Behavioural Analysis of the Enterprise JavaBeans^{TM} Component Architecture

Shin NAKAJIMA[1] and Tetsuo TAMAI[2]

[1] NEC Corporation, Kawasaki, Japan
[2] Graduate School of the University of Tokyo, Tokyo, Japan

Abstract. Rigorous description of protocols (a sequence of events) between components is mandatory for specifications of distributed component frameworks. This paper reports an experience in formalizing and verifying behavioural aspects of the Enterprise JavaBeans^{TM} specification with the SPIN model checker. As a result, some potential flaws are identified in the EJB 1.1 specification document. The case also demonstrates that the SPIN model checker is an effective tool for behavioural analysis of distributed software architecture.

1 Introduction

Software component technology is gaining importance in constructing large-scale distributed applications such as E-Commerce systems, and has also been widely accepted in the industry as a new technology for object-oriented software reuse [21]. Notable examples are COM/DCOM, JavaBeans/Enterprise JavaBeans, and a new component model of CORBA proposed by OMG. Systems can be constructed by implementing a component that encapsulates application logic and making it run with existing components in a pre-defined execution environment.

A *component* is a constituent of a system, and is a reusable unit that has a definite interface for exchanging information with other constituents. The main feature of the technology is an integration framework that is based on a specific computational model for components and supports the basic information exchange protocols among them. User-defined components can only be run successfully if they conform to the specification that the framework assumes. This implies that the integration framework specification should be described in an unambiguous manner [10].

Current component frameworks, however, have not been successful in providing precise specifications in their documents. Specification documents use a natural language (English) and informal diagrams for illustrative purposes. It is not uncommon to find ambiguities, inconsistencies or even *bugs* in the informal documents. Thus, the application of formal techniques is necessary for creating rigorous specification documents for component frameworks. Actually, Sullivan et al. [19] have identified some ambiguities in the description of the COM aggregation and interface negotiation specification by using a precise description written in the Z notation. Sousa et al. [18] have shown that modeling with Wright [1], an Architecture Description Language (ADL), is effective for behavioural analysis of the EJB 1.0 specification.

This paper reports an experience using the SPIN model checker [5] for behavioural analysis of the Enterprise JavaBeans component architecture. The main contributions of the present work can be summarized as follows. (1) Results show that the SPIN model checker is an adequate tool for behavioural analysis of distributed software architecture as compared with other tools such as Wright [1] or Darwin [12]. (2) The case identifies there are some potential flaws in the EJB 1.1 document [20] in terms of behavioural specification.

2 Enterprise JavaBeans

Enterprise JavaBeans^{TM} is a component architecture for distributed business applications. This section briefly describes the Enterprise JavaBeans (EJB) component architecture and presents some of its behavioural specifications. The material in this section is based on the EJB 1.1 specification document [20].

2.1 The Component Architecture

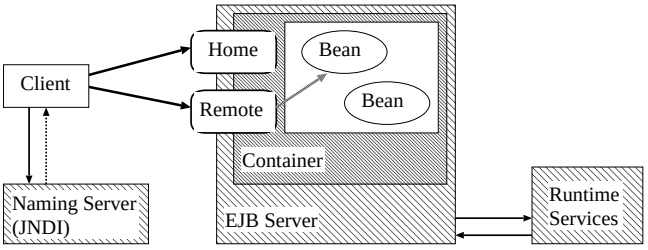

Fig. 1. The EJB Architecture

Figure 1 illustrates an overview of the EJB component architecture, as well as introduces several principal participants. `Client` is a client Java application that uses a `JNDI` Naming Server to look up an object reference of a `Home` interface. `Home` is a FactoryProxy responsible for creating a new `Bean` instance, and returns an object reference of a `Remote`, which is a RemoteProxy [9]. The `Remote` accepts requests from the `Client` and delegates them to the `Bean`, which executes these requests under the control of the `Container`. In addition, the architecture provides several runtime-services such as passivation, persistency, and garbage collection. All client requests to the `Bean` are made through the HomeProxy or the RemoteProxy objects. The `Container` may intercept the original requests and then control invocations on the `Bean`, while making it possible for the runtime-services to interleave between client request invocations.

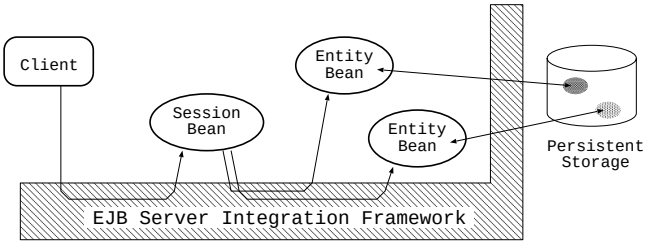

Fig. 2. Session and Entity Beans

There are two types of Enterprise JavaBeans : Session Beans and Entity Beans (Figure 2). An Entity Bean represents data stored in persistent storage and is a shared resource. The internal values of an Entity Bean are synchronized with the contents of persistent storage through the runtime service. A Session Bean, on the other hand, executes on behalf of a single client, and may implement long-lived transaction style application logic to access several Entity Beans.

