

Formal Modeling of the Enterprise JavaBeans™ Component Integration Framework

João Pedro Sousa and David Garlan

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
{jpsousa|garlan}@cs.cmu.edu
<http://www.cs.cmu.edu/~able/>

Abstract. An emerging trend in the engineering of complex systems is the use of component integration frameworks. Such a framework prescribes an architectural design that permits flexible composition of third-party components into applications. A good example is Sun Microsystems' *Enterprise JavaBeans™ (EJB)* framework, which supports object-oriented, distributed, enterprise-level applications, such as account management systems. One problem with frameworks like EJB is that they are documented informally, making it difficult to understand precisely what is provided by the framework, and what is required to use it. We believe formal specification can help, and in this paper show how a formal architectural description language can be used to describe and provide insight into such frameworks.

Keywords: Software architecture, software frameworks, component integration standards, component-based software, Enterprise JavaBeans.

1 Introduction

Component integration frameworks¹ are becoming increasingly important for commercial software systems. The purpose of a component integration framework is to prescribe a standard architectural design that permits flexible composition of third-party components. Usually a framework defines three things: (a) the overall structure of an application in terms of its major types of constituent components; (b) a set of interface standards that describe what capabilities are required of those components; and (c) reusable infrastructure that supports the integration of those components through shared services and communication channels.

A successful framework greatly simplifies the development of complex systems. By providing rules for component integration, many of the general problems of component mismatch do not arise [8]. By providing a component integration platform for third-party software, application developers can build new

¹ Component integration frameworks are sometimes referred to as *component architectures*

applications using a rich supply of existing parts. By providing a reusable infrastructure, the framework substantially reduces the amount of custom code that must be written to support communication between those parts.

A good example of a framework is Microsoft's Visual BasicTM system, which defines an architecture for component integration (Visual Basic Controls), rules for adding application-specific components (such as customized widgets, forms, graphics, etc.), and code that implements many shared services for graphical user interfaces (for example, to support coordination and communication among the parts via events.)

Another, more recent example is Sun's Enterprise JavaBeansTM (EJB) architecture. EJB is intended to support distributed, Java-based, enterprise-level applications, such as business information management systems. Among other things, it prescribes an architecture that defines a standard, vendor-neutral interface to information services including transactions, persistence, and security. It thereby permits application writers to develop component-based implementations of business processing software that are portable across different implementations of those underlying services.

One critical issue for users and implementors of a framework is the documentation that explains what the framework provides and what is required to instantiate it correctly for some application. Typically a framework is specified using a combination of informal and semi-formal documentation. On the informal side are guidelines and high-level descriptions of usage scenarios, tips, and examples. On the semi-formal side one usually finds a description of an application programmer's interface (API) that explains what kinds of services are provided by the framework. APIs are formal to the extent that they provide precise descriptions of those services – usually as a set of signatures, possibly annotated with informal pre- and post-conditions.

Such documentation is clearly necessary. However, by itself it leaves many important questions unanswered – for component developers, system integrators, framework implementers, and proposers of new frameworks. For example, the framework's API may specify the names and parameters of services provided by the infrastructure. However, it may not be clear what are the restrictions (if any) on the ordering of invocations of those services. Usage scenarios may help, but they only provide examples of selected interactions, requiring the reader to infer the general rule. Moreover, it may not be clear what facilities *must* be provided by the parts added to the framework, and which are optional.

As with most forms of informal system documentation and specification, the situation could be greatly improved if one had a precise description as a formal specification of the framework. However, a number of critical issues arise immediately. What aspects of the framework should be modeled? How should that model be structured to best expose the architectural design? How should one model the parts of the framework to maintain traceability to the original documentation, and yet still improve clarity? How should one distinguish optional from required behavior? For object-oriented frameworks what aspects of the object-oriented design should be exposed in the formal model?

In this paper we show how one can use formal architectural modeling to provide one set of answers to these questions. The key idea is to provide an abstract structural description of the framework that makes clear what are the high-level interfaces and interactions, and to characterize their semantics in terms of protocols. By making explicit the protocols inherent in the integration framework, we make precise the requirements on both the components and on the supporting infrastructure itself. This in turn yields a deeper understanding of the framework, and ultimately supports analysis of its properties. Furthermore, we can validate that the model is a useful abstraction of “reality” by checking that the model exhibits the properties that are required informally in the specification of the software framework.

In the remainder of this paper we describe our experience in developing a specification of Sun’s Enterprise JavaBeans integration framework. The primary contributions of this paper are twofold. First, we show how formal architectural models based on protocols can clarify the intent of an integration framework, as well as expose critical properties of it. Second, we describe techniques to create the model, and structure it to support traceability, tractability, and automated analysis for checking of desirable properties. These techniques, while illustrated in terms of EJB, shed light more generally on ways to provide formal architectural models of object-oriented frameworks.

2 Related Research

This work is closely related to three areas of prior research. The first area is the field of *architectural description and analysis*. Currently there are many architecture description languages (ADLs) and tools to support their use (such as [11], [17], [14], [13]). While these ADLs are far from being in widespread use, there have been numerous examples of their application to realistic case studies. This paper contributes to this body of case studies, but pushes on a different dimension – namely, the application of architectural modeling to component integration frameworks.

Among existing ADLs the one used here, Wright, is most closely related to Rapide [11], since both use event patterns to describe abstract behavior of architectures. Wright differs from Rapide insofar as it supports definition of connectors as explicit semantic entities and permits static analysis using model checking tools. As we will see, this capability is at the heart of our approach for modeling integration frameworks.

