Technical Report 157

Checking OCL Constraints in Distributed Component Based Systems

Achim D. Brucker Burkhart Wolff

July 2001

Institut für Informatik
Albert–Ludwigs–Universität Freiburg
Georges-Köhler-Allee 52
D-79110 Freiburg, Germany
Tel: +49 (0)761 203-8240, Fax: +49 (0)761 203-8242

{brucker,wolff}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{brucker,wolff}

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 design 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, Distributed Systems

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

Contents

1 Introduction

Developing state of the art software systems requires powerful software engineering tools that support the development process. In many development process models there is a common understanding that a process should consist of requirement analysis, specification, implementation and validation. At present, in industry and academia different emphasis is put on tool support; while in industry, the effective production of large software systems is a major concern, in academia correctness and quality of software systems is the predominant research goal.

Following the needs of industrial CASE tool supported development, the *Unified Modeling Language (UML)* [15] was proposed by the CASE tool industry, leading to a standardization process under the direction of the Object Management Group (OMG). UML is a graphical notation for object oriented data modeling and process modeling. In particular UML offers *class diagrams* for the description of the static system structure. In an industrial context, the generation of code stubs, which is possible out of an class diagram is already an substantial achievement. This stub generation reduces the amount of code that has to be written, and thus the development time, by a large amount.

On the academic side, research was mainly focused on the *validation* and *verification* part of the software development process. Thus, formal specification languages, having their roots in mathematical logic have been proposed. Prominent examples of such languages are Z [17] and VDM [10] which provide powerful techniques for refinement and validation, e.g. test data generation or even verification.

There are several reasons, that *formal methods* are gaining more and more attention in industry, e.g. caused by the increasing use of software in security and safety critical areas like e-commerce or life–saving systems (e.g. airbag systems). Also the customer protection is more and more improved through stricter laws about product liability. Among other things, these are the reasons for an increased interest, in industry and academia, in testing and verification techniques. Driven by this need, new specification languages have been proposed, that attempt to close the gap between industry and academia. One of these languages proposals is the *Object Constraint Language (OCL)*, a textual extension of the UML, promising the vision of easy use, object-orientation and the adaption of analysis methods well known from formal methods. There is already some tool support for OCL, for example the type checker and runtime constraint checker developed at the university Dresden [5].

On the technological front a new challenge appeared some years ago in the shape of component based systems. The overall idea of this approach is to partition the system in several parts (components), that are language and machine independent and can be connected via a network. The intensive reuse of code in form of components is accepted in research and industry as an way for speeding up development and enabling the organization of systems with increasing size. The increasing complexity of such systems enlarges the demand for quality assurance and — as a prerequisite — the need for specification and validation. Thus, the acceptance of component technologies, often called middleware, like J2EE/EJB from Sun Microsystems or the Common Object Request Broker Architecture (CORBA) [12], from the Object Management Group (OMG) is opening a new field for formal modeling techniques.

In this report we present an application of formal specification, in particular we show how to generate dynamic constraint checks from preconditions, postconditions, and class invariants specified in the context of UML class diagrams. Our main contribution is to provide *concepts* and *design patterns* the specification of distributed components. We integrate runtime constraint checking code generation techniques using UML/OCL based specifications of distributed J2EE/EJB components. Our implementation in a commercial 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.

The rest of this paper is structured as follows: First we will give an introduction to the used specification formalisms proposed by the OMG: the Unified Modeling Language and the Object Constraint Language. In the following section we give an overview of distributed systems in general and the J2EE architecture in particular. On this basis we present in section 4 a new level of abstraction for describing a distributed component. We will show, how this abstraction facilitates testing of distributed systems. Further we show, that this abstraction also is easing the design of such system, by introducing three different modeling styles, similar to design patterns [7]. In Section 6 we discuss some technology–details about CASE tool support for our our design patterns. Further we will outline some specialties of J2EE/EJB and their impact on specification and runtime checking. Finally we give a comparison with CORBA.

2 A Short Introduction to UML and OCL

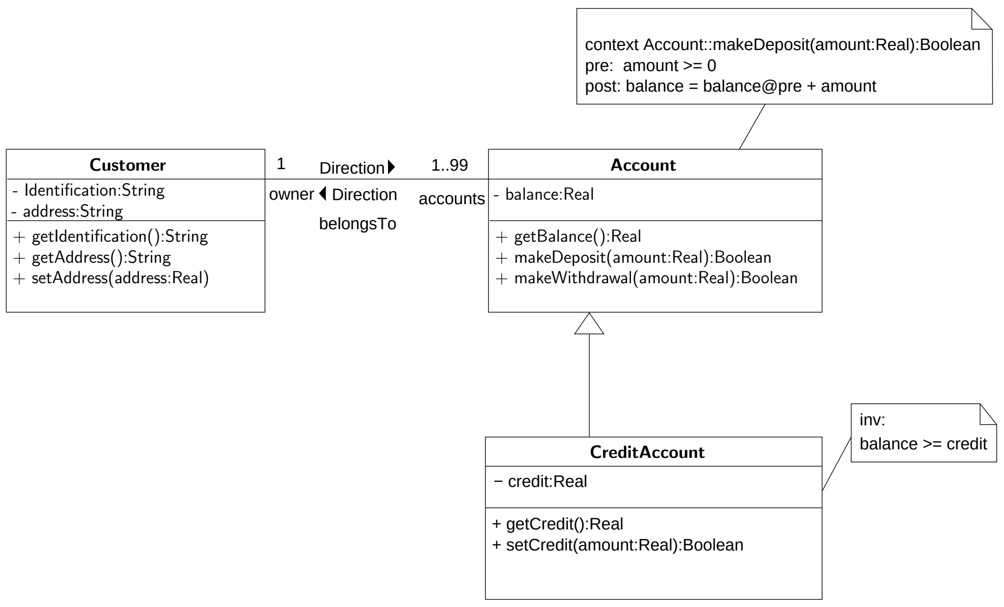
2.1 The Unified Modeling Language

The *Unified Modeling Language (UML)* [15, 16] is a standard from the Object Management Group (OMG), proposed as a semi-formal specification language for object oriented specification and design. The Object Constraint Language (OCL) [3, 19] is part of the UML standard since version 1.1. OCL is an attempt to extend the UML by a more formal specification language. UML itself contains several different diagram types, for example state-chart diagrams, collaboration diagrams and many others, whereas OCL is a textual specification language, which is suited for refinement of the UML diagrams by writing constraints on them. We will discuss some details of OCL in the next section.

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. . .) of classifiers (classes, interfaces, templates. . .) used in the system. In this paper will concentrate our effort on this diagram type. In the context of class diagrams, OCL is used to write class invariants, preconditions and postconditions of methods.