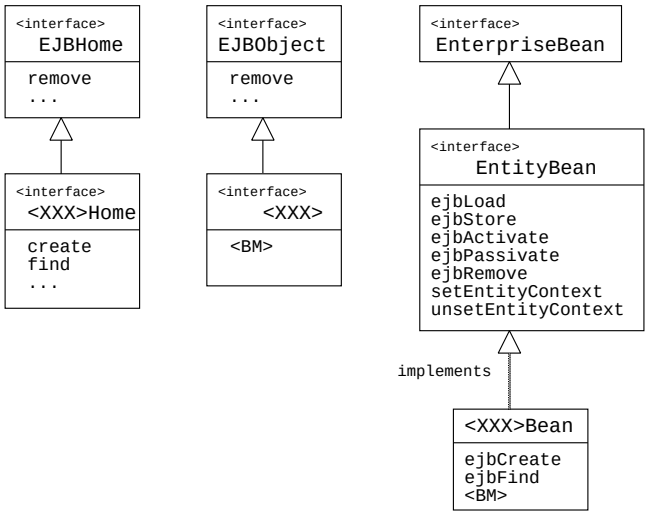

Fig. 3. Class Diagram

Figure 3 presents a fragment of the class diagram for Entity Beans. In the figure, `EJBHome`, `EJBObject`, `EnterpriseBean`, and `EntityBean` are the four Java interfaces that the EJB component framework provides. The `EJBHome` interface

describes the client's view of the `Home` shown in Figure 1, while `EJBObject` corresponds to the `Remote`. The `EntityBean` interface specifies the basic APIs that should be implemented by every Entity Bean. The `EnterpriseBean` is a common super-interface for both `EntityBean` and `SessionBean`[3].

In order to develop a new Entity Bean, three Java constructs must be defined: (1) `<XXX>Home` as a sub-interface of `EJBHome`, (2) `<XXX>` as a sub-interface of `EJBObject`, and (3) `<XXX>Bean` as an application class for implementing the `EntityBean`. The interface `<XXX>Home` adds application-specific `create` and `find` methods. The interface `<XXX>` should have application-specific business methods, which are denoted here by `<BM>`.

The class `<XXX>Bean` is a concrete bean definition that implements all of the methods specified by the `EntityBean` interface, as well as provides application-specific methods: `ejbCreate` and `ejbFind` are the counterparts of `create` and `find` of `<XXX>Home`, respectively. `<BM>` directly corresponds to `<BM>` of `<XXX>`. When a client invokes, for example, create method of `<XXX>Home`, the `Container` intercepts the request and then delegates it to `<XXX>Bean` instance in the form of an `ejbCreate` method invocation. Most of the methods defined in `EntityBean` are APIs for runtime-services. The passivation service uses `ejbPassivate` and `ejbActivate`, while the persistency service invokes `ejbLoad` and `ejbStore`.

The EJB 1.1 specification document describes the roles of the participants and all of the APIs in the form of Java interface descriptions. It is important to note that most of the methods are accompanied by exceptions in addition to normal functionalities. All explanations are written in a natural language.

2.2 Behavioural Specifications

Behavioural specifications show an external view of component architecture in terms of event sequences. With the EJB architecture, an event corresponds to a method invocation. Thus, the behavioural specifications consist of traces of method invocations. Below shows some example descriptions of behavioural aspects adapted from the EJB 1.1 specification document.

Figure 4 is a lifecycle model of an Entity Bean (adapted from Figure 23 on page 102 [20]). The lifecycle consists of three states, and each method contributes to a transition between the states. For example, a business method invoked by a client can only be executed when the Entity Bean is in its `ready` state. In view of the lifecycle model, the passivation service is an event initiator that makes the Entity Bean move between the `ready` and `pooled` states by using `ejbPassivate` and `ejbActivate`.

In addition, the document employs Object Interaction Diagrams (OIDs) to illustrate typical event sequences. Figure 5 is an example of an OID for the Entity Beans (simplified and adapted from Figure 29 on page 139 [20])[4]. The

[3] It is not shown in the figure.

[4] Division of labour between the two participants in a rectangle, `EJBObject` and `Container`, is dependent on a particular implementation, and thus the rectangle may sometimes be treated as one entity (on page 62 [20]).

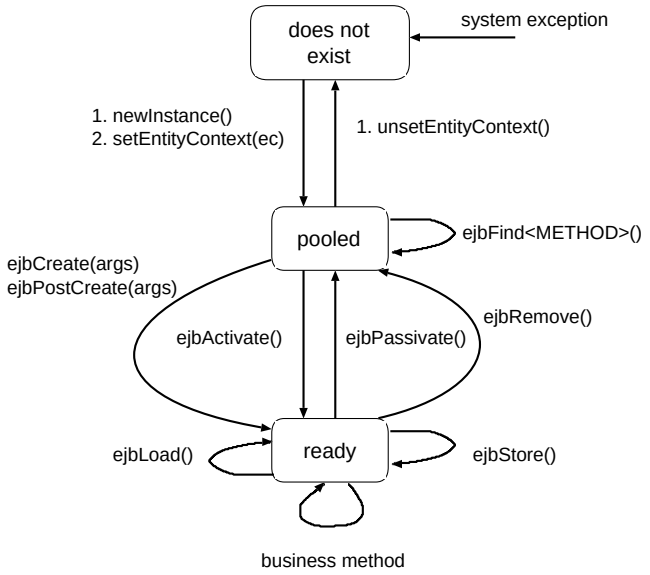

Fig. 4. Lifecycle of Entity Beans

diagram depicts various portions of event sequences that relate to a passivation service and business method invocation by a client.

A client request is directly delegated to the Entity Bean when the bean is in the `ready` state (see top of the diagram). The passivation service may spontaneously decide that the bean should be swapped out. This is accomplished through a sequence of method invocations; `ejbStore` followed by `ejbPassivate`. In Figure 4, it can be seen that the bean is now in the `pooled` state. At this point, a client may invoke another business method (see middle of the diagram). Since the bean is not in the `ready` state, the business method cannot be executed, and is thus suspended. The EJB server is then responsible for returning the bean to the `ready` state by issueing `ejbActivate` and `ejbLoad`. Finally, the suspended request is executed by the bean.

In addition to lifecycle models, the EJB 1.1 specification document describes behavioural aspects such as those described through OIDs. An OID, however, is just an example trace and is not a complete specification. One must infer from OIDs the intention of the original designer, as well as obtain precise specification descriptions from them. Furthermore, the document uses a natural language to describe temporal constraints related to particular runtime-service methods. For example, on p. 113 it says that the `Container`[5] invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance. These constraints are used as sources for behavioural properties to be verified.

[5] It is identified here with the entity represented by the rectangle in Figure 5.

