A language and framework for supporting an active approach to component-based software integration

Suzanne W. Dietrich, Susan D. Urban, Amy Sundermier, Yinghui Na, Ying Jin, Sunitha Kambhampati
Department of Computer Science & Engineering
Arizona State University, Tempe, AZ 85287-5406, U.S.A.
dietrich@asu.edu, s.urban@asu.edu

Keywords: component-based integration, active rules, events

Received: June 25, 2001

The IRules project at Arizona State University applies active rule technology to the integration of distributed, black-box software components. The goal of IRules is to provide an environment in which an application is developed through the integration of software components using active rules that are known as integration rules. Using the IRules Component Definition Language (CDL), the application integrator first describes a purchased, black-box component within the IRules environment to allow access to the properties and methods defined by the purchased component. In addition, CDL allows for the definition of named extents, stored and derived attributes, externalized relationships and events to enhance the features of the purchased components to support application development. After defining the desired interface for the component, the application integrator then develops the application using active integration rules that define the interaction of the components in response to events. This paper presents the Component Definition Language and its resulting framework that supports the IRules active approach to component-based software integration.

1 Introduction

Electronic commerce (e-commerce) and other Web-based applications are inherently distributed in nature since a client is not expected to be co-located with the data for the application. However, many Web-based applications are primarily three-tiered architectures with a presentation layer using a browser to interface with the user, a middle tier generally built with an object-oriented or component-based interfacing technology, and a database as the persistence layer. Commercial component standards such as Enterprise JavaBeans (EJB) from Sun Microsystems or COM+ from Microsoft are typically adopted for the middle tier, allowing the developers of an application to focus on creating code to represent business needs while relying upon a commercial component container to supply infrastructure services. The current state of middle-tier component software primarily addresses the requirements of three-tier architectures, simplifying the development of Web applications by providing an application-programming layer between the presentation and storage layers that decreases development time. However, e-commerce applications require access to data and software from many different sources. As a result, the simplifying assumption of one underlying database in the persistence layer as in most three-tiered architectures is too restrictive for some distributed applications.

There is a conflict between the goal of a commercial component container vendor to make three-tier web application development simpler and faster versus the longer-term goal of software engineering to make component-based application development a reality. Component-based software engineering research encourages the idea of building applications from purchased components. Using purchased components directly to build an application, however, is difficult to accomplish if the application developer must work with the limitations of an interface defined by the vendor of the component. This paper addresses some of the challenges inherent in application development using purchased components. We refer to these components as "black-box" components, since we assume the component must be used without modification to source code.

Our research focuses on adapting database technologies to the area of software component integration. As described in (Silberschatz & Zdonik 1997), certain forms of database functionality need to "break out of the box" to better serve the needs of applications that depend on distributed sources of information. Active rule processing technology (Widom & Ceri 1996) is an example of a database component that can provide useful services to advanced applications if the appropriate technology exists for the use of rules in distributed environments. Traditionally, active rules have been used to transform passive, centralized database systems into reactive systems that respond to database and external events through the use of rule processing features. Active rules are typically formatted as Event-Condition-Action (ECA) rules. When an event occurs, if an optional condition holds, then a specified action is performed.

The Integration Rules (IRules) project at Arizona State University (http://www.eas.asu.edu/~irules) is

investigating the middle-tier, rule processing technology necessary for the use of active rules in the integration of distributed, black-box software components (Urban et. al. 2001a, Urban et. al. 2001b). The intended use of this rule processing technology is for the specification of event-based processing logic in the development of component-based applications for distributed environments, where the granularity of the components can range from low-level database objects to an entire software system.

The IRules approach builds upon the use of the Enterprise JavaBeans (EJB) software component model specification from Sun Microsystems (J2EE 2001). The EJB component model promotes the vision of separating component services from the business logic of the components. Assuming that all databases and software sources of the application environment are encapsulated using EJBs, the application integrator uses the IRules Component Definition Language (CDL) to extend the definition of a purchased software component to declare named extents, additional attributes, externalized relationships and component events. Since there is no inherent support for direct object references between distributed components, *externalized relationships* (Rumbaugh 1987) play an important role in associating the purchased components that are being integrated in the distributed IRules environment.

Once components are defined in the IRules environment using CDL, application integrators can create distributed applications using the IRules Integration Rule Language together with application transactions. Integration rules provide a re-active capability to the environment so that as distributed components and external sources generate event notifications, integration rules invoke methods on components or perform higher-level application transactions. The purpose of this paper is to provide a description of the IRules Component Definition Language and the framework of component metadata and wrappers that are generated to support the IRules active rule architecture approach to the integration of purchased software components.

In the following sections, we outline the details of the IRules approach to component integration. Section 2 first provides an overview of related work. In Section 3, we provide an overview of the IRules approach, introducing an investment example that will be used to illustrate IRules concepts throughout the rest of the paper. Section 4 introduces the IRules Component Definition Language, illustrating the definition of named extents, additional attributes, externalized relationships and component-generated events. Section 5 elaborates on the static component metadata that is generated as a result of the compilation of CDL. Section 6 provides the details on how the IRules environment supports the enhancements to the purchased components using wrappers. The paper concludes in Section 7 with a summary of our work and a discussion of future research directions.

2 Related Work

Recent work on component integration has focused on the architecture of software interconnection based on the underlying component model. Software architectures such as COM+ (Microsoft 2000), CORBA (OMG 1998), and Enterprise JavaBeans facilitate the integration by supporting the development of systems from independently developed components.

One approach to the interoperability of components is event-based. In (Barrett et. al. 1996), the Event-Based Integration (EBI) framework was proposed as a high-level, general, and flexible reference model for event-based software integration. This approach outlines architectural concepts for interconnection through events. In (Ma & Bacon 1998), the CORBA-Based Event Architecture (COBEA) is a general event-driven architecture for building distributed active systems. COBEA extends the CORBA Event Service by supporting the publish-register-notify model and provides filtering, fault-tolerance, and access control services. COBEA is a general event-driven architecture for distributed active systems, rather than an implemented system.

There have been some initial results on the use of ECA rules for integrating distributed components. In (Pissinou & Vanapipat 1996) and (Pissinou et. al. 1997), an ECA rule approach is used in component interoperation. Distributed applications are modeled as Distributed Active Objects by adding wrappers on top of the components that do not have triggers so that ECA rules can be used in distributed environments. The ECA Object Service is based on the CORBA specification. Objects communicate by method invocations and service requests. The rule object is an independent CORBA object that is isolated from the application objects. The work in (Pissinou et. al. 1997) describes an architecture for executing rules in a distributed environment. In (Chakravarthy & Le 1998) ECA rules are proposed to solve distributed interoperation of components that have an OMG IDL interface. The project focuses on the specification, detection and management of composite events (Le & Chakravarthy 1998). The system uses the CORBA event service and implements conditions and actions by method calls. In (Bultzingsloewen et. al. 1996, Koschel & Lockemann 1998), the CORBA-Based Distributed Information System named C^{2}offein was developed to use ECA rules for distributed component interoperation. Wrappers are used for read access to the underlying data source and primitive event detection. The CORBA push model is used for event detection. C^{2}offein provides a concrete architecture and implementation of how to use ECA rules to integrate heterogeneous information sources.