In figure 2.1 an example of an UML class diagram, annotated with OCL constraint is given. The class diagram shows a simple banking scenario. The functional behavior of the methods makeDeposit(), belonging to the class **Account**, is given by the specification of its precondition and postcondition. This class has several methods and one sub-class **CreditAccount** which is inherited from **Account**. The purpose of the class **CreditAccount** is introducing a new account type in our system, which allows to make debts only up to a specific credit limit. Further we have a class **Customer** with some attributes and methods. We also model a relation between **Account** and **Customer**, in UML such a relation is called an *association*: Every instance of the class. **Account** "belongs to" an instance of class **Customer**. In detail, the association belongsTo requires, that every instance of class **Account** is associated with exactly one instance of class **Customer**. It characterizes that every account has a unique owner. In the other direction, the association models that an instance of class **Customer** is related to a set of instances of class **Account**, the size of this set is limited to number between one and 99. In this aspect, the association characterizes, that a customer must have at least one account and no more than 99 accounts. Being of type **Account** means in this context to be an instance

Figure 2.1 Modeling a simple bank scenario with UML


context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
Customer
- Identification:String
- address:String
+ getIdentification():String
+ getAddress():String
+ setAddress(address:Real)
1
Direction
owner
Direction
belongsTo
1..99
accounts
Account
- balance:Real
+ getBalance():Real
+ makeDeposit(amount:Real):Boolean
+ makeWithdrawal(amount:Real):Boolean
CreditAccount
− credit:Real
+ getCredit():Real
+ setCredit(amount:Real):Boolean
inv:
balance >= credit

of class **Account** or class **CreditAccount**. The range limits for association are called *multiplicities* in the Unified Modeling Language.

2.2 The Object Constraint Language

The *Object Constraint Language (OCL)* is textual extension of the UML based on mathematical logic, and thus OCL is in the tradition of other data-oriented formal specification languages like Z [1, 17, 21] or VDM [10]. OCL is a side-effect free classical logic with equality and undefinedness that allows to specify constraints on graphs of object instances.

In the UML (and also in Java [11]) an interface is only an implementation obligation, that describes the methods a class has to provide. An interface can not describe any attributes. For specifying methods on the interface, there is a strong need for having access to attributes (the state) of classes. To solve this gap between the need of specification and the possibilities of interfaces we generate accessor methods for every attribute, taking its visibility (private, public, protected) into account. This can be done automatically, e.g. for every attribute a:type a method GetA():type is provided. As we see later, this solve also a technical problem in the area of runtime checking of invariants.

In the context of class diagrams, the Unified Modeling Language standard [15], allows for the specifications of invariants, precondition and postconditions for every classifier used. We restrict ourself to the following constraints:

Invariants for classifiers. Following the OCL standard [14, page 6-52], these invariants must hold *for all instances* of this classifiers *at any time*. In practice, this seems to be to strict, so we suggest the following semantics in the context of class diagrams:

- If the invariant belongs to a class or an attribute of this class, the invariant should hold before and after any method invocation of the corresponding class. We will call these invariants *class invariants*. A class–invariant can be expressed as a constraint, that is part of the precondition and postconditions of any method[1] of that class.
- If the invariant belongs to an specific method, the invariant should hold before and after the innovation of this method. We will call these invariants *method invariants*. In general method invariants describe the invariant part during execution of a specific method. This invariant is more special and often more strict than the class invariant. A method invariant can be expressed as a constraint that is part of the precondition and postcondition of that method.
- If the invariant belongs to an association we convert this invariant to a class-invariant at all opposite ends. For more details, see the discussion at the end of this section.

We assume that we never access class attributes directly, instead we always use accessor methods. This is necessary since controlling direct access to attributes is technically not feasible. In contrary it is technically simple to check any kind of constraint directly before or after a method call, e.g. through wrapping. Furthermore, this is no restriction when using Enterprise Java Beans, because we always have to communicate with the bean implementation through a remote interface or home interface. Because an interface cannot have any attributes, we have to provide the accessor methods anyway. See section 3.2 for a detailed discussion of these interfaces.

Preconditions for methods are describing the requirements on the program state before method invocation.

Postconditions for methods are describing the requirements on the state after method invocation, provided the precondition was fulfilled on the state before the invocation.

[1]An special exception must be made for the constructor and destructor of classes, e.g. the constraint has only to be part of the postcondition of the constructor and the precondition of the destructor, see section 7.1 details.

Further, the concept of associations with multiplicities in UML can be understood as abbreviation for relations with certain constraints, which can be expressed in OCL. It is easy to see, that the multiplicity expressed at the end of the association can be converted directly to an invariant at the other ends. For example for the association with the roles owner and accounts in the class diagram shown in figure 2.1 can be 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 (e.g. is in the set accounts) is owned by this customer.

3 A Short Introduction to Distributed Systems Using J2EE/EJB

3.1 The Java Enterprise Edition

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 Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) and the J2EE/EJB architecture from Sun Microsystems.

At the moment, our approach only considers the J2EE (Java 2 Platform, Enterprise Edition) architecture, which is widely used in the area of e-commerce. The J2EE based middleware is especially designed as an extension of the "normal" development kit of the Java language [11]. The J2EE architecture provides a wealth of services, like the Java Naming and Directory Interface (JNDI), Java Transaction Service (JTS) or the EJB container. The latter represent an integral part of the J2EE model and is discussed in the next section.

J2EE/EJB follows strictly the thin client approach; this means, that the main purpose of the client is presenting the results and all of the computing work is done on the server side. In general, J2EE proposes the classical three–tier client server architecture. The system is divided in a thin client side presentation layer (first tier) and a server side consisting of two tiers, see figure 3.1. In detail the server consist of a server side presentation layer which does the communication with the clients, the server side business logic layer (both layers constitute the second tier) in which the business model is implemented, and the information management and storage system, called enterprise information system (third tier). In this paper, we will concentrate on the server side, in particular the implementation of the business model (server side business logic layer).

Figure 3.1 The J2EE application model.

