An Integration Rule Processing Algorithm and Execution Environment for Distributed Component Integration

Ying Jin
California State University, Sacramento
Department of Computer Science
Sacramento, CA 95819, USA
E-mail: jiny@ecs.csus.edu

Susan D. Urban, Suzanne W. Dietrich and Amy Sundermier
Arizona State University
Department of Computer Science and Engineering
Tempe, AZ 85287-8809, USA
E-mail: s.urban@asu.edu, dietrich@asu.edu

Keywords: active databases, software component integration, rule processing algorithm, transaction, management

Received: July 25, 2005

The Integration Rules (IRules) Project[*] provides an active, rule-based approach for supporting event-driven activity in applications involving distributed software component integration. This paper presents the execution model, transaction model, and integration rule execution algorithm of the IRules environment. The paper begins with an overview of the IRules language framework to establish the context for the use of events and rules in the integration process, with Enterprise JavaBeans (EJBs) serving as a component model. The paper then elaborates on the integration rule processing algorithm and execution environment. The rule execution model supports traditional active rule coupling modes, and defines a new immediate asynchronous mode to support concurrent execution of triggered rules and transactions. The transaction model is based on the flexible transaction model, providing a means to coordinate global transaction execution with the transactional features of EJB containers. IRules component wrappers also provide support for the global transaction context as well as the synchronization of method execution with the nested execution of integration rules. The paper defines the semantics of coupling modes in terms of cycles and levels of rule execution, presenting the integration rule processing algorithm for coordinating the execution of events and methods on components with the nested execution of integration rules in the context of the transaction model. The details of the algorithm are presented using Unified Modeling Language (UML) activity diagrams, providing a generic approach that can be used as the foundation for rule processing in other distributed environments. An investment application is used to illustrate the concepts presented in this paper.

Povzetek: Predstavljen je algoritem za integracijo pravil.

1 Introduction

The development of advanced enterprise applications often requires the integration of distributed software components and services. Standard component models and distributed computing tools, such as the Common Object Request Broker Architecture (CORBA) [1] and Enterprise JavaBeans (EJBs) [2], have been developed to facilitate the integration process in distributed environments. However, component integration could be a difficult process in some cases, since application integrators must not only mediate the semantics of component interactions, but must also be skilled in low-level knowledge of middleware programming, event handling, and transaction management. This difficulty motivates the need for a more declarative approach to the integration process.

In response to this need, the Integration Rules (IRules) Project has developed an active rule-based approach to distributed component integration, using *integration rules* to provide a declarative approach to event-driven integration activity [3, 4, 5, and 6]. The IRules project is based on the concept of active database rules. Active database systems extend traditional databases by supporting mechanisms to automatically monitor and react to events that are taking place either inside or outside of the database system [7 and 8]. Active database rules, known as Event-Condition-Action (ECA) rules, are the core of any active system.

[*] This research was supported by National Science Foundation under Grant No. IIS-9978217.

Similar to an active rule, an integration rule consists of an event, a condition, and an action. The event of an integration rule is generated from distributed sources. The condition is expressed as a query over distributed components. If the condition evaluates to true, the action is invoked to execute methods on components or to invoke application transactions that capture integration logic. Integration rules can therefore be used to separate event processing from the main integration logic of an application. Furthermore, event handling and rule processing are managed within the transactional environment of the IRules system, shielding the low-level details of event handling and transaction management from integrators. Integrators can therefore focus on integration logic rather than low-level programming details.

The IRules approach to component integration consists of a language framework described in [4] and an execution environment for processing rules and transactions [6] over distributed components. The execution environment presented in this paper consists of a rule processing algorithm and transaction management system, illustrating a rule-based approach to the integration of components with well-defined interfaces based on the EJB component model [2]. The integration of black-box components introduces several challenges to the development of a rule and transaction processing framework for integration rules. First of all, components cannot be modified and they are typically not aware of their participation in the integration framework. As a result, components alone do not provide necessary behavior for participating in more global rule and transaction processing activities. Furthermore, the EJB component model has its own notion of transactional behavior, which is beyond the control of an external environment such as IRules. The execution of integration rules within a transactional context requires suitable control logic at the global IRules level to overcome the restrictions of the underlying EJB component model. Furthermore, active rules can trigger other active rules, thus forming a nested structure as a result of cascaded rule execution. Since the nested execution of rules and their transaction control in a distributed environment may span across several distributed locations, distributed rule processing is more challenging than that of centralized active rule environments.