Our own past work in the area of active database systems has influenced the research presented in this paper. In particular, our work with the ADOOD RANCH (Dietrich et. al. 1992) project resulted in a declarative language for

the integration of active, deductive, and object-oriented language concepts (Urban et. al. 1997), together with a framework for capturing the metadata of such an environment (Abdellatif et. al. 1999) and an execution model that supports the incremental examination of the database state during rule processing (Abdellatif 1999). Our approach to the use of derived attributes in the IRules Component Definition Language, as well as the structure of integration rules, extends our results from the ADOOD RANCH project to distributed domains. The work in (Ayyaswamy 1999) represents our initial investigation of a CORBA architecture for distributed ECA rule processing for the purpose of maintaining constraints in a loosely-coupled, federated database environment. More recently, we have performed a comparison of CORBA (OMG 1998), Java (J2EE 2001), and Jini (Arnold 2000) technologies for evaluating different architectural options for the execution of integration rules (Saxena 2000, Urban et. al. 2001c).

The IRules project differs from the above research projects in several aspects. First, IRules is based on the Enterprise JavaBeans component model. Second, IRules builds its own distributed environment. The compilation of the IRules Component Definition Language automatically generates the code for the wrappers of the black-box components rather than hard-coding the wrappers. Third, the IRules project is also investigating transaction management, conflict resolution, and failure handling issues in its distributed rule processing environment.

The externalized relationships of the IRules environment share some similarities with the CORBA Relationship Service (OMG 2000). The Relationship Service supports the definition and creation of relationships between distributed CORBA objects. As with IRules externalized relationships, the related objects do not have to be aware of the relationship. One obvious difference is that the Relationship Service is for relating CORBA objects while the IRules externalized relationships are designed for black-box components that adhere to the EJB component model.

3 IRules Overview

This section provides a high-level overview of the IRules project to establish the basis for a more detailed presentation of the IRules Component Definition Language and its supporting metadata and wrapper framework in the following sections. The IRules project adopted the Enterprise JavaBeans (EJB) server-side component model for the Java programming language (J2EE 2001). Due to space limitations, we assume prior knowledge of EJBs.

To illustrate the IRules approach to the integration of EJB components, this section presents an Investment application that is used throughout the remainder of this paper. This application is depicted in Figure 1, illustrating four different containers with purchased software components. The Portfolio container maintains current information in the form of entity beans about client portfolios, including information about current and past stock holdings and the orders under which stocks were bought and sold. The Portfolio container also provides a session bean with application logic to conduct buying and selling of stocks. The Pending Order container provides entity beans for storing pending orders that are waiting for execution when a particular market condition is met. The Stocks container represents locally managed information about stocks and their current prices as entity beans. This information exists independently of the portfolios that own them. We are assuming that the information in the Stock container is updated based on current stock prices from external sources. The Stock container can also generate events to signal changes in value depending upon buy/sell transactions in the stock market. Finally, the User container uses entity beans to store billing information about portfolio accounts and the users that are associated with accounts. The User container also provides a session bean with procedures for billing users for stock buy and sell transactions.

There are implied relationships between the four containers in Figure 1. Portfolios contain specific stocks. Pending orders are related to a portfolio and represent buy and sell transactions on stocks. Portfolios are owned by a specific account, and accounts are billed for buy and sell transactions. In general, the application programmer must know of the specific relationships and write procedural code to achieve the integration. In a typical Web-based, three-tier architecture, this approach may be satisfactory. Advanced distributed applications, however, may require the interconnection of components in containers provided by multiple companies from distributed locations. These types of applications can benefit from an environment that provides greater support in understanding and establishing relationships between distributed components.

In addition to illustrating the four independent containers of our sample application, Figure 1 also illustrates the the IRules approach to component interconnection. The lines between containers represent externalised relationships of the desired object model, defining relationships between components on different servers. For example, we wish to represent the fact that a Portfolio may have orders waiting for execution, where the order information is stored in the Pending Order component. The Pending Order component is also defined to act upon a specific type of stock. The application integrator uses the IRules Definition Language to define a distributed application.