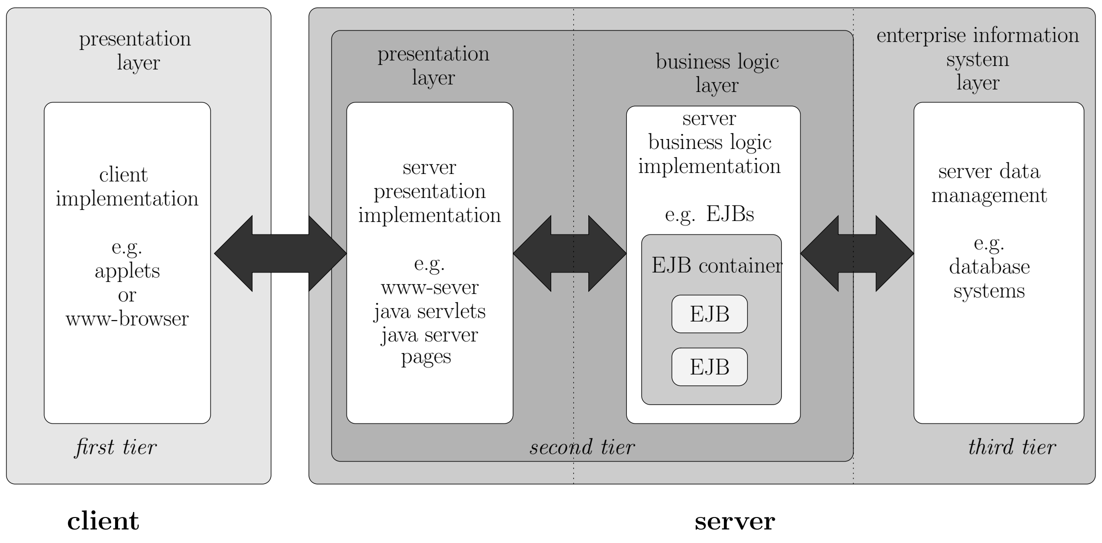
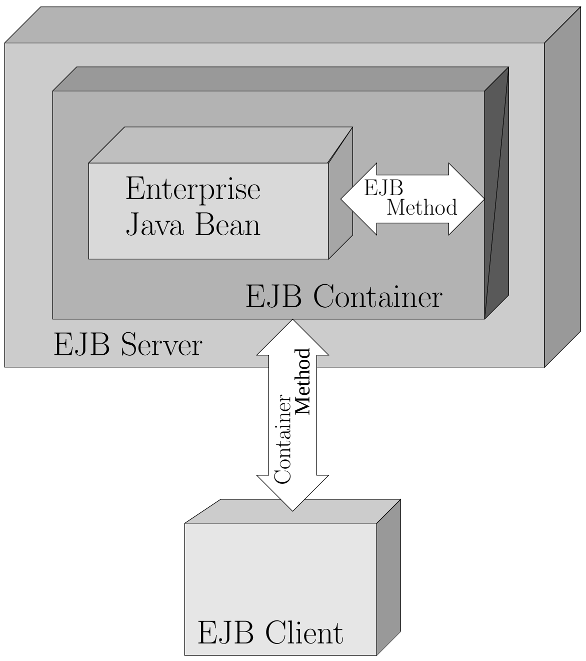


Figure 3.2 The basic organization of an EJB container.



3.2 Enterprise Java Beans

The distributed component in the J2EE architecture is called *Enterprise Java Bean (EJB).* The life cycle of an EJB is managed by an EJB container, which is the main part of the server side business logic layer.

The typical architecture of an EJB container is shown in figure 3.2; it consists of the EJB server on which one or several EJB containers are running. An EJB container is a runtime environment providing infrastructure for one or many Enterprise Java Beans. The EJB client is only able to communicate with one or more EJB containers; the client is not able to address an Enterprise Java Bean directly. The main task of the EJB container is to delegate, through special interfaces, the request of a client to the corresponding EJB. An Enterprise Java Bean consists mainly of:

remote interface: In this interface the business-methods are grouped together, most of the communication with the clients is done using this interface. It is recommended to call this interface after its use, e.g. **Account**.

home interface: In this interface the functionality for the management of the beans life-cycle is provided. Often the term *finder methods and factory methods* is used, to refer to the methods handling the life-cycle management. These methods are an integral part of the home interface. It is recommended by the EJB standard to use the postfix **Home** to mark the home interface, e.g. **AccountHome**.

bean implementation: This class implements the home interface and the remote interface. In principle, this is the core of the EJB. It is recommended to use the postfix **EJB** (using the postfix **Bean** is also usual) to name the implementation, e.g. **AccountEJB**.

Summarizing, the bean implementation is a *refinement* of exactly one home interface and one remote interface.

As every normal Java class, the bean–implementation can implement a lot more interfaces than the remote-interface and home-interface. The decision, which of these interfaces are used as remote–interface (or home–interface) is described in the *deployment descriptor* and is not fixed during compilation. Instead, this binding is done in a special step (called deployment) before installing the EJB in its container (runtime environment).

4 Concepts of an EJB–Specification

Based on existing techniques to handle individual OCL formulae, we will observe in the following sections 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", as a kind of design 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

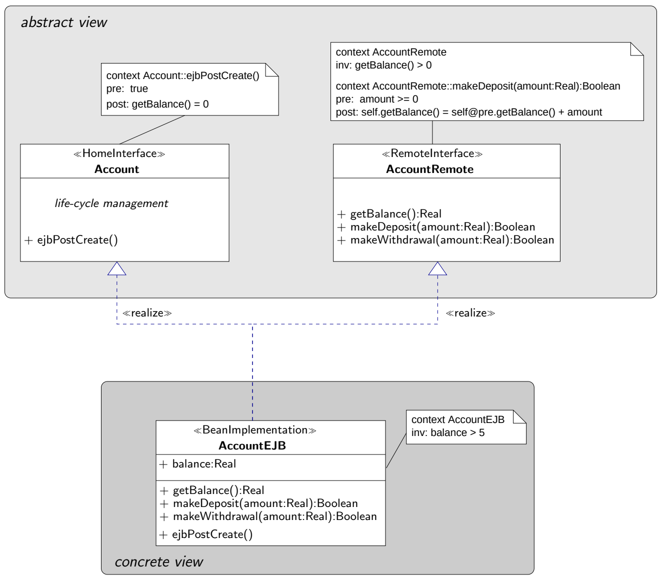
The original version of EJBs does not provide a concept of a "specification" of an EJB. Thus, adding logical specification concepts to existing EJB technology needs some adaption, both on the syntactical (what are the right signatures of formulae drawn from EJB interfaces?) and on the semantical side (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* of the signatures 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* (see figure 4.1) of the EJB by its bean implementation (together with its signature consisting of its class declaration). The bean implementation may have further variables and methods with private or protected visibility, hence the concrete view is much more detailed than the abstract view defined by (H, R).

4.1.1 The Syntactical Side

With respect to the syntactical side, we propose to join 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[1]) of (H, R) build together with the canonical accessor functions for the public variables of I the signature of the abstract

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

Figure 4.1 Abstract view and concrete view



view. As an example for a canonical accessor function, consider getBalance() in figure 4.1, 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(), see section 7.1) 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 [18], can not uphold in a specification. But considering the properties of the underlying middle–ware architecture, we can exploit the fact, that the J2EE/EJB standard guarantees the disjointness of the signatures H and R. This justifies our method to construct the signature of the interface.

