws LoginFailedException;

  @IAStep(from = "ready", to = "ready")
  public String add(Product p);

  @IAStep(from = "ready", to = "start")
  public void checkout();
}
```

Listing 1: An interface automaton definition for a Java interface

The output actions (`success`, `Exception1` ...`ExceptionN`) are not related to method calls, but only to the outcome of state `A'`.

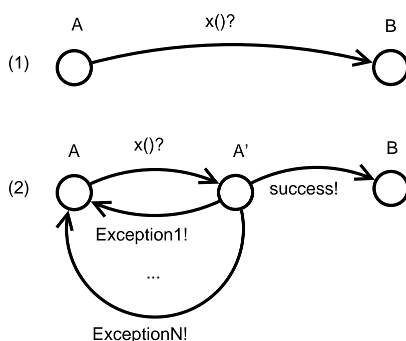

Figure 4: Patterns for modeling interface automata

5. MODEL USAGE IN JAVA ENTERPRISE APPLICATIONS

The interface automaton definition integrated with Java interface descriptions allows for using the automaton in component systems. As shown in figure 2, in the Java Enterprise Edition, Java interfaces describe the functionality of the application logic components, called Enterprise Java Beans (EJBs). Thus the automata definitions can be used for describing permitted call sequences to EJBs. Due to the integrated description of interface automata, the automata definitions are available for usage not only at development time, but also at the run time of the system, by using reflection mechanisms [12] for Java annotations. In the following, the possible usage of the models is explained.

5.1 Development Time

The formal notation of interface automata allows for statically checking the models for deadlocks and other properties. In addition to static checks, the behaviour of interface automata can be simulated prior to the system deployment to ensure the correct behaviour or to gain an understanding of the sequential contract. As the notation can be transformed in any other notation for interface automata or I/O-automata [13], existing model checking and simulation tools (e.g. UPPAAL [14]) can be used to verify the automata attached to interfaces.

It is also possible to perform static code analysis for component interactions. This requires to extract models that describe how interfaces are used by other components. Approaches for this already exist, e.g. in the *Jadet* tool [15]; for our purpose, the tool must be adapted to extract calls to EJB interfaces from the control flow to create an interface automaton, which can then be matched with the automaton describing the service.

As the contract is formally defined and directly attached to the interface, component evolution is supported by the approach. When the implementation of an EJB is changed, the sequential contract is not touched by the changes, as it is attached to the interface instead of the business code. In contrast, when the sequence checks are mixed-up with the code of the business methods, any changes in the business code might affect the sequential contract. The same applies to component exchange. When an EJB is exchanged against another implementation, e.g. an implementation of another supplier, the interface containing the contract does not implicitly change with the implementation.

The static nature of code annotations allows for using graphical editors for an easier understanding of more complex automata. A graphical editor could be embedded into development environments for Java programs (e.g. Eclipse [16]). Such an editor could visualize the automaton of interface classes while editing the interface's source code.

5.2 Run Time

The integration of interface automata descriptions using Java annotations allows for run time access to the definition. At development time the complete definition is available programmatically. This enables to create common frameworks to use automata at run time. The knowledge about the current state of the interface automaton can be used to enforce the correctness of call sequences. A common framework can evaluate calls to an interface and reject calls that are not allowed. To accomplish this, a technique must exist that can observe and prevent method calls. Examples for such techniques are aspect-oriented programming (AOP) [17] and dynamic proxies [18].

As a proof-of-concept we implemented such a framework for the Java Enterprise Edition. This framework evaluates the

interface automata attached to EJBs in a system and rejects calls if they do not comply with the sequential contract. When calls are rejected, the reason is given in an exception. The exception also includes a trace, which names the last states before the rejection and the steps taken to get to these states. To accomplish this task, the prototype uses the interceptor concept already contained in the JEE specification. Interceptors use AOP techniques to intercept method calls in an JEE environment. The prototype uses this concept to attach an observer to each bean with an interface automaton. This observer traces the calls to a bean interface and the bean instance implementing that interface. Subsequently, all method calls to these interfaces are verified by the observer with respect to the interface automaton. When the sequential contract is broken, the observer will reject the call. Figure 5 visualizes this concept. The left side shows the interaction of a component with the interface `ShoppingCartBean` when it has no explicitly modeled sequential contract. As explained in the motivation, the business component must check the permission of the call in the business methods. The right side shows the same scenario with the observer in use: The observer is notified about method calls, verifies them against the interface automaton, and rejects the second login method, since it is not valid in the example interface automaton shown in listing 1.

For attaching an observer to an EJB, only one annotation is necessary. Listing 2 shows a bean implementing the interface `IShoppingCart`. The annotation `@Interceptors` refers to a class from our runtime framework that implements the observer. The observer class is by this means the central instance for the enforcement and monitoring of sequential contracts.

