Towards a Formal Framework for JavaBeans^{TM} and Enterprise JavaBeans^{TM}

November 2, 2001

Kwang Wu Lee   Dr. Padmanabhan Krishnan*

*supervisor

Abstract

This project aims to provide a framework for the formal specification of JavaBeans and Enterprise JavaBeans (EJB), Sun Microsystems' component technology.

We develop a list of properties that distinguishes beans from a Java class. For example, we formalise the notion of session beans, home/remote interfaces, etc. We also briefly touch upon the use of JavaBeans/EJB technology in a particular application.

Contents

CHAPTER 1

Introduction

1.1 Motivation

To meet an ever-increasing and sophisticated demand for software, many new technologies have been proposed. For large-scale applications, object-oriented software component technology is seen as a solution to overcome the complexities of modern software.

The idea of using components is well-known in hardware engineering. Hardware engineers build systems quickly and reliably from existing components. In software engineering, a component is a reusable unit that has a definite interface for exchanging information with other constituents of a system. The use of these components overcomes the limitations of the traditional library or module approach. Commercial software systems now employ a variety of component technologies to address different needs, e.g. Microsoft's COM+, Sun Microsystems' JavaBeans/Enterprise JavaBeans (EJB), and OMG's component model of CORBA.

We have chosen JavaBeans/EJB for our study of its architecture, as it is a popular component technology, and is platform independent.

In such large scale object-oriented technology, precise specifications are vital. User-defined components can only be run successfully if they conform to the specification that the framework assumes [NT01]. Unfortunately, the specification of these architectures and designs is too often only written in an informal language, i.e. plain English, and diagrams.

For this project our primary resource is the official Enterprise JavaBeans (EJB) specification 1.1, a 314-page document written in a natural language. The problems with this document are:

1. various related ideas are spread throughout the document,
2. the lack of precise definitions makes it very difficult to know exactly what a bean is, and
3. in many cases the document only provides examples of method sequences and code fragments, rather than formal rules. The generalisation of such example cases is not clear.

Our goal is to formulate a precise definition of a bean and the properties automatically associated with it. We attempt to capture the essence of JavaBeans/EJB as a collection of axioms based on predicates that describe the structure and behaviour of beans.

1.2 Related Research

Sousa and Garlan [SG00] discuss formalising EJB using an architecture description language (ADL). Their major focus is the interconnection between the client and the server associated with the EJB components. The paper uses Wright [All97] as the specification language, and the FDR (Failures/Divergences Refinement) [fdr92, Ros98] model checker to verify the specification.

They use two major concepts: a component and a connector. Each component is described as an interface and a computation, and connectors are described in terms of roles and ports. The distinction between a component and a connector is not well defined and depends on the role that they assume. The paper argues that the identification of the connector is important because the component integration framework provides a mediating infrastructure between components by the user of the framework. The port, role, computation, and glue are defined by a variant of Communicating Sequential Processes (CSP) [Hoa85], which is used to define each behaviour of a component and connector in terms of communications of processes.

Nakajima and Tamai [NT01] report a successful use of the model checker SPIN [Hol91] to analyse the behavioural properties of the EJB specification. They show how the EJB architecture is modelled as a Promela (the language used by SPIN) process model and discuss various technical issues. The verification of properties was formulated in terms of LTL (Linear Temporal Logic).

1.3 Report Structure

The remainder of this report is organised as follows:

Chapter 2 introduces model checking and theorem proving, including the theorem prover PVS [Rus97], and describes further background on JavaBeans/EJB. In Chapter 3, the specification of JavaBeans and the formalisation steps taken are discussed. We demonstrate an example implementation `OurButton` and determine whether it satisfies a minimal valid JavaBean definition. Syntactical issues are the main focus rather than semantical correctness, to gain experience with formalisation.

Some details of the EJB specification are presented in Chapter 4 and the basic framework structure is also discussed. We avoided formalising syntactical requirements because these can be checked by a verifier tool that comes with a standard Java 2 Enterprise Edition Software Development Kit (J2EE SDK). We provide a collection of predicates that describes the semantical features of beans. This is used to obtain a checklist that identifies the properties of beans.

In Chapter 5, an EJB implementation of the Jalapeño business scenario from "Enterprise JavaBeans by Examples" [JF00] is studied. Specific cases related to EJB requirements are identified and formalised as a specification. Each specification contains a list of the general properties of the beans and the specific properties of the example and situation.

Finally, Chapter 6 presents conclusions and further work.

1.4 Acknowledgements

I would like to thank Dr. Padmanabhan Krishnan for guiding me with this project, and also Jane McKenzie for proofreading this paper many times.

CHAPTER 2

Preliminary

2.1 Overview of JavaBeans and EJB

There are many different kinds of component technology, each emphasising different aspects. These can be divided into two major groups: client-side components, and enterprise environment components for the server-side. Client-side technology addresses ~~that~~ the development of software using components aimed for the end user. Software development tools have sophisticated graphical user interface (GUI) environments, and developers buy off-the-shelf components and assemble them. Developers can also create their own components for distribution or sale. Server-side or enterprise oriented components are not supported by the visual aspects of the graphical user interface, but rather by the underlying database services.

In this section, we explain JavaBeans and EJB and discuss their differences.

2.1.1 JavaBeans

JavaBeans was released before EJB, and its main emphasis is the visual aspect of programming, suitable for clients.

The JavaBeans API (application programming interface) specification defines JavaBeans as *"a reusable software component that can be manipulated visually in a builder tool"*. The specification briefly defines JavaBeans, then refines the initial definition by categorising the different types of beans and enumerating the optional features. The specification also defines terms and design patterns[†] as well as standard APIs.

The typical features that distinguish a JavaBean are: (1) introspection so that a builder tool can analyse how a bean works, (2) customisation so that when using an application builder a user can customise the appearance and behaviour of a bean, (3) the use of events as a simple communication metaphor to connect beans, (4) properties both for customisation and for programmatic use, and (5) persistence so that a bean can be customised in an application.

We will touch upon some of these aspects later when we formalise the descriptions.

[†]The term *design pattern* is used as a narrow definition as opposed to the common definition of design pattern in the literature. In the specification, these refer only to naming conventions that enable the builder tool to make understood meta information of the component.

2.1.2 Enterprise JavaBeans

The EJB specification defines its architecture as *"a component architecture for the development and deployment of component-based distributed business applications"*. The EJB architecture is one of several specification parts of the J2EE platform architecture and can be understood best within the context of the J2EE environment.

The Big Picture: Java 2 Enterprise Platform

In the late 1980s and early 1990s, most enterprise applications were based on a client/server architecture or two-tier architecture [MS]. As depicted in the client part (see Figure 2.1), the presentation logic and business logic is combined tightly. The presentation logic refers to an abstract interface to end users such as a graphical user interface, and the business logic refers to the specific processing of business tasks that require computation, such as calculating the interest of a bank account.

Later, the three-tier (see Figure 2.2) and multi-tier architectures were introduced. They provide greater flexibility than the traditional two-tier system by separating the business logic from the presentation parts, enabling business logic to be captured as components and reused in other situations.

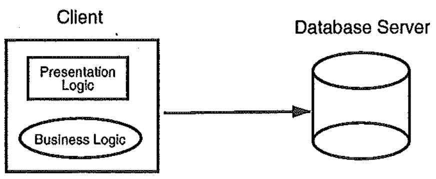


Figure 2.1: Two-Tier Architecture (client/server architecture)

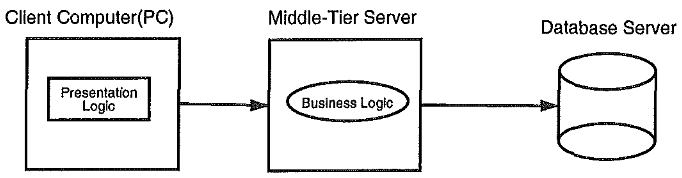


Figure 2.2: Three-Tier Architecture

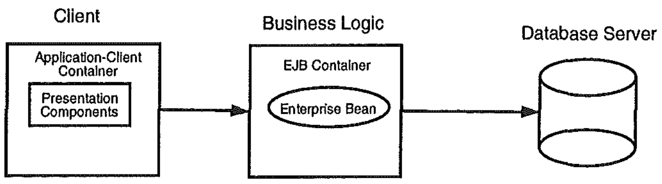


Figure 2.3: J2EE Application Programming Model for Three-Tier Applications

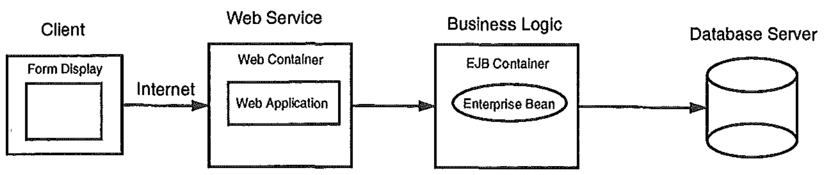


Figure 2.4: J2EE Application Programming Model for Web-Based Applications

J2EE supports the multi-tier system (see Figure 2.4), and Enterprise JavaBeans corresponds to the business logic components of the system. Figure 2.3 shows how the EJB architecture corresponds to the Middle-Tier Server in Figure 2.2.

