Grey-Box Specifications for Object-Oriented Runtime Components

Yannick Welsch
Software Technology Group, University of Kaiserslautern
y_welsch@cs.uni-kl.de

Abstract

We present a formal specification technique for object-oriented program components based on their boundary message behaviour. Component specifications describe restrictions on the set of message traces for a component without referring to actual implementations. Finally, we provide a framework to link specifications with abstract states to their implementations.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Languages, Verification

1. Introduction

Specification techniques for object-oriented programs do not often scale very well. Whereas some techniques describe properties at the program (e.g. class) level other techniques have a model which has to be manually mapped onto the concrete program for verification and checking. The issues resulting are due to the fact that object-oriented programming does not provide the right semantical layer between single classes and large, complete, executable programs.

We introduce the notion of component at the program level and our specifications describe restrictions on component behaviours, which result directly from given component implementations. Specifications and implementations are related by the messages passing component boundaries. This allows various (behavioural) specification techniques to be used within our specification framework.

In the next sections, we shortly describe the related work, our component model, the relation of specifications to component behaviours and a framework for embedding concrete specification techniques. For a more detailed description we refer to [9].

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

2. Related Work

Specification techniques like JML [5] and Spec♯ [1] describe properties on classes / interfaces and their corresponding methods. However, no or very little[1] support for larger components is provided. Whereas earlier versions of JML only allowed for a descriptive nature of program behaviour, recent support for model programs [8] was added. However these model programs maintain a too strong mapping to the code in our opinion.

The concept of Trace-Assertions in Jass [6] is also similar to our trace-based approach. The main difference is that we specify the partial traces for components which can be large groups of objects, whereas their approach describes traces on the class instance level. Typestates [4] also relate concrete program states to abstract specification states but lead again to purely descriptive specifications. We reuse the idea of the Refinement Calculus approach [2] where our refinement relation is based on the valid message traces of a component implementation and its specification.

Our technique tries to combine the advantages of the approaches at the program and model level by using abstract states and the messages (types and values) of the underlying programming language.

3. Component Model

The components we consider are called boxes. A box is basically a runtime entity which aggregates an evolving set of objects. Box signatures define the type boundary for boxes in order to encapsulate implementation types of a box and decouple internal and external types, allowing a box implementation to be changed without changing the visible behaviour.

At the language level, we have box declarations and expressions to create new instances of a box. A box B with interface type I is declared as "box $B : I$". The box signature can be derived from I. To create a runtime instance of a box, we have to associate a given box declaration with an implementation. This is done using the "box B with C" syntax, where C denotes the implementation class compatible with B (subtype of I).

[1] Spec♯ has a notion of component, which is however too restricted for our purposes.

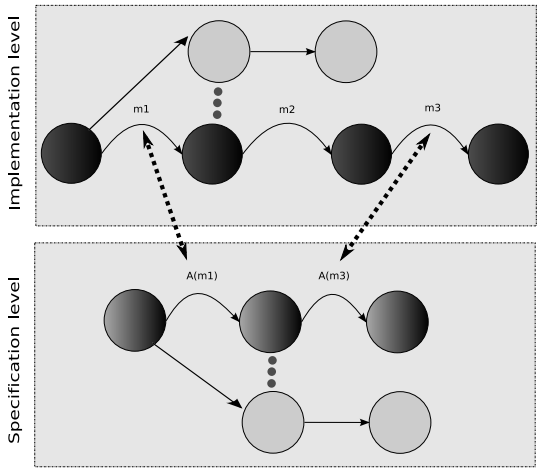

Figure 1. Linking of exectuable specification to implementation by messages occuring at box boundaries

Message Behaviour To get to a message oriented view where a program is characterized by the possible message traces occurring in a run, the program state space is partitioned into component state spaces. Method calls are transformed into sets of messages, which size depend on the number of box boundaries passed. Different types of messages are distinguished, inter- and intra-box messages, call and return messages, and for inter-box messages, their direction (in or out).

The behaviour of a component can then be defined as how it reacts (with which outgoing message) to a history of incoming messages. The exact mapping between standard (similar to Featherweight Java) and message semantics is covered in [7].

4. Specifications

Our concept of box specifications describes the boundary behaviour of a box by putting restrictions on the valid message traces passing the boundary of a box. Specifications are simply transition systems which can have generic abstract states. We leave it to a concrete specification technique (and language) to define these states. Transitions are triggered only by messages and state evolution is left again to a concrete specification handler. We call concrete specification handler a specification technique (with its appropriate language) and box specification the specification of a box for which there must be some concrete specification handler which can interpret it. We introduce a box specification declaration as "map spec $H < S >$ to B" which maps a concrete specification handler H initiated with a box specification S to a box B. We describe an initiated specification handler $H < S >$ as a pair of transition relation $\longmapsto_{H_S}$ and a value $initState_{H_S}$. The transition relation $\longmapsto_{H_S}$ maps the specification state $\mathcal{T}$ of a box to the next state $\mathcal{T}'$, depending on the message m occurring at the boundary.

In Figure 1 we illustrate how the message traces of the implementation are related to the specifications. By A we denote an abstraction function, as in most of the cases, our specifications only consider certain aspects of a message. As our specifications are able to define the external operation of a component, we call them grey-box specifications.

5. Implementation

We have implemented the component model for the Java programming language using the Java 5 annotation mechanism and bytecode rewriting. We provide an API for observing message sequences occurring at box boundaries and a framework to embed generic specification handlers. As proof of concept we have defined a custom specification language with Java-like syntax. Using our implementation, it is possible to check if, for a certain run, a box implementation satisfies its specification.

6. Conclusion

We defined the interaction of programs with their component specifications. This allows to give future component specification languages a precise semantics. Our specification technique results in a unique combination of abstract states at the specification level while using the types and values of the underlying programming language.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: an Overview. CASSIS, volume 3362 of *LNCS*, pages 49–69. Springer, 2005

[2] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. Lecture Notes in Computer Science. Springer, 1998

[3] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, 2005. Technical Report

[4] R. DeLine and M. Fähndrich. Typestates for Objects, 2004. In 18th ECOOP

[5] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML Reference Manual (Draft). Last update: October 2007 `http://tinyurl.com/jmlrefman`

[6] M. Plath. Trace-Zusicherungen in Jass: Erweiterung des Konzepts "Programmieren mit Vertrag", 2000. Master's Thesis

[7] A. Poetzsch-Heffter, J.-M. Gaillourdet, and J. Schäfer. Towards a Fully Abstract Semantics for Object-Oriented Program Components, 2008. `http://tinyurl.com/papbox08`

[8] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs. Iowa State Univ. TR #07-04A

[9] Y. Welsch. Grey-Box Specification and Runtime Testing of Object-Oriented Runtime Components, 2008. Master's thesis. `http://tinyurl.com/ywelschgb08`