The IRules execution model presented in this paper supports the traditional dimensions of active rule execution, with extensions for use in a distributed environment. The Integration Rule Processing (IRP) algorithm controls rule processing in a distributed environment, fully supporting immediate, deferred, and decoupling modes of execution. The immediate asynchronous mode is a new coupling mode defined in this research to support concurrent execution of triggering transactions and triggered rules, thus improving the performance of distributed rule processing. The IRP algorithm also provides support for the nested execution of immediate rules. Handling immediate rules in a distributed environment requires system control for synchronization between a triggering transaction and the triggered rules. The synchronization process requires suspension of the method execution of the EJB component, allowing the generation of events before and after the method execution, with the immediate execution of rules in response to the events. The IRP algorithm described in this paper contributes to the use of immediate coupling modes and nested rule execution within the IRules framework. These features for the nested execution of immediate rules in a distributed environment have not been addressed by previous research, especially in the context of component integration.

Our research uses Unified Modeling Language (UML) activity diagrams [9] to present the logic of the rule execution algorithm. The algorithm is generic so that it can be used in other environments for rule processing, although the specific implementation of the IRP algorithm within the IRules environment is supported by the IRules transaction management [6], wrapper design [10], and synchronization algorithms [11].

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 provides an overview of the IRules approach with a presentation of the language framework and system architecture. Section 4 presents the integration rule coupling modes as well as the transaction model and the transactional support found in wrappers for the synchronization of rule and method execution. Section 5 describes the rule processing algorithm and provides an example of rule execution using an investment application. The paper concludes in Section 6 with a summary and discussion of future research directions.

2 Related Work

There are several active database research projects that have influenced the development of the IRules environment, including relational active databases, such as POSTGRES [12] and Starburst [12], as well as object-oriented active databases, such as HiPAC [14], SAMOS [15], ADOOD RANCH [16], and REACH [17]. Active rules also exist in a limited form in commercial database systems as database triggers [18].

Active rule execution algorithms have been addressed in centralized environments, such as the research outlined in the introduction to active database systems in Widom and Ceri [8], as well as the work of Fraternali and Tanca [19] and Warshaw [20]. The algorithm in [8] provides a high level abstract view of rule processing, repeatedly retrieving a triggered rule, evaluating the condition, and performing the action if the condition evaluation is true. Similarly, the algorithm in [19] presents three phases of active rule processing. In the triggering phase, the algorithm builds a set of rules that are triggered. In the consideration phase, the algorithm first gets rules from

the set of rules constructed in the triggering phase, and then evaluates the condition part of each rule. In the action phase, if the condition evaluation of a rule returns non-empty bindings, the algorithm will perform the action of the rule. Compared to [8] and [19], this paper presents an active rule processing algorithm that was specifically designed to assist with component integration in a distributed environment. The IRules algorithm not only covers the three basic phases of active rule processing for a set of triggered rules, but also elaborates on the nested execution of integration rules for distributed components, which is a challenging extension to past work on centralized rule execution algorithms. The IRules algorithm also describes how to execute a rule according to four types of coupling modes and how to react to different types of events in the context of distributed component integration. Whereas the algorithm in [20] uses state transition to characterize rule execution according to different coupling modes, the IRules algorithm uses a coordinate system to describe the semantics of rule execution.

In addition to centralized active database systems, there are several research projects on active, rule-based distributed systems. In the system described in [21], ECA rules are used to provide distributed communication for the components that describe interfaces in the Object Management Group (OMG) Interface Definition Language (IDL) [22]. The project focuses on the specification, detection and management of composite events.

C^2offein [23] is a CORBA-based system with a comprehensive design of distributed event detection. The underlying data sources are wrapped to enable read access in a CORBA environment. If a database does not support an active mechanism, the wrapper queries the database at regular intervals to detect changes in the data. Clients can also call event detection before any update operation to the database. Rule processing is supported using a production rule expert system shell.

The FRAMBOISE (FRAMework using oBject OrIented technology for Supplying active mEchanisms) project [24] is an object-oriented framework formed by a toolbox to provided active database functionality, such as event definition, event detection, and rule execution. The set of architectural components that separates the active functionality from the underlying DBMS is called an activity service. A database event detection connector, condition evaluation connector, and database action execution connector regulate the interaction between the activity service and the underlying DBMS.

