Enriching OSGi Service 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: Architecture description languages define component interfaces with sequential contracts, which allow for static analysis of method call sequences in component or service interactions. However, component and service platforms like OSGi for Java do not provide mechanisms for the specification or enforcement of such sequential contracts. Thus the contracts are only defined in the documentation which might be outdated when long-living systems evolve at the implementation level. This vision paper proposes to attach formal sequential models, in our case *interface automata*, to the interface definition of OSGi services, so that the modeling information is permanently and tightly coupled to the implementation. This enables consistent documentation, static analysis of component interactions at design time, and real-time enforcement of behavioural contracts at run time. By this means, component interactions can be seamlessly verified in long-living systems when components and their connections are added or changed over time.

1 Motivation

In component-based and service-oriented software architectures, systems consist of components which are represented to their context by interfaces that describe their functionality. The implementation of this functionality is hidden to the environment, and accessed through the published interfaces. Beugnard et al. [BJPW99] 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 of current programming languages usually allow for defining syntactic level interfaces, i.e. the methods available for invocation. For defining behaviour level interfaces, programming language extensions like the Java Modeling Language (JML) [LC05] can be used.

Components often assume that their methods are called in certain orders, e.g. when a component requires authentication before permitting the invocation of further methods.

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

While these synchronization level interface descriptions are important for the interface's users, they are usually not explicitly and formally stated in current interface descriptions of programming languages like Java and their component and service frameworks, like OSGi [OSG05]. Hence, instead of focusing on the business aspects only, methods must validate the current state to prevent errors and possibly inconsistent data. This is usually performed at the beginning of the code in the method body, in which object attributes are inspected for reasonable values, and exceptions are thrown when an invalid state was identified.

Several formal techniques exist for describing sequential contracts of interfaces. As an instance of them, *interface automata* [dAH01] define permitted call sequences based on state machines. These models and related tools can be used for verifying component or service interaction with respect to the sequential description at design time. In this case, the models reside in the software documentation. However, when designing architectures for long-living software, it must be considered that the documentation might be unavailable or become inconsistent when the software evolves. In this contribution we propose to attach interface automata directly to OSGi service interface descriptions, thus enriching the component interfaces with *synchronization level* information. At design time, the models can be used for static analysis regarding component interactions. At run time, the model information attached to the interfaces can be used to ensure that only valid sequences of method calls are performed. This is of interest especially for system evolution, when components are changed or added that are not known beforehand, since valid interaction sequences can be ensured permanently once they are defined in the program code.

This paper is organized as follows: In section 2 we show how to enrich OSGi service interfaces with interface automata definitions. The use of these models at design time and run time is presented in section 3. A short overview of related work is given in section 4 before the paper is concluded in section 5.

2 Approach

Interface automata [dAH01] are essentially finite state machines with input, output, and internal actions, where each input defines a received method call and each output defines an outgoing method call. Systems of interface automata communicate via input and output actions, so that service and component interactions can be described with sequential contracts. Methods of interfaces are input actions that are executed when the method is called. These actions trigger transitions (called *steps*) in the automaton. If the interface automaton has a step with the called method in the current state, the call is permitted.

OSGi services are components that can be described with interface automata, since they publish functionality with a Java interface and thus a set of methods. To reach our objective of coupling sequential modeling information to these interfaces, we use meta data annotations in Java source code [Sun04]: The annotation `@InterfaceAutomaton` can be attached to OSGi service interfaces and contains information about the sets of states and steps. Each state is represented by a nested annotation `@State` denoting the state's name. One of the states is marked as initial, representing the entry point of the sequential

```
@InterfaceAutomaton(
  states={ @State(value="loggedOut",initial=true),
           @State("loggedIn") },
  steps={
    @Step(from="loggedOut", action="login(java.lang.String,java.lang.String)",
         to="loggedIn"),
    @Step(from="loggedIn", action="execute(java.lang.String)", to="loggedIn"),
    @Step(from="loggedIn", action="logout()", to="loggedOut")} )
public interface IShell {
  public void login(String user, String password);
  public void logout();
  public String execute(String command);
}
```

Listing 1: An interface automaton definition for a Java interface

contract. Each step is represented by a nested annotation `@Step` referencing an originating state, a Java method signature as input action, and a target state. Since we consider provided interfaces only, each action is an input action in this context.