EJB Architecture for Component-Based Development

In the EJB architecture, the container and the business logic area are the central focus. Hence often we abstract the multi-tier system as a three-tier system for simplicity, with the major parts being the client, beans, container, and database server.

Clients never directly access an EJB bean instance. Instead, it is accessed through the *home interface* and the *remote interface.* The home interface is used to create and destroy bean instances. The remote interface is used to call the business method of a bean.

Clients access beans through the JNDI (Java Naming and Directory Interface) service. The JNDI service provides a location-independent (or location-transparent) way of interacting between clients and EJBs so that it avoids tight couplings and maintains a flexible architecture to increase bean reusability.

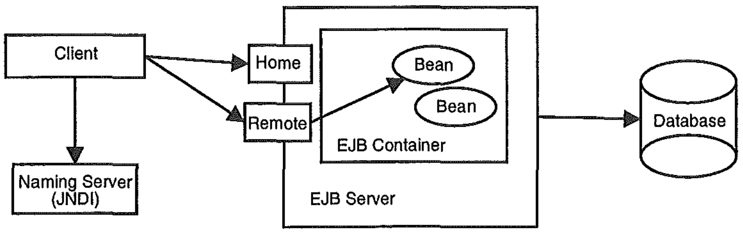


Figure 2.5: EJB Architecture

Session Bean and Entity Bean

in

The EJB specification defines two types of beans: session beans and entity beans.

A session bean is suitable for managing a session, and executes on behalf of a single client. It may implement a long-lived transaction style application logic to access several entity beans. Session beans have two options that can be specified at the time of deployment: *stateful* or *stateless*. Session beans are intended to be stateful, and stateless beans are defined as a special case.

An entity bean models an object as data in the database. It also allows shared access for multiple users.

Deployment of the Enterprise JavaBeans Issue

One of the major issues of EJB is how to deploy beans in a distributed environment, considering their scalability, maintainability, and customisability. A deployment tool (deploytool) is included with J2EE SDK. The tool can be viewed as a transformer or simply as a function which takes an EJB-JAR (Java ARchive) file and various options as its parameters and outputs results, which means that it can also be analysed formally.

2.2 Model Checking and Theorem Proving

The recent trend of applying formal verification to software has focused on automatic verification so that the gap between theory and practice in software engineering can be reduced. Model checking is a popular method and many supporting tools have been developed. The hardware industries, for instance Intel, are already using model checking to identify defects in their hardware design. Typical properties verified include absence of deadlock, fairness, etc.

In spite of success in the hardware area, and some parts of the software domain, it is not feasible to apply model checking to real software at an industrial scale due to the *state space explosion* problem and *infinite state space* problem [CGP99]. The amount of memory required to represent the formulae of the software is too large. It has been shown that the best known model checking algorithm has a complexity $|M| \cdot 2^{\mathcal{O}(|\phi|)}$, where M is the structure and ϕ is the formula, and $|M|$ and $|\phi|$ represent their respective sizes. In practice, M is very often larger than the available memory.

Theorem proving is a possible alternative to model checking, as it does not suffer from the state explosion problem because it does not search all possible cases. Another advantage is that the proof steps can be obtained, providing insight into the reasoning.

However, theorem proving is a manual process and therefore tedious to use.

2.2.1 Introduction to Theorem Prover PVS

The Prototype Verification System (PVS) is a higher order logic (HOL) based theorem prover [OSRSC99a, OSRSC99b, OSRSC99c]. In practice, it is used as a proof assistant or proof checker, because human intervention is still needed to prove a theorem.

Notation Used in the Specification

In this section, we discuss a specific example, the theory of `stacks`, which is an excerpt from the PVS Language Reference [OSRSC99a].

```
stacks[t: TYPE+] : THEORY
 BEGIN

  stack: TYPE+
  empty: stack
  nonemptystack?(s: stack) : bool = s ≠ empty
  nonemptystack_exists: AXIOM ∃ (s: stack) : nonemptystack?(s)

  push: [t, stack → (nonemptystack?)]
  pop: [(nonemptystack?) → stack]
  top: [(nonemptystack?) → t]

  x, y: VAR t

  push_top_pop: AXIOM
    ∀ (s: stack) : nonemptystack?(s) ⇒ push(top(s), pop(s)) = s

  pop_push: AXIOM ∀ (s: stack) : pop(push(x, s)) = s

  top_push: AXIOM ∀ (s: stack) : top(push(x, s)) = x

  pop2push2: THEOREM
    ∀ (s: stack) : pop(pop(push(x, push(y, s)))) = s
 END stacks
```

This example illustrates various aspect of PVS including the use of type parameters at the theory level, the general format of declarations, and the use of predicate subtyping to define the type of nonempty stacks.

The first line introduces a theory named *stacks* that is parameterised by a type *t* (the formal parameter of *stacks*). The keyword *TYPE+* indicates *t* is a nonempty type, and the keyword *TYPE* can be empty.

The uninterpreted (nonempty) type *stack* is declared, and the constant *empty* and variable are declared to be of type *stack*. The defined predicate *nonemptystack?* is then declared on elements of type *stack*; it is true for a given *stack* element if and only if that element is not equal to *empty*.

From line 7 to line 9, the functions *push*, *pop*, and *top* are declared. Note that the predicate *nonemptystack?* is being used as a type in specifying the signatures of these functions. The variables *x* and *y* are then declared.

From line 11 to line 16, the axioms are declared for *push*, *pop*, and *top*, which make *push* a stack constructor and *pop* and *top* stack accessors.

Note that the axioms are valid for all stacks. In other words, all stacks satisfy these properties.

Finally, there is the theorem *pop2push2*, that can easily be proved by two applications of the *pop_push* axiom.

2.2.2 About PVS Modelling Scheme

Using the theorem prover PVS, Krishnan [Kri00] proposed schemes to formalise UML (Unified Modelling Language) diagrams in terms of PVS specifications. Since the EJB specification describes similar diagrams, such as state transition diagrams and object interaction diagrams, these schemes were applied to formalise certain aspects of JavaBeans/EJB.

CHAPTER 3

Specification of JavaBeans

Key aspects of our formal specification of JavaBeans are presented in this chapter. First we outline a general approach, then discuss each specification in detail.

3.1 General Approach

The general approach of formalising informal specifications is to have a variety of predicates and construct axioms based on these predicates. Consider the following example: there are classes x and y, and we want a predicate that checks whether a given class is x or not. To specify such a predicate we need a type that specifies classes. Let us assume that we also need similar predicates for methods and properties. These can be captured as the following formal specification:

```
classX?: [Class → bool]
methodY?: [Method → bool]
propertyZ?: [Property → bool]
```

classX?, *methodY?*, and*propertyZ?* are simple predicates. Note that different types, e.g. *Class*, *Method*, and *Property*, are used to specify these predicates. These types are defined in TheoryJava.

Class and *Method* correspond to the classes and the methods in Java, respectively. *Property* is also from the property concept in the Java language, but it is a composite of instance variables and special methods (i.e. setters/getters).

As we intend to specify structural details, there are no definitions associated with behaviour. The basic infrastructure for the behaviour description will be developed in the next chapter.

3.2 General JavaBeans Requirements

In this section, we discuss:

- the precise definition of JavaBeans at a verifiable level,
- syntactic requirements that are used to expose JavaBeans information to an environment, and
- predicates that are useful to differentiate beans.

3.2.1 Minimal Valid JavaBean

As our goal is to study JavaBeans formally, the precise definition of what a JavaBean is, is our focus. The problem of the official JavaBeans description is that it only states a conceptual definition. The immediate problem is that one cannot differentiate a given class from a JavaBean. Describing the exact and valid definition of JavaBeans is extremely difficult because there are so many different types of JavaBeans. For example, a given class can be a JavaBean by definition, but in a programming sense, it may not be valid, and so on.