```
@Management(IShoppingCart.class)
@Interceptors(IAInterceptor.class)
public class ShoppingCartBean implements
    IShoppingCart { ... }
```

Listing 2: The observer is attached to a bean in one row of source code. The framework only needs this information and the automaton itself for tracing the interfaces sequential contract and to reject illegal calls.

The information about the interface automata states in a running system also allows for creating a dashboard for monitoring the system. Such a dashboard could give administrators valuable information about the occurrence of errors in the system. The state traces allow for giving precise information about the reasons for these errors.

6. DISCUSSION

The approach presented here aims at using formal techniques for the specification of sequential contracts in Java, and at enabling their usage in the context of the Java Enterprise Edition. We will now discuss the approach with respect to this objective and the goals stated in the motivation.

From a functional perspective, the goals are reached in general: With the definition of meta data annotations as explained in section 4, the semantics of interface automata can be used inside Java. On the one hand, permitted call sequences can be specified. On the other hand, accessing and interpreting the models is possible in order to build tools that prevent method calls that do not comply with the model specifications. The usage of annotations and reflection mechanism entails that all information is consistently available in the coherent notation of the programming language, and thus directly usable at run time. Java interfaces can thus carry information not only about structural properties, but also about sequential contracts. In contrast to a generative approach, which is usually followed with domain-specific languages and in model-driven development, the code does not need to be synchronized with an externally defined model. The usage of reflective accessible language constructs allows to integrate validation frameworks into applications. These frameworks can use the modeling notation to ensure correctness of programs with respect to interaction sequences. By this means, validation with respect to invocation sequences does not have to be mixed-up with the business logic and is thus separated from it.

Usage of these specifications is only possible if appropriate component models are available that allow to observe component interactions and – depending on the goal – manipulate it, too. We used business logic components in the Java Enterprise Edition here to show that the integration of the approach into existing frameworks is possible if these requirements are fulfilled. The integration into JEE could be implemented with few lines of code.

A disadvantage of using interface automata as formal basis for sequential contracts is its expressiveness. This basis disallows e.g. the definition of sequences of the form $add^n remove^n$, where *add* and *remove* are methods and n is an unknown number of calls. Hence the approach has the same issues in expressiveness as for example the approach of Cheon and Perumandla [4].

The interface automata described here are limited to a single thread, since the EJBs in focus are intended to be used in one single thread only. Nevertheless, the models allows for describing a wide range of sequential contracts, that are often in use with systems using the Java Enterprise Edition, which is the goal of this paper.

The automata notation is tightly coupled with the interface definition. As shown in section 5, this notation implies that the automata design follows a certain pattern. This constraints the freedom for modeling contracts. In our use cases, these modeling constraints did not have a negative impact. The behaviour of business component interfaces seem to apply well to this pattern. However, we cannot exclude that this constraint decreases the applicability.

Considering the purpose of using sequential contracts, the strategy for handling prohibited method calls is important. We focused on ensuring valid sequences and thus reject invalid calls. Other strategies could include postponing or scheduling invalid calls until their requirements are met. In addition, concurrency handling must be considered. In contrast to method invocations in components, steps in interface automata are not time consuming. When the component can be accessed concurrently, the interface automaton's

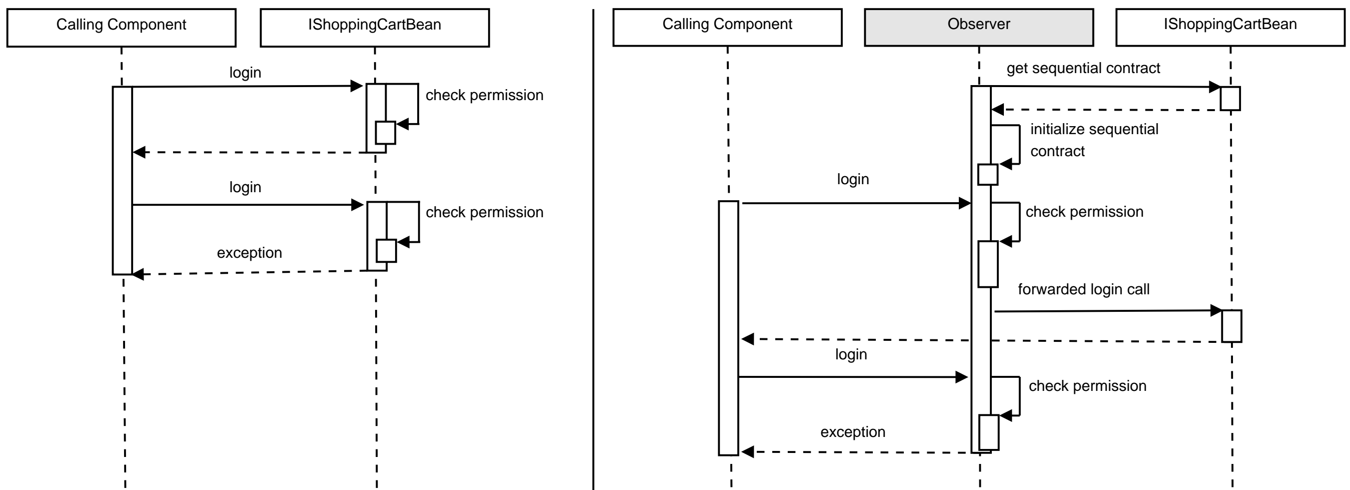

Figure 5: On the left side, the callee needs to check whether the calls respect the sequential contract. On the right side, these checks are processed by the observer. The callee can focus on the business issues.

