Testing Distributed Component Based Systems Using UML/OCL

Achim D. Brucker Burkhart Wolff

Institut für Informatik, Albert–Ludwigs–Universität Freiburg, {brucker,wolff}@informatik.uni-freiburg.de

Abstract

We present a pragmatic approach using formal methods to increase the quality of distributed component based systems: Based on UML class diagrams annotated with OCL constraints, code for runtime checking of components in J2EE/EJB is automatically generated. Thus, a UML–model for a component can be used in a black–box test for the component. Further we introduce different design patterns for EJBs, which are motivated by different levels of abstraction, and show that these patterns work smoothly together with our OCL constraint checking.

A prototypic implementation of the code generator, supporting our patterns with OCL support, has been integrated into a commercial software development tool[1].

Keywords: *OCL, Constraint checking, EJB, J2EE, Design by Contract, Design pattern*

1 Introduction

The commercial success of CASE tools supporting the *Unified Modeling Language* (UML) [9], defined by Object Management Group (OMG), indicates that diagrammatic methods are a promising approach to introduce semi–formal techniques into conventional software engineering practice. Extended by the *Object Constraint Language* (OCL) [8], UML attempts to bridge the gap between semi–formal and formal techniques, even though OCL is still under development and many nitty–gritty problems of its formal semantics have not yet been settled.

On the technological front, a new challenge appeared some years ago in the shape of *component based systems*. The overall idea is to partition a system into parts (*components*) that are language and machine independent and can be connected via a network. Offering flexible migration to legacy systems and enabling reuse of code, component technologies are viewed in industry as a way of speeding up development and organizing systems with increasing size. However, the increasing complexity of such systems also increases the need for specification and validation. Thus, the component technologies like J2EE/EJB [10] or CORBA [7] open a new field for formal specification and its applications.

In this paper, we present such an application of formal specifications: namely dynamic constraint checking of preconditions and postconditions and class invariants generated from UML diagrams annotated with OCL constraints. Our main contribution is to provide *concepts*, *design patterns* and integrated constraint checking *code generation techniques* for UML/OCL-specifications of distributed J2EE/EJB-components. Our implementation in a CASE tool enables black-box testing of components and provides thus the technical basis for more advanced approaches such as systematic test case generation or formal refinement proofs.

Our paper is structured as follows: After a brief description of component based systems using J2EE and a description of UML/OCL, we will adopt common specification techniques to J2EE and provide design patterns for EJBs.

2 Component based distributed systems

Distribution is a key issue in modern system design. Almost every system using the Internet can be seen as a distributed system consisting of one or more servers and clients. In typical applications, the servers provide information or data management, while the clients collect user data and display results.

In the last years several middleware component standards for designing and implementing client server architectures were introduced. The most well known of these are the CORBA (Common Object Request Broker Architecture) from the OMG and J2EE/EJB from Sun Microsystems. Any of these standards provides a way for describing the interface of the distributed components, as used by the clients. Seen from the formal methods per-

[1]This work was partially funded by Interactive Objects Software GmbH (`http://www.io-software.com`) in collaboration with the Software Engineering Group at the University Freiburg.

spective, interfaces are just signatures, and it is natural to enrich them by formulae specifying the functional or the dynamic behavior; if these formulae have a well–defined semantics, it is possible to analyze and to reason over the expected behavior of a component.

In our setting with the J2EE/EJB middleware and UML/OCL, we have to consider that the EJB standard requests a split in the interface of the component into two parts: The *home interface* describing the functions for life–cycle management (such as object creation and destruction) of the EJB and the *remote interface* describing the functional behavior. The home interface and remote interface are implemented by the *bean implementation*. Together, these three parts build an *Enterprise Java Bean (EJB)*, the *distributed component* in the J2EE model. As we will see later, this interface splitting has a great impact on the organization of the specification and the black–box testing of the EJB.

3 UML/OCL: An overview

One of the more important diagram types of the UML is the *class diagram* showing the static structure of the software design; its main purpose is to illustrate the dependencies (any kind of associations, inheritance,etc.) of classifiers (components, classes, interfaces, templates,etc.) used in the system.