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 precondition and stronger postconditions as the former one. In our example in figure 4.1, 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 [21] for example). In our setting, we chose to use only a very simple data refinement notion which requires that any formula describing a postcondition of the abstract view is implied by the formulae describing the corresponding postcondition of the concrete view. Also the preconditions of the concrete view have to be implied by the preconditions of the abstract view[2]. 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.

Recalling the two widely accepted testing technologies: White–Box–Testing and Black–Box–Testing, we follow the Black–Box scenario. This provides two posibilities for generationg testing code, for the abstract view (H, I) or for the concrete view I.

For an implementor, the latter technique is probably the preferable one, while for a developer of a piece of software build of EJB based components (which could be only available as binary), the former technique is the preferable. Of course, if the code of the implementation is available, the code generator can offer mixed strategies by conjoining the constraints of both views.

[2]Keep the direction of the implication for preconditions and postconditions in mind!

[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 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 prescribed by the EJB standard [18]. Driven by this need, we suggest to extend an EJB by additional information to an *extended EJB* pattern. In a CASE tool, these extended EJBs 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 sections, we will discuss three different design patterns substantiating the idea of extended EJBs.

5.1 The CompactBean Design Pattern

The *CompactBean* design pattern (see figure 5.1) 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 Design 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 (see figure 5.2 for details). 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.

Figure 5.1 The CompactBean Design Pattern

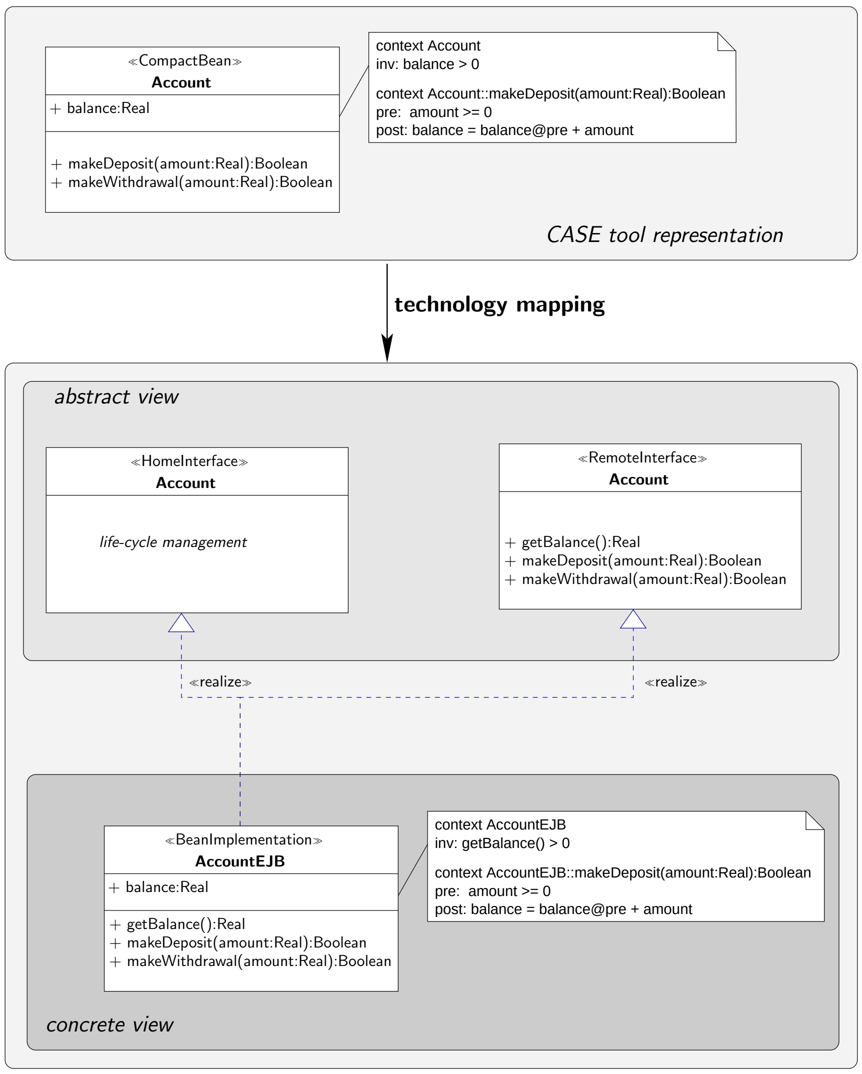
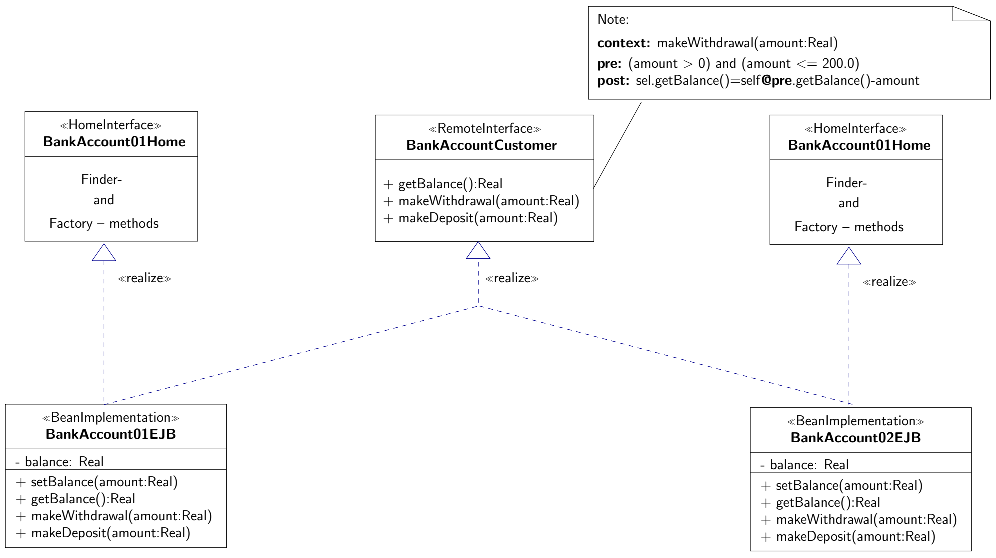

≪CompactBean≫
Account
+ balance:Real
+ makeDeposit(amount:Real):Boolean
+ makeWithdrawal(amount:Real):Boolean
context Account
inv: balance > 0
context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
CASE tool representation
technology mapping
abstract view
≪HomeInterface≫
Account
life-cycle management
≪RemoteInterface≫
Account
+ getBalance():Real
+ makeDeposit(amount:Real):Boolean
+ makeWithdrawal(amount:Real):Boolean
≪realize≫
≪realize≫
≪BeanImplementation≫
AccountEJB
+ balance:Real
+ getBalance():Real
+ makeDeposit(amount:Real):Boolean
+ makeWithdrawal(amount:Real):Boolean
context AccountEJB
inv: getBalance() > 0
context AccountEJB::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
concrete view

Figure 5.2 The ExpandedBeanHome pattern: Several bean implementations realizing the same remote interface


Note:
context: makeWithdrawal(amount:Real)
pre: (amount > 0) and (amount <= 200.0)
post: sel.getBalance()=self@pre.getBalance()-amount
≪HomeInterface≫ BankAccount01Home
Finder- and Factory – methods
≪RemoteInterface≫ BankAccountCustomer
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)
≪HomeInterface≫ BankAccount01Home
Finder- and Factory – methods
≪realize≫
≪realize≫
≪realize≫
≪BeanImplementation≫ BankAccount01EJB
- balance: Real
+ setBalance(amount:Real)
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)
≪BeanImplementation≫ BankAccount02EJB
- balance: Real
+ setBalance(amount:Real)
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)

In this scenario the partitioning of the extended EJB into (H, R, I) triples is straightforward: 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).