state may differ from the component's state that it represents. This has to be considered for the integration into frameworks that allow for concurrent access to component instances. However, this is not an issue in the presented integration for EJBs, as concurrent calls to bean instances are serialized by the EJB container. Hence these beans are not concurrently accessible.

In summary, interface automata and their usage in Java as presented here fulfill the goal to make sequential contracts available in Java. Not all model features are used, but the subsets of interest in the context of existing frameworks are applicable. We therefore think that this approach is promising, and also plan to evaluate it further by using the approach in software projects within our department.

7. CONCLUSION

In current programming languages and frameworks, component interfaces are usually described structurally. This only includes the signatures of the available methods. However, stateful components often assume certain orders of method calls, e.g. an authentication before further methods may be called. While several formal techniques exist that allow for describing call sequences, these abstract concepts need to be mapped to current programming languages. In this contribution we presented an approach for representing sequential contracts in Java, using a notation that is integrated with the Java interface notation. Our notation allows for programmatically using the contract information at development time and at run time. In addition, we implemented a framework for evaluating calls to Java Enterprise Beans at run time, which detects and rejects method calls that do not comply with the sequential contract of interfaces.

However, some questions are still open: The concepts used in this paper focus on single-threaded access to components. The sequential contract definition and its implementation used in this paper must be reconsidered in the future for handling concurrent access.

As future work, we plan to develop tools to edit the interface automaton definitions graphically, and to transform contract information into the formats necessary for existing model checking tools. As another step, we want to statically verify the compliance of client code with the behavioural contract. Due to feedback from the industry, the generation of test stubs on the basis of permitted call sequences will also be considered as future work.

8. REFERENCES

[1] Sun Microsystems, Inc.: JSR 316: Java^{TM}Platform, Enterprise Edition 6 (Java EE 6) Specification (December 2009) `http://jcp.org/en/jsr/detail?id=316`.

[2] de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM. (2001) 109–120

[3] Sun Microsystems, Inc.: JSR 175: A Metadata Facility for the Java^{TM}Programming Language (2004) `http://jcp.org/en/jsr/detail?id=175`.

[4] Cheon, Y., Perumandla, A.: Specifying and Checking Method Call Sequences of Java Programs. Software Quality Journal **15**(1) (March 2007) 7–25

[5] Leavens, G., Cheon, Y.: Design by Contract with JML. Draft, available from jmlspecs. org **1** (2005) 4

[6] Heinlein, C.: Advanced thread synchronization in java using interaction expressions. Objects, Components, Architectures, Services, and Applications for a Networked World **2591/2009** (2009) 345–365

[7] Pavel, S., Noye, J., Poizat, P., Royer, J.C.: Java Implementation of a Component Model with Explicit Symbolic Protocols. In: In Proceedings of the 4th International Workshop on Software Composition (SC'05), volume 3628 of LNCS, Springer-Verlag (2005) 115–124

[8] Sun Microsystems, Inc.: JSR 318: Enterprise JavaBeans^{TM}3.1 - Proposed Final Draft (March 2008) `http://jcp.org/en/jsr/detail?id=318`.

[9] Reussner, R., Becker, S., Happe, J., Koziolek, H.,

Krogmann, K., Kuperberg, M.: The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe (TH), Germany (May 2007)

[10] Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer **32**(7) (1999) 38–45

[11] Object Management Group: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1: CORBA Interfaces. Technical report, Object Management Group (January 2008) `http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF/`.

[12] Demers, F.N., Malenfant, J.: Reflection in logic, functional and object-oriented programming: a short comparative study. In: In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. (1995) 29–38

[13] Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing. PODC '87, New York, NY, USA, ACM (1987) 137–151

[14] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1**(1–2) (Oct 1997) 134–152

[15] Wasylkowski, A., Zeller, A., Lindig, C.: Detecting Object Usage Anomalies. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ESEC-FSE '07, New York, NY, USA, ACM (2007) 35–44

[16] The Eclipse Foundation: ECLIPSE website `http://www.eclipse.org/`.

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwing, J.: Aspect-Oriented Programming. ECOOP '97 - Object-Oriented Programming: 11th European Conference **LNCS 1241** (1997) 220–242

[18] Sun Microsystems: Dynamic Proxy Classes, Java SE Documentation (January 2010) `http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html`.