The second related area is research on the *analysis of architectural standards*. An example close in spirit to our work is that of Sullivan and colleagues, who used Z to model and analyze the Microsoft COM standard [18]. In our own previous work we looked at the High Level Architecture (HLA) for Distributed Simulation [2]. HLA defines an integration standard for multi-vendor distributed simulations. We demonstrated that Wright could be used to model this framework and identify potential flaws in the HLA design. EJB differs from HLA in that it provides a different set of challenges. In particular, unlike HLA, EJB is

an object-oriented framework; it has a diverse set of interface specifications; and it has weaker (but more typical) documentation.

The third related area is *protocol specification and analysis*. There has been considerable research on ways to specify protocols using a variety of formalisms, including I/O Automata [12], SMV [4, 5], SDL [10], and Petri Nets [15]. While our research shares many of the same goals, there is one important difference. Most protocol analysis assumes one is starting with a complete description of the protocol. The problem is then to analyze that protocol for various properties. In contrast, in architectural modeling of systems like EJB, protocols are typically *implicit* in the APIs described in the framework documentation. Discovering what the protocols are, and how they determine the behavior of the system is itself a major challenge.

3 Enterprise JavaBeans™

3.1 Background

One of the most important and prevalent classes of software systems are those that support business information applications, such as accounting systems and inventory tracking systems. Today these systems are usually structured as multi-tiered client-server systems, in which business-processing software provides services to client programs, and in turn relies on lower level information management services, such as for transactions, persistence, and security (see Fig. 1.)

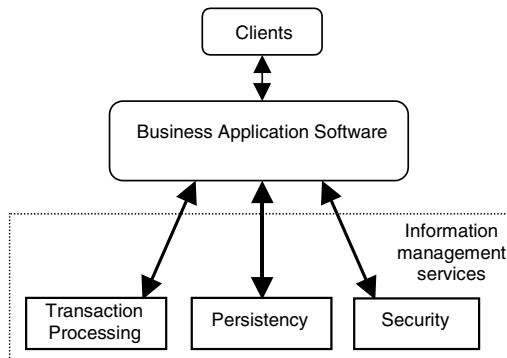


Fig. 1. A three-tiered business application

Currently one of the problems with writing such software is portability: application software must be partially rewritten for each vendor's support facilities because information management services provided by different vendors often have radically different interfaces.

Additionally, clients of application software are faced with a huge variety of interfaces to those applications. While some differences are inevitable, given that different applications must provide different capabilities, one would wish for certain levels of standardization for generic operations such as creating or deleting business process entities (such as accounts).

To address this problem several vendors have proposed component integration frameworks for this class of system. One of these is Sun Microsystems' Enterprise JavaBeans™ framework, a component architecture for building distributed, object-oriented, multi-vendor, business applications in the Java programming language. The basic idea of the framework is to standardize on three things. First, the framework defines a standard interface to information management services, insulating application software from gratuitous differences in vendors' native interfaces. Second, the framework defines certain standard operations that can be used by client software to create, delete, and access business objects, thereby providing some uniformity across different business applications software. Third, the framework defines rules for composing object-oriented business applications using reusable components called *beans*.

By standardizing on these aspects of an information management application, EJB intends to promote application portability, multi-vendor interoperability, and rapid composition of applications from independently developed parts.

The remainder of this section elaborates on the elements of EJB that are necessary to follow the formalization in Sect. 6.

3.2 Overview of Enterprise JavaBeans™

Sun's "Specification of the Enterprise JavaBeans™ Architecture" [6], (henceforth, *EJB spec*) defines a standard for third parties to develop Enterprise JavaBeans™ deployment environments (henceforth, *EJB servers*). An application running in one of these environments would access information management services by requesting them of the EJB server, via the EJB API, in the way prescribed by the EJB spec.

Figure 2 illustrates a system with a remote client calling an application that implements some business logic, for which Orders and Accounts are relevant operational entities. In the object-oriented paradigm, such entities are termed *objects*. An object can be viewed as a unit that holds a cohesive piece of information and that defines a collection of operations (implemented by methods) to manipulate it.

The EJB framework defines particular kinds of objects, termed Enterprise JavaBeans™ (*beans*, for short). Beans must conform to specific rules concerning the methods to create or remove a bean, or to query a population of beans for the satisfaction of some property. Hence, whenever client software needs to access a bean, it can take some features for granted.

It is the job of EJB server *providers* to map the functionality that the EJB spec describes into available products and technologies. In version 1.0, released in March 1998, the EJB spec covers transaction management, persistence, and

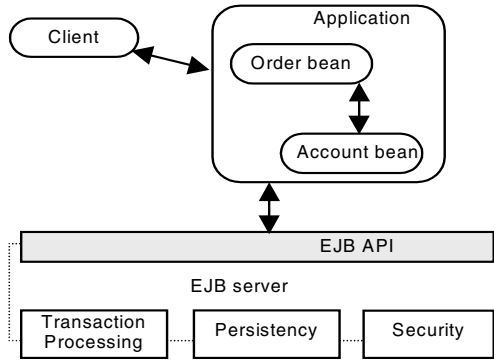


Fig. 2. The EJB server offering access to information management services.

security services.² The EJB spec does not regulate how these services are to be implemented, however: they may be implemented by the EJB server provider, as part of the server; or they may rely on external products, eventually supplied by other vendors. Such products, however, are invisible to the beans.