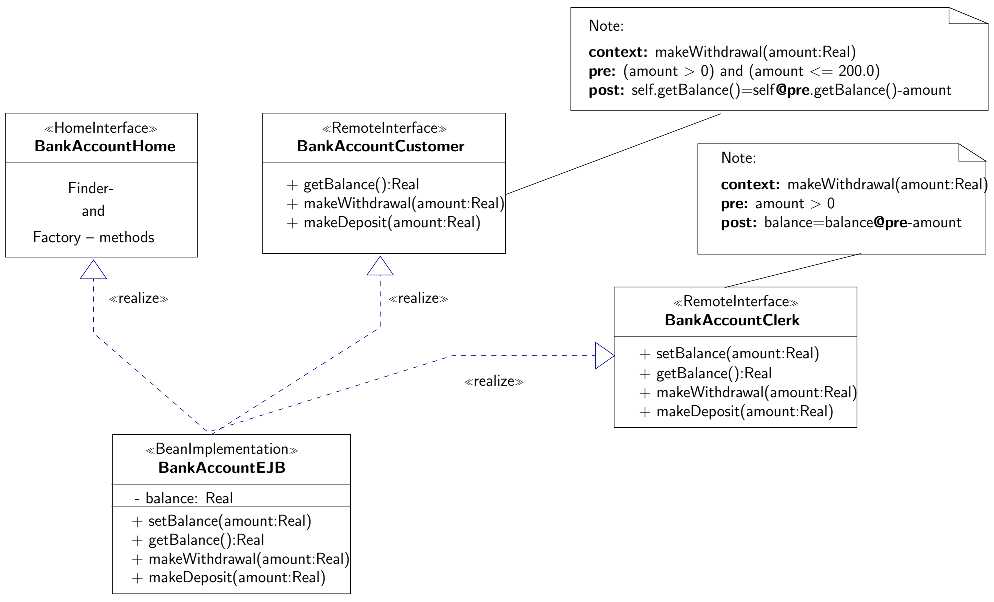
In the ExpandedBeanHome setting, the designer is able to specify OCL formulae on several implementations, thus we have to check for every triple (H, R_j, I_j) that I_j is a refinement of (H_j, R). We implement this by embedding runtime checking code into every bean implementation.

5.3 The ExpandedBeanRemote Design Pattern

The *ExpandedBeanRemote* pattern is based on the idea of providing different ways of access to the same implementation (see figure 5.3. 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

Figure 5.3 Several remote interfaces implemented by the same bean implementation


Note:
context: makeWithdrawal(amount:Real)
pre: (amount > 0) and (amount <= 200.0)
post: self.getBalance()=self@pre.getBalance()-amount
≪HomeInterface≫
BankAccountHome
Finder-
and
Factory – methods
≪RemoteInterface≫
BankAccountCustomer
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)
Note:
context: makeWithdrawal(amount:Real)
pre: amount > 0
post: balance=balance@pre-amount
≪realize≫
≪realize≫
≪RemoteInterface≫
BankAccountClerk
+ setBalance(amount:Real)
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)
≪realize≫
≪BeanImplementation≫
BankAccountEJB
- balance: Real
+ setBalance(amount:Real)
+ getBalance():Real
+ makeWithdrawal(amount:Real)
+ makeDeposit(amount:Real)

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 Tool Integration

For standard Java programs, there is already an OCL type checker [6] and constraint checking code generator [20] available, developed at the University of Dresden. Based on these tools, we prototypically integrated OCL support for the CompactBean approach into the commercial CASE tool ArcStyler [9]. Beside the OCL specific details we discussed in the last section, we present in this section some general strategies for integrating our design patterns for EJBs smoothly into a CASE tool environment.

6.1 The General Structure of a CASE Tool

It is convenient to broaden the scope of our discussion from the issue of code generation to its integration into the basic framework of a conventional CASE tool, see figure 6.1 for an overview. We have a repository R, that is able to store any kind of UML objects and diagrams. Whereas in the following discussion we only use class diagrams. For data exchange reasons, we also haven an export/import mechanism for files in the XML Metadata Interchange (XMI) [13] format, an XML based format for representing UML diagrams which is also defined in the UML specification [15]. We have an partial bijective function $\mathit{import} : \mathrm{XMI} \to R$ which imports the content of the XMI file in our repository, whereas the inverse function import^{-1} exports our repository R in the XMI format. Moreover, we need to display the content of our repository, for example during editing, on some presentation data D. This is done by a partial function $\mathit{view} : R \to D$. Further, we need some function $\mathit{update} : D \to R$ for updating the repository with the changes made in the displayed diagram. The code generation, which also maps our design patterns to a concrete technology, is implemented by some partial function $\mathit{code} : R \to C$ that maps the content of the repository to Java code based on J2EE.

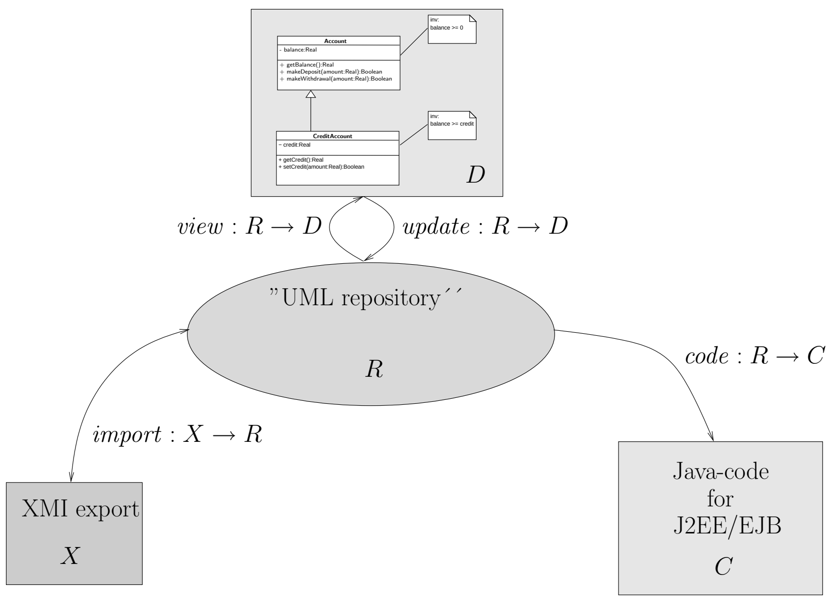
6.2 Support for our Design Patterns

6.2.1 Supporting the CompactBean Pattern

For supporting the *CompactBean design pattern* within an generic CASE tool, we have to consider the following issues:

- For representing a *whole* EJB, we introduce the stereotype ≪CompactBean≫.