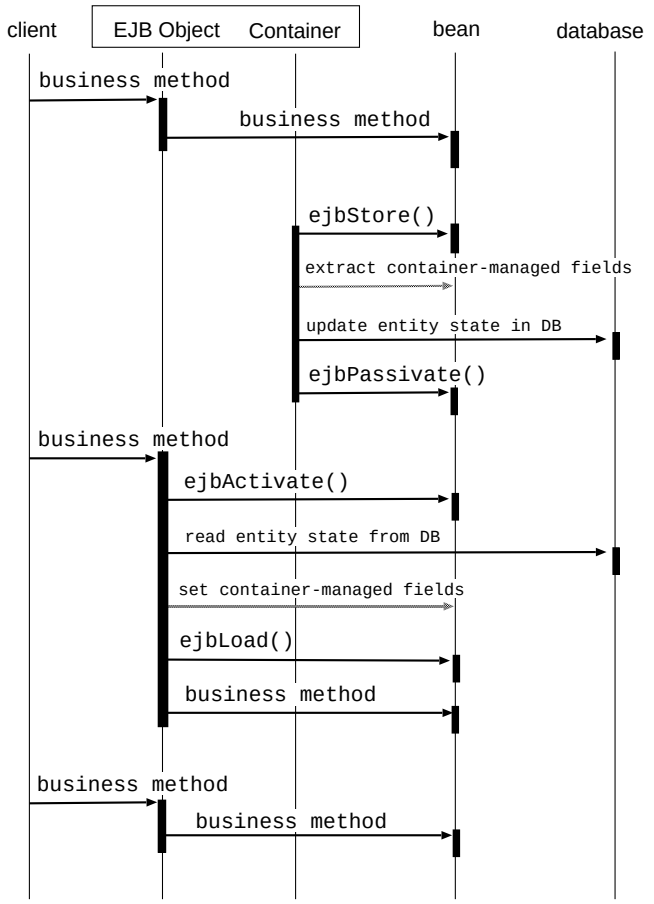

Fig. 5. Object Interaction Diagram

3 Formalization

This section focuses on the issues and approach related to formalizing the component architecture in Promela as compared with other specificands such as communication protocol and distributed algorithm [5] or program verification [6] which Promela has been used successfully.

3.1 Issues and Approach

Since the specificand, the EJB framework, has different characteristics from those that have been formalized and analyzed in Promela/SPIN, there are several issues that must be considered before formalization. First, the original specification is centered around APIs, and an API defines exceptional cases as well as

a normal functionality[6]. Each API is a Java method accompanied by possible application-specific and system exceptions.

Second, one cannot predetermine the behaviour of clients that access bean instances. Dependent on the possible client behaviour, the EJB framework must show valid behaviour regardless of the client. In particular, some clients may terminate, while others may not.

Third, since behavioural specification is based on event sequences, Linear-time Temporal Logic (LTL) formulae to represent properties should involve atomic proposition that describes an occurrence of a particular event, method invocation. In contrast, validity of atomic propositions in LTL is determined by states. This makes it necessary to devise a way to encode an event occurrence.

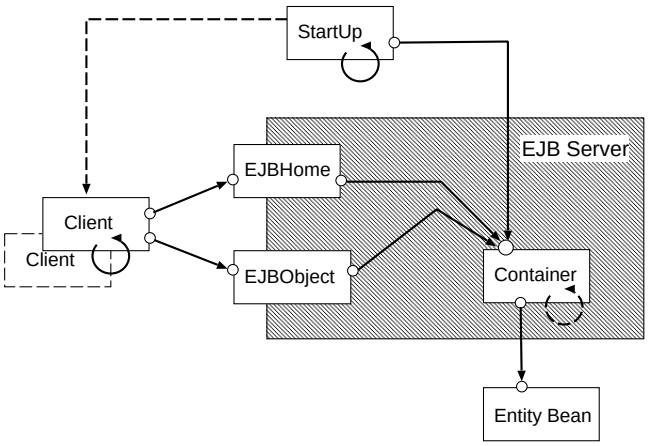

Fig. 6. Promela Processes

Figure 6 shows the overall configuration of the Promela processes. Three processes with a cyclic arrow are event initiators. The `Client` process issues a request such as create or business method, and is thus considered to be an event initiator. The role of `Container` is dual. It accepts events from other processes as well as generates events that implement runtime services such as passivation or persistency. The `StartUp` process needs further explanation. The `StartUp` process is responsible for setting up the initial environment necessary for a `Client` and a `Bean` to be executed properly. For example, Entity Beans are not required to be created by a client, but are created in advance. The `StartUp` process for such a case is responsible for creating and initializing the Entity Beans. Thus, introducing the `Client` and `StartUp` processes as event initiators is an approach that can be used to deal with the the second issue above. This

[6] The Wright group has observed similar characteristics in the cases [2][18].

approach also contributes to creating a well-structured formal model because the functionality of the EJB server and processes for setting up verification environment are clearly separated. In regard to the first and the third issues, one can make use of the Promela language constructs, which will be discussed in detail using actual Promela code in Section 3.2.

3.2 Promela Model

In order to describe method invocation on an object and return from the method, one first introduces two communication channels between the caller and calleé processes. And an extra channel is defined for transporting possible exceptions from the object.

```
#define  NBUF  1
chan mthd = [NBUF] of { short };
chan retv = [NBUF]  of { short };
chan excp = [NBUF] of { short };
```

The `chan mthd` is used for invoking method and messages flow from the `Container` to the `EntityBean`. The other two, `retv` and `excp`, are for messages going in the opposite direction. The channel definitions assume that the method name and the values are `short` and properly `#define`d.