A typical example of the symbiosis between an EJB server and an external product would be for an EJB server provider to offer access to one or more industry standard databases. The customer organization could then develop new applications that access existing corporate databases, using the persistency services provided by the EJB server. All that the developers of the new application would need to be aware of is the logical schema of the existing databases.

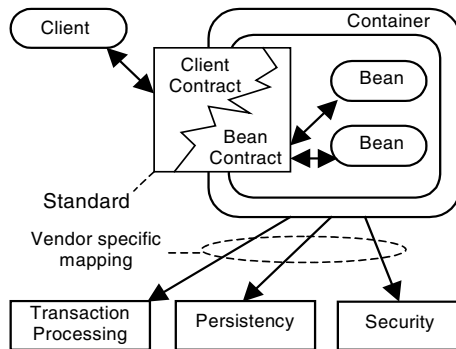


Fig. 3. The EJB container.

² Actually, version 1.0 views persistency services to be optional.

The EJB spec refers to the collection of services that both the beans and the client software use as a *container* (see Fig. 3). A container provides a deployment environment that wraps the beans during their lifecycle. Each bean lives within a container. The container supports (directly or indirectly) all aspects that the bean assumes about the outside world, as defined in the EJB spec.³ The protocols that regulate the dialog between a bean and its container are termed the *bean contract*.

The container also supports a set of protocols, termed the *client contract*, that regulate the dialog between client software and a bean. The client contract defines two interfaces that a client uses to communicate with a specific bean: the *Home Interface* and the *Remote Interface*. Both interfaces are implemented at deployment-time by special-purpose tools supplied by the EJB server provider.⁴ The Remote Interface reflects the functionality of the bean it represents, as it publishes the so-called business methods of the bean. Each bean has one such interface. The Home Interface contains the methods for creation and removal of beans, as well as optional methods for querying the population of beans (*finder methods*). There is one such interface per bean class.

To use the services of a bean a client first obtains a reference to the bean's class Home Interface using the Java Naming and Directory Interface™ (JNDI). Using this reference, the client software can call a `create` method in the class's Home Interface, thus obtaining a reference to the bean's Remote Interface implemented by the container. The Remote Interface then delegates subsequent method calls to the corresponding bean. The fact that the client uses JNDI to obtain a reference to the Home Interface of the class is a necessary condition for distribution transparency. Any piece of software, including a bean, may use the client contract to communicate with some bean if the software does not know (or care) where the target bean is actually being deployed. Such software calls the interfaces in the container holding the target bean using Java's Remote Method Invocation.

An EJB server manages the population of beans that reside in main memory in a way that is transparent to the client software. As the population of beans inside a container grows beyond a certain limit, determined by the EJB server, the container sends some number of the least recently used beans to secondary memory. The EJB spec refers to the beans that are subject to this operation as *passivated*. Since every call to a bean flows through the interfaces in the

³ This does not mean the container restrains beans from accessing the world outside EJB. For instance, a bean may include Java Database Connectivity (JDBC) code to access a database directly. However, in doing so, the bean sacrifices implementation independence and distribution transparency.

⁴ In Java, the Home and Remote Interface are termed `EJBHome` and `EJBObject`, respectively. These two interfaces in the EJB spec are *extended* by user-written, domain-specific, Java interfaces. Such domain-specific Java interfaces are read by the deployment tools to produce the container-specific classes that implement the two interfaces. The latter classes are, however, invisible to the user. For the sake of clarity we will continue to refer to the user-specified interfaces as Home and Remote Interface.

container, it is the container that relays the call to the bean, as appropriate. So, whenever a method call is addressed to a passivated bean, the bean is brought back to primary memory by the container. The EJB spec refers to beans that are subject to this latter operation as *activated*.

Although passivation and activation are transparent to the client calling the bean, it is not so to the bean itself. Before being passivated, the bean is required to release the shared resources it acquired previously, so as not to lock them during passivation time. Likewise, upon activation, the bean may have to reacquire the resources to serve the client's request. Therefore, in order to allow the bean to perform these actions, the container issues synchronization messages to the bean just before passivation and immediately after activation, before the client's call is relayed (`ejbPassivate` and `ejbActivate`, in Fig. 4.)

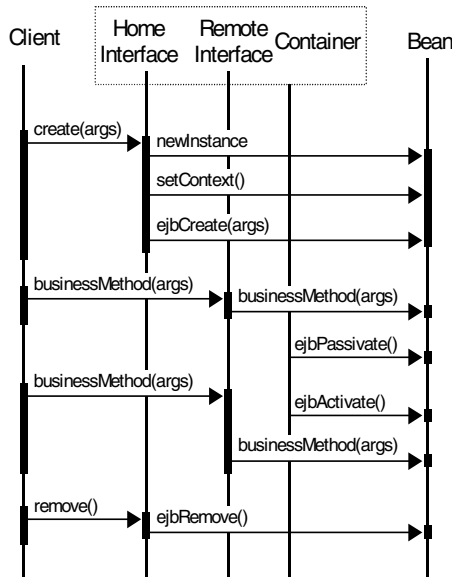


Fig. 4. Sample event trace for the lifecycle of a bean.