Figure 6.1 The general structure of a CASE tool.



- The classifier with stereotype ≪CompactBean≫ contains all attributes and member-functions that are necessary to implement the business model.

- There is no need to model the life cycle management explicitly, but it is possible to do so, by introducing finder– and factory methods, following the naming standards prescribed by the Enterprise Java Bean specification [18].

Using the CompactBean pattern, the details of the EJB, namely its remote interface, home interface and the bean implementation are abstracted away. This mapping from our abstract representation to a specific technology (illustrated in figure 5.1) is done automatically during code generation. In the general CASE tool diagram (see figure 6.1) only the function *code* is affected: For every *CompactBean* a corresponding remote interface containing the declarations of the business methods and accessors-methods for every public attribute of the CompactBean is generated. Moreover the home interface and a skeleton of the bean implementation, which contains implementations for the finder methods and factory methods, is generated.

In the ≪CompactBean≫ pattern, the generation of constraint checking code is straightforward: constraints attached to the ≪CompactBean≫ classifier are mapped to checking code in the bean implementation. This reflects intention of the designer, because in this scenario he will "identify" a Enterprise Java Bean by their bean implementation.

6.2.2 Supporting the ExpandedBeanHome Pattern

The *ExpandedBeanHome* pattern allows the use of several pairs of home interfaces and bean implementations (see figure 5.2). Using this design pattern have to consider the following issues:

- For representing the extended EJB we introduce the stereotype ≪ExpandedBean-Home≫. Additionally stereotypes for representing the internals of the EJB (≪RemoteInterface≫, ≪HomeInterface≫, ≪BeanImplementation≫) are provided.
- We distinguish the levels of abstraction described in the last section, namely the abstract view and the concrete view.
- The different abstraction views should be supported by additional view and update functions.
- We provide function *expand* to support the conversion of a CompactBean into an ExpandedBeanHome; this enables better tool support.
- Of course a new code generation scheme is required.
- As in the CompactBean pattern, the ExpandedBeanHome pattern is neutral with respect to the XMI export and import.

For a valuable support of OCL constraints within a CASE tool using the Expanded-BeanHome design pattern, we have to consider the places where OCL constraints should be written. In particular we have to think about the "operational semantics" of constraints attached to the remote interface or at any of the several bean implementations, respectively the remote interfaces. In the following, when we speak about attaching an constraint to an implementation of interface, we describe the place, where the constraint is specified. For example, if the constraint is a postcondition of a method the term "attaching to a interface" should be understood as specifying the constraint as postcondition of method m in the context of the specific interface. We propose the following strategy for using constraints using the ExpandedBeanHome scenario:

remote interface: In this approach, the remote interface is unique for an EJB, therefore all the OCL constraints attached to this classifier should hold for all instantiations of that EJB. The constraint attached to the remote interface should describe the business model in greater detail, but only OCL constraints are allowed, that have a concrete semantics in the context of the remote interface, e.g. all classifiers used have to be declared in that interface (except query-functions, as described in earlier).

home interface: The home interface is unique for every bean implementation and can therefore contain implementation specific constraints together with “global” constraints (e.g. for the finder methods) that should hold true for all instantiations of that EJB. As in the case of the remote interface, all classifiers used in the OCL constraint have to be declared in that interface.

bean implementation: Constraints added to an bean implementation should hold only for that specific implementation, therefore constraints, that should hold for all instantiations and all implementations of that EJB have to be attached on all bean implementation of that Enterprise Java Bean

6.2.3 Supporting the ExpandedBeanRemote Pattern

The *ExpandedBeanRemote* design pattern allows the use of several pairs of home interfaces and bean implementations (see figure 5.3). Using this pattern have to consider the following issues:

- For representing the whole EJB we introduce the stereotype ≪ExpandedBeanRemote≫. Additionally stereotypes for representing the internals of the EJB (≪RemoteInterface≫, ≪HomeInterface≫, ≪BeanImplementation≫) are provided.
- We distinguish the different levels of abstraction described in the last section: the abstract view and the concrete view.
- The different abstraction views should be supported by additional *view* and *update* functions.
- We provide function $expand'$ to support the conversion of a CompactBean into an ExpandedBeanRemote, this enables better tool support.
- A new code generation scheme is required.
- As in the CompactBean pattern, the ExpandedBeanRemote pattern is neutral with respect to the XMI export and import.

As in the ExpandedBeanHome pattern, the ExpandedBeanRemote pattern requires a short discussion about the “operational semantic” of constraints attached to one of the interfaces or the implementation. In detail we propose for the ExpandedBeanRemote design pattern:

remote interface: In this approach, the remote interface describes a special “way to access” the EJB. Therefore constraint attached to this interface should only describe the business model of this special access in detail.

home interface: The home interface is unique for every bean implementation and can therefore contain implementation specific constraints together with “global” constraints (e.g. for the finder methods) that should hold for all instantiations of that EJB.

bean implementation: Constraints added to an bean implementation should hold for all EJBs generated on the basis of the specification of the extended Bean used. Therefore, OCL formulae describing parts of the business model that should hold for all “ways to access”, must be attached on the bean implementation.