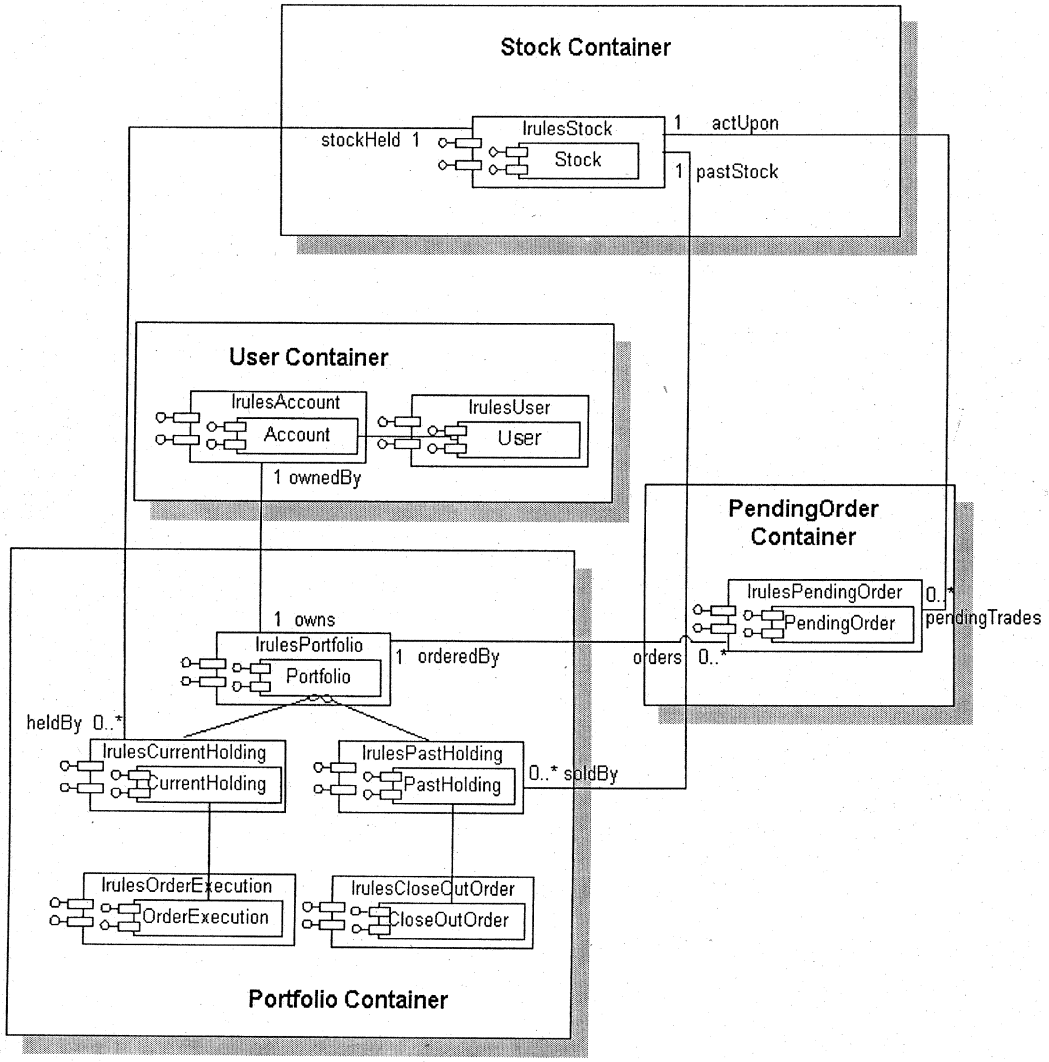

Figure 1: Investment Example

The IRules Definition Language consists of four sublanguages: (1) the Component Definition Language (CDL), (2) the Event Definition Language (EDL), (3) the Integration Rule Language (IRL), and (4) the IRules Scripting Language (ISL). Using CDL, the application integrator first describes a purchased, black-box component within the IRules environment to allow access to the properties and methods defined by the purchased component. In addition, CDL allows for the definition of a named extent, stored and derived attributes, externalized relationships and events to enhance the features of the purchased components to support application development. EDL provides a language for the definition of external and system-level events. After defining the desired interface for the components and events, the application integrator then develops the application using active integration rules via the IRL, which defines the interaction of the components in response to events. ISL provides the application integrator with a more complete approach to transaction development over the object model of the distributed application. Currently, we are investigating JACL (DeJong & Laird 1997) and its extensions as the foundation of the ISL.

Figure 2 illustrates a high-level architectural view of the IRules processing environment. In the IRules architecture, the object manager uses component metadata and the abstract IRules wrapper interface to provide the rule processor with the appropriate references to remote interfaces as needed to process rules and transactions. Thus the object manager encapsulates the choice of the EJB component model from the other system components in the IRules framework and architecture. The metadata manager stores information about the IRules object model of the application, resulting from the compilation of the IRules Definition Language. The compilation of CDL also results in the generation of wrappers for the purchased components, which provide required information for supporting the IRules environment. The object manager and the metadata manager are used by the transaction and rule

processor to execute the application logic of the environment. Transactions specified in the scripting language can execute methods on entity and session beans, where the object manager is first consulted to locate the component required for the execution of the method. The execution of such methods can send event notifications via the IRules wrappers to the event handler, denoting the before and after points in the execution of such methods. The event handler communicates with the transaction and rule processor to trigger integration rules. The execution of integration rules triggers additional application transactions, beginning a new cycle in the execution of methods on EJB components. A more complete description of the execution environment for IRules can be found in a companion paper (Urban et. al. 2002). The following sections elaborate on the CDL and the metadata and wrapper framework required to support a rule-based approach to software component integration.

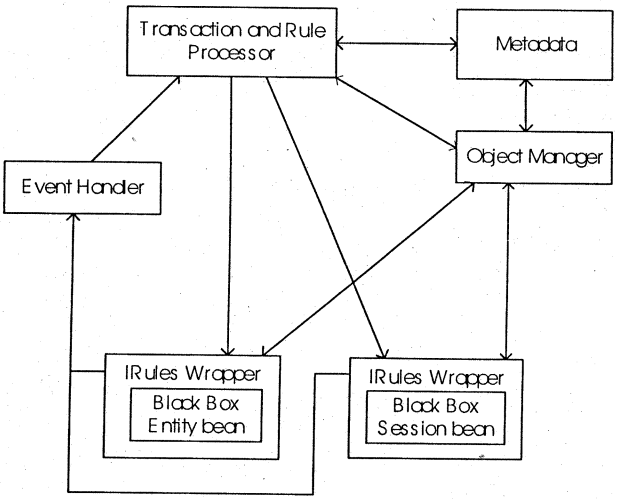

Figure 2: IRules Architectural Overview

4 Component Definition Language

The Component Definition Language provides the application integrator with a tool to describe purchased, black-box software components to the IRules environment. Recall that IRules assumes the EJB component model and components can be either entity beans or session beans. The application integrator defines only additional properties using CDL. In other words, the behavior of the black-box EJB is not redefined in CDL, since it is available using reflection. Figure 3 presents the CDL for the Investment Example presented in the previous section. Specific examples from this figure will be used to describe CDL in more detail.

The syntax of CDL is loosely based on the syntax of the Object Definition Language (ODL) of the Object Data Management Group (ODMG) standard (Cattell et. al. 2000). ODL defines the classes that an object-oriented database (OODB) manages in its persistent store. For each class, the database designer may define an extent, which is a named collection of the objects of that type in the database, and the properties and behavior associated with that class. The term *property* refers to *attributes*, describing characteristics of the object, or *relationships*, defining associations between objects. The relationship between two objects is automatically maintained by the database system. When one side of the relationship is updated, the database system is responsible for maintaining the inverse relationship.

The IRules Component Definition Language provides the application integrator with the ability to define named extents, attributes, relationships and events to be associated with a black-box component deployed in the IRules environment. The applicability of these definitions depends on the type of the EJB, such as entity beans or session beans.

4.1 Entity Beans

Each entity bean is described to the IRules environment by a component declaration, giving the name of the black-box component (*ComponentName*). Entity beans are identified by the implements EntityBean clause of the component definition. Entity beans defined within the IRules environment may include a named extent, additional attributes, externalized relationships and events. In the abstract syntax shown below, italicized identifiers represent names that are filled in based on the specification of the application:

```
component ComponentName implements EntityBean
(extent ExtentName)
{ attribute     AttributeType      AttributeName
                { OptionalAttributeDefinition };
  relationship RelationshipType  RelationshipName
                inverse InverseRelationshipName;
  event         IRulesEventName(EventParameters)
                { method EventDefinition };                }
```

The *ExtentName* provides the name of an extent that the application integrator can use in the specification of the integration rules to iterate over objects of the type *ComponentName*. In the Investment CDL of Figure 3, each entity bean has a defined extent. By convention, the name of the extent is the plural of the name of the component. For example, the component Stock has an extent named stocks.