With version 1.1 of the UML standard, UML was extended by the textual formal specification language OCL, which is a classical logic with equality and undefinedness based on the signature of side–effect free UML methods. Since UML‘s type discipline offers built–in type constructors such as *set*, *seq* or *bag*, OCL enables to specify constraints on graphs of object instances. In the context of class–diagrams, OCL is used for specifying class–invariants, preconditions and postconditions of methods.

In figure 1, a UML class diagram showing a simple banking scenario is given. The functional behavior of the methods makeDeposit(), belonging to the class **Account**, is given by the specification of its preconditions and postcondition. From **Account**, a class **CreditAccount** is inherited, which allows only debts up to a specific credit limit; this is specified by a class invariant. Further, we model by an *association* that every instance of type **Account** (here **Account** or **Credit Account**) "belongs to" a **Customer**. The concept of associations with multiplicities (similar to "cardinalities" in E/R diagrams) can be understood as relations with certain constraints made explicit by appropriate OCL formulae. In general, an invariant at an association end (e.g. multiplicities) can be converted to class invariants on all opposite ends [8]. In our example, the multiplicities were transformed to the following OCL formulae:

```
context Customer
    inv: (1 <= self.accounts.size())
         and
         (self.accounts.size() <= 99)

context Account
    inv: (1 = self.owner.size())
```

Using invariants for associations, we can also describe if such a relation is partial, injective, surjective or bijective. In our example we would like to express that the associations belongsTo is surjective:

```
context Customer
    inv: self.accounts.forall(a | a.owner = self)

context Account
    inv: self.owner.accounts->includes(self)
```

This guarantees that every account a customer controls (particularly, which is in the set accounts) is owned by this customer.

4 Concepts for EJB-Specification

Based on existing techniques to handle individual OCL formulae, we will observe in the following subsections that the semantic relations between collections of OCL constraints annotated to parts of an EJB can be described as a data refinement. This observation leads to the development of a code generation scheme for constraint checking code of an individual EJB. In the next section, we will introduce the concept of an "extended Bean" pattern that offers the potential to extend this scheme to systems with *n* to *m* relations between home and remote interfaces on the one hand and EJB implementations on the other. This kind of systems is required by engineering practice.

4.1 General Principles

For standard Java programs, there is already an OCL constraint checking code generator [4, 5, 11] available, developed at the University of Dresden, that could be integrated into our work. It provides a type–checker for OCL–formulae and a collection of libraries for their execution. The problem lies in the right *integration* of this tool into the context of EJB.

The original version of EJBs does not provide a concept of a "specification". Thus, adding logical specification concepts to existing EJB technology needs some adaption, both on the syntactical (what are the right signatures of formulae drawn from EJB interfaces?) and