7 Technological Details

7.1 Checking OCL Constraints using EJBs

When generating constraint checking code for J2EE/EJB, we have to take care of some specialties of this middleware architecture. First we have to consider that an EJB has no constructor, instead we have create–, finder– and remove–methods, controlling the life cycle (e.g. activation, passivation or destruction) of the EJB. Technically, we can handle these methods like any "ordinary" method, with one exception: The J2EE architecture restricts the type of exceptions, that can be thrown in such a method, see [18] for details. We have to respect these restrictions, when we decide to throw an exception whenever a OCL constraint is violated. We suggest checking invariants of Enterprise Java Beans, concerning these methods controlling the life cycle of the EJB, in the following manner:

ejbCreate() is called whenever an EJB is created. Here we suggest not to check any invariants, because after an successful creation of the EJB it is guaranteed, that ejbPostCreate() is called and if the creation fails, then checking invariants is meaningless.

ejbPostCreate() is called whenever an EJB is successfully created (it is called directly after the call of ejbCreate()). To guarantee that the invariants holds after its creation we must check the invariants in the postcondition of this method.

ejbRemove() is called whenever an EJB is deleted from memory, so we can handle it like an "normal" destructor, e.g. the invariants are checked in the precondition of ejbRemove().

ejbActivate() is called whenever a EJB is activated, therefore we handle it like a kind of constructor. This means the invariant is checked in the postcondition of ejbActivate(). This guarantees that the invariants holds, when a client accesses the EJB for the first time after activation.

ejbPassivate() is called whenever a EJB is passivated, therefore we handle it like a kind of destructor. That means the invariant is checked in the precondition of ejbPassivate(). This guarantees that only an EJB is passivated, whose invariants holds.

ejbLoad() is used for resets the bean state to a specific state stored in the underlying database. Here we also suggest checking the invariant in the postcondition of the method call, like ejbActivate().

ejbStore() is used for writing the beans state into the underlying database. Here we suggest checking the invariant in the precondition and postcondition of the method call, guaranteeing that only states where the invariant holds are stored into the database and also that the EJB is in an state conforming the invariant after calling ejbStore().

The tasks of these methods and their use is discussed in [18]. The finder methods can be treated as usual methods obviously, since a finder method does not change the system state (is side–effect free), there is no need to check any invariants in the precondition or postcondition of a finder method.

7.2 Using Different Distributed Component Technologies

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 own syntax and concepts, 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 Common Object Request Broker Architecture (CORBA) [12], also defined by the OMG, a special language for describing the client accessible interface is defined: the Interface Definition Language (IDL). 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 Enterprise Java Beans, the concrete view is a real refinement of the abstract view. In contrast to EJB, however 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 guarantee serializability of transaction, therefore one has the whole problematic (concurrency, call–backs) of distribution while specifying and constraint checking. For the case of *non–reentrant* EJBs combined with container managed persistence, the EJB container guarantees the serializability of transactions. When a call–back occurs the transaction is rolled back.

8 Conclusion and 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. Of course, post–hoc checking of violations of precondition 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 [8, 2, 4]. 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).

Further, our experience shows, that using such a powerful middleware architecture like J2EE/EJB allows us to specify complex client – server applications in a relatively easy manner. Under the assumption of an correct implementation of the middleware architecture and limiting ourself to specifying the business model we can also simplify our specification by using a middleware; this simplified version can neglect the aspects of distribution and concurrency.

On the other side, we have also new fields of activity for the specification. One of these are based on transactions. Within enterprise information systems several operations on Enterprise Java Bean are grouped together. Such a group of operations is often called transaction. It is an important property of a transaction, that it can be canceled, in a well defined manner (doing a rollback), when something went wrong during its execution. Therefore we see constraints on transactions as a useful extension to our actual model. Specifying precondition and postconditions and also invariants for transaction would allow us to specify “legal” states of the underlying database.

Bibliography

[1] Formals Specification – Z Notation – Syntax, Type and Semantics. June 2000. http://www.cs.york.ac.uk/~ian/zstan/. Consensus Working Draft 2.6.

[2] David Carrington, Ian MacColl, Jason McDonald, Leesa Murray, and Paul 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, October 1998. http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?98-22.

[3] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, Technische Univerität München, 1999. http://wwwbroy.informatik.tu-muenchen.de/reports/CKR+99.html.

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

[5] OCL Compiler Suite. http://dresden-ocl.sourceforge.net/. Implementation of an OCL compiler and code generator for Java. For details see [6] and [20].

[6] Frank Finger. *Design and Implementation of a Modular OCL Compiler*. Diploma thesis, Technische Universität Dresden, March 2000. http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1994. ISBN 0-201-63361-2., 416 pages.

[8] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. *Lecture Notes in Computer Science*, 1212:52–71, 1997. http://swt.cs.tu-berlin.de/~santen/.

[9] ArcStyler: The IT–Architecural IDE for J2EE/EJB, July 2001. `http://www.arcstyler.com`.

[10] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. 0-13-880733-7.

[11] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The JavaLanguage Specification.* Addison-Wesley Europe, Amsterdam, The Netherlands, second edition, 2000. ISBN 0-201-31008-2. `http://java.sun.com/docs/books/jls/index.html`.

[12] CORBA/IIOP 2.2 Specification. February 1998. `ftp://ftp.omg.org/pub/docs/formal/98-02-01.pdf`.

[13] OMG XML Metadata Interchange (XMI) Specification. November 2000. `ftp://ftp.omg.org/pub/docs/formal/00-11-02.pdf`.

[14] *Object Constraint Language Specification*, chapter 6. In Object Management Group [15], February 2001. `ftp://ftp.omg.org/pub/docs/ad/01-02-13.pdfl`. Version 1.4.

[15] OMG Unified Modeling Language Specification (draft). February 2001. `ftp://ftp.omg.org/pub/docs/ad/01-02-13.pdfl`. Version 1.4.

[16] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley Longman, Inc., Reading, MA, USA, 1998. ISBN 0-201-30998-X.

[17] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, second edition, 1992. ISBN 013-978529-9. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.

[18] Sun EJB 2 specification. 2000. `http://www.javasoft.com/products/ejb/docs.html`.

[19] Jos Warmer and Anneke Kleppe. *The Object Contraint Language: Precise Modelling with UML.* Addison-Wesley Longman, Inc., Reading, MA, USA, 1999. ISBN 0-201-37940-6. `http://www.klasse.nl/ocl-boek/intro.htm`. This book covers only OCL 1.1.

[20] Ralf Wiebicke. *Utility Support for Checking OCL Business Rules in Java Programs.* dilploma thesis, Technische Universität Dresden, December 2000. `http://rw7.de/ralf/diplom00/ocl-java.ps`.

[21] Jim Woodock and Jim Davies. *Using Z.* Prentice Hall, 1996. ISBN 0-13-948472-8. `http://softeng.comlab.ox.ac.uk/usingz/`.

Glossary

A

accessor method A method for setting or getting an class attribute, usually the accessor method for getting an attribute `a:type` is called `getA():type` and the method for setting this attribute is called `setA(a:type)`.

activation The process of transferring an enterprise bean from secondary storage into memory.

association The semantic relationship between two or more classifiers that specifies connections among their instances. A association can be described in more detail by using multiplicities.

B

bean implementation The class, that implements the methods of an Enterprise Java Bean. Together with the home interface and the remote interface the bean implementation forms the EJB.

business logic The code that implements the functionality of an application. In the Enterprise Java Bean model, this logic is implemented are described in the remote interface and are implemented by the methods of an enterprise bean.

business method A method that implements one aspect of the business model. A business-method is part of the business logic.

business model A model that describes the functional behavior of the system or a part of the system. The business model is implemented by the business logic.