An attribute defined in CDL specifies default behavior to get and set the attribute's value: the method *getAttributeName* returns *AttributeType*, and the method *setAttributeName* takes an argument of *AttributeType* to which the attribute value is set. This attribute will be stored as part of the IRules wrapper for the component. In Figure 3, the Portfolio component has a stored attribute, named lastPortfolioValue of type float. The IRules wrapper automatically provides the accessor (getLastPortfolioValue and setLastPortfolioValue) methods for the stored attribute.

```
component Stock implements EntityBean
 (extent stocks)
 {relationship set <CurrentHolding> heldBy
    inverse CurrentHolding::stockHeld;
  relationship set <PendingOrder> pendingTrades
    inverse PendingOrder::actUpon;
  relationship set <PastHolding> soldBy
    inverse PastHolding::pastStock;
  event beforeSetPrice(NewPrice)
   {method before setPrice(NewPrice)}; };
component User implements EntityBean
 (extent users) {};
component Account implements EntityBean
 (extent accounts)
 {relationship Portfolio owns inverse Portfolio::ownedBy;};
component Portfolio implements EntityBean
 (extent portfolios)
 {attribute float lastPortfolioValue;
  attribute float portfolioValue {
    portfolioAI.calculatePortfolioValue(Portfolio self)};
  relationship Account ownedBy inverse Account::owns;
  relationship set<PendingOrder> orders
    inverse PendingOrder::orderedBy; };
```

```
component CurrentHolding implements EntityBean
 (extent currentHoldings)
 {relationship Stock stockHeld inverse Stock::heldBy;};
component PastHolding implements EntityBean
 (extent pastHoldings)
 {relationship Stock pastStock inverse Stock::soldBy;};
component PendingOrder implements EntityBean
 (extent pendingOrders)
 {relationship Stock actUpon inverse Stock::pendingTrades;
  relationship Portfolio orderedBy inverse Portfolio::orders;
  event afterCreatePendingOrder
    (pnId,portId,stockId,numOfShares,desPrice,action)
 {method after
  create(pnId,portId,stockId,numOfShares,desPrice,action)};};
component OrderExecution implements EntityBean
 (extent orderExecutions) {};
component CloseOutOrder implements EntityBean
 (extent closeOutOrders) {};
component PortfolioSession implements SessionBean
 {event afterSellStock(stockId,price,portId,numOfShares)
 {method after sellStock(stockId,price,portId,numOfShares)};};
component PortfolioAI implements SessionBean {};
```

Figure 3: Component Definition Language for the Investment Example

An attribute may also be a derived attribute, meaning that its value is computed using a predefined method. In the Investment example shown in Figure 3, the Portfolio component has a derived attribute portfolioValue of type float. When the portfolioValue attribute is referenced (using the getPortfolioValue method), its value will be computed using the calculatePortfolioValue method defined in the portfolioAI session bean. The suffix AI in this example stands for Application Integrator, since it is the responsibility of the application integrator to define the meaning of a derived attribute. At this point in time, IRules allows this logic to be coded as a method of a session bean. We have introduced the self syntax here to indicate that the method is called on the Portfolio object itself. We are planning to allow for additional parameters to the method call, which could include properties of the purchased components and the properties defined as part of the enhanced IRules environment.

We have briefly explored the use of the Enterprise JavaBeans Query Language (EJB QL) to declaratively specify the meaning of a derived attribute. The current specification of EJB QL within the EJB 2.0 specification has several limitations that discourage its use within IRules at this time. One limitation restricts values returned from a query to be either an existing object or part of an existing object. Another limitation restricts the traversal of relationships to only those deployed in the same container (and the same ejb-jar file). Since the goal of IRules is to provide externalized relationships across distributed components in multiple containers, these limitations are too restrictive. Therefore, we have provided the application integrator with a more general option to specify the required logic as a method of a session bean.

Externalized relationships play an important role in specifying the associations between the black-box components being integrated. In the Investment example, the association ownedBy in the Portfolio component relates a portfolio to its associated account. The inverse relationship owns in the Account component associates the account to its portfolio. Section 6 describes how these externalized relationships are maintained in the IRules wrapper for the black-box component.

The application integrator ultimately specifies event-based integration rules to glue the black-box components together. When an event occurs, if an optional condition holds, then a specified action is performed. At the component level, the application integrator defines IRules events that the integration rules monitor based on method calls to the underlying black-box component. The IRules environment supports the generation of an event before or after a method call. For example, in Figure 3, the component Stock defines beforeSetPrice as an event that the IRules environment monitors, which is raised before the call to the setPrice method in the underlying black-box Stock component. The event parameter newPrice is obtained from the newPrice parameter to the setPrice method call.

We also plan to have the IRules environment support the selective monitoring of internal events from black-box components that are compliant with the Java Message Service API (JMS), which is the event service adopted

for EJBs. The details for providing this support are currently being investigated.

4.2 Session Beans

Session beans must also be declared to the IRules environment by a component declaration using the implements SessionBean clause, allowing IRules access to the properties and methods defined by the purchased component. A session bean may also define events to be monitored at the IRules level.

```
component ComponentName implements SessionBean
{ event     IRulesEventName(EventParameters)
          { method EventDefinition };          }
```

The *EventDefinition* is consistent with events defined for entity beans. The event is specifying the interception of a before/after method call to the underlying black-box session bean. In Figure 3, the CDL component definition for the session bean PortfolioSession defines the IRules event afterSellStock, which is raised after the method call to sellStock. We are also investigating a mechanism to support the selective monitoring of an internal event of a session bean by the IRules environment.

5 Component Metadata

Metadata is the data maintained by the system that describes the data in the system itself. For example, relational databases use metadata to represent the data of any application. In a similar manner, the IRules system uses metadata to represent both the components and the processing logic (application transactions and integration rules) of the application, allowing the system components of the architecture to be data-driven by the metadata describing the application. The IRules environment will store metadata as the result of compiling the IRules Definition Language. This section describes the metadata stored by the compilation of CDL. The current prototype of the IRules component metadata is written using serialized Java objects. We plan to investigate the use of the JavaSpaces service in Jini for the distributed metadata implementation.

Figure 4 gives a UML diagram illustrating the static component metadata generated as a result of the compilation of the IRules Component Definition Language. The metadata stored for an IRules Wrapper includes its name, the name of its black-box component and its JNDI name. JNDI is the abbreviation for the Java Naming and Directory Interface, which provides location and organization services in a distributed computing environment. The wrapper includes an association to the black-box component that it wraps. The JNDI name of the black-box component is obtained from the deployment descriptor for this purchased component.

An IRules Wrapper is itself an EJB of the same type as the EJB that it is wrapping. An IRules Wrapper that is an entity bean must store the name of its associated extent and properties, which are relationships and attributes. A relationship has a cardinality, such as single-valued or multi-valued. In this case, the IRulesRelationship metadata class shows a multivaluedFlag that is set to true for a multivalued relationship. A relationship also has an inverse, which is indicated by a recursive association in Figure 4 to the IRulesRelationship class, which gives the inverse relationship. An attribute has a type, and may be explicitly stored or derived. A derived attribute, as shown by the IRulesDerivedAttribute metadata class, records the name of the method and its session bean that is called to derive its value based on input parameters. The names of the attributes associated with the black-box component are also maintained in the metadata as BlackBoxAttribute since the IRules Wrapper acts as its proxy. Similarly, the IRulesMethod class represents the methods of the black-box components and the default accessor methods for the IRulesAttributes.

The events associated with a component are either method events or internal events, and are illustrated in Figure 4 by an association to an eventStub. The eventName provides an access path into the event metadata, which also includes external and system-level events that are defined using EDL.