The `EntityBean` in Figure 6 is represented by a simple Promela process that waits for method invocation from the `mthd` channel. The fact that any method can be invoked at any time is expressed using a `do...od` statement with an `endLoop` label. Each entry in `do...od` corresponds to a method, and its body is just an `if...fi` statement, which in turn has more than one branch. Each branch corresponds to either an exception or a normal termination. Since the `if...fi` has channel send statements in its guard positions, any of the branches can be executed at any time, which simulates a non-deterministic choice between exceptions and normal terminations. Therefore, one can encode exceptions in the Promela model, which resolves the first issue in Section 3.1 in a compact manner.

```
#define  ejbActivate   106
#define  ejbPassivate  110
#define  BM            120
...

proctype EntityBean ()
{
endLoop:
 do
  :: mthd?ejbActivate -> if :: retv!Void  :: excp!SysError fi
  :: mthd?ejbPassivate -> if :: retv!Void  :: excp!SysError fi
  :: mthd?BM -> if :: retv!Value :: excp!AppError :: excp!SysError fi
  ...
 od
}
```

The `EJBObject` in Figure 6 is an example process that terminates. It is a RemoteProxy object for accepting requests from the client, and delegating them to the `Container`, and it ends its lifecycle after handling of `remove` method by the client. Since an `EJBObject` is capable of accepting requests from more than one client[7], it must distinguish between each of the clients' requests. In the Promela description below, the `remote` channel carries two channels as well as a method name as its formal parameters. By sending different channel parameter values each time, it is possible to simulate a situation in which each message sent corresponds to a different method invocation event.

```
chan remote = [NBUF] of { short, chan, chan };
chan retval[NC] = [NBUF] of { short };
chan except[NC] = [NBUF] of { short };

proctype EJBObject()
{
   chan returnValue;   chan exceptionValue;   short value;

progressLoop:
endLoop:
  do
    :: remote?remove,returnValue,exceptionValue
       -> { request!reqRemove; retvalFC?value
            -> returnValue!value; goto endTerminate }
          unless { exceptFC?value -> exceptionValue!Error; goto endTerminate }

    :: remote?BusinessMethod,returnValue,exceptionValue
       -> { request!reqBM; retvalFC?value -> returnValue!value }
          unless { exceptFC?value -> exceptionValue!Error }
  od;

endTerminate:
   skip
}
```

The `Client` in Figure 6 is a Promela process that accepts two channel parameters when it runs. Since it invokes a request either on a HomeProxy or on a RemoteProxy, the `Client` is an event initiator that always sends messages via, for example, the `remote` channel, and waits for completion of the invoked method execution.

```
proctype Client(chan ret; chan exc)
{
  ...
MainLoop:
    do
```

[7] More precisely, the `EJBObject` for Session Bean should be ready for multiple requests while the one for Entity Bean need not.

```
    :: remote!BusinessMethod,ret,exc ->
        if :: ret?value -> skip  :: exc?value -> goto endClient fi
    :: remote!remove,ret,exc ->
        if :: ret?value -> goto wrapUp :: exc?value -> goto endClient fi
    :: home!remove,ret,exc;
        if :: ret?value -> goto wrapUp :: exc?value -> goto endClient fi
   od;
  ...
}
```

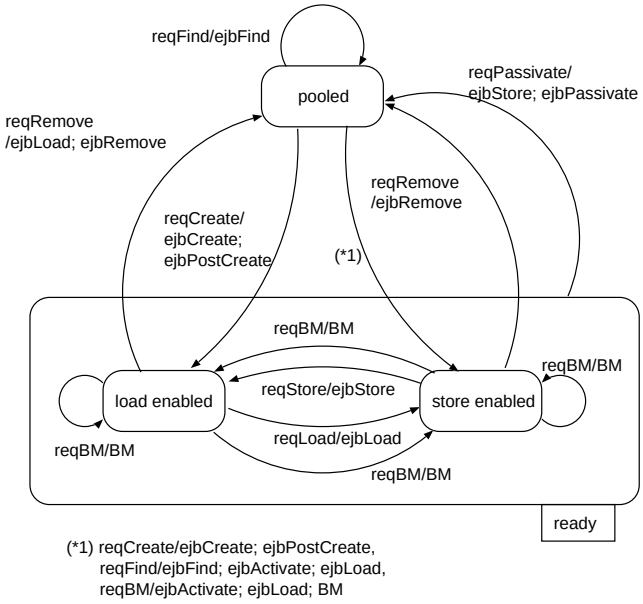

Fig. 7. Container for Entity Beans

As for the `Container` process, Figure 7 only shows the main behaviour in terms of the state transition diagram a lá Statechart [14] because the Promela code is too lengthy to be shown here[8]. The diagram is mainly derived from the lifecycle model in Figure 4, and appropriately augmented by studying the descriptions in other parts of the original document. The `ready` and `pooled` states in the diagram correspond to the states with the same labels in Figure 4; however, the `ready` state is modeled as a super-state with two sub-states. Such elaboration is needed for a proper interpretation of potential interference related to persistency service, which was found necessary during the formalization

[8] It is some 250 lines of Promela codes.

process.

The diagram shows all of the necessary transitions for implementing the main behaviour of the `Container`. For example, a transition from `pooled` to `load enabled` is notated as

```
reqCreate / ejbCreate;ejbPostCreate,
```

which indicates the following behaviour: when the `Container` receives a `reqCreate` from the `EJBHome`, the `Container` invokes two consecutive methods, `ejbCreate` and `ejbPostCreate`, on the `EntityBean`[9].

Finally, one goes back to the third issue in Section 3.1, which is related to representation for describing the occurrence of a particular event as an atomic proposition in LTL formulae. In the above description of `EntityBean`, one can observe that the process is, at a certain time, in a state in which it is ready to receive a particular event. When, for example, an `ejbActivate` message is at the head of the `mthd` channel, the `EntityBean` process can be considered to be in a state where `mthd?ejbActivate` is executable. It is also possible to check whether or not a state satisfies such a condition by using a square brackets notation such as `mthd?[ejbActivate]`, which can be used as a required atomic proposition representing a particular event occurrence.