C

class A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

class diagram A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

class invariant A class-invariant is an constraint that should hold true before and after any invocation of an method or any access to an class attribute. This definitions is different from the intention of the OCL standard [14, page 6-52], where it is postulated that a invariant should hold for *any* instance *at any time*.

classifier A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

collaboration diagram A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

Common Object Request Broker Architecture (CORBA) A middleware architecture, standardized by the OMG.

component A software (sub)system that can be factored out and has a standardizable or reusable interface. Components are usually language and machine independent and can be connected via a network.

Computer Aided Software Engineering tool (CASE tool) The generic term CASE can be used to mean any computer–based tool for software planning, development, and evolution. We use this term for describing complex engineering tools, that support the development chain from specification to code generation.

concurrency The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. 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.

constraint A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extensibility mechanisms in UML.

container managed persistence (CMP) The data transfer (storing the state information) between an entity bean's variables and resource manager managed by the entities bean's container.

D

data refinement In set based specification, data refinement is the process of showing that one set of operations is implementd by another set on a different state space.

E

EJB container A container implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, naming, and other services. An EJB container is provided by an EJB or J2EE server.

EJB server A software that provides services to an EJB container. For example, an EJB container typically relies on a transaction manager that is part of the EJB server to perform the two-phase commit across all the participating resource managers. The J2EE architecture assumes that an EJB container is hosted by an EJB server from the same vendor, so does not specify the contract between these two entities. An EJB server may host one or more EJB containers.

enterprise information system The applications that comprise an enterprise's existing system for handling company-wide information. These applications provide an information infrastructure for an enterprise. An enterprise information system offers a well defined set of services to its clients. These services are exposed to clients as local or remote interfaces.

Enterprise Java Bean (EJB) A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and secure.

entity bean An Enterprise Java Bean that represents persistent data maintained in a database. An entity bean can manage its own persistence or it can delegate this function to its container. An entity bean is identified by a primary key. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.

F

factory method A method implementing the *factory method pattern*, which defines an interface for creating objects.

finder method A method defined in the home interface and invoked by a client to locate an entity bean.

formal method Formal methods are techniques and tools based on mathematics and mathematical logic that support the description, construction and analysis of hardware and software systems.

H

home interface One of two interfaces for an enterprise bean. The home interface defines zero or more methods for managing an enterprise bean. The home interface of a session bean defines create and remove methods, while the home interface of an entity bean defines create, finder, and remove methods.

I

inheritance The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior.

interface Description of the externally visible behavior of a class, object, or other entity. In the case of a class or object, the interface includes the signatures of the operations.

Interface Definition Language (IDL) Implementation language–independent language for specification of distributed CORBA components.

invariant A invariant is an constraint that should hold true before and after any state transition. This definitions is different from the intention of the OCL standard [14, page 6-52], where it is postulated that a invariant should hold for *any* instance *at any time*. Special kinds of invariants are the class invariants and method invariants.

J

Java 2 Platform, Enterprise Edition (J2EE) An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multi–tiered, web-based applications.

Java 2 Platform, Standard Edition (J2SE) The core Java technology platform.

Java Naming and Directory Interface (JNDI) An API that provides naming and directory functionality, e.g. JNDI is used by clients to locate EJB objects.

Java Transaction Service (JTS) An API that allows applications and J2EE servers to access transactions.

M

method invariant A constraint must hold true before and after invocation of that method.

middleware A general term for any programming technique that serves to "glue together"or mediate between two separate and usually already existing programs. A common application of middleware is to allow programs written for access to a particular database to access other databases.

multiplicity A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a subset of the non-negative integers.

N

non–reentrant A possible property of an EJB. For non–reentrant EJBs, the EJB container throws exception if several threads/clients are using the same interface. This Guarantees that only non concurrent accesses to the EJB can take place.

O

Object Constraint Language (OCL) A (semi-) formal constraint language for making UML specifications more formal. It is part of the UML standard [15]. The purpose of OCL is specifying all kind of constraints like invariants, preconditions, postconditions or constraints related with finite-state-machines.

Object Management Group (OMG) The OMG was founded in April 1989 by eleven companies. In October 1989, the OMG began independent operations as a not-for-profit corporation. Through the OMG's commitment to developing technically excellent, commercially viable and vendor independent specifications for the software industry, the consortium now includes about 800 members. The OMG is moving forward in establishing CORBA as the "Middleware that's Everywhere"through its worldwide standard specifications: CORBA/IIOP, Object Services, Internet Facilities and Domain Interface specifications, UML and other specifications supporting Analysis and Design.

P

passivation The process of transferring an enterprise bean from memory to secondary storage. See also activation.

postcondition A constraint that must be true at the completion of an operation.

precondition A constraint that must be true when an operation is invoked.

private This model element is only visible (can only be accessed) within the classifiers it is defined in.

protected This model element is only visible (can only be accessed) within the classifiers it is defined in and all derived classifiers (subtypes).

public This model element within the classifier it is defined and also outside this classifier.

R

remote interface One of two interfaces for an enterprise bean. The remote interface defines the business methods callable by a client.

S

state-chart diagram A diagram that shows a state machine.

state machine A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

stereotype A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre–existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extendibility mechanisms in the UML.

T

three–tier A special type of client/server architecture consisting of three well-defined and separate processes, each running on a different platform: The first implements the user interface, which runs on the user's computer (the client). The second tier is the functional modules that actually processes the data. This middle tier, or middleware runs on a server which is often called the application server. The third tier is the database management system, often called enterprise information system that stores the data required by the middle tier. This tier runs on a second server.

transaction An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back. Transactions enable multiple users to access the same data concurrently.

U

Unified Modeling Language (UML) A general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects. UML builds on previous notational methods such as Booch, OMT, and OOSE. It is being developed under the auspices of the Object Management Group (OMG).

V

validation The process of proving empirically the correctness of an implementation against a specification. Normally done by testing.

verification The process of proving mathematically the correctness of an implementation against a specification. Often done by using formal methods.

visibility An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

X

Extensible Markup Language (XML) A markup language that allows you to define the tags (markup) needed to identify the content, data, and text, in XML documents. An XML document must undergo a transformation into a language with style tags under the control of a stylesheet before it can be presented by a browser or other presentation mechanism. Two types of style sheets used with XML are CSS and XSL . Typically, XML is transformed into HTML for presentation. Although tags may be defined as needed in the generation of an XML document, a Document Type Definition (DTD) may be used to define the elements allowed in a particular type of document. A document may be compared with the rules in the DTD to determine its validity and to locate particular elements in the document.

XML Metadata Interchange (XMI) An open information interchange model intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way, bringing consistency and compatibility to applications created in collaborative environments. XMI is intended to be either stored in a traditional file system or streamed across the Internet from a database or repository.

Index