3.3 The Enterprise JavaBeans™ Specification

The EJB spec [6] released by Sun is a 180-page document, in which the concepts and their interplay are described in English, much in the same way as Sect. 3.2. A few informal state diagrams complement the explanation. There are also some chapters dedicated to the presentation of illustrative scenarios of interactions described using event trace diagrams. For instance, the event trace in Fig. 4 is an adaptation of the ones in pages 32 to 36 of the EJB spec. The document has

an appendix enumerating the Java API that the elements of the architecture should follow. The signature and purpose of each method is briefly described, in English, along with an enumeration of the exceptions that may be raised. No pre- and post-conditions are provided.

Although voluminous, documentation such as this has two intrinsic problems. First, related information is spread throughout the document. For example, to determine what sequence of method calls a bean must follow to request a typical service from the container, the reader must locate the explanation in the text (hopefully covering all relevant operations), refer to the API method descriptions, examine any examples of sample executions, and consult the list of possible raised exceptions.

Second, the lack of a precise definition makes it difficult for a reader to resolve inconsistencies and ambiguities, and to determine the intended semantics of the framework. As an example of unresolvable inconsistencies, in one place the documentation says the Home Interface should “define *zero* or more `create` methods” (page 14), while in another it says “*one* or more `create` methods” (page 20). Without a single place in the document that has the precise definition, it is impossible to determine which of the two (if either) is correct (even assuming we can determine what a `create` method should do).

As another example, consider the issue of the interaction between bean deletion and bean passivation. Suppose a client decides to remove a bean that the client has not accessed in some time. If the container has passivated that bean, it is not clear what happens. The normal rules of method invocation would imply that the bean would first have to be activated (reacquiring all resources needed for its normal operation), only to be immediately removed. This seems like a strange kind of behavior, and it is not clear if it is intended by the standard.

Finally, as with any documentation that only provides *examples* of method sequences, rather than formal *rules*, it is impossible for a reader to be sure what generalization is intended.

It seems clear that much could be gained by a formal unambiguous specification of EJB as a supplementary (or even central) resource for framework implementers, bean providers, and developers of client software. In the remainder of this paper we examine one such specification.

4 Wright

Wright is a formal language for describing software architecture. As with most architecture description languages, Wright describes the architecture of a system as a graph of components and connectors. Components represent the main centers of computation, while connectors represent the interactions between components. While all architecture description languages permit the specification of new component types, unlike many languages, Wright also supports the explicit specification of new architectural connector types [1].⁵

⁵ Wright also supports the ability to define architectural styles, check for consistency and completeness of architectural configurations, and check for consistent specifica-

A simple Client-Server system description is shown below:

```

Configuration SimpleExample
  Component Server
    Port Provide = <provide protocol>
    Computation = <Server specification>
  Component Client
    Port Request = <request protocol>
    Computation = <Client specification>
  Connector C-S-connector
    Role Client = <client protocol>
    Role Server = <server protocol>
    Glue = <glue protocol>
  Instances
    s: Server
    c: Client
    cs: C-S-connector
  Attachments
    s.Provide as cs.Server;
    c.Request as cs.Client
end SimpleExample.

```

This example shows three basic elements of a Wright system description: component and connector type declarations, instance declarations, and attachments. The instance declarations and attachments together define a particular system configuration.

In Wright, the description of a component has two important parts, the *interface* and the *computation*. A component interface consists of a number of *ports*. Each port defines a point of interaction through which the component may interact with its environment.

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. A Wright description of a connector consists of a set of *roles* and the *glue*. Each role defines the allowable behavior of one participant in the interaction. A pipe has two roles, the source of data and the recipient. The glue defines how the roles will interact with each other.

The specification of both components and connectors can be parameterized, either with a numeric range – allowing a variable number of ports or roles with identical behaviors – or with a process description – instantiating the generic structure of a component (or connector) to a specific behavior. A typical case of parameterization is a Client-Server connector that allows the attachment of a variable number of Clients, multiplexing their requests according to rules defined in the glue protocol:

tions of components and connectors. In this paper we restrict our presentation to just those parts of Wright that concern the specification of EJB. See [3] for further details.

```

Connector C-S-connector(nClients:1..)
  Role Client1..nClients = <client protocol>
  Role Server = <server protocol>
  Glue = <client multiplexing glue protocol>

```

Each part of a Wright description – port, role, computation, and glue – is defined using a variant of CSP [9]. Each such specification defines a pattern of events (called a process) using operators for sequencing (“ \rightarrow ” and “ $;$ ”), choice (“ \sqcap ” and “ \sqbox ”), parallel composition (“ \parallel ”) and interruption (“ \triangle ”).⁶

Wright extends CSP in three minor syntactic ways. First, it distinguishes between *initiating* an event and *observing* an event. An event that is initiated by a process is written with an overbar. Second, it uses the symbol \S to denote the successfully-terminating process.⁷ (In CSP this is usually written “SKIP”.) Third, Wright uses a quantification operator: $\langle \text{op} \rangle x : S \bullet P(x)$. This operator constructs a new process based on the process expression $P(s)$, and the set S , combining its parts by the operator $\langle \text{op} \rangle$.

For example, $\sqcap i:1,2,3 \bullet P_i = P_1 \sqcap P_2 \sqcap P_3$.

5 Component or Connector?

When defining the architectural structure of a framework, a key question is what are the connectors. This question is important because many frameworks are essentially concerned with providing mediating infrastructure between components that are provided by the user of the framework. Making a clear distinction between the replaceable componentry, and the mechanisms that coordinate their interaction greatly improves the comprehensibility of the framework.