6 Wrappers

One of the goals of the IRules project is to integrate commercial-off-the-shelf (COTS) components using containers produced by commercial vendors. The IRules wrappers play an important role in enhancing the interface of a purchased software component, providing the additional behavior required for interacting with the IRules environment. The wrappers provide a mechanism to act as a proxy to the original black-box component and to add the definition of IRules extents, attributes, externalized relationships, and events. This section describes how the IRules environment wraps both entity beans and session beans to become part of an IRules distributed application.

Figure 5 provides an overview of the IRules approach to wrapping black-box EJB components, assuming the naming conventions for EJBs. The EJB Layer in the diagram reinforces the description of the EJB component model. The EJBHome interface represents the life-cycle methods of the component. The EJBObject interface, also known as the remote interface, defines the signature of business methods for changing attribute values and carrying out business logic functions that are specific to the EJB component. The right-most column represents the implementation of the enterprise bean. The Wrapper Abstract Layer provides the behavior that is inherited by every IRules wrapper. The Wrapper Implementation Layer shows the IRules Wrapper for the BlackBox component. The lowest layer of the diagram is the Component Layer and identifies the BlackBox EJB.

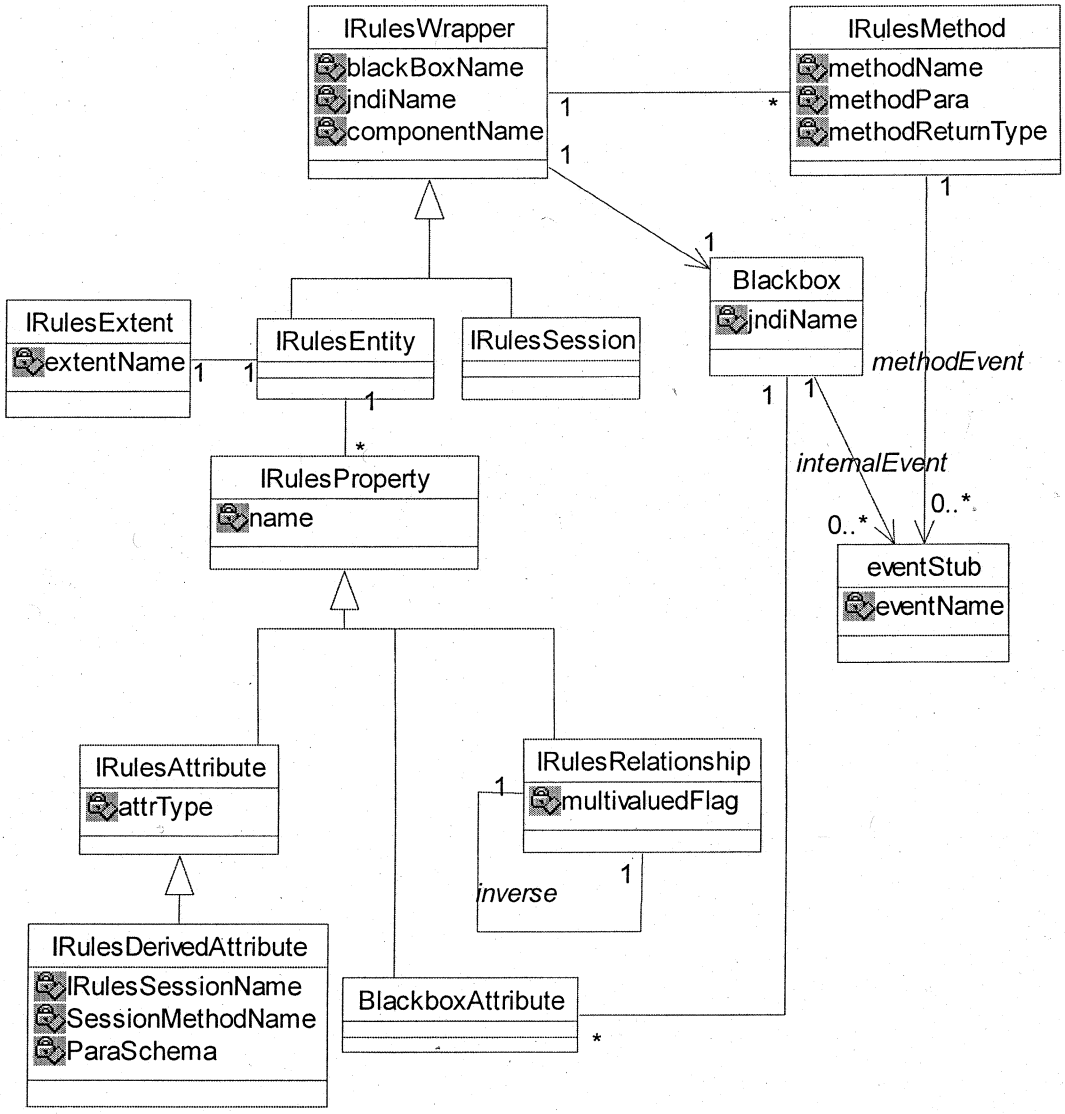

Figure 4 Static Component Metadata

To develop the functionality of the wrappers, we have implemented a prototype for the Investment application that provides proof of concept of the technology. Since our initial prototype, the IRules wrapper design has already been refined based on changes to the EJB specification. The EJB 2.0 specification introduced a new container-managed persistence (CMP) contract. For container-managed entity beans, the deployment descriptor indicates which fields and relationships of the bean are to be maintained by the container. The actual mapping of these container-managed fields (CMF) to a persistent store happens in a server-specific way and is not included in the deployment descriptor. In the case of the BEA Systems Weblogic Server (BEA 2001) that we are using for our implementation, an XML file specifies the object-to-relational mapping between the CMFs and container-managed relationships (CMRs) to the underlying relational database store. The only container-managed relationships supported are those between entity beans deployed at the same time (in the same ejb-jar file). Since this limitation on CMRs is too restrictive for the IRules environment, the IRules wrappers explicitly provide a mechanism to store and retrieve the externalized relationships from the underlying persistent storage.

6.1 Entity Beans

Since the IRules wrapper for a deployed black-box entity bean stores persistent data and needs to be shared between clients, the wrapper is also an entity bean. The IRules wrapper defines container-managed persistent fields for (1) the reference to the black-box entity bean it is wrapping, (2) stored attributes, and (3) externalized relationships. The wrapper also includes code that is generated from the compilation of CDL to provide accessor methods for attributes, manipulation methods for relationships, a proxy to method calls, and support for raising events to trigger rules.