Listing 1 shows a Java interface with an interface automaton annotation. This automaton is defined by 2 states `loggedOut` and `loggedIn`, 3 input actions (the interface method definitions), and 3 corresponding steps. The contract defines that the first method to be called is `login`, before the methods `execute` and finally `logout` may be called.

3 Usage of the Models

When interface automata are attached to OSGi services as described above, component interaction can be verified with appropriate tools, thus ensuring valid invocation sequences even if the participating components change. The usage of annotations for embedding the interface automaton into the program code has several advantages: Annotations are typed meta data that can be accessed programmatically at design time and run time of the software, as the automaton definition is directly embedded into the source code. Thus the modeling information can be used for different kinds of verification.

3.1 Design Time

In OSGi, some Java interfaces are service descriptions used by the framework during execution. Since the models are not only filed in the documentation, but coupled to these service descriptions, they are always available with the source code when long-living systems evolve, even when the documentation is outdated. The static nature of annotations allows for graphical editing for an easier understanding of more complex automata because they can be read and written by an according editor as any other notation.

Since the Java methods define the actions in the source code unambiguously, it is possible to perform static code analysis for component interactions. This requires to extract models

that describe how services are used by other components. Approaches for this already exist, e.g. in the JADET tool [WZL07]; for our purpose, they must be adapted to extract calls to OSGi service interfaces from the control flow to create an interface automaton with output actions, which can then be matched with the automaton describing the service.

3.2 Run Time

Since the annotations are accessible at run time, they can be used in the OSGi Service Platform for monitoring and enforcing the compliance of calls to service interfaces with their behavioural contract. To accomplish this, a technique must exist that can observe and prevent method calls. Examples for such techniques are aspect-oriented programming (AOP) [KLM+97] and dynamic proxies [Sun10].

As a proof-of-concept[1] we implemented the integration for the OSGi implementation Eclipse Equinox [Ecl11a] in combination with Equinox Aspects [Ecl11b], which enables AOP in an OSGi environment. This proof-of-concept includes business components as services which are described by interfaces with interface automata definitions, and one *observer* component which observes and possibly rejects interface calls using AOP. The observer reads interface automata from the annotations first when service components with interface automata descriptions are deployed. Subsequently, all method calls to these interfaces are verified with respect to the model. When the sequential contract is broken, the observer will reject calls.

Figure 1 visualizes this concept. The left side shows the interaction of two business components without the explicitly modeled sequential contract. As explained in the motivation, the business components must themselves check the permission of the call. The right side shows the same scenario with the proposed concept 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 section 2.

4 Related Work

The extension of interfaces with further information is subject to many publications. However, most of them focus on implementing behavioural contracts, i.e. pre- and postconditions for method calls. It is possible to emulate call sequence specifications using behavioural descriptions with dedicated internal state fields that are checked in preconditions and set in postconditions of method invocations. This approach is argued to be error-prone and hard to understand since the sequential contract is modeled indirectly [CP07].