According to the document "How to be a Good Bean" [PLC97], all JavaBeans must support persistence to hold the customised information. Hence, it is mandatory to implement the interface `java.io.Serializable` or `java.io.Externalizable`. The predicate *supportSerializability?* checks whether the given bean implements either of these two interfaces so that the state of a JavaBean can be stored.

```
supportSerializability?(cls: Class) : bool =
    implements?(cls, java_io_Serializable) ∨
     implements?(cls, java_io_Externalizable)
```

Another mandatory requirement from the document [PLC97] is that a bean constructor must have either an explicit or an implicit *no parameters constructor* so that the method `java.Beans.instantiate` can instantiate the bean. The predicate *hasNoParamConstructor?* takes a Java class and indicates whether it has a zero parameter constructor.

```
hasNoParamConstructor?: [Class → bool]
```

There is no explicit statement about the minimal requirements to be a valid bean in the official JavaBeans specification [Ham97]. However, based on other sources [PLC97] and practical commonsense, one can conclude that to implement a valid JavaBean at least two requirements must be satisfied: (1) it should support serializability, and (2) its class must have a zero parameter constructor.

Formalisation of these gives the following predicate *minimalValidJB?*:

```
minimalValidJB?(cls: Class) : bool =
    supportSerializability?(cls) ∧ hasNoParamConstructor?(cls)
```

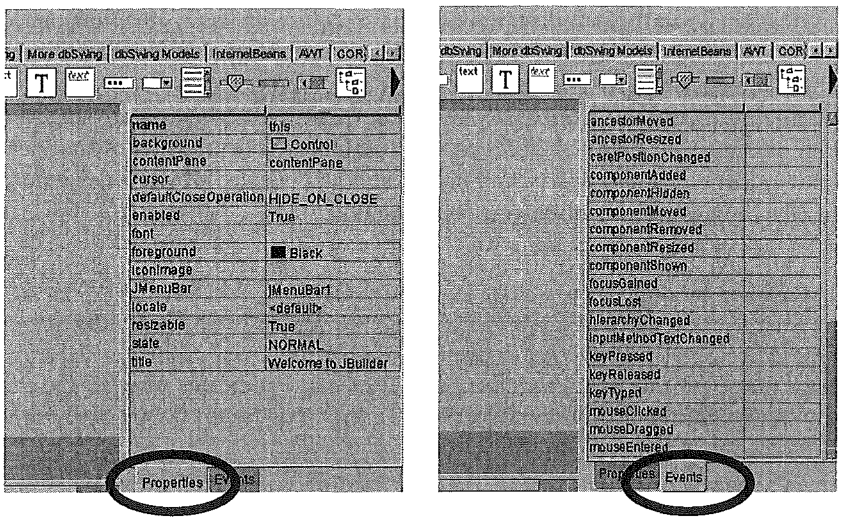
The axiom *minimalValidJB?* is defined in terms of *supportSerializability?* and *hasNoParamConstructor?*, stating that to be a minimal valid JavaBean it must support serializability and have a zero parameter constructor. □

3.2.2 Introspection and its Design Patterns

Properties and events play a key role when customising and combining components. Many commercial GUI builder tools use these two concepts (see Figure 3.1 from Borland JBuilder 4). Developers customise individual components through the properties editor and connect them using the events editor.

To enable the GUI builder to recognise the properties and events of a component automatically, some form of standard agreement or rule is necessary. JavaBeans uses the term *design patterns* (actually just a naming convention) for conventional names, and type signatures for sets of methods and/or interfaces that are used for standard purposes.

The whole process of determining which properties, events, and methods a Java Bean supports at runtime and in the builder environment is called *introspection*.



name	this
background	☐ Control
contentPane	contentPane
cursor	
defaultCloseOperation	HIDE_ON_CLOSE
enabled	True
font	
foreground	■ Black
iconImage	
JMenuBar	jMenuBar1
locale	<default>
resizable	True
state	NORMAL
title	Welcome to JBuilder

ancestorMoved	
ancestorResized	
caretPositionChanged	
componentAdded	
componentHidden	
componentMoved	
componentRemoved	
componentResized	
componentShown	
focusGained	
focusLost	
hierarchyChanged	
inputMethodTextChanged	
keyPressed	
keyReleased	
keyTyped	
mouseClicked	
mouseDragged	
mouseEntered	

Figure 3.1: Properties Editor (Left) and Events Editor (Right)

Naming Conventions for Properties

Introspection by reflection relies on naming conventions (i.e. the design pattern). For example, a simple property access method depends on the following patterns:

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> x);
```

We represent these patterns in the fashion of a predicate prototype. These predicates identify the patterns. Intuitively, these predicates play the role of a filter or an information indicator.

For example, let x be a simple property and TypeX be its type. In addition, let *getX* be the getter of x and *setX* be the setter of X, and assume that these two conform to the design pattern described above. Then *simplePAM?(getX)* is true, and *simplePAM?(setX)* is true.

All the following predicates have the suffix 'PAM' (Property Access Method Pattern). We define three additional predicates as follows:

simplePAM?:	[Method $\rightarrow$ bool]
booleanPAM?:	[Method $\rightarrow$ bool]
indexedPAM?:	[Method $\rightarrow$ bool]

If a bean has at least one bound property, the bean should have the following pair of event listener registration methods. These methods are used to interact with the events.

```
public void addPropertyChangeListener(PropertyChangeListener x);
public void removePropertyChangeListener(PropertyChangeListener x)
```

Similarly, if a bean has at least one constrained property, the bean should have the following pair of event listener registration methods:

```
public void addVetoableChangeListener(VetoableChangeListener x);
public void removeVetoableChangeListener(VetoableChangeListener x)
```

Note that bound and constrained properties also require semantic specifications (see the JavaBeans specification [Ham97] Section 7.4). For example, in the case of bound properties, when a property change occurs on a bound property, the bean should call the `PropertyChangeListener.propertyChange` method on any registered listeners, passing a `PropertyChangeEvent` object that encapsulates the locale-independent name of the property and its old and new values. However, we do not consider the semantics in the following formalisation:

boundProperty?: [Property $\rightarrow$ bool]
constrainedProperty?: [Property $\rightarrow$ bool]

The predicate *boundProperty?* indicates whether the given property conforms to the bound property. The predicate *constrainedProperty?* indicates whether the given property conforms to the constrained property.

Naming Conventions for Event-Listener Registration

The design patterns for event-listener registration serve similar functions to the property access method pattern, but for events instead. The methods used to register event notifications should conform to the standard design pattern for event-listener registration. For a normal standard multicast event-listener registration method, the following design pattern must be followed:

```
public void add<ListenerType>(<ListenerType> listener);
public void remove<ListenerType>(<ListenerType> listener);
```

The predicate for the multicast event-listener registration method is suffixed with 'ELRM' (event-listener registration method).

multicastELRM?: [Method $\rightarrow$ bool]

The JavaBeans specification also defines a design pattern for a unicast event-listener registration method. Note that the difference is at the *adder* (i.e. methods which follow the `add<ListenerType>` pattern), which can throw the exception `java.util.TooManyListenerException` if too may listeners have already been registered.

The design pattern for the unicast event-listener registration method is:

```
public void add<ListenerType>(<ListenerType> listener) throws
    java.util.TooManyListenerException;
public void remove<ListenerType>(<ListenerType> listener);
```

This is captured as follows:

unicastELRM?: [Method $\rightarrow$ bool]

The predicate *unicastELRM?* indicates whether a given method conforms to the design pattern. □

Other event handling methods can be formalised in a similar fashion.

3.2.3 BeanInfo

Instead of relying on reflection, JavaBeans allows a builder tool to introspect a bean by providing a *BeanInfo* class. The associated BeanInfo class must follow a name convention; ⟨BeanName⟩ becomes ⟨BeanName⟩BeanInfo. For instance, a BeanInfo class for JavaBean 'Foo' is 'FooBeanInfo'.

The predicate *validBeanInfoName?* checks whether the suffix is 'BeanInfo'.

```
validBeanInfoName?: [Class → bool]
```

Providing a BeanInfo class is the bean provider's decision and is optional. The predicate *validBeanInfoName?* can be used with the predicate *hasMatchingJB?* to find any dangling BeanInfo interface and provide a warning message.

The predicate *hasMatchingJB?* indicates whether or not the given JavaBean and a BeanInfo class match.

```
hasMatchingJB?: [JBClass, Class → bool]
```

3.2.4 Implementing Threads

Typically, a JavaBean runs under the web browser's control and is implemented with multi-threading supports so that the whole browser's CPU time is not used. In other words, the bean can be run in a separate thread, which increases the user's perception of responsiveness.

This multi-threading may result in unexpected results such as a deadlock or livelock situation. Thus, we provide a predicate that checks whether a JavaBean supports multi-threading or not.