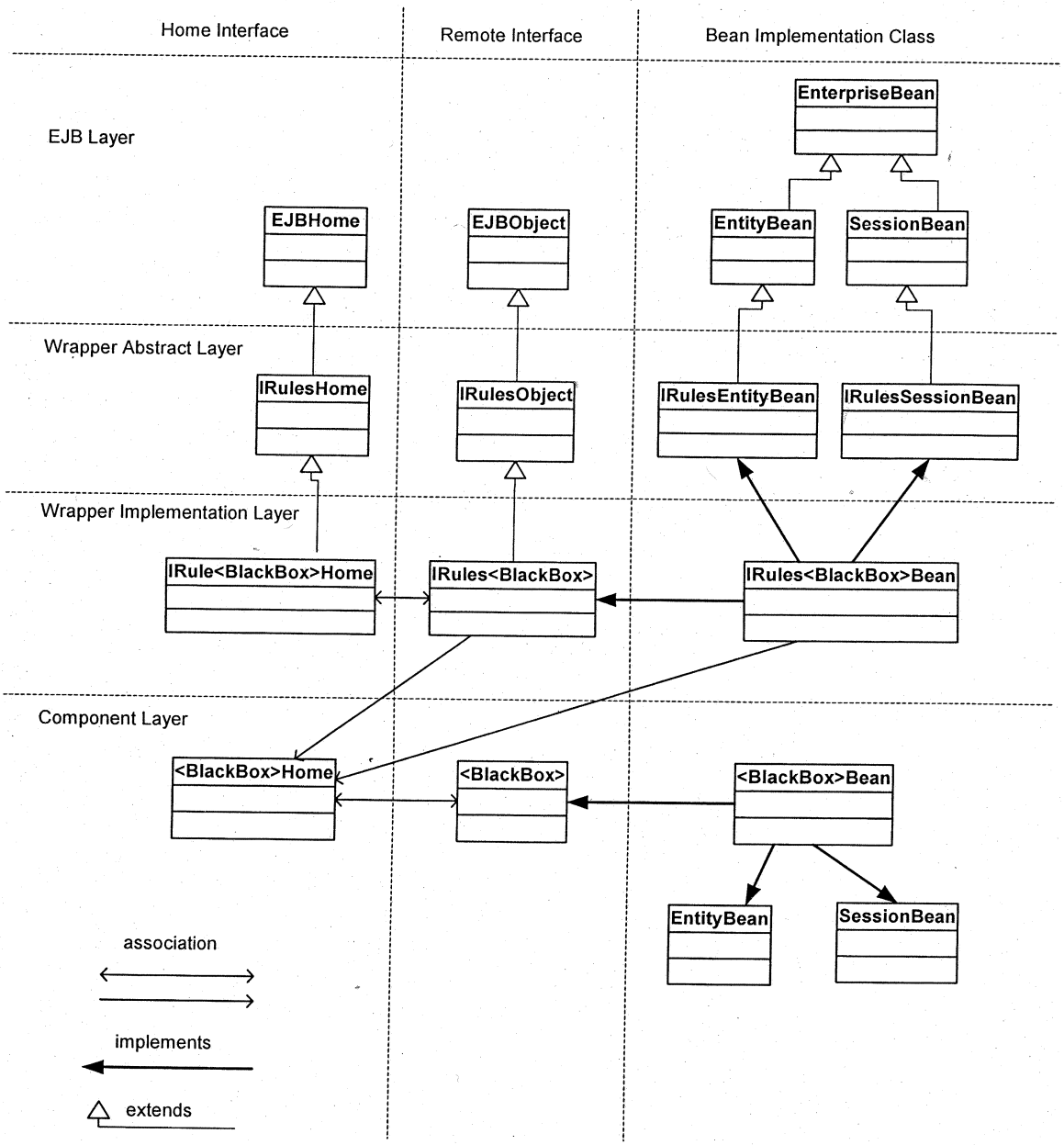

Figure 5: IRules Wrapper Overview

The code for an IRules wrapper is generated by the compilation of CDL. The instances of the wrapper are created using the create method in the Home interface of the wrapper. The parameters to the create method include the primary key of the black-box EJB that the wrapper is wrapping and any initial values for stored attributes and externalized relationships. Note that the primary key of the IRules wrapper is set to the same value (and type) as the primary key of its wrapped purchased component.

The wrapper establishes a reference to its black-box component storing its handle, which is a persistent network reference to an EJB object, as a CMF called cmpForHandle. In a container-managed entity bean, the data type for a CMF can be a Java primitive type or a Java serializable type. The container is responsible for transferring data between an entity instance and the underlying persistent storage, and provides a get and set method for each CMF. To store the EJB handle to the black-box component, the EJB handle is cast to the type java.lang.Object and used as a parameter to the setCmpForHandle method, which is supplied by the container. The following code snippet illustrates how the handle to a PendingOrder entity bean, given by the variable p, is stored in the IRules wrapper:

```
Handle handleToBB = p.getHandle();
Object objHandleToBB = (Object)handleToBB;
setCmpForHandle(objHandleToBB);
```

To retrieve the handle as the appropriate type, the wrapper implements a refToBlackBox method that calls the

getCmpForHandle method to retrieve the serialized handle and casts it to the type of the associated purchased component, in this case PendingOrder:

```
public PendingOrder refToBlackBox()
    {Object obj = getCmpForHandle();
    Handle handle = (Handle)obj;
    EJBObject ejbobj = handle.getEJBObject();
    PendingOrder blackBox = (PendingOrder)ejbobj;
    return blackBox;}
```

There is nothing added to the wrapper itself to support named extents. When the CDL is compiled, the extent name is entered into the static component metadata. The extent name is only used by the application integrator in the IRL application specification to iterate over the extent of a component. The underlying implementation of the IRL will use the findAll method provided in the home interface to realize the extent.

CDL allows the definition of both stored and derived attributes. Since a stored attribute persists as a CMF of the IRules wrapper, the container is responsible for the get and set methods. A derived attribute is not stored but virtual. Its value is derived using the get method in the wrapper that is generated as the result of compiling CDL. The get method calls the method of the session bean that derives the value of the attribute. Consider as an example the getPortfolioValue method in the wrapper for Portfolio that calls the calculatePortfolioValue method of the portfolioAI session bean. The getPortfolioValue method first gets the home interface of the wrapped portfolioAI session bean. This lookup functionality is abstracted in the method lookupAIHome() that uses JNDI to locate the home interface. The call to calculatePortfolioValue calculates the portfolio value with the corresponding black-box portfolio EJB as an input parameter.

```
public float getPortfolioValue()
    {IRulesPortfolioAIHome home = lookupAIHome();
    IRulesPortfolioAI ai = home.create();
    Portfolio self = refToBlackBox();
    float value = ai.calculatePortfolioValue(self);
    return value;}
```

Externalized relationships are also implemented as a CMF in the IRules wrapper, since the current restriction of CMRs in the EJB 2.0 specification limits relationships to entity beans deployed in the same ejb-jar file. Relationships are associations that can be single-valued or multivalued. A single-valued relationship is stored in a manner similar to the reference to the black-box component, storing the serialized handle to the related object. Multivalued relationships use a Vector to store the handles of the multiple related objects. Since a Vector is serializable, it is then stored in the CMF for the multivalued relationship. The wrapper provides the required translation between the CMF and the required types.