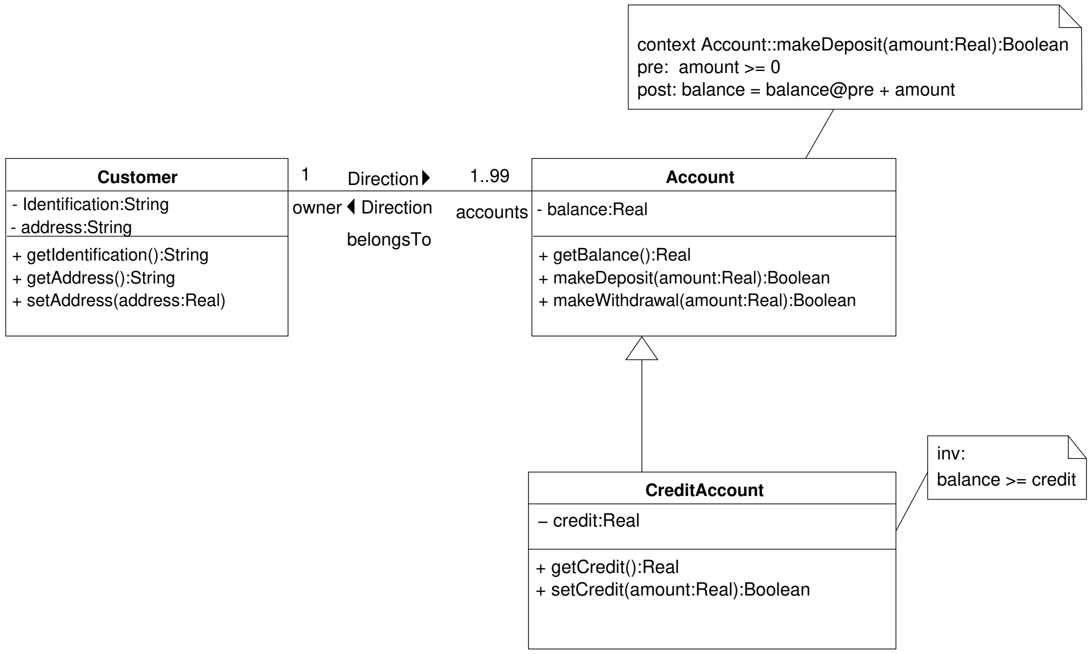

Figure 1: Modeling a simple bank scenario with UML

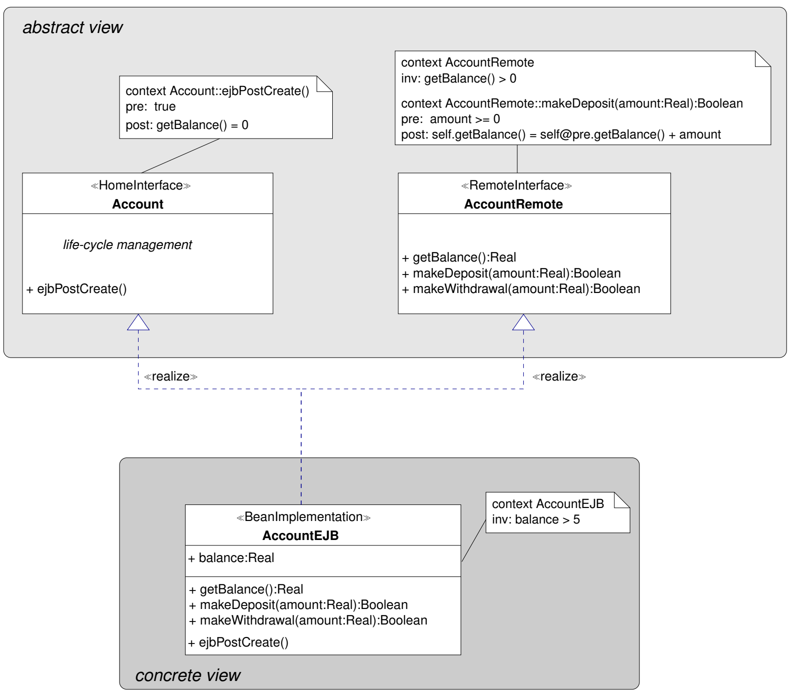

Figure 2: Abstract view and concrete view

on the semantical side (do the checks mirror the intended semantics?). For an EJB that consists of exactly one home interface H, one remote interface R and one bean–implementation I, we define its *abstract view* as the "union" of H and R. The abstract view represents an interface describing all methods accessible by the client. Further we define the *concrete view* (for an extract, see figure 2) of the EJB by its bean implementation (together with its signature consisting of its class declaration).

4.1.1 The syntactical side

With respect to the syntactical side we propose to merge the signature of H and R and enrich it by some accessor functions derived from I. More precisely, all side–effect free functions (called *query–functions* in the UML[2]) of (H,R) build together with the *canonical accessor functions* for the public variables of I the signature of the abstract view. As an example for a canonical accessor function, consider getBalance() in figure 2, which is a derived canonical accessor function for the public attribute balance of I; recall that accessor methods of the public attributes are special query–functions. Our proposal is motivated by the fact that the H and R are not independent from a specification point of view: For example, for specifying initial values of attributes such as balance we have to write a formula as postcondition of the function that creates the EJB. This "create–function" (e.g. ejbPostCreate()) clearly belongs to the home interface (where the life–cycle is described) whereas the accessor method for the public attributes belongs to the remote interface. This shows, that the partitioning of the interface of an EJB, as stated by the J2EE standard [10], can not uphold in a specification.