The SPIN feature for automatic translation of LTL formulae can also be used. For example, as described in the last paragraph of Section 2.2, the EJB specification requires that the `Container` invokes at least one `ejbLoad` between `ejbActivate` and the first business method in the instance. To verify the property, the SPIN can be used to generate a *never automaton*, where the process description needs appropriate `#define`s.

```
spin -f "! [](q8 && <>q2) -> (! q2 U q12))"

#define  q2  mthd?[BusinessMethod]
#define  q8  mthd?[ejbActivate]
#define  q12 mthd?[ejbLoad]
```

Note that the LTL formula is *negated* because the SPIN model checker handles negative claims in the form of never automaton for verification.

4 Behavioural Analysis

This section presents some concrete examples of behavioural properties and the results of analysis.

4.1 Entity Beans

As discussed in Section 3.1, an appropriate client process is necessary for analyzing the behaviour of the EJB server. Three varieties of clients are formulated, and

[9] As can also be seen in Figure 4.

their behaviour can be compactly expressed in terms of event sequences generated by the client. (1) client-1 is a standard client that generates {*create*, *find*}; BM^*; *remove*, (2) client-2 starts with a find method (*find*; BM^*; *remove*), and (3) client-3 does not end with remove ({*create*, *find*}; BM^*). The individual client uses a somewhat different `StartUp` process because each run needs a different event sequence for setting up the environment to successfully start the client. A standard checking command for deadlock-freeness (`run -q`) is executed on each client model, and all succeed.

Next, various properties, formulated in terms of LTL formulae, are verified against the standard client-1. The first property of interest is the behaviour of invoking a business method (BM) on a passivated Entity Bean (Figure 4), and can be validated by the following two LTL formulae. When an Entity Bean instance is in the `pooled` state and a client requests a BM on the instance, `ejbActivate` is eventually executed (E1). The formula (E2) states that an `ejbLoad` is invoked between the `ejbActivate` and the BM on the instance.

$$\text{(E1)}\quad \Box(pooled \wedge BM\_Client \rightarrow \Diamond ejbActivate)$$

$$\text{(E2)}\quad \Box(ejbActivate \wedge \Diamond BM \rightarrow (\neg BM\ \mathbf{U}\ ejbLoad))$$

In a similar fashion, the EJB server invokes `ejbStore` between the last business method (`BM`) and `ejbPassivate`. (E3) is also satisfied.

$$\text{(E3)}\quad \Box(BM \wedge \Diamond ejbPassivate \rightarrow (\neg ejbPassivate\ \mathbf{U}\ ejbStore))$$

The EJB server must obey a set of "trivial progress" properties. The properties can be compactly expressed as (E4), showing that a client request M *leads to* an actual method invoked on an instance.

$$\text{(E4)}\quad \Box(M\_Client \rightarrow \Diamond M\_Bean)$$

At first, this seems to be of no interest at all because such properties are trivially satisfied. However, they actually reveal several interesting characteristics of the EJB server.

In the case of the client's create request, in which M_Client is `create` and M_Bean is `ejbCreate`, the expected formula is satisfied. The `ejbCreate` method, however, should be accompanied by an `ejbPostCreate` (Figure 4), and thus the LTL formula should be (E5).

$$\text{(E5)}\quad \Box(create \rightarrow \Diamond(ejbCreate \wedge \Diamond ejbPostCreate))$$

The formula becomes false because there are situations in which `ejbCreate` raises a system exception and thus `ejbPostCreate` is not executed. This shows that there is some complexity in taking into account the possible occurrence of exceptions, which is mandatory for behavioural analysis of the EJB server.

The next property has to do with "spontaneous allocation" of a fresh bean instance. The EJB server has the liberty of keeping more than one behaviourally equivalent Entity Beans and allocating them for client requests. (E6) is a property to represent one such behaviour, which is found to be true.

(E6) $\Box(ejbRemove \rightarrow \Diamond(BM\_Client \rightarrow \Diamond BM))$

Behaviour relating to a remove request is of particular interest because the trivial progress property (E7) does not hold.

(E7) $\Box(remove \rightarrow \Diamond ejbRemove)$

Analyzing the output trace reveals that a possible *livelock* caused by an infinite iteration of `ejbStore` and `ejbLoad` prevents the EJB server from the expected progress. In addition, (E7) can be checked by setting the weak-fairness flag (`run -f -a`). This filters out situations that has the previous livelock. The formula, however, still fails because of potential interference between `ejbRemove` and `ejbPassivate`. To study the situation, please refer to Figures 4 and 7. The state-diagrams show that two transitions, `ejbRemove` and `ejbPassivate`, are possible from the `ready` to the `pooled` state. Also `ejbRemove` is initiated by a client `remove`, while `ejbPassivate` is an event from the runtime passivation service that is generated in an asynchronous way independent of any client requests. The identified situation is that the bean instance moves to the `pooled` state by `ejbPassivate` while a client explicitly requests a `remove`. And thus it does not lead to an occurence of `ejbRemove`. The interference is a potential flaw in the EJB 1.1 specification document [20] that can be seen by careful examination of (E7).

It is also possible to try a trivial progress for the case of business method (BM).

(E8) $\Box(BM\_Client \rightarrow \Diamond BM\_Bean)$

The formula proves to be false because of a possible livelock of the `ejbStore` and `ejbLoad` as discussed in relation to (E7). Analysis under the weak-fairness condition also leads to a failure. This is because an internal exception occurs in the EJB server. For example, when a client issues a BM request on an instance in the `pooled` state, it involves more than one method execution (see E1 and E2). It is possible that one of the method, for example `ejbActivate`, raises an exception. If this is the case, the EJB server cannot continue the service on the instance, and thus the property does not hold. If the property (E9), which takes into account the exception, is used, it is found to be successful under the weak-fairness condition.