Consider as an example, the pendingTrades relationship defined in the Stock component that represents the set of pending orders for the stock. The following code snippet illustrates the functionality of the addPendingTrades method that adds a PendingOrder instance to this multivalued relationship. The getCmpForPendingTrades() method provided by the container returns the CMF for the relationship, which is called cmpForPendingTrades. The retrieved object is cast to a Vector and the handle to the IRules wrapper for the pendingOrder is added to the Vector before it is made persistent by the call to the container-provided method setCmpForPendingTrades.

```
public void addPendingTrades(IRulesPendingOrder ir)
    {Object obj = getCmpForPendingTrades();
    Vector relatedPendingOrder = (Vector)obj;
    Handle handle = ir.getHandle();
    relatedPendingOrder.addElement(handle);
    Object ref = (Object)relatedPendingOrder;
    setCmpForPendingTrades(ref);}
```

The IRules Definition Language allows the application integrator to refer to purchased components and their methods. Therefore, the underlying implementation of these languages must translate a call to a method on the purchased component to its IRules wrapper, allowing the hooks into the IRules environment. Thus, the IRules Wrapper acts as a proxy for calling a method on its associated black-box component. Whenever a method on a black-box EJB is called from within the IRules environment (from an action of an integration rule or an application transaction), the IRules environment passes the control of execution to the corresponding method of the IRules wrapper. Every method in the black-box EJB has a corresponding method with the same method name in its IRules wrapper. The arguments to the method in the wrapper include all the parameters to the corresponding method in the black-box component and in the same order. There are additional parameters to pass the transaction context.

The Component Definition Language allows for the definition of events that are raised before or after a method call on the underlying black-box component. The IRules Wrapper for the component is responsible for triggering these method events to the IRules environment. Consider the case where an event is raised after a method call. After completing all of the preliminary actions needed by the IRules environment, the IRules Wrapper delegates to the business method of the black-box component to execute the business logic. After executing the method of the black-box component, control returns to the IRules Wrapper, which is then responsible for triggering the after method event. The wrapper bundles all the necessary information including the transaction context into a common IRules event data structure and publishes the occurrence of the afterEventName to the IRules topic via the JMS messaging service. The IRules Event Handler notifies the rule processor when a new event is detected and rule processing is done. Further detailed information about the execution environment can be found in a companion paper (Urban et. al. 2002).

6.2 Session Beans

The IRules wrapper for a deployed black-box session bean is also a session bean. Similar to the wrapper of an entity bean, the IRules wrapper wraps all of the business methods in the underlying black-box session bean and acts as a proxy to method calls on the purchased component. The IRules wrapper also generates the method-based events that are defined in the CDL. Since session beans do not represent persistent data, the IRules session bean wrapper does not support the additional attributes or externalized relationships supported by the IRules entity bean wrapper.

The IRules session bean wrapper is a stateful session bean. The IRules Wrapper holds a reference to the underlying black-box stateful session bean to maintain the conversational state across method calls. For a black-box stateless session bean, the IRules Wrapper creates a new instance of the underlying black-box component before invoking any methods on the purchased component. In the initial prototype that has been implemented, all of the required information to access the underlying black-box session bean, like the JNDI name of the black-box component it is wrapping and the EJB server URL has been stored as part of the state information of the IRules wrapper. The IRules session bean wrapper is designed as a stateful session bean to retain this information for the entire lifecycle of the IRules wrapper bean. The current design could be modified to obtain the information from environment variables, thus opening up the possibility of having an IRules stateless session bean wrapper that wraps a stateless black-box session bean.

7 Summary and Future Directions

This paper presented the IRules Component Definition Language and the metadata and wrapper framework required for supporting the IRules active approach to component-based software integration. The IRules environment acts as a mediator (Gamma et. al. 1995) in the integration process by encapsulating the logic describing the interconnections between components using integration rules.

The implementation described in this paper and its companion paper (Urban et. al. 2002) on transaction and execution control is a prototype of the technology based on the Investment example. Work is underway to develop a general-purpose system that uses Jini as the basis of the distributed computing environment.

There are also language issues to be investigated and implemented. We are currently developing a compiler for CDL that uses JavaSpaces for the storage of metadata and automatically generates the EJB wrapper code for the components. Although we have an initial design for the IRL, we are in the process of investigating condition evaluation techniques for IRL rule conditions that involve distributed query processing.

Acknowledgements

We want to thank Rohini Patil for her assistance in the refinement and illustration of the component metadata diagram.

This work was partially supported by a grant from the National Science Foundation (IIS-9978217).

References

[1] T. Adbellatif. An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States, Ph.D. Dissertation, ASU, Dept. of Computer Science and Engineering, Fall 1999, 375 p.

[2] T. Abdellatif, R. Chan, S. W. Dietrich, B. Siddabathuni, A. Sundermier, and S. D. Urban, "Meta-Data Components in Support of an Active Deductive Object-Oriented Database System," Proc. of the 3rd IEEE Meta-Data Conference, Bethesda, MD, On-line publication: http://computer.org/conferen/proceed/meta/1999/, paper #16, 1999.

[3] K. Arnold, The Jini™ Specifications: 2nd Ed., Addison-Wesley Publishers, NJ, 2000.

[4] K. Ayyaswamy, "The Design and Implementation of a CORBA based environment for Distributed Constraint Maintenance," M.S. Thesis, Dept. of Computer Science and Engineering, ASU, 1999.

[5] D. Barrett, L. Clarke, P. Tarr, and A. Wise, "An Event-Based Software Integration Framework," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 4, October 1996.

[6] BEA Systems Weblogic Server, http://edocs.beasys.com/wls/docs61/index.html

[7] G. Bultzingsloewen, A. Koschel, and R. Kramer, "Active Information Delivery in a CORBA-based Distributed Information System," Proc. of the First International Conference on Cooperative Information Systems (CoopIS'96), Brussels, Belgium, June 1996.

[8] R. G. G. Cattell, ed., The Object Database Standard, ODMG 3.0, Morgan Kaufmann, 2000.

[9] S. Chakravarthy and R. Le, "ECA Rule Support for Distributed Heterogeneous Environments," Proc. of the International Conference in Data Engineering, Orlando, 1998.

[10] M. DeJong and C. Laird, "TCL+Java = A Match Made for Scripting," http://www.sunworld.com/sunworldonline/swol-11-jacl.html.

[11] S. W. Dietrich, S. D. Urban, J. V. Harrison, and A. P. Karadimce, "A DOOD Ranch at ASU: Integrating Active, Deductive, and Object Oriented Databases," Data Engineering Bulletin, Special Issue on Active Database Systems, vol. 15, no. 1-4, December, 1992, pp. 40-43 (see also http://www.eas.asu.edu/~adood)

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Design, Addison-Wesley, Reading, MA, 1995.

[13] Java™ 2 Platform Enterprise Edition 1.3, http://java.sun.com/j2ee/

[14] A. Koschel and P. Lockemann, "Distributed Events in Active Database Systems: Letting the Genie Out of the Bottle," Journal of Data and Knowledge Engineering (DKE), vol. 25, pp. 11-28, 1998.

[15] R. Le and S. Chakravarthy, "Support for Composite Events and Rules in Distributed Heterogeneous Environments," Technical Report, Computer and Information Science and Engineering Dept., University of Florida, January 1998.