4.1.2 The semantical side

With respect to the semantical side, we already observed that the organization of EJBs suggests the distinction of an abstract view "implemented by" a more detailed concrete view. The latter may provide private variables, more methods, stronger invariants, and weaker preconditions and stronger postconditions as the former one. In our example in figure 2, the concrete view invariant requires balance to be larger than 5, while the abstract view relaxes this condition to balance>0. In the community of formal methods, the relation between abstract and more concrete views on a system and their semantic underpinning is well–known under the term *refinement*. Various refinement notions have been proposed (As for Z, see [12] for example). In our setting, we chose to use only a very simple data refinement notion which requires that any formula of the abstract view is implied by the formulae of the concrete view. Of course, following the approach taken in this paper, we do not attempt to formally prove such a relationship. Rather, we will generate code for runtime–checking the formulae (constraints) both on the abstract and the concrete view. Thus,

1. if only violations against abstract view constraints (but not concrete ones) occur, we can conclude that the abstract view is *not* a refinement (as it should be),

2. if only violations against the concrete view constraints occur (but not the abstract ones) the specification of I is too tight for its purpose.

On this basis, coding constraint checks is straight forward: formulae of the abstract and concrete view are converted to check code that is executed at the entry and/or the exit of the method bodies in the implementation; preconditions only at the entries, postconditions only at the exits, and invariants at both[3]. An obvious exception is made when entering or leaving object creation or destruction methods. Note that our coding scheme results also in constraint checks for internal (e.g. recursive) method invocations; a naive coding scheme based on wrappers of an interface would behave differently.

5 Design Patterns for Enterprise Java Beans

While modeling distributed systems using J2EE/EJB, there often arises the need for a more detailed model of the internal structure of an EJB as intended by the EJB standard [10]. Driven by this need, we suggest to extend an EJB by additional information to an *extended EJB* pattern. In a CASE tool, these patterns form the basis of a "technological mapping", i.e. a mapping of a pattern to a specific EJB implementation.

For example, in an extended EJB we will allow more than one bean implementation or more remote interfaces and show how to handle constraints during the mapping to standard EJB technology. Because an EJB is represented by a (H,R,I) triple, this boils down to the question of constructing (H,R,I) triples from extended EJBs.

In the following, we will discuss three different patterns.

[2] These functions have the UML attribute isQuery set to true.

[3] In the UML standard, it is required to check invariants "at any time"; we deliberately relaxed this requirement for both practical and conceptual reasons and treat invariants more like "loop–invariants" allowing intermediate states *inside* an implementation violating the invariant.

5.1 The CompactBean Pattern

The CompactBean pattern (see figure 3) is directly motivated by the highest possible level of abstraction of an EJB: the *abstract view*. This pattern allows an easy way to develop EJB applications, by abstracting away all technical details.

Using the abstract view (i.e. the pair (H,R)) we can directly refer to the discussion of the last section. In this pattern, there is no possibility for the designer to annotate the interface of I with OCL formulae. Therefore we can only check the abstract view at runtime by generating constraint checking code directly into the implementation.

5.2 The ExpandedBeanHome Pattern

Motivated by the need to provide technologically optimized (different) bean implementations (and thus home interfaces) of the same remote interface, we suggest the ExpandedBeanHome pattern. Its necessity occurs, for example, when an extended bean should provide an optimized implementation for different runtime environments. Thus the CompactBean pattern allows the specification of an extended EJB composed of several pairs (H_j,I_j) and a unique remote interface R that is implemented by every bean implementation of this extended bean.

In this scenario the partitioning of the extended EJB into (H,R,I) triples is straight-forward: We extend every pair (H_j,I_j) by the same remote interface R. For every such triple an EJB is generated, thus the number of EJBs is equal to the number of home interfaces (and thus bean implementations).