(E9) $\Box(BM\_Client \rightarrow \Diamond(BM\_Bean \vee Exception))$

As mentioned above, several trivial progress properties are not satisfied due to either possible exceptions or some other anomalous situation. However, it is easy to confirm that there exists at least one desirable sequence for satisfying each property. One may try a negation of a property that corresponds to a desirable behaviour expressed in a deterministic manner. (E10) is an example case for a business method (BM).

(E10) $\neg(\Diamond(BM\_Client \wedge \Diamond BM\_Bean))$

The SPIN protocol analyzer (`pan`) fails to show the correctness of the property and generates a sequence that leads to a failure. The sequence of minimal length is exactly what one would expect. Below is an edited output of the trace in the form of a Message Sequence Chart (MSC).

```
q\p   Container   EJBObject   Bean   Client
-------------------------------------------------
 13   .           .           .      remote!BM,6,8
 13   .           remote?BM,6,8
  1   .           req!BM
  1   req?BM
  3   mthd!BM
  3   .           .           mthd?BM
```

Therefore, it is true that at least one event sequence fulfills the trivial progress property for BM.

4.2 Session Beans

Session Beans have two options that can be specified at the time of deployment: stateful or stateless. The two options are not very different to the bean developer; however, they exhibit quite different runtime behaviour. Thus, formalization and analysis of Session Beans are actually conducted to model two independent behaviours, although some common behaviours are effective for both models. As in the case of Entity Beans, one introduces a variety of clients with an adequate `StartUp` process for each client case.

Common Behaviour One important property in terms of behavioural specification is in relation to concurrency control. According to the document, if a client request arrives at an instance while the instance is executing another request, the container must throw an exception for the second request. Session Beans must satisfy such a "non-reentrant" property. The property can be formulated in an LTL formula such as (S1).

$$\begin{aligned}\text{(S1)}\ \ &\Box(\mathit{Invoked}\_1 \wedge \Diamond \mathit{BM}\_2 \\ &\quad \wedge (\neg(\mathit{Return}\_1 \vee \mathit{Exception}\_1)\ \mathbf{U}\ \mathit{Exception}\_2) \\ &\rightarrow \Diamond(\mathit{Return}\_1 \vee \mathit{Exception}\_1))\end{aligned}$$

Some remarks on the formula are in order: (a) the server is in a state in which it has already accepted the first BM ($\mathit{Invoked}\_1$), (b) the second BM is requested afterward ($\Diamond \mathit{BM}\_2$), (c) the first BM is not completed before the second BM request ends with an exception ($\neg(\mathit{Return}\_1 \vee \mathit{Exception}\_1)\ \mathbf{U}\ \mathit{Exception}\_2$), and then (d) the first BM request results either in a normal termination or in an exception ($\Diamond(\mathit{Return}\_1 \vee \mathit{Exception}\_1)$).

As for concurrency control, two varieties of clients are used to examine the behaviour: (1) client-s1 is a simple process for invoking only one BM, (2) client-s2 is an infinite process that generates BMs iteratively (BM^+). In addition,

the standard check for deadlock-freeness is executed on the following three runs: (a) two client-s1's are involved, (b) two client-s2's are involved, and (c) three clients-s2's are involved. All are found to be deadlock-free; however, the size of the searched state space is different for each run (Table 1)[10].

Table 1. Analysis of Non-reentrant Property

Run	♯States	♯Transitions	Depth
(a)	3,742	8,217	66
(b)	6,161	12,225	295
(c)	75,769	167,884	986

Although the state space for the present EJB server model is not large in comparison to other published cases of practical interest, reducing the size by an appropriate abstraction is still important for efficient checking since the sizes drastically increase as in Table 1. To prove the property (S1), the case (a) was used because of the simplicity.

Stateful Session Beans In addition to the three client models explained above, two other models are used: (1) *create*; BM^*; *remove*, and (2) *create*; BM^*. The first client model is a standard one used in most analyses. The second one, however, is mandatory for properties involving *timeout*. The EJB server must generate a *timeout* event for garbage-collecting of possibly unused Stateful Session Bean instances either in the `method_ready` or in the `passive` state. In the case of the `method_ready` state, the timeout event must be followed by an `ejbRemove` on the instance. From the `passive` state, on the contrary, instances just disappear without any further events.

The timeout makes the situation somewhat complicated for a Stateful Session Bean. For example, the trivial progress property for business method (BM) is not satisfied. This is because an instance is forced to leave the `ready` state due to a possible timeout event. The only alternative is to verify the property (S2) that takes into account the timeout situation.

$$\text{(S2)}\quad \Box(BM\_Client \rightarrow \Diamond(BM\_Bean \vee timeout))$$

Stateless Session Beans For a client model of Stateless Session Beans, the simplest one capable of generating BM^* is sufficient. This is because the EJB server is responsible for the creation and deletion of Stateless Session Beans. A create request by a client becomes *no-op* for Stateless Session Beans.

Stateless Session Beans do not have any conversational state, and all bean instances are equivalent when they do not serve a client-invoked method. Stateless

[10] The metrics are the case for a Stateless Session Bean.

Session Beans are similar to Entity Beans in that the EJB server will allocate a suitable bean instance even if one has already been removed. This is because a Stateless Session Bean has a "spontaneous allocation" property that an Entity Bean has. This property can be confirmed by checking the LTL formula (E6) mentioned above.

A property specific to Stateless Session Beans is "spontaneous removal or automatic garbage collection." Stateless Session Beans do not allow an explicit remove operation by clients. The EJB server is responsible for deciding whether an instance is no longer necessary and invoking an `ejbRemove` method on it. In the present Promela model, the `Container` invokes `ejbRemove` in a non-deterministic manner. The LTL formula for the property is (S3), which says that `ejbRemove` is eventually invoked when the instance is in the `ready` state.

(S3) $\Box(ready \rightarrow \Diamond ejbRemove)$

The result is false due to a *livelock* of continuous BMs. The counter example sequence (livelock of BMs) is exactly the case in which the client accesses the bean heavily and thus the EJB server does not issue any `ejbRemove` on the bean. This confirms that timeout is generated only when the client is in a *think* time and does not invoke any BM at all. This is in accordance with the EJB 1.1 specification.