```
multiThreadedRun?(jb: JBClass) : bool =
   implements?(jb, java_lang_Runnable)
```

The predicate *multiThreadedRun?* is defined in terms of the predicate *implements?* and indicates whether `java.lang.Runnable` is implemented. □

Based on these definitions we have proved that a particular example from Bean Development Kit 1.1 (BDK), `OurButton`, is a valid JavaBean.

3.3 The Structure of the Basic Framework for JavaBeans

Figure 4.4 shows the dependency of the PVS theory: TheoryJavaBeans depends on TheoryJava.

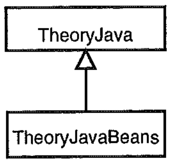


Figure 3.2: Theory Dependence Graph

- TheoryJava
  This contains basic Java types (Class, Interface, and Property) and basic object-oriented modelling frameworks (extends?). In addition, there are some axioms that ensure that some types are disjoint.

- TheoryJavaBeans
  This contains JavaBeans predicates. These include minimal valid JavaBeans, BeanInfo name checking, multi-thread support checking, and design patterns of JavaBeans (which enable reflection based on introspection).

CHAPTER 4

Specification of Enterprise JavaBeans

In this chapter, we analyse a subset of the requirements in the EJB specification. It should be noted that it is beyond the scope of this project to construct a complete formalisation of the EJB specification. Therefore, we will describe only some properties of the EJB architecture.

Each specification will be explained and the formalisation approach described.

4.1 General Approach

The properties of the EJB specification can be classified into various aspects of properties such as behavioural and protocol, etc. Past research [NT01, SG00] has focused on the verification of only one aspect. Our approach is to focus on constructing the list of predicates and axioms that distinguish a given EJB from a Java class.

4.1.1 State Predicates

In this section, we discuss state based predicates, which are used to describe behaviours. The details of the following is described in Krishnan's work [Kri00].

```
aStatePred?: [Method, State → bool]
```

The predicate *aStatePred?* is called a *state predicate.* State predicates are very useful in describing the behaviour of a system. For example, let *methodX* be an instance of type *Method* and *st* be an instance of type *State.* An informal description of *aStatePred?(methodX, st)* checks that the method *methodX* is called at the state *st.*

Based on these state predicates a relation *strongPast?* can be defined. For example, let *st1* and *st2* be instances of the type *State.* Then *strongPast?(st1, st2)* means that state *st1* occurs before state *st2.*

These states and state predicates are used to capture the behaviours of a system in temporal logic style. Although we have translated the sequence diagrams and state-transition diagrams in the EJB specification into a PVS specification, these are not presented herein, since the translation follows the work described in [Kri00, CDH$^+$00, Har00].

4.2 General EJB Requirements

In this section, we present the specifications and their formalisation. Each specification includes the EJB specification section and the page number.

General Specification 1 (EJB.5.1 page 40) *The client view of a bean is location-independent. A client running in the same Java Virtual Machine (JVM) as the session object uses the same API as a client running in a different JVM on the same or different machine.*

The EJB architecture achieves location-independence by the JNDI mechanism, which requires the following procedures (see the left of Figure 4.1).

First, the reference of the context must be obtained before any JNDI lookup operations. Thus, the client must call the constructor `InitialContext()`, which returns a reference to the context interface.

Then, clients may call the `lookup` method, which will return an object. The EJB specification requires that clients use the `javax.rmi.PortableRemoteObject.narrow` method to perform type conversion of references of the EJB home and remote interfaces.

Figure 4.1 describes the parallel state machine diagram for client and session beans, which is an extended version of the EJB specification that only describes the life cycle of a session bean (EJB.5.6). The method name starting with "call/" and "get/" is for method synchronisation. For example, if the client calls a "call/home.create()" method, then in the container "get/home.create()" a transition will occur.

Note that although the diagram is incomplete it is a precise formal notation. Potential exceptions, crashes, termination, and all valid combinations of the JVM operation are omitted in the client part of the diagram.

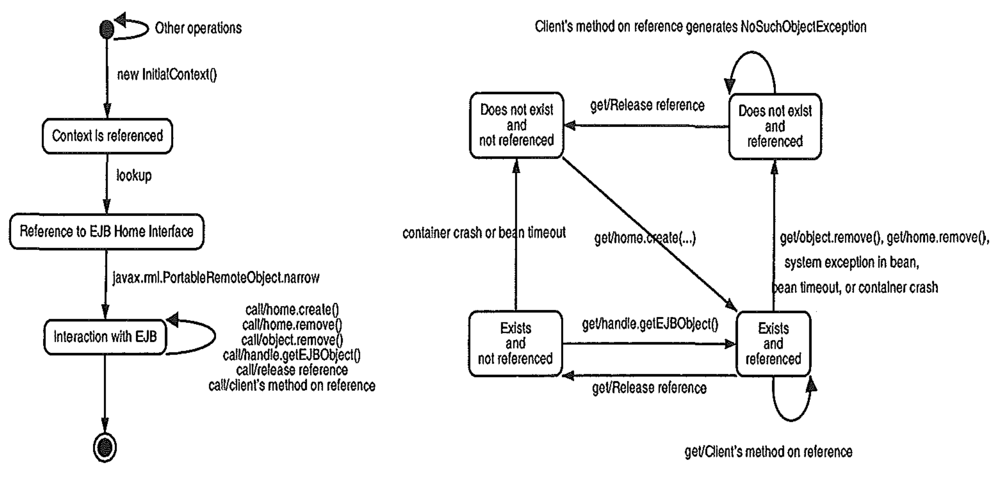
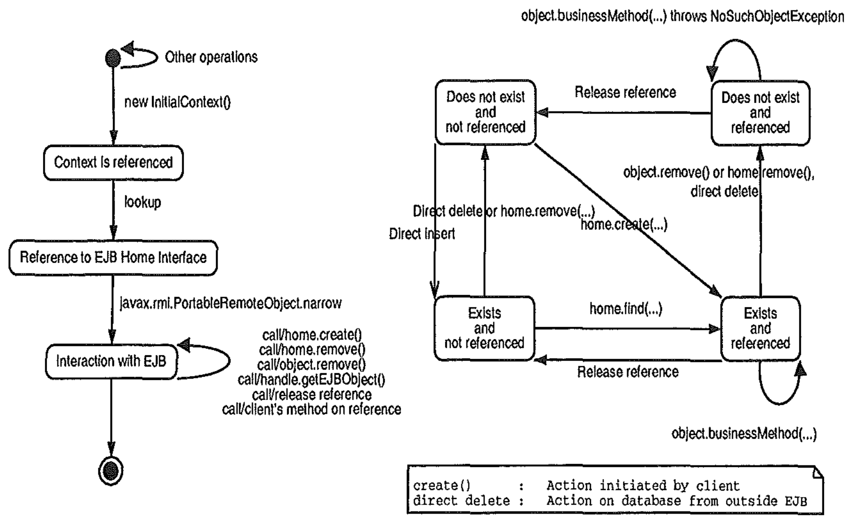

Other operations
new InitialContext()
Context is referenced
lookup
Reference to EJB Home Interface
javax.rmi.PortableRemoteObject.narrow
Interaction with EJB
call/home.create()
call/home.remove()
call/object.remove()
call/handle.getEJBObject()
call/release reference
call/client's method on reference
Client's method on reference generates NoSuchObjectException
Does not exist and not referenced
get/Release reference
Does not exist and referenced
container crash or bean timeout
get/home.create(...)
get/object.remove(), get/home.remove(), system exception in bean, bean timeout, or container crash
Exists and not referenced
get/handle.getEJBObject()
Exists and referenced
get/Release reference
get/Client's method on reference

Figure 4.1: Parallel State Transition Machine of a Client (left) and Life Cycle of a Session Object (right)

Figure 4.2 describes the parallel state machine diagram of an entity session bean. The same rules of Specification 1 are applied to the entity session bean version.



Other operations
new InitialContext()
Context is referenced
lookup
Reference to EJB Home Interface
javax.rmi.PortableRemoteObject.narrow
Interaction with EJB
call/home.create()
call/home.remove()
call/object.remove()
call/handle.getEJBObject()
call/release reference
call/client's method on reference
object.businessMethod(...) throws NoSuchObjectException
Does not exist and not referenced
Release reference
Does not exist and referenced
object.remove() or home.remove(), direct delete
Direct delete or home.remove(...)
Direct Insert
home.create(...)
Exists and not referenced
home.find(...)
Exists and referenced
Release reference
object.businessMethod(...)
create() : Action initiated by client
direct delete : Action on database from outside EJB

Figure 4.2: Parallel State Transition Machine of a Client (left) and Life Cycle of an Entity Object (right)

> *Note that the HOL specifications of the two parallel state transition machines are not presented because the encoding of the state transition is standard.*

□

General Specification 2 (EJB.5.8.1 page 46) *The specification requires that a stateful session bean has a unique identity that is assigned by the container at create time.*

To a bean developer, the object identities of session beans may become important if different session objects of the same type need to be created and managed. For example, suppose a customer started a shopping session, suspended the shopping session temporarily for a day or two, and later completed the session. To resume the session, the object identity may need to be obtained.

To explain the uniqueness of the object identity of session beans, consider the following code:

```
FooHome fooHome = ...; // obtain home of a
                       // stateful session bean
Foo foo1 = fooHome.create(...);
Foo foo2 = fooHome.create(...);
```

The above code illustrates that the interface `FooHome` is a specific home interface. `foo1` refers to the first instance returned by invoking `create`, and `foo2` refers to the second instance returned by invoking `create` again.

In other words, the specification requires that after the assignment of `foo1` and `foo2`, `foo1` $\neq$ `foo2`.

From the specification and the above code illustration, we derive the axiom *uniqueness_of_stateful_session_bean*:

```
create: Method
ofType?: [InstanceVariable, JavaType → bool]
returnValue: [InstanceVariable, Method, State → Instance]
fooHome: InstanceVariable

uniqueness_of_stateful_session_bean: AXIOM
  ∀ (hInterface: Interface) :
    ∀ (st1, st2: State) :
      homeInterface?(hInterface) ∧
      ofType?(fooHome, hInterface) ∧
      callWithRef?(fooHome, create, st1) ∧
      callWithRef?(fooHome, create, st2) ∧ st1 ≠ st2
      ⇒
       returnValue(fooHome, create, st1) ≠
       returnValue(fooHome, create, st2)
```

The variables *st1* and *st2* are of type *State*, and correspond to different time instances. For example let *st1* be the state instance when the first `fooHome.create()` is called and *st2* be the state instance when the second `fooHome.create()` is called. Then, for all session beans within the same home interface, if the `create` method is invoked at *st1* and at *st2* they should return a different value.

The predicate *ofType?* indicates whether a given instance variable matches a Java type exactly (i.e. no subtype is allowed). Thus, ofType?(fooHome, hInterface) means the reference *fooHome* is a home interface type.

The function *returnValue* returns an instance for invoking a given method with respect to the instance variable at a given state. That is, the value of `foo1` is the same as *returnValue(fooHome, create, st1)*.

The above axiom specifies that the instance returned from invoking the `create` method at different times (*st1* and *st2*) should be different (`foo1` at *st1* and `foo2` at *st2*). □

General Specification 3 (EJB.5.8.2 page 46–47) *All session objects of the same stateless session bean within the same home have the same object identity, which is assigned by the container.*

Specification 3 is similar to Specification 2 but for stateless beans. As mentioned in Section 2.1.2 deployment of beans is important in a distributed environment. Stateless beans deployed at different times will have different identities only if the deployment is in different homes.

```
object_identity_of_stateless_session_bean: AXIOM
  ∀ (hInterface: Interface) :
    ∀ (st1, st2: State) :
      homeInterface?(hInterface) ∧
      ofType?(fooHome, hInterface)
      ⇒
       returnValue(fooHome, create, st1) =
       returnValue(fooHome, create, st2)
```

The same predicates *ofType?* and function *returnValue* in Specification 2 are used again in Specification 3.

Now `foo1` = `foo2` for all *st1* and *st2*. Note that in this case, *st1* and *st2* can be either equal or not. □

General Specification 4 (EJB.4.3.1 page 35) *A session object is removed when the EJB container crashes. The client has to reestablish a new session object to continue computation.*

In a typical Internet commercial site, while a customer (client) is selecting items, he/she may experience a sudden transaction failure such as a temporary network failure before checking out the items. This is a very unfortunate situation for both the customer and the company because the customer has to select the items again, and the company may lose the business transaction if the user discontinues the purchase.

The most desirable solution is to recover to the state that existed before the crash occurred. However, the automatic recovery by the container depends on the type of beans; a session object is removed when an EJB container crashes, and the container does not support recovery. Thus, we need to ensure that there is an associated client side recovery mechanism.

This restriction imposes the responsibility for recovery to the client side process if a session bean must be used and a guarantee of recovery is necessary.

The restriction is captured as follows:

```
ejbCrash?: [State → bool]
continueComputation?: [EJBObjectType, State → bool]
reestablish?: [EJBObjectType, State → bool]

session_bean_reestablish: AXIOM
  ∀ (b: SessionObjectType) :
    ∀ (st1, st3: State) :
      (strongPast?(st1, st3) ∧
      ejbCrash?(st1) ∧
      continueComputation?(b, st3))
      ⇒
       (∃ (st2: State) :
          reestablish?(b, st2) ∧
          strongPast?(st1, st2) ∧
          strongPast?(st2, st3))
```

The predicate *ejbCrash?* indicates whether the EJB container has crashed at a given state. The predicate *continueComputation?* indicates whether a given bean continues its execution at a given state. These two predicates are applied for all states to capture that these two events can happen at any time.

The predicate *reestablish?* indicates whether a given bean has been reactivated after the crash at a given state. The existential quantifier is used as we only need one state where the reactivation occurs. Figure 4.3 shows that *ejbCrash?* occurs at *st1*, *reestablish?* occurs at *st2*, and *continueComputation?* occurs at *st3*.

The axiom *session_bean_reestablish* describes this exactly. *SessionObjectType* ensures that the specification only applies to session beans not to entity beans. Using the predicate *strongPast?*, we specify the states in the order *st1* < *st2* < *st3*. If *ejbCrash?* occurs and computation is continued at some later time, then reestablishment must have occurred between the two events. □

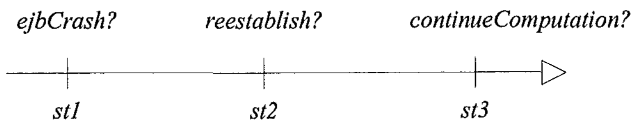


Figure 4.3: EJB Container Crash/Recovery Process

General Specification 5 (EJB.5.4 page 43) *A client never directly accesses instances of the session bean's class. A client always uses the session bean's remote interface to access a session bean's instance.*

The bean provides a service to clients by exposing the public interface of the business methods. However, session beans have an important restriction: a client can not directly access the bean instance. A session object supports the business logic methods of the object. The session object delegates the invocation of a business method to the session bean instance.

According to the EJB specification, the class that implements the session bean's remote interface is provided by the container; its instances are called session `EJBObjects`. Note that this container responsibility of providing an EJBObject is discussed later in Specification 6.

Specification 5 limits the client's and container's behaviours and must be followed, but this is not easily checked by a syntactic verifier.

An informal description of the above specification can be stated as: if a client accesses a bean instance, then the following must be guaranteed: (1) the client must always indirectly access a session bean instance; and (2) a client must access a session object instance (remote interface) that exists between the client and the bean instance.

The above requirement is specified formally as:

```
directAccess?: [Instance, Instance → bool]
access?: [Instance, Instance → bool]

acess_ax: AXIOM
  ∀ (x, y: Instance) :
    access?(x, y) ⇒
      (directAccess?(x, y) ∨
        (∃ (z: Instance) : directAccess?(x, z) ∧ access?(z, y)))

indirectAccess?(x, y: Instance) : bool =
    access?(x, y) ∧ ¬ directAccess?(x, y)

ClientType: TYPE FROM Instance
EJBBeanInstance: TYPE FROM Instance
containerProvided?: [SessionObjectType, EJBBeanInstance → bool]
isSessionObjectInstance?: [Instance → bool]

client_never_directly_access_bean: AXIOM
  ∀ (c: ClientType, b: EJBBeanInstance) :
    isSessionObjectInstance?(b) ∧ access?(c, b) ⇒
      (∃ (so: SessionObjectType) :
          indirectAccess?(c, b) ∧
            access?(c, so) ∧ containerProvided?(so, b))
```

Here we define the predicate *directAccess?* to indicate whether the instance given as the first argument directly accesses the instance given as the second argument.

The predicate *access?* is the generalised version of the predicate *directAccess?*. It indicates whether the instance given as the first argument accesses the instance given as the second argument either directly or indirectly.

The axiom *access_ax* corresponds to the recursive definition of the predicate *access?*, which has basically the same structure as the predicate *strongPast?*. It is either *directAccess?* or a recursive version of indirect access.

We also define the predicate *indirectAccess?* as "instance x can access instance y but not directly".

Finally, the axiom *client_never_directly_access_bean* specifies that a client can only access indirectly. □

General Specification 6 (EJB.5.4 page 43) *The class that implements the session bean's remote interface is* ***provided by the container****. The instances of the class are called session* `EJBObject`.

Specification 6 restricts container providers. From a user's point of view, bean developers do not have to be concerned with this. However, this specification is related to Specification 5 because clients must access a session object.

Informally, if a session bean is *deployed* at a container, the container must produce a session `EJBObject` of a remote interface for the client. Note that here we introduce the concept of *deploying*, which is not clearly stated in Specification 6 but is required.

This specification is captured as follows:

```
deployedAtContainer?: [EJBType, EJBContainerType → bool]
implementsRemoteInterface?: [EJBType → bool]
container_provide_session_EJBObject: AXIOM
  ∀ (bean: EJBType) : isSessionBean?(bean) ⇒
     (∀ (container: EJBContainerType) :
        deployedAtContainer?(bean, container) ⇒
          implementsRemoteInterface?(bean) )
```

First we need the predicate *deployedAtContainer?*, which says whether a given EJB is deployed at a given container.

The predicate *implementsRemoteInterface?* indicates whether a given EJB implements a remote interface or not.

The axiom *container_provide_session_EJBObject* specifies that if a bean is a session bean, and is deployed at a container, then the bean has a remote interface. □

General Specification 7 (EJB.5.8.3 page 47) *The object identifier of a session object is, in general, opaque to the client. The results of* `getPrimaryKey()` *on a session* `EJBObject` *reference results in* `java.rmi.RemoteException`.

For both bean developers and container providers, this specification is unclear because of the term "opaque". We interpret "opaque" in a strict sense to be safe; *as "is hidden from" or "its information is not allowed to be accessed"*.

The J2EE verifier checks whether the method `getPrimaryKey()` is specified with the keyword `throws`, e.g. `public java.lang.Object getPrimaryKey() throws java.rmi.RemoteException` at a syntactic level. However, the actual implementation of throwing the exception is not checked since the empty body implementation has passed the verification test. We specify this at a semantic level.

This specification is captured as:

```
belongTo?: [Method, Instance → bool]
methodsAlwaysThrows?: [Method, Exception → bool]

getPrimaryKey_throw_remote_exception: AXIOM
  ∀ (so: SessionObjectType, getPrimaryKey: Method) :
     belongTo?(getPrimaryKey, so) ⇒
        methodsAlwaysThrows?(getPrimaryKey, java_rmi_RemoteException)
```

We define the predicate *belongTo?* to check whether a given method belongs to a given instance. That is, the class of the instance has the method.

The predicate *methodsAlwaysThrows?* indicates whether a given method always throws a given exception. Note that this predicate checks at a *semantic level* as well as at a syntactic level.

Finally, the axiom *getPrimaryKey_throw_remote_exception* states that if `getPrimaryKey` is invoked for any session objects, the method `getPrimaryKey` must throw *java_rmi_RemoteException*. □

General Specification 8 (EJB.6.5.1 page 53) *The bean's container calls the* `setSessionContext` *method to associate a session bean instance with its context maintained by the* ***container***. *Typically, a session bean instance retains its session context as part of its conversational state.*

According to the sequence diagram (EJB.6.7.2), when a session object is created, to associate context information, the container calls `setSessionContext`. If, during its life cycle, the bean instance needs to invoke the methods of the SessionContext interface, the instance should retain the SessionContext reference in an instance variable as part of its conversational state [MS].

```
public class ExampleBean implements javax.ejb.SessionBean {
    ...
    SessionContext ctx;
    ...
    public void setSessionContext(SessionContext sc) {
        this.ctx = sc;
        ...
    }
    ...
}
```

As the above code shows, when `setSessionContext` is invoked with the parameter `sc`, `sc` receives the SessionContext reference. Note that the instance variable `ctx` is declared in the class body. By assigning `this.ctx = sc`, now the instance retains the SessionContext reference as part of its conversational state.

This specification is also clearly a restriction on both the bean developer and the container provider. Note that this property is not checked by the J2EE verifier.

```
contextAssociationNeeded?: [EJBBeanClassType → bool]
containerCall?: [Method → bool]
retainSessionContext?: [EJBBeanClassType → bool]
isSessionBeanClass?: [EJBBeanClassType → bool]
setSessionContext: Method

implements_setSessionContext: AXIOM
  ∀ (b: EJBBeanClassType) :
    isSessionBeanClass?(b) ∧ contextAssociationNeeded?(b) ⇒
      containerCall?(setSessionContext) ⇒
        retainSessionContext?(b)
```

The predicate *contextAssociationNeeded?* indicates whether a given EJB class needs context information. The predicate *containerCall?* decides whether a given method is called by the container. The predicate *isSessionBeanClass?* determines whether a given EJB class is a session bean class. The predicate *retainSessionContext?* determines whether a given EJB class retains its context reference.

Finally, the axiom *implements_setSessionContext* specifies that if a bean is a session bean class and needs context information, then the container invokes the method `setSessionContext`. If the container calls `setSessionContext()`, then the bean instance must retain the reference. □

General Specification 9 (EJB.5.5 page 43) *The home interface of a session bean must not define any finder methods.*

The EJB specification requires that one or more finder methods must be defined in an entity bean's home interface. The name of the finder method must start with the prefix `find`, such as `findSmallAccounts(...)`, and provide a way to find an entity object or collection of entity objects within the home.

The finder method is only applicable to an *entity bean.* Unlike a session bean, if an entity object or collection of entity objects already exist in the database, the client can load from the database to the container.

We do not formalise this as it as an example of a specification requirement.

□

General Specification 10 (EJB.8.4 page 92) *All entity objects are considered to be* ***persistent objects.***

So far we have focused mostly on session beans. Here we present an entity bean specification. The three major services provided by the container are transaction, persistence, and security. The exact definition of persistence is difficult to describe concisely. We say something is persistent when a given object can be stored permanently, i.e. in a database system. The entity bean is specifically designed to represent persistent objects (or entities). For example, a registered customer using commercial sites should be stored in a database.

```
persistent?: [EJBType → bool]
isEntityBean?: [EJBType → bool]