Heinlein [Hei09] 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

[1] The prototype can be downloaded at http://www.s3.uni-duisburg-essen.de/index.html?op=view&id=342

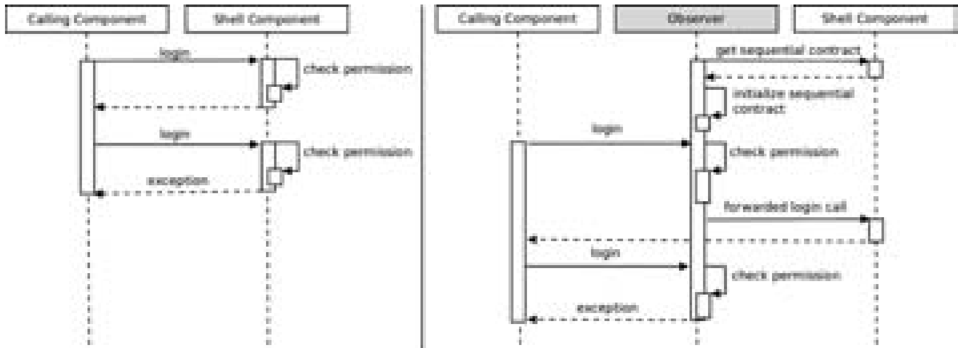

Figure 1: On the left side, the callee needs to check whether the calls respect the sequential contract. On the right side, these checks are processed by the contract enforcing component. The callee can focus on the business issues.

checks for the constraints. Our approach does not require changes in the language, and is thus easier to adopt. Pavel et al. [PNPR05] introduce a framework which allows to check and reject method calls to Enterprise JavaBeans [Sun08] components based on Symbolic Transition Systems. In their approach, the state checks have to be implemented manually or with a precompiler directly into the business methods. In contrast to our approach, the validation code is integrated in the business methods, which makes them harder to read and understand, especially when maintaining or evolving a legacy software.

5 Conclusion and Future Work

In current programming languages and frameworks, component or service interfaces are usually described structurally. This only includes the method signatures. However, 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 concepts are not reflected in current programming languages and frameworks. In this contribution we presented a concept for representing sequential contracts in Java for checking the compliance of a callers behaviour at run time, and for rejecting calls that break the sequential contract. We also presented how to integrate this concept in existing frameworks, using the OSGi Service Platform for a proof-of-concept.

As a benefit of the implementation, the business methods do not need to validate the component's state, but the development can focus on the business requirements. This eases program comprehension in software maintenance and evolution especially in long-living systems, when documentation might be absent, because the sequential contracts are not defined in the documentation, but directly in the code.

As future work we plan to build tools for developing the automata graphically, synchronizing them with the code, and performing static analysis as introduced above. This allows for validation of the approach in large development projects. We will also improve the no-

tation for interface automata in code. In addition, we plan to integrate interface automata in further component frameworks and service platforms, e.g. Enterprise JavaBeans. As another step, we will statically verify the compliance of sequential contracts in complex architectures, using input and output interfaces for components.

References

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[CP07] Yoonsik Cheon and Ashaveena Perumandla. Specifying and Checking Method Call Sequences of Java Programs. *Software Quality Journal*, 15(1):7–25, March 2007.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120, 2001.

[Ecl11a] Eclipse Foundation. Equinox, 2011. `http://www.eclipse.org/equinox/`.

[Ecl11b] Eclipse Foundation. Equinox Aspects, 2011. `http://eclipse.org/equinox/incubator/aspects/`.

[Hei09] Christian Heinlein. Advanced Thread Synchronization in Java Using Interaction Expressions. *Objects, Components, Architectures, Services, and Applications for a Networked World*, 2591/2009:345–365, 2009.

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

[LC05] G.T. Leavens and Y. Cheon. Design by Contract with JML. *Draft, available from jmlspecs. org*, 1:4, 2005.

[OSG05] OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*. IOS Press, Inc., 2005.

[PNPR05] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. 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*, pages 115–124. Springer-Verlag, 2005.

[Sun04] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[Sun08] Sun Microsystems, Inc. JSR 318: Enterprise JavaBeans™3.1 - Proposed Final Draft, March 2008. `http://jcp.org/en/jsr/detail?id=318`.

[Sun10] Sun Microsystems. Dynamic Proxy Classes, Java SE Documentation, January 2010. `http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html`.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 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, pages 35–44, New York, NY, USA, 2007. ACM.