From our perspective, the entities that are a locus of application-specific computation are best represented as components. The infrastructure that is prescribed by the framework to assure the interconnection between application components is a likely candidate to be represented as a (set of) connector(s).

In general, however, it may not always be obvious what should be represented as a component and what should be represented as a connector. Consider the system illustrated in Fig. 5a, consisting of three components: A, B, and C. In some cases the purpose of C is to enable the communication between A and B, using an A-C protocol over connector X, and a C-B protocol over connector Y. If those two protocols are completely independent, it makes sense to represent C as a distinct component, and keep X and Y as separate connectors.

On the other hand, if events on X are tightly coupled with those on Y (or vice versa), then it may make more sense to represent the protocol between X

⁶ We assume familiarity with CSP. For details on the semantics of the mentioned operators see the extended version of this paper in electronic format, available from Springer Verlag.

⁷ Wright uses a non-standard interpretation of external choice in the case in which one of the branches is \S : specifically, the choice remains external, unlike, for example, the treatment in [16]. See [3] for technical details.

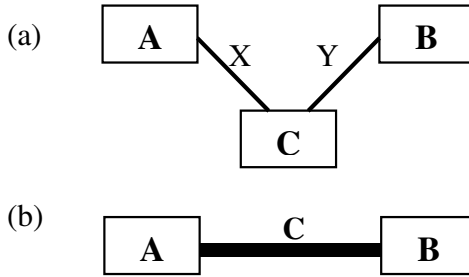


Fig. 5. Component or connector?

and Y directly using a single connector, as indicated in Fig. 5b. In this case, the connector itself encapsulates the mediating behavior of C as *glue*.

Representing a complex piece of software as a connector is a judgement call that is enabled by describing connectors as first class architectural entities. This perspective departs from a notion of connection that is restricted to relatively simple mechanisms like method calling, event announcing, or data pipelining. It requires the ability to describe the protocols that go on at each end of the connector (the *roles* in Wright) as well as the rules that tie those protocols together (the *glue*). In addition, it requires the ability describe complex topologies of connection, beyond simple point-to-point, like having multiple clients communicating with a server over the same set of protocols (a parametric multi-role connector in Wright – see Sect. 4.)

6 Formalizing Enterprise JavaBeans™

Turning now to EJB (as illustrated in Fig. 3), it seems clear that clients and beans should be represented as components. Each performs significant application-specific computation, and is best viewed as a first class type of computational entity in the architectural framework. However, as the actual computations of the clients and beans cannot be defined at the framework level (since they will be determined when the framework is used to develop a particular application), we will represent those components parametrically. That is, the actual application code will be used to instantiate them at a later time.

What about the EJB container? While it would be possible to represent it as a component, as in Fig. 5a, it seems far better to consider it a rich connector, as in Fig. 5b. Not only is the container primarily responsible for bridging the gap between clients and beans, but also the container-client and container-bean sub-protocols are so tightly interwoven that it makes sense to describe them as a single semantic entity (i.e., the connector glue). For example, the effect of a remote method call from a client to a bean is mediated by the container so that if the target bean is passivated it can be activated using the container-bean activation protocol. The resulting general structure is illustrated in Fig. 6.

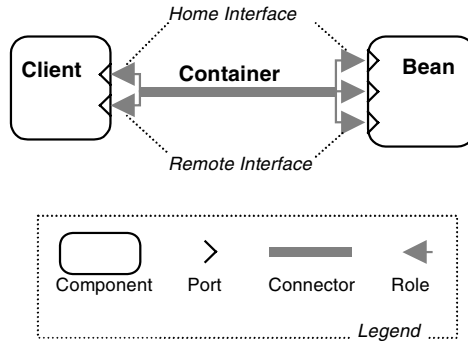


Fig. 6. One Client connected to one Bean.

In this case the Remote and Home interfaces become roles in the Container connector that both a Client and a Bean interact with. In Wright this structure is described (schematically) as:

```

Configuration one-Client-one-Bean
  Component Client (BusinessLogic: Process)
    Port UseHomeInterface = <...>
    Port UseRemoteInterface = BusinessLogic
    Computation = <...>
  Component EJBBean (BusinessLogic: Process)
    Port BeanHome = <...>
    Port JxBean = <...>
    Port RemoteInterface = BusinessLogic
    Computation = <...>
  Connector Container (BusinessLogic: Process)
    Role HomeInterface = <...>
    Role RemoteInterface = BusinessLogic
    Role UseBeanHome = <...>
    Role UseJxBean = <...>
    Role UseRemoteInterface = BusinessLogic
    Glue = <...>
  Process SomeBusinessLogic = <...>
  Instances
    A: Client(SomeBusinessLogic)
    B: EJBBean(SomeBusinessLogic)
    C: Container(SomeBusinessLogic)
  Attachments
    A.UseHomeInterface as C.HomeInterface
    A.UseRemoteInterface as C.RemoteInterface
    C.UseBeanHome as B.BeanHome
    C.UseRemoteInterface as B.RemoteInterface
    C.UseJxBean as B.JxBean
end one-Client-one-Bean.

```

As indicated earlier, we use a placeholder process `BusinessLogic` as a parameter to clients, beans, and the Container connector. (The connector is parameterized by the business logic because it also needs to know about the `BusinessLogic` protocol.)

The Wright specification of the configuration also defines the attachments between the ports of each component and the corresponding roles in the Container. The next sections examine each part in turn.