In the ExpandedBeanHome setting, the designer is able to specify OCL formulae on several implementations, thus we have to check for every triple (H,R_j,I_j) that I_j is a refinement of (H_j,R). We implement this by embedding runtime checking code into every bean implementation.

5.3 The ExpandedBeanRemote Pattern

The ExpandedBeanRemote pattern is based on the idea of providing different ways of access to the same implementation. This can be useful for modeling security related controls. For example an EJB can implement a (unique) remote interface for every role it is interacting with. In this scenario the designer specifies a unique pair (H,I) and several remote interfaces R_j. We build the (H,R_j,I) triple by combining every remote interface R_j with the pair (H,I). For every such triple an EJB is generated, thus the number of EJBs equals the number of remote interfaces.

In the ExpandedBeanRemote setting, the designer is able to specify OCL formulae on the bean implementations, thus we have to check for every triple (H,R_j,I) that I is a refinement of (H,R_j). We implement this by embedding runtime checking code into every bean implementation.

6 Conclusion and Future Work

6.1 Related Work

Considering other distributed component technologies, there seems to be a common understanding for describing the interface of a distributed component. Whereas every technologies defines its unique syntax, the idea is the same: Every component is described by an "interface" which contains methods and attributes accessible from the client.

Looking at the widely used CORBA (Common Object Request Broker Architecture) [7], also defined by the OMG, a special language for describing the client accessible interface is defined: the IDL (Interface Definition Language). In contrast to EJB, CORBA does not regulate the internal structure of the component interface, thus the problems that are based on the partitioning of the abstract view in a home interface and a remote interface do not exist. Nevertheless a CORBA component has abstract view defined with the help of the IDL (which is seen by the clients) and a concrete view defined by its implementation. As in the case of EJBs, the concrete view is a real refinement of the abstract view. Also, in contrast to EJB, an interface of a CORBA component can allow the access to attributes directly. Hence, the compliance to the requirement of accessing attributes only via accessor–methods has to be checked in an additional step.

Plain CORBA does not guarantees serializability of transaction, therefore one has the whole problematic (concurrency, call–backs) of distribution while specifying and constraint checking. The J2EE architecture provides, in contrast to CORBA, some support for guaranteeing serializability and non–concurrency: By using only *non–reentrant* EJBs and container managed persistence, the EJB container guarantees the serializability of transactions. For example, when a call–back occurs the transaction is rolled back. This allows us to ignore, during specification, most of the problems that are normally caused by distribution. Therefor, under these prerequisites (and supposing a correct middleware implementation) the specification of large distributed systems is possible in a relatively easy manner.