entity_persistent: AXIOM
   ∀ (b: EJBType) : isEntityBean?(b) ⇒ persistent?(b)
```

The predicate *persistent?* indicates whether a given bean is persistent or not. The predicate *isEntityBean?* indicates whether a given bean is an entity bean or not.

The axiom *entity_persistent* specifies that all entity beans are persistent beans. □

4.3 The Structure of the Basic Framework for EJB

The formalisation of the EJB architecture is divided into two major areas: session beans and entity beans, each of which contain a client view and a container contract. Figure 4.4 shows its corresponding PVS theory's structure.

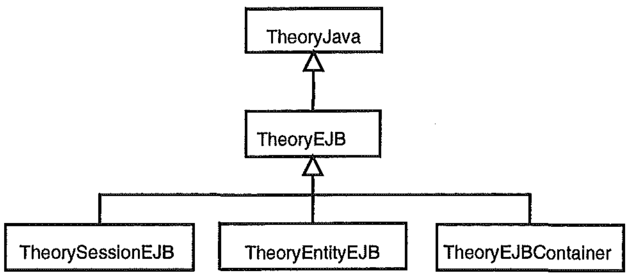


Figure 4.4: Theory Dependence Graph

- TheoryEJB
  This contains EJB specific types, and J2EE EJB interfaces and classes are defined here. State predicates' frameworks from Krishnan's work are also defined.
- TheoryEntityEJB
  This contains entity bean specific types, and predicates are defined. The life cycle of the state-transition diagram specified in the EJB white paper has been modelled as predicates.
- TheorySessionEJB
  This contains session bean specific types, and predicates are defined. The life cycle has been modelled as predicates.
- TheoryEJBContainer
  This contains container related sequence diagrams in the EJB specification that have been translated. The translated container related sequence diagrams in the EJB specification are found in the TheoryEJBContainer

CHAPTER 5

Specification of an EJB Application: Jalapeño Case Study

In the previous chapter, general specifications of EJB were discussed. In this chapter, a specific application, "Enterprise JavaBeans by Example" [JF00], is discussed in detail.

5.1 Brief Jalapeño Business Scenario

The Jalapeño business scenario (Henri's Stuffed Peppers Restaurant), which is a verbatim excerpt from page 143–144 of "Enterprise JavaBeans by Example", is as follows:

> Henri wants to enter a new market by offering his Stuffed Peppers menus through an online ordering system on the Internet. Online ordering will allow the customers to connect through the Internet to the ordering system. The customer will be able to select from a list of items. The customer's selection will then be transferred to the server. The server will calculate a price for the item and send the price back to the client. The customer can either accept or decline. If the customer accepts, an order will be printed at the restaurant.

5.2 General Behaviours

Figure 5.1 illustrates what is happening when the client creates a new order.

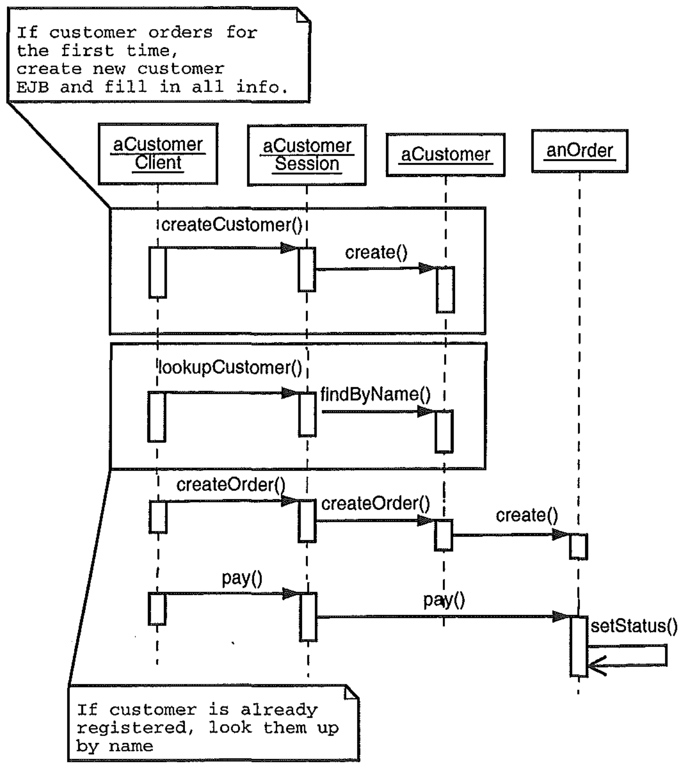


Figure 5.1: Create New Order (EJB by example P. 148)

We know `CustomerSession` is a session bean and `Order` and `Customer` are entity beans. Since these are business methods, we can refine the sequence diagram. This provides us with our first specification:

Specification 1 *If a client makes an order for the first time, then a customer must be created.*

Customers are modelled as arrival events. A customer who orders for the first time needs to fill in the address information and payment method, which is a tedious process. This information needs to be saved in the restaurant database, and when the customer returns, by identifying the customer, the address and payment method can be loaded, and this step can be skipped.

We assume that customers identify themselves by choosing the HTML page for the first time customer. As mentioned, `CustomerSession` is a session bean and `Customer` is an entity bean. The method `createCustomer` is a business method, and when they are invoked the support and restriction of session beans and the entity bean rule must be applied.

1. The customer identifies that he/she is a first time visitor to the restaurant.
2. After the identifying process, `createCustomer` is invoked.
3. The `create` method of the `Customer` entity bean will be created in the database.

This is formalised as follows:

```
firstCustomer?: [State → bool]
equippedSessionBeanSupport?: [EJBObjectType, Method → bool]
equippedEntityBeanSupport?: [EJBObjectType, Method → bool]
createCustomer: BusinessMethod
create: Method

jala_unknownCustomer: AXIOM
  ∀ (st1: State) :
    firstOrder?(st1) ⇒
      (∀ (obj: CustomerSessionObjectType) :
         ∃ (st2: State) :
           strongPast?(st1, st2) ∧
           call?(createCustomer, st2) ∧
           equippedSessionBeanSupport?(obj, createCustomer)
            ⇒
            (∀ (obj: CustomerObjectType) :
               ∃ (st3: State) :
                 strongPast?(st2, st3) ∧
                 call?(create, st3)))
```

The predicate *firstOrder?* identifies whether an order has been made at a given state for the first time. The predicate *equippedSessionBeanSupport?* indicates whether a given EJB object is a session object and the business method belongs to the bean. These two predicates distinguish EJB modelling from normal sequence diagram modelling as they are specific only to EJB. The business method *createCustomer* and the method *create* correspond to the first boxed group of message sequences. Note that the method *create* is not considered as a business method since it belongs to the home interface.

Finally, the axiom *jala_unknownCustomer* describes the message sequence of Specification 1. For every *CustomerSessionObjectType* there will be some state where the *create* method will be invoked. □

Specification 2 *If the customer has ordered at least once, then the customer information will be obtained from the previously recorded database.*

As described in Specification 1, customers who are already registered do not need to fill in their address and payment method again. They only need to identify themselves.

The method calls in the "lookupCustomer" box in Figure 5.1 show the process of finding the customers' information from the database. Here `lookupCustomer` is a business method associated to the session bean `CustomerSession`, and the finder method `findByName` is a business method of the entity bean `Customer`.

Similarly, this can be captured as follows:

```
lookupCustomer: BusinessMethod
findByName: Method

jala_knownCustomer: AXIOM
  ∀ (st1: State) :
    ¬ firstOrder?(st1) ⇒
     (∀ (obj: CustomerSessionObjectType) :
        ∃ (st2: State) :
          strongPast?(st1, st2) ∧
          call?(lookupCustomer, st2) ∧
          equippedSessionBeanSupport?(obj, createCustomer)
           ⇒
           (∀ (obj: CustomerObjectType) :
              ∃ (st3: State) :
                strongPast?(st2, st3) ∧
                call?(findByName, st3) ∧
                equippedEntityBeanSupport?(obj, create)))
```

The predicate *firstOrder?* is used again in Specification 2 but *not* as a first order. The predicate *equippedEntityBeanSupport?* is introduced here, which indicates whether a given EJB object is an entity object and the business method belongs to the bean. Finally, the axiom *jala_knownCustomer* corresponds to the looking up customer part of the sequence diagram. The description of the axiom is as in Specification 1. □

The remainder of the message sequences in Figure 5.1, i.e. creating a new order and paying, can be described similarly.

5.3 The Structure of the Ordering System

The way that clients can retrieve product information as shown in Figure 5.2.

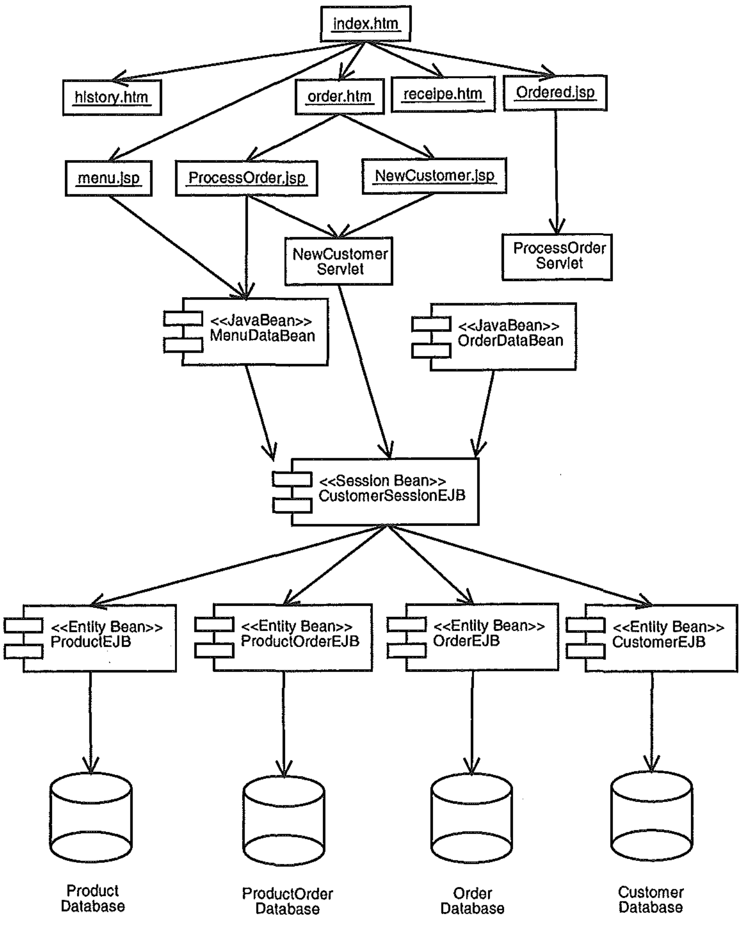


Figure 5.2: Ordering System Structure

The following steps are a slightly simplified description from "Enterprise Java-Beans by Example" [JF00], page 164, and show how a request travels from a client to the database server. The steps correspond to the path `JSP--MenuDataBean--CustomerSessionEJB--ProductEJB`

1. A Java Server Page (JSP) request is made by a browser. Then, the servlet creates an instance of a JavaBean (embedded in the JSP), and the bean creates a new `CustomerSession` EJB.
2. `CustomerSession` calls the `findAllProducts()` method of `Product`.
3. `Product` retrieves all the products stored on the database and creates product entities for all the products. It then returns an `Enumeration` of `Product`.
4. `CustomerSession` returns the product's `Enumeration` to `MenuDataBean`.
5. `MenuDataBean` iterates through `Product Enumeration` and stores the product name, product number, and price fields of each `Product` entity EJB as elements of three, corresponding, multi-valued properties of `MenuDataBean`.

6. The JSP (or, more correctly, JSP-generated servlet) uses the `MenuDataBean`'s get methods to retrieve the product name, product number, and price for all the products that have been retrieved. These data are inserted into the appropriate places in the JSP.

7. The resulting HTML is sent to the browser. A product menu appears on the browser.

The above specification is captured as follows:

```
menuSelection?: [Bean, State → bool]
menuItem: Bean
findAllProducts: BusinessMethod
returnPage?: [Bean, State → bool]

jala_menuReturn: AXIOM
  ∀ (st1: State) :
     menuSelection?(menuItem, st1) ⇒
      (∀ (obj: CustomerSessionObjectType) :
          ∃ (st2: State) :
           strongPast?(st1, st2) ∧
           call?(create, st2) ∧
           equippedSessionBeanSupport?(obj, create)
            ⇒
            (∀ (obj: ProductObjectType) :
                ∃ (st3: State) :
                   strongPast?(st2, st3) ∧
                   call?(findAllProducts, st3) ∧
                   equippedEntityBeanSupport?(obj, findAllProducts)
                    ⇒
                    (∃ (st4: State) :
                        strongPast?(st3, st4) ∧
                        returnPage?(menuItem, st4))))
```

The predicate *menuSelection?* indicates whether a given bean is selected at a given state, i.e. the submission from the user's menu selection. The predicate *returnPage?* indicates whether the resulting HTML page based on a given bean's request is returned at a given state.

The axiom *jala_menuReturn* captures the procedure described above in detail. The predicate *equippedSessionBeanSupport?* and *equippedEntityBeanSupport?* refine that the business methods within the general specification discussed in Chapter 4. For instance, method *findAllProducts* is a business method of the entity bean *Product* so that all the requirements or services provided from the container must be related to it. □

Specification 3 (MenuDataBean must be a valid JavaBean) *JavaBean `MenuDataBean` must be a valid JavaBean. Also it should follow valid procedures such as JNDI interaction to create a `CustomerSesssion` Bean.*

Specification 3 is derived from the argument: `MenuDataBean` must be at least a valid JavaBean. In this multi-tier system, this specification shows that the JavaBeans' validity also plays a role.

This is captured as follows:

```
MenuDataBean : Class
satisfyClientRequirement? : [Class → bool]

MenuDataBean_is_client : AXIOM
   minimalValidJB? (MenuDataBean) ∧
   satisfyClientRequirement? (MenuDataBean)
```

The *MenuDataBean* is defined as a *Class*. The predicate *satisfyClientRequirement?* checks whether a given class satisfies the mandatory JNDI procedure mentioned in Chapter 4.

The axiom *MenuDataBean_is_client* specifies that the *MenuDataBean* satisfies the predicate *minimalValidJB?* and *satisfyClientRequirement?*. □

Specification 4 (CustomerSession must also satisfy the EJB client specification) *Since `CustomerSession` creates a `Product` bean and invokes its business methods, General Specification 1 should be applied to `CustomerSession` as a client as well as a session bean.*

This case clearly shows that a bean can behave as a client as well as a server component by providing services (i.e. business methods) to other EJB components. Bean developers will naturally apply client-side programming to the `CustomerSession` bean. However, it is worth clearly stating that *any bean can be a client, and if it is used as a client then all the restrictions of clients must be carefully examined.*

As Figure 5.2 shows the `CustomerSession` bean creates a `Product` and calls the business method `findAllProducts` 5.1. Simply, we impose all the client's requirements on the `CustomerSession` bean.

This is formalised as follows:

```
CustomerSessionBean : EJBBeanClassType
ejbSatisfyClientRequirement? : [EJBBeanClassType → bool]

CustomerSession_is_client : AXIOM
   ejbSatisfyClientRequirement? (CustomerSessionBean)
```

The *CustomerSessionBean* is an instance of type *EJBBeanClassType*. The predicate *ejbSatisfyClientRequirement?* checks whether a given bean class satisfies the mandatory JNDI procedure.

Finally, the axiom *CustomerSession_is_client* specifies that the *CustomerSessionBean* holds the predicate *ejbSatisfyClientRequirement?*. □

This concludes our brief description of the example. It has shown the link between the application specific specification and general specifications associated with JavaBeans/EJB.

CHAPTER 6

Conclusion and Further Work

6.1 Conclusion

We have taken the first step towards our goal of providing a framework for JavaBeans and EJB. Firstly, we identified the key properties of JavaBeans and EJB. In the chapter on JavaBeans specification, we identified key properties of JavaBeans, which described the minimal valid JavaBean from an ordinary Java class. This was based on the syntax associated with JavaBeans.

In the EJB study, the original description was not adequate. Like the JavaBeans' case, no precise definition was given for either a session bean or an entity bean. We focused on the semantics of session beans by constructing several predicates and axioms based on these. The composition of the two state-transition machines in a parallel state transition diagram, capturing the requirements in a temporal logic style, and use of recursive definitions have been developed.

In the Jalapeño study, we showed how the general specifications of JavaBeans/EJB can be applied to a specific application. Firstly, the business methods of the normal sequence diagram are distinguished from normal methods by associating our predicates. Secondly, we extended the general JavaBeans/EJB requirements to the scenario specific deployment diagram, and each characteristic of the components was developed.

Other papers have focused on the connections of beans, container, and client or on the analysis of the flow semantics of beans. We focused on what JavaBeans/EJB is and provided a framework for it.

6.2 Further Work

We have not shown how the proof of the specification can be verified. While we have used PVS-HOL to specify the systems, the verification is beyond the scope of this project due to the size of the EJB specification. Thus, the effectiveness of our approach needs further investigation.

In Chapter 5, the formalisation of the Jalapeño business scenario showed that we need an extensible theory to connect from the general EJB specification to the specific case.

Finally, a translator, which translates from a given implementation to formulae, is vital. This would automate the translation of Java code to the bulk of the PVS specification, so that the proof process becomes the major focus.

Bibliography

[All97] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, CMU, School of Computer Science, January 1997. CMU/SCS Report CMU-CS-97-144.

[CDH$^+$00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu an d Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Softw are Engineering*, June 2000.

[CGP99] E. M. Clarke, Jr, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[fdr92] *Failures Divergence Refinement: User Manual and Tutorial, 1.2β.* Formal Systems (Europe) Ltd., Oxford, UK, 1992.

[Ham97] G. Hamilton. *JavaBeans^{TM} API Specification 1.01.* Sun Microsystems, 1997.

[Har00] D. Harel. From Play-In Scenarios To Code: An Achievable Dream. In T. Maibaum, editor, *Proceedings of FASE'00*, LNCS 1783, pages 22–34. Springer Verlag, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[JF00] H. Jubin and J. Friedrichs. *Enterprise JavaBean^{TM} by Example.* Prentice Hall PTR, Upper Saddle River, New Jersey, 2000.

[Kri00] P. Krishnan. Consistency Checks for UML. In *7th Asia Pacific Software Engineering Conference*, pages 162–169, Singapore, Dec 2000.

[MS] V. Matena and B. Stearns. *Applying Enterprise JavaBeans^{TM}: Component-Based Development for the J2EE^{TM} Platform.* Addison Wesley, Boston.

[NT01] S. Nakajima and T. Tamai. Behavioural Analysis of the Enterprise JavaBeans^{TM} Component Architecture. In *Model Checking LNCS 2057*, pages 163–182, 2001.

[OSRSC99a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, 1999.

[OSRSC99b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, 1999.

[OSRSC99c] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, 1999.

[PLC97] E. Pelegri-Llopart and L. P. G. Cable. *How to be a Good Bean*. Sun Microsystems, 1997.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[Rus97] J. Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. In *Tutorial presented at FORTE X/PSTV XVII '97: Formal Description Techniques and Protocol Specification, Testing and Verification*, November 1997.

[SG00] J. P. Sousa and D. Garlan. *Formal Modeling of the Enterprise JavaBeans^{TM} Component Integration Framework*. Technical report, School of Computer Science, Carnegie Mellon University, 2000.