6.1 The Client

The specification of a Client component is:

```

Component Client (BusinessLogic: Process)
  Port UseRemoteInterface = BusinessLogic
  Port UseHomeInterface
    = create → ( GoHomeInterface
                  △ noSuchObjectException
                    → UseHomeInterface )
                  △ remove → ( § [] removeException → § ))
  Where GoHomeInterface
    = getEJBMetaData → GoHomeInterface
  Computation = create → CallBean
  Where CallBean
    = ( (UseRemoteInterface || GoHomeInterface)
        △ noSuchObjectException → create → CallBean )
        △ remove → ( § [] removeException → § )

```

It has two ports for accessing the Bean: `UseHomeInterface` and `UseRemoteInterface`. As noted above, the latter is defined by a process that describes the application logic implemented by the Bean and is passed to the Client as a parameter (`BusinessLogic`).

The process describing the client's view of the Home Interface consists of three events: `create` and `remove`, with the obvious meaning, and `getEJBMetaData`, which is a service provided by the container that returns meta-information about the methods supported by the bean. Note that the port is initialized by a `create` event and terminated by a `remove` event. The auxiliary process definition `GoHomeInterface`, describes the Home Interface perspective of what may go on between the creation of a bean and its removal: getting the bean's meta-data.

An event that may occur at any time after the creation, `noSuchObjectException`, corresponds to an exception being raised by the container. In fact, the EJB spec says that “a Client must always be prepared to recreate a new instance (of a bean) if it loses the one it is using” (pp. 24).⁸ Hence, if the Client gets a `noSuchObjectException`, it should go back to create another bean. The

⁸ In a distributed computing environment, it is possible to loose communication with a remote server. The distribution transparency provided by EJB, however, has the potential to hide from the client whether the reinitialized home interface is directed to the same, recovered, server or to another that supports the same bean class.

Wright specification exhibits this property in both the specification of the process `GoHomeInterface` and in the process `CallBean` in the Client's computation: the occurrence of a `noSuchObjectException` event causes the Client to reinitialize the Home Interface by issuing a `create` event. In Sect. 7 we see how less trivial properties can be checked by the use of automated tools.

The main body of computation, once it is initialized by a `create`, is the parallel composition of the processes `UseRemoteInterface` and `GoHomeInterface`. What goes on in this composition is dictated by the application logic, passed as a parameter to the client, in parallel with the initialized Home Interface. Finally, at any time (after initialization) the client may decide to remove the bean. This is signaled by the client-initiated `remove` event interrupting the process described above (using the Δ operator). However, the Client must be prepared to handle a `removeException`, thrown by the Container. After a `remove`, either the computation successfully terminates, or it accepts a `removeException`, after which it also terminates. The EJB spec does not define how components should handle exceptions. So we only note the fact that an exception may be received. It should be clear now that the specification of the `UseHomeInterface` port is actually a view of the Client's computation, restricted to the events recognized by the Home Interface.

The `HomeInterface` role in the container expresses the possible behaviors of the client that attaches to this role:

```

Connector Container (BusinessLogic: Process)
  Role HomeInterface =  $\overline{\text{create}}$   $\rightarrow$  GoHomeInterface
  Where GoHomeInterface
    = (  $\overline{\text{getEJBMetaData}}$   $\rightarrow$  GoHomeInterface
       $\square$  noSuchObjectException  $\rightarrow$  HomeInterface )
       $\square$   $\overline{\text{remove}}$   $\rightarrow$  (  $\$$   $\square$  removeException  $\rightarrow$   $\$$  )

```

The process specification for this role is equivalent to the process in the `UseHomeInterface` of the Client component, in the sense that it will generate the same set of traces. After being initialized by `create`, the attached component will choose (internally) whether or not to `remove` the bean. If the component chooses not to remove the bean, it may initiate a request for meta-data. It also admits a `noSuchObjectException`, which resets the role. If the component chooses to remove the bean, it admits a `removeException`, but terminates afterwards, in either case.⁹

6.2 The Container and the Bean

In the container, there are three Wright roles that are involved in the creation of a bean. The first is the `HomeInterface` role, as discussed in Sect. 6.1, to which

⁹ Again, for simplicity, we focus on a single run of the protocols between the client and the container, in order to distinguish between a situation where the protocol demands a reset, from a situation where it runs through successfully and could go back to create another bean.

the client attaches. The other two are the `UseBeanHome` and `UseJxBean` roles, to which the bean attaches:

```

Connector Container (BusinessLogic: Process)
  alpha Created =  $\alpha$ UseJxBean \ {setContext, ejbRemove}
  ...
  Role UseBeanHome = newInstance  $\rightarrow$  ejbCreate  $\rightarrow$  §
  Role UseJxBean   = setContext  $\rightarrow$  GoJxBean
  Where GoJxBean
    = ejbPassivate  $\rightarrow$  ejbActivate  $\rightarrow$  GoJxBean
      □ ejbRemove  $\rightarrow$  UseJxBean
  Glue = ...
  Where BeanLive
    = create  $\rightarrow$  newInstance  $\rightarrow$  setContext  $\rightarrow$  ejbCreate
       $\rightarrow$  ( RUNCreated
          △ remove  $\rightarrow$  ejbRemove  $\rightarrow$  § )
  ...

Component EJBBean (EJBObject: Process)
  Port BeanHome = newInstance  $\rightarrow$  ejbCreate  $\rightarrow$  §
  Port JxBean   = setContext  $\rightarrow$  GoJxBean
  Where GoJxBean
    = ejbPassivate  $\rightarrow$  ejbActivate  $\rightarrow$  GoJxBean
      □ ejbRemove  $\rightarrow$  §

```

Since it is often the case that a protocol refers to events in more than one role, the perspective that a specific role has of a protocol is limited by the alphabet of the role. It is the *glue* that links what goes on in each role, thus completing the protocol followed by the connector.