[16] C. Ma and J. Bacon, "COBEA: A CORBA Based Event Architecture," 4th Conference on Object Oriented Technologies and Systems (COOTS), New Mexico, April 1998.

[17] Microsoft Corporation. COM+, http://www.microsoft.com/com/tech/complus.asp

[18] Object Management Group: The Common Object Request Broker, Architecture and Specification, Revision 2.3, December 1998.

[19] Object Management Group: Relationship Service Specification, version 1.0, April 2000.

[20] N. Pissinou and K. Vanapipat, "Active Database Rules in Distributed Database Systems: A Dynamic Approach to Solving Structural and Semantic Conflicts in Distributed Database Systems," Computer Systems Science and Engineering, vol. 1, pp. 35-44, 1996.

[21] N. Pissinou , K. Makki, and R. Krishnamurthy, "An ECA Object Service to Support Active Distributed Objects," Informatics and Computer Science, pp. 63-104, 1997.

[22] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," Proc. of OOPSLA, 1987, pp. 466-481.

[23] A. Saxena, An Evaluation of Distributed Architectures for the Integration of Black-Box Software Components, M.S. Thesis, ASU, Dept. of Computer Science and Engineering, Tempe, AZ, Fall 2000.

[24] A. Silberschatz and S. Zdonik, "Database Systems – Breaking Out of the Box," ACM SIGMOD Record, vol. 26, no. 3, September 1997.

[25] S. D. Urban, A. P. Karadimce, S. W. Dietrich, T. Ben Abdellatif, and R. Chan, "CDOL: A Comprehensive Declarative Object Language," Data and Knowledge Engineering, vol. 22, 1997, pp. 67-111.

[26] S. Urban, S. Dietrich, A. Saxena, A. Sundermier, "Interconnection of Distributed Components: An Overview of Current Middleware Solutions," Journal of Computer and Information Science in Engineering, vol. 1, no. 1, March 2001, pp. 23-31.

[27] S. D. Urban, S. W. Dietrich, Y. Na, Y, Jin, A. Sundermier, and A. Saxena, "The IRules Project: Using Active Rules for the Integration of Distributed Software Components," Proc. of the 9th IFIP Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems, Hong Kong, April 2001, pp. 265-286.

[28] S. D. Urban, A. Saxena, S. W. Dietrich, and A. Sundermier, "An Evaluation of Distributed Computing Options for the Integration of Black-Box Software Components," Proc. of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, June 2001, pp. 100-109.

[29] S. D. Urban, S. W. Dietrich, A. Sundermier, Y. Na, Y. Jin, and S. Kambhampati, "Distributed Software Component Integration: A Framework for a Rule-Based Approach," To appear in the Handbook of Electronic Commerce in Business and Society, P. Lowry, J. Cherrington, and R. Watson (editors), CRC Press, 2002.

[30] J. Widom and S. Ceri, Active Database Systems: Triggers and Rules for Advanced Database Processing, Morgan Kaufmann Publishers, San Francisco, 1996.

Volume 25 Number 4 November 2001 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

Special Issue:

Component Based Software Development

Guest Editors:

M.B. Juric, I. Rozman, D. Deugo

The Slovene Society Informatika, Ljubljana, Slovenia

Volume 25 Number 4 November 2001 ISSN 0350-5596

Informatica

An International Journal of Computing and Informatics

Special Issue of the *Informatica* - An International Journal of Computing and Informatics

Dedicated to

Component based software development

Component based software development (CBSD) has become the predominant way of developing, packing, deploying and using software. CBSD influences all aspects of software development, which was reflected in a large number of submitted articles. Initially we received 77 contributions, which made the review process and the final selection very difficult.

For the special issue of the Journal Informatica, dedicated to Component Based Software Development we have selected fourteen quality papers, which cover different aspects of CBSD, including integration, modeling and design, patterns, agents, security, formal specifications, fault-tolerance, discussion of management processes for CBSD and a case study.

The first paper, "A Language and Framework for Supporting an Active Approach to Component-Based Software Integration" by Suzanne W. Dietrich, Susan D. Urban, Amy Sundermier, Yinghui Na, Ying Jin, and Sunitha Kambhampati, presents the IRules Component Definition Language and the environment, which acts as a mediator in the integration process by encapsulating the logic describing the interconnections between components using integration rules. It also presents the wrapper framework required for supporting the IRules active approach to component-based software integration.

The second paper, "DEPA (Design Pattern Application) – A Component-based Model for Applying Design Patterns in Software Development", by Katrina Ji and Sean Chen, discusses the lack of a formal model in applying design patterns. The authors present the DEPA model that allows a systematic way of applying design patterns in software development projects, particularly to those projects with resource constraints.

The third paper, "An Approach for Modeling Components with Customization for Distributed Software", by X. Xie and S. M. Shatz, discusses an approach for blending Petri net concepts and object-oriented features to develop a specification approach for distributed component software systems. A key result is the definition of a "plug-in" structure that can be used to create "subclass" object models, which correspond to customized components.

The fourth paper, "A Uniform Component Modeling Space" by Duane Hybertson, presents a component modeling space as a context for supporting component-based software development and accumulating component-related knowledge. It provides a uniform structure for modeling components and modeling systems in which the components may be integrated. The uniform structure can serve as the basis for an organized repository of knowledge of components and systems in which they can be used.

The fifth paper, "An Agent-Based Component Platform for Dynamically Adaptable Distributed Environments" by Rainer Weinreich and Reinhold Plösch, argues that increased flexibility can be achieved by using agent technology and agent platforms as powerful component environments. The authors present an adaptable component platform which incorporates mobile agent platforms and describe how important issues of component deployment, configuration and security are supported by the environment.

The sixth paper, "MobiDoc: A Mobile Agent-based Framework for Compound Documents" by Ichiro Satoh, presents a mobile-agent-based framework for building mobile compound documents, called MobiDoc, where the compound document can be dynamically composed of mobile agent-based components and can migrate itself over a network as a whole, with all its embedded agents.

The seventh paper, "BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems", by Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault, answers the software engineering needs of the design of knowledge-based system engines in that it presents a framework composed of reusable and adaptable software components.

The eighth paper, "A Security Assurance Framework for Component Based Software Development", by Ashwin Kumar M. V. N., Arun K. Singh, and Ramesh Babu S., presents a framework to assure security of components. The framework uses Aspect Oriented Programming paradigm to capture security characteristics of the components and weaves the corresponding security checks into them. It also introduces a novel verification mechanism to ensure that the COTS components are developed as per security contract.

The ninth paper, "The ABCs of Specification: AsmL, Behavior, and Components" by Mike Barnett and Wolfram Schulte, shows how to use AsmL, an executable specification language, to provide behavioral interfaces for components. This allows clients to fully understand the meaning of an implementation without access to the source code.

The tenth paper, "Towards Rigorous and Effective Functional Contract for Components" by F. J. Galan Morillo, V. Diaz and J. M. Canete Valdeon, proposes a