4.3 Discussion

The EJB 1.1 document [20] describes specifications from various viewpoints, thereby making it necessary to have a consistent model incorporating all of the viewpoints scattered throughout the document. As most of the descriptions are written in a natural language, the document uses many ambiguous "words." One specific example is related to the persistence service; invocations of the `ejbLoad` and `ejbStore` can be *arbitrarily* mixed with invocations of business methods. The intention of the author of the document is understandable, but, the word "arbitrarily" needs a concrete interpretation. The model in Section 3.2 adapts an interpretation that can be seen in the state-model in Figure 7. The model still leads to a livelock situation, which is revealed in the analysis.

Although the lifecycle model such as the one in Figure 4, can be used as a basis for formalizing the behavioural aspect, the model in the document is too simple. Reaching the final model in Figure 7 requires a thorough reading of the document and feedback from the analysis. This could be made easier if the EJB document had chapter(s) that concentrated on precise descriptions of behavioural aspects.

After obtaining a formal model, it is possible to conduct behavioural analysis in which properties are expressed in terms of LTL formulae. Thanks to numerous literature on the use of LTL formulae [3][8][13], one finds it not hard (though not easy either) to formulate properties using LTL formulae. It, however, still requires a trial-and-error in formulating and checking the LTL formulae. This is partly because a naive *leads-to* property such as (E4) does not hold in nearly

all cases. It was a surprise at first, but was found immediately from the fault traces that many interesting situations, either possible exceptions or other run-time anomalies, were not considered in the formulae. The trial-and-error process was a great help in understanding the behaviour of the EJB server. The SPIN model checker could be described as a *light-weight* design caclulator [16] used in iterative processes for refining the formal models.

Some LTL formulae were deemed to be false because of a possible livelock in the EJB server, some of which could be avoided by setting a weak-fairness flag. One must be very careful about properties proved under the fairness condition because a real system may incorporate a scheduler that behaves differently from what was assumed at the time of the analysis [4]. In the present study, however, the source of livelock is where the actual implementation should be taken care of. For example, a livelock with `ejbLoad` and `ejbStore` can be attributed to the present design artifact written in Promela, which is believed to be a faithful model in accordance with the presentation of the original document. This is understandable because the document does not mention anything about the implementation. As in proving (E7) and (E9), analysis under the fairness condition revealed other interesting situations, which are important to the present study. As shown in Section 4.1, proving (E7) results in a failure, which indicates that the bean instance moves to the `pooled` state by `ejbPassivate` even when a client explicitly requests a `remove`. The analyses have revealed potential flaws in the EJB 1.1 document.

In summary, most of the problems are identified in the formalization process in which a consistent model to integrate various aspects is obtained. Early stages of behavioural analysis can contribute to *debugging* the model as well as being a great help in understanding the specificand. Finally, note that the EJB 1.1 specification document [20] is not a design document. It is more or less abstract and lacks information detailed enough for use in software development. However, because the document is used as a *reference* for those involved in the EJB technology, more detailed description is mandatory. As the present study revealed with the formalization and analysis, improvements must be made in the EJB 1.1 specification document.

5 Comparisons

Behavioural analysis has been most successful in software architecture. Wright [1] and Darwin [12] are the two practical tools that have been applied to distributed software infrastructure of non-trivial complexities [2][11]. Comparing Wright and Darwin with the approach using the SPIN model checker in some degrees of detail is mandatory for discussing whether the SPIN model checker is an adequate tool for behavioural analysis of distributed software architecture.

Wright [1] follows the *Component-Connector* model of software architecture [17], which provides a general framework for describing various architecture styles from a unified viewpoint. To explain briefly, a *component* is a computational entity while a *connector* is an abstract infrastructure that *connects* the

participating components. *Connector* is responsible for how the components exchange information, which basically describes the behavioural specification of the specificand architecture. In particular, Wright adopts Communicating Sequential Process (CSP) as a rigorous notation for describing the behavioural specifications of *connectors*. The concrete syntax of Wright can be considered as a syntax-sugaring of a large CSP process description in a structured manner.

CSP is the essence of Wright in terms of behavioural specification, and its formal analysis can be conducted by means of FDR (Failures/Divergences Refinement) [15], a model-checker for CSP. Since FDR is based on failure-divergence semantics, various properties such as deadlock-freeness are checked through a refinement test ($\sqsubseteq$). Wright formulates several verification problems using its surface syntax.

Using FDR to analyze behavioural aspects of software architecture requires some familiarity with failure-divergence semantics and how to express various properties in terms of the refinement relationship, thus making it less accessible for a wide audience. Another drawback with Wright is that the specificand must be modeled as a connector if behavioural analysis is applied. In other words, Wright does not provide a methodology for analyzing systems with more than one connector.

Sousa et al. [18] apply Wright to formalizing and analyzing the EJB component integration framework as defined in the EJB 1.0 specification. The entire specification of the EJB server, including the Container and two client-accessible proxies, is modeled as a single large connector. Thus, traceability between the original specification and the resultant formal model is weak. The present formal model written in Promela (Section 3.2) shows a more intuitive mapping between the two; an object in the original document is modeled as a Promela process.

Additionally, Sousa et al. identify a potential flaw relating to an interference between delegation of business method and `ejbPassivate`, as well as provide a remedy. They also discuss that the flaw may be due to their modeling but not to the EJB 1.0 specification. The same flaw, however, did not manifest itself in the present case study with Promela. On the contrary, the study in Section 4.1 identifies other flaws such as one relating to a potential interference between an execution of `remove` request and `ejbPassivate`.

Darwin [12] uses diagram notation for the structural aspects and FSP (Finite State Processes), a variant of CSP, to describe behavioural specifications, which can be analyzed by the LTSA (Labeled Transition System Analyzer) model checker. The basic model of Darwin is the *Component-Port* model, and does not have explicit notion of Connector. The model is basically equivalent to the Promela model; a component and a port can be mapped to a Promela process and a channel respectively.