In order to single out each piece of the glue that corresponds to a particular protocol in the software framework, we introduce auxiliary process definitions. `BeanLive` is one of them. Since this is a glue process, it takes the viewpoint of the container: hence, the `create` event is initiated by the environment (in the `HomeInterface` role). After receiving a `create`, the container initiates the `newInstance` event in the `UseBeanHome` role, sets the newly created bean's runtime context (`setContext` in the `UseJxBean` role,) and signals the new bean to run the appropriate initialization method (`ejbCreate` in `UseBeanHome`).

The `BeanLive` process then accepts any event in the alphabet of the `UseJxBean` role, except for `setContext` (part of the initialization) and `ejbRemove` (part of the termination). When interrupted by a `remove` event in the `HomeInterface` role, the `BeanLive` process signals the bean to run the appropriate termination method (`ejbRemove` in the `UseJxBean` role) and then terminates.¹⁰

The Container relays the business logic events in the role `RemoteInterface` (to which the Client attaches) to the role `UseRemoteInterface` (to which the

¹⁰ The roles take the viewpoint of the environment (of the components that attach to the roles,) as opposed to the viewpoint of the container. So, the parity of initiation is reversed in the glue and in the roles. Note also that the processes in the roles `UseBeanHome` and `UseJxBean` match the processes in the corresponding ports in the `Bean` component, `BeanHome` and `JxBean`.

Bean attaches). The glue process `Delegate` assures this by simply stating that any event e in the `RemoteInterface` role is followed by the (container-initiated) same event \bar{e} in the `UseRemoteInterface` role.

Recall now that the container may decide to passivate a bean according to a least recently used policy. The glue process `SwapBean` (see below) accepts any event in the alphabet of the `Container`,¹¹ except for the events `ejbPassivate` and `ejbActivate`. Whenever the container decides to initiate an `ejbPassivate` event, the `SwapBean` process waits for the next event in the `RemoteInterface` role. After that, and before the event is relayed to the `UseRemoteInterface` role, an `ejbActivate` event is interleaved. The parallel combination of the processes `SwapBean` and `Delegate` in the glue produces the desired effect: the business logic events are normally relayed, but whenever the bean was passivated, it receives an activation event just before the business logic event is sent.

```

Connector Container (BusinessLogic: Process)
  alpha Activated =  $\alpha$ Container \ {ejbPassivate, ejbActivate}
  Role UseJxBean = setContext  $\rightarrow$  GoJxBean
  Where GoJxBean
    = ejbPassivate  $\rightarrow$  ejbActivate  $\rightarrow$  GoJxBean
       $\square$  ejbRemove  $\rightarrow$  UseJxBean
    ...
  Role RemoteInterface = BusinessLogic
  Role UseRemoteInterface = BusinessLogic
  Glue = BeanLive
        || Delegate
        || SwapBean
  Where Delegate =  $\square$  e:  $\alpha$ RemoteInterface  $\bullet$ 
    RemoteInterface.e  $\rightarrow$  UseRemoteInterface. $\bar{e}$   $\rightarrow$  Delegate
  Where SwapBean
    = RUNActivated  $\triangle$  ejbPassivate
       $\rightarrow$  (  $\square$  e:  $\alpha$ RemoteInterface  $\bullet$  RemoteInterface.e
         $\rightarrow$  ejbActivate  $\rightarrow$  UseRemoteInterface. $\bar{e}$   $\rightarrow$  SwapBean )
    
```

7 Using the Model

By precisely specifying the implied protocols of interaction for EJB, one achieves a number of immediate benefits. First, the formal specification is explicit about permitted orderings of method calls, and about where the locus of choice lies. Second, the specification makes explicit where different parts of the framework share assumptions. In particular, the role of `BusinessLogic` as a parameter helps clarify the way in which assumptions about the application-specific behavior are shared among the parts of the framework. Third, the model helps clarify some of the more complex aspects of the model by localizing behavior. For example, the murky role of passivation becomes clear in the `Container` glue.

Furthermore, it is also possible to submit the model to formal analysis via model checking tools. To do this we used the FDR™ model checker for CSP [7]

¹¹ Taken here as the union of the alphabets in all roles.

to check for deadlocks in the container.¹² In addition to checking for deadlocks, FDR can also be used to make sure that specific required behaviors¹³ still hold in the overall result of the composition of all local specifications. For that we use the CSP notion of process refinement. Specifically, we can check if a process describing the desired behavior is *refined* by the overall specification; for instance, if a process describing the client's recovery after a container failure is refined by the one-Client-one-Server specification. If that is the case, that means that the intended behavior was not lost due to a mistake during the process of specifying all the interacting behaviors.

For the current model, analysis revealed one significant problem. The problem concerns a possible race condition between the delegation and passivation processes inside the Container. Suppose that the Client initiates an event in the `RemoteInterface` role. Then, before the `Delegate` process relays the event to the bean through the `UseRemoteInterface` role, the `SwapBean` process, operating concurrently, decides to passivate the bean. Now, the `Delegate` process must relay the received business logic event to the `UseRemoteInterface` role, before it can accept the next event in the `RemoteInterface` role. However, the `SwapBean` process just issued an `ejbPassivate` notification to the bean, and hence it waits for the next event in the `RemoteInterface` role to reactivate the bean. Therefore, the processes that go on inside the Container cannot agree on what to do next, and the connector deadlocks.