In the system described in [25], active rules are used to glue together existing applications in a distributed environment. Active rule processing is implemented through event, condition, and action services. The condition object of a rule subscribes to the event object of the rule, while the action object subscribes to the condition object of the rule. The system uses the publish/subscribe service implementation of X^2TS [26] as a notification mechanism between event, condition, and action, where X^2TS is based on the CORBA Notification Service. X^2TS can also provide additional transaction control mechanisms such as exception handling over the basic CORBA Transaction Service. In contrast, the IRules rule manager controls when to evaluate a condition and execute an action according to coupling modes, rather than through a publish/subscribe service.

The above distributed rule projects have all been based on the use of the CORBA standard. In contrast, the IRules project requires access to distributed resources that advertise services using the EJB component model. The use of distributed rules for the integration of EJB component technology has not been addressed by the existing research. The IRules project is also using Jini connection technology [27] as the primary means for distributed object computing, rather than CORBA as in other projects. Furthermore, the rule processing algorithm of IRules can handle nested rule execution in a distributed environment. Existing distributed rule projects have not addressed cascaded rule execution within distributed transactions.

Using active rules to control the flow between activities of a workflow system has been adopted by a number of projects, such as the project described in [28]. More recent work on workflow uses ECA rules both inside and outside of activities. The work in [29] is a centralized workflow management system, where ECA rules are used for constraint management inside tasks, as well as for the control of the execution order of tasks. In [30], the workflow system named CapBasED-AMS uses ECA rules to specify the security authorization requirements imposed on a task as well as the execution sequence. The $TriGS_{flow}$ system of [31] is introduced as a framework for workflow management, where ECA rules encapsulate and realize coordination policies. In the WIDE (Workflow on Intelligent Distributed Database Environment) project [32], active rules are used in a workflow management system for exception handling. Compared to component integration, the flow movements from task to task in a workflow environment are well-defined compared to the interconnection of software components. A workflow system has more control over the tasks that are executed, while access to component services, especially black-box components, may be more restrictive.

Active rules are also used in the composition of web services. The research in [33] proposed the SELF-SERV (compoSing wEb accessibLe inFormation & buSiness sERVices) system to compose services within a peer-to-peer paradigm. SELF-SERV includes a declarative language based on state charts and a peer-to-peer service execution model. A statechart consists of states and transitions. Transitions are labeled by ECA rules. Compared to IRules, SELF-SERV is for the composition of web services, while IRules is for the integration of software

components. ECA rules are used as the "glue" of the IRules integration to specify event-driven integration logic, while state charts are the "glue" for the composition of SELF-SERV. ECA rules are used in SELF-SERV for state transition, but the logic of execution is not controlled by ECA rules. In the IRules environment, integration logic is composed from the use of application transactions together with integration rules that respond to events.

3 The IRules Approach

There are two important aspects of the IRules environment: the IRules Definition Language (IRDL) for application specification and the IRules execution environment for integration rule and transaction execution. IRDL supports the definition of components, events, rules, and application transactions for distributed component integration. The execution environment consists of the execution model for integration rules, transaction management for rule execution, and the rule execution algorithm that coordinates the execution model with transaction management. The IRules execution environment is described in detail in Section 4. This section overviews the language framework and the architectural design of the IRules environment.

3.1 The IRules Definition Language

The IRDL consists of four sub-languages: the Component Definition Language (CDL) for defining IRules components, the IRules Scripting Language (ISL) for describing application transactions, the Event Definition Language (EDL) for defining events, and the Integration Rule Language (IRL) for defining active rules. The IRDL was initially reported in [3, 5] with refinements of the language presented in [10, 11, 34, 35]. The examples of the language presented in this section originally appeared in [4, 6] and are repeated here to make the paper self-contained.

3.1.1 Component Definition Language (CDL)

The CDL establishes an object model for application integration activity [3, 5, and 10]. The current implementation of IRules is based on the EJB component model, assuming that all of the components of the environment are EJB components. To support component interconnection with active rules, IRules adds a semantic layer on top of existing EJB components. This layer is the IRules wrapper layer. IRules wrappers are automatically generated after CDL is compiled. IRules wrappers provide additional functionality to black-box components, such as defining externalized relationships between distributed components, and specifying extents, derived attributes, and stored attributes for each component. The IRules wrapper layer also defines the events generated before and after method calls on components as well as the events that are internal to black-box components. The details of IRules wrappers can be found in [10].