LTSA is a well-designed model-checker that can be used even by a novice. LTSA is easy to use, but restricts itself in terms of the power of verification. For safety analysis, a deterministic FSP process (property automaton) is used to show *correct* behavior. LTSA model checks a product of the target and the property automaton. For liveness analysis, FSP provides a declarative way to

specify a set of progress labels, which is equivalent to $\Box\Diamond q$ and $\Box(p \rightarrow \Diamond q)$ in LTL formulae, and is less expressive than the full LTL used in the SPIN model checker.

Sullivan et al. [19] formalize structural aspects of the COM model in the Z notation, and point out that there is a conflict between aggregation and interface negotiation in the original specification. This is a successful non-trivial result of applying formal methods to component architectures. Jackson and Sullivan [7] employ Alloy to formalize the model, which was originally formulated in the Z notation, and uses Alcoa, an automatic analysis tool to show that the same flaws manifest themselves in the specification. Thus, they demonstrate the effectiveness of the automatic analysis tool. The present case study deals with the behavioural aspects of the EJB framework, and it does not deal with the structural ones. The use of both approaches may be necessary for analyzing advanced component architectures.

Kobryn [9] uses a UML Collaboration diagram to model component architectures. First, a pattern for component frameworks is introduced, and then the pattern is instantiated to the two important frameworks, EJB and COM+. Since the approach makes use of UML Collaboration diagram, the main concern is to illustrate the participant roles and structural relationships between the participants. Behavioural analysis is not conducted. The present case study concentrates on behavioural analysis using the SPIN model checker, but was limited to the EJB framework. However, it can be easily applied to other component frameworks such as COM+ because COM+ and EJB can be instantiated from a general pattern, as illustrated by Kobryn in his paper. Finally, the configuration of the Promela processes in Figure 6 is almost identical to that of Kobryn's pattern[11]. This ensures that the model in the present case study is *natural*, and thus shows a sufficient traceability with the original document.

6 Conclusion

This paper describes how the SPIN model checker was used for behavioural analysis of the Enterprise JavaBeans component architecture. This is a case study on applying a model-checking technique to a non-trivial real-world software artifact. Using concrete examples, the present work was able to demonstrate that the SPIN model checker can be used as an effective tool for behavioural analysis of distributed software architecture. Further, the case was also able to successfully identify several potential flaws in the EJB 1.1 specification document.

Finally, further work is needed on integration of the UML-based approach, for example, in [9] and the SPIN-based model that is amenable to automatic behavioural analysis. This is inevitable for the model-checking technology to gain a wide acceptance from software engineers.

[11] The authors did not know about the work in [9] before writing this paper.

References

1. Allen, R. and Garlan, D. : Formalizing Architectural Connection, Proc. ACM/IEEE ICSE'94 (1994).
2. Allen, R., Garlan, D., and Ivers, J. : Formal Modeling and Analysis of the HLA Component Integration Standard, Proc. ACM SIGSOFT FSE'98, pp.70-79 (1998).
3. Dwyer, M.B., Avrunin, G.S., and Corbett, J.C. : Patterns in Property Specifications for Finite-State Verification, Proc. ACM/IEEE ICSE'99 (1999).
4. Godefroid, P. and Holzmann, G.J. : On the Verification of Temporal Properties, in Proc. PSTV'93, pp.109-124 (1993).
5. Holzmann, G.J. : The Model Checker SPIN, IEEE trans. SE, vol.23, no.5, pp.279-295 (1997).
6. Holzmann, G.J. and Smith, M.H. : Software Model Checking: Extracting Verification Models from Source Code, Proc. FORTE/PSTV'99 (1999).
7. Jackson, D. and Sullivan, K. : COM Revisited: Tool-Assisted Modelling and Analysis of Complex Software Structures, Proc. ACM SIGSOFT FSE'00 (2000).
8. Janssen, W., Mateescu,R., Mauw, S., Frennema, P., and van der Stappen, P. : Model Checking for Managers, Proc. 6th SPIN Workshop (1999).
9. Kobryn, C. : Modeling Components and Frameworks with UML, Comm. ACM, vol.43 no.10, pp.31-38 (2000).
10. Leavens, G. and Sitaraman, M. (ed.) : *Foundations of Component-based Systems*, Cambridge University Press 2000.
11. Magee, J., Kramer, J., and Giannakopoulou, D. : Analysing the Behaviour of Distributed Software Architectures: a Case Study, Proc. IEEE FTDCS'97 (1997).
12. Magee, J., Kramer, J., and Giannakopoulou, D. : Software Architecture Directed Behavior Analysis, Proc. IEEE IWSSD'98, pp.144-146 (1998).
13. Manna, Z. and Pnueli, A. : *The Temporal Logic of Reactive and Concurrent Systems : Specification*, Springer-Verlag 1991.
14. OMG : OMG Unified Modeling Language Specification v1.3 (2000).
15. Roscoe, A.W. : *The Theory and Practice of Concurrency*, Prentice Hall 1998.
16. Rushby, J. : Mechanized Formal Methods: Where Next?, Proc. FM'99, pp.48-51 (1999).
17. Shaw, M. and Garlan, D. : *Software Architecture*, Prentice Hall 1996.
18. Sousa, J. and Garlan, D. : Formal Modeling of the Enterprise JavaBeans^{TM} Component Integration Framework, Proc. FM'99, pp.1281-1300 (1999).
19. Sullivan, K., Marchukov, M., and Socha, J. : Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model, IEEE trans. SE, vol.25, no.4, pp.584-599 (1999).
20. Sun Microsystems, Inc. : Enterprise JavaBeans^{TM} Specification, v1.1 (1999).
21. Szyperski, C. : Components and the Way Ahead, in [10], pp.1-20 (2000).

This article was processed using the LaTeX macro package with LLNCS style