A simple correction for the deadlock is:

```
Connector Container (EJBObject: Process)
...
Where Delegate
= (  $\prod e: \alpha\text{RemoteInterface} \bullet$ 
     $\text{RemoteInterface.e} \rightarrow \text{UseRemoteInterface.e} \rightarrow \text{Delegate}$  )
 $\prod \text{ejbPassivate} \rightarrow \text{Delegate}$ 
...
```

That is, the `Delegate` process must prevent passivation between receiving an event in the `RemoteInterface` role and relaying it to the `UseRemoteInterface` role. One way to model it in CSP is to explicitly allow the `ejbPassivate` event outside the mentioned “critical section”.

While arguably one might attribute the detected problem to *our* specification, and not to Sun's EJB spec, it does point out a place where the complexity of the specification can lead to errors that might be hard to detect otherwise. Without a precise model and effective automated analysis tools to identify problem areas, such errors could easily be introduced, undetected, into an implementation.

¹² Translation from Wright to FDR is accomplished semi-automatically using the Wright tool set. See [1].

¹³ For instance, Sun's document (pp. 24) states that any implementation of the EJB protocol between a client and an EJB server must allow the client to recover from EJB server crashes.

8 Conclusions and Future Work

In this paper we have outlined a formal architectural model of part of Sun's EJB component integration framework. In doing this we have attempted to shed light both on EJB itself, and on the way in which one can go about modeling object-oriented architectural frameworks. The key idea in our approach is to take an architectural view of the problem that makes explicit the protocols of interaction between the principle parts of the framework. In particular, we have shown how representing the framework's mediating infrastructure as a connector with a well-defined protocol helps to clarify the overall structure of the framework and to localize the relationships between the various method calls that connect the parts.

The use of formal architectural modeling languages to represent frameworks such as EJB opens up a number of important questions to investigate. First, while our specification focused on certain properties of the framework, there are many others that one might want to model. For example, although potential deadlocks are highlighted by our model, we do not handle important issues such as performance, reliability, and security. For many frameworks finding notations that expose such properties will be crucial.

Second, given a formal specification, such as the one we have presented, it should be possible to influence conformance testing. Currently, conformance to a framework can only be loosely checked – for example, by making sure that an implementation provides the full API. However, given a richer semantic model, it should be possible to do much better.

Third, the EJB spec uses inheritance to organize the presentation of many of its concepts. For example, the `SessionBean` class inherits behavior from the `EnterpriseBean` class, which in turn inherits from the `java.io.Serializable` class. In contrast, the formal model that we have presented is essentially flat. To come up with our model we had to fold together the implicit semantic behavior defined in several classes. It would have been much nicer to have been able to mirror the inheritance structure in the architectural specification. While such extension is relatively well-understood with respect to signatures, it is not so clear what is needed to handle interactive behaviors – such as protocols of interaction. Finding a suitable calculus of protocol extension is an open and relevant topic for future research.

Acknowledgments

This research was supported by the US Defense Advanced Research Projects Agency and Rome Laboratory, USAF, under Cooperative Agreement F30602-97-2-0031, and by the US National Science Foundation under Grant CCR-9357792. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Rome Laboratory, the US Department of Defense, or the US National Science Foundation. The US Government is authorized to reproduce

and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

References

- [1] Robert Allen and David Garlan. A formal basis for architectural connection. In *ACM Trans. on Software Engineering and Methodology*, July 1997.
- [2] Robert Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Sixth Intl. Symposium on the Foundations of Software Engineering (FSE-6)*, Nov. 1998.
- [3] Robert Allen. A Formal Approach to Software Architecture. PhD thesis, CMU, School of Computer Science, January 1997. CMU/SCS Report CMU-CS-97-144.
- [4] Edmund Clarke et al. Automatic verification of finite state concurrent systems using temporal logic specifications. In *ACM Trans. on Programming Languages and Systems*, April 1986.
- [5] Edmund Clarke et al. Verification Tools for Finite-State Concurrent Systems. A Decade of concurrency - Reflections and Perspectives. *Springer Verlag LNCS 803*, 1994.
- [6] Vlada Matena, Mark Hapner, Enterprise JavaBeans™, Sun Microsystems Inc., Palo Alto, California, 1998.
- [7] Failures Divergence Refinement: User Manual and Tutorial, 1.2β. Formal Systems (Europe) Ltd., Oxford, England, 1992.
- [8] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.
- [9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [10] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
- [11] David C Luckham, et al. Specification and analysis of system architecture using Rapide. In *IEEE Trans. on Software Engineering*, April 1995.
- [12] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT LCS, 1988.
- [13] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings ESEC'95*, Sept. 1995.
- [14] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. In *IEEE Trans. on Software Engineering*, April 1995.
- [15] J.L. Peterson. *Petri nets*. ACM Computing Surveys, September 1977.
- [16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [17] Mary Shaw, et al. Abstractions for software architecture and tools to support them. In *IEEE Trans. on Software Engineering*, April 1995.
- [18] K.J. Sullivan, J. Socha, and M. Marchukov. Using formal methods to reason about architectural standards. In *1997 Intl. Conf. on Software Engineering*, May 1997.