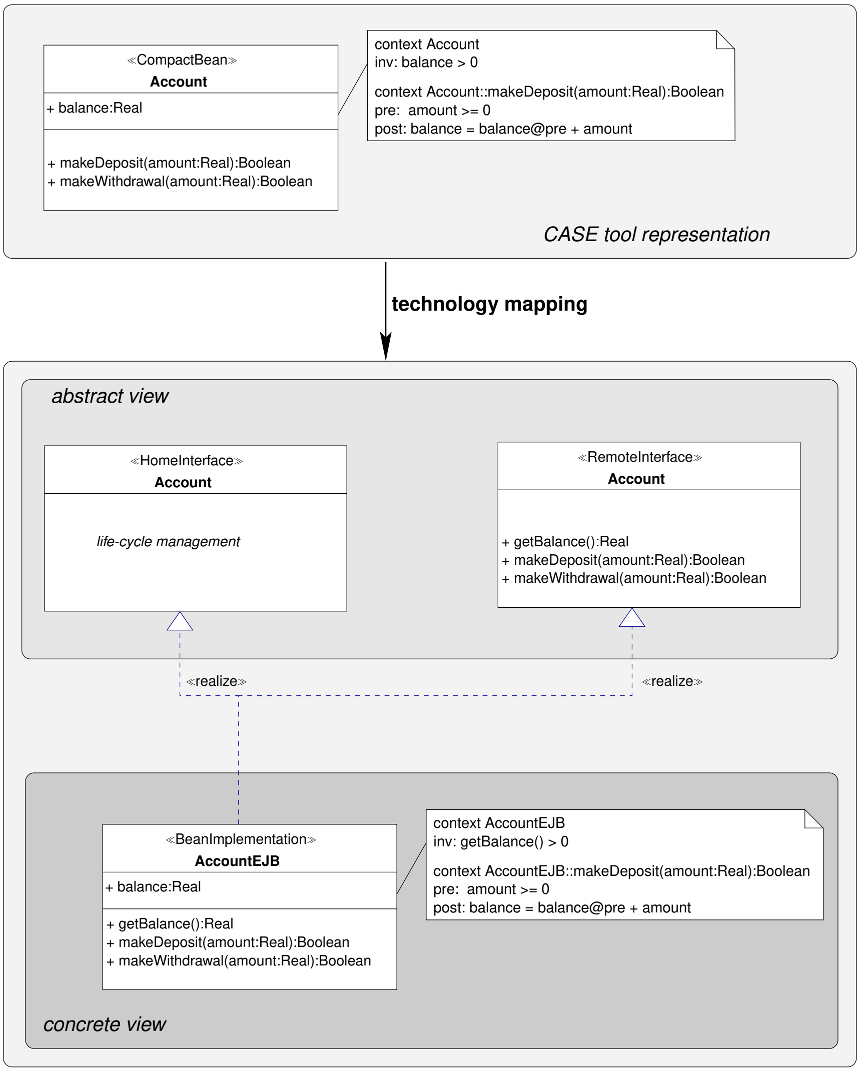

Figure 3: The CompactBean Pattern

6.2 Future Work

We presented a pragmatic approach to use diagrammatic specifications for dynamic testing of software components based on state–of the art component technology; an extended version of this paper is available as technical report [1].

Of course, post–hoc checking of violations of preconditions and postconditions and class invariants of software components is rather an a posteriori debugging method than a systematic a priori approach of analyzing a piece of software. However, we intend to complement our approach by a test–case generation technique similar to [6, 2, 3]. Thus a specification is also used to generate systematically test–cases along predefined testing hypotheses from the specification. Such a technique requires real theorem proving and a declarative (instead of an operational) semantics of OCL; a suitable embedding of OCL into Isabelle/HOL is in preparation. Such an embedding would also allow for a formal proof of refinement of an abstract view by the concrete one (allowing to omit the checks of the abstract level) or the verification of an implementation against the concrete level (allowing to omit the checks of the concrete level).

References

[1] A. D. Brucker and B. Wolff. Checking OCL Constraints in Distributed Systems Using J2EE/EJB. Technical Report 157, Albert-Ludwigs-Universität Freiburg, July 2001.

[2] D. Carrington, I. MacColl, J. McDonald, L. Murray, and P. Strooper. From Object-Z specifications to ClassBench test suites. Technical Report 98-22, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, Oct. 1998.

[3] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 268–284. Springer, Apr. 1993.

[4] OCL Compiler Suite. Available at: `http://dresden-ocl.sourceforge.net/`.

[5] F. Finger. *Design and Implementation of a Modular OCL Compiler*. Diploma thesis, Technische Universität Dresden, Mar. 2000.

[6] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with Isabelle. *Lecture Notes in Computer Science*, 1212:52–71, 1997.

[7] CORBA/IIOP 2.2 Specification, Feb. 1998. Available as: `ftp://ftp.omg.org/pub/docs/formal/98-02-01.pdf`.

[8] *Object Constraint Language Specification*, chapter 6. In Object Management Group [9], Feb. 2001. For further info see: `http://www.omg.org/uml`.

[9] OMG Unified Modeling Language Specification (draft), Feb. 2001. For further info see: `http://www.omg.org/uml`.

[10] Sun EJB 2 specification, 2000. For further info see: `http://www.javasoft.com/products/ejb/docs.html`.

[11] R. Wiebicke. Utility support for checking OCL business rules in java programs. dilploma thesis, Technische Universität Dresden, Dec. 2000.

[12] J. Woodock and J. Davies. *Using Z*. Prentice Hall, 1996.