As an example of CDL, Figure 1 defines two externalized relationships and one event for a pending order component within an investment application. The first line of the component definition indicates that the StockBroker_PendingOrder is an EntityBean component. The second line of the definition specifies an extent that can be used to query all pending order instances, a feature that is useful in the specification of integration rule conditions (see Section 3.1.4 for an example). Assuming that StockBroker_Stock, StockBrocker_Portfolio, and StockBroker_PendingOrder are implemented as separate distributed components, the first relationship definition in Figure 1 illustrates an externalized, bi-directional relationship between components: a StockBroker_PendingOrder is orderedBy a StockBroker_Portfolio, while in the inverse direction, a StockBroker_Portfolio orders a StockBroker_PendingOrder. The second relationship defines the relationship between StockBroker_PendingOrder and StockBroker_Stock: a StockBroker_PendingOrder actsUpon a StockBroker_Stock while a StockBroker_Stock has pendingTrades on the StockBroker_PendingOrder. The CDL also defines an event afterSetAction that is to be raised after the setAction operation on StockBroker_PendingOrder. Note that the CDL definition of StockBroker_PendingOrder does not repeat any of the method definitions of the original component definition. CDL is used to enhance the component definition with IRules functionality.

3.1.2 IRules Scripting Language (ISL)

In the IRules environment, ISL describes well-defined sequences of processing logic as application transactions [3, 5]. ISL is based on JACL [36], which is the Java version of the Tool Command Language (TCL) [37]. An ISL example in an investment application is shown in Figure 2. The clientWantsToSellStock application transaction consists of two steps: create a pending order component and print the information of this pending order. The newInstance command is a JACL extension that abstracts a sequence of statements into one command, thus making the script concise and easy to reuse.

```
Component StockBroker_PendingOrder implements
EntityBean
(extent pendingOrders)
{ relationship StockBroker_Portfolio orderedBy inverse
StockBroker_Portfolio::orders;
  relationship StockBroker_Stock actUpon inverse
StockBroker_Stock::pendingTrades;
  event afterSetAction (pnAction) {method after
setAction(string pnAction)};}
```

Figure 1: CDL of PendingOrder Component

```
application transaction clientWantsToSellStock(String pnId,
String portfolioId, String stockId, int numOfShares,float
```

```
desiredPrice, String action, Stock actUpon, Portfolio
orderedBy)
tcl newInstance
{
set pn [newInstance PendingOrder $pnId $portfolioId
$stockId $numOfShares $desiredPrice $action $actUpon
$orderedBy $irulesId];
 printPendingOrderInfo $pn $irulesId;
}
```

Figure 2: ISL Example for the clientWantsToSellStock transaction

3.1.3 Event Definition Language (EDL)

There are four different types of IRules events [3, 5, and 11]: *method events*, *application transaction events*, *internal events*, and *external events*. A method event is generated before or after the execution of a method on a component. An application transaction event is generated before or after the execution of an application transaction. An internal event is an event generated by a black-box component. An external event is generated by sources external to the IRules environment. EDL describes application transaction events and external events. Method events and internal events are defined in CDL following EDL syntax. Figure 1 illustrates the definition of a method event afterSetAction that is generated after the setAction operation in the Stock component.

Figure 3 shows an application transaction event definition in EDL. The specification has the syntax similar to the method event specification in Figure 1. The key word appTrans identifies that the event is an application transaction event. The event has an event name afterSellStockOnNewPO and five parameters. The event parameters are constructed by the projection of the parameters of the application transaction by parameter name.

```
event afterSellStockOnNewPO(stockId, price, portfolioId,
numOfShares, pn)
{
appTrans after sellStockOnNewPO(String stockId, float
price, String portfolioId, int numOfShares,
stockBroker.PendingOrderComponent.PendingOrder  pn);
}
```

Figure 3: EDL for the afterSellStockOnNewPO event

3.1.4 Integration Rule Language (IRL)

IRL is a language for defining integration rules [3, 5, 34, and 35]. IRL is based on the traditional ECA rule format in active database systems. An integration rule includes an event, a condition, and an action. A condition includes a Boolean clause and an optional query over the object model to define a binding structure for data that satisfies the condition. The action part consists of an optional from clause and a do clause. The from clause iterates through the binding structure passed from the condition. The do clause executes the action in the format of a method call or an application transaction.

```
create rule  clientWantsToSellStockRule
event     afterClientWantsToSellStock(pnId, portId, stockId,
numOfShares, desPrice, pnaction, actUpon, orderedBy)
condition immediate
          when pnaction = "sell"
          define stockAndPendingOrder as
                    select struct ( stk: s, newPo: pn )
                    from s in stocks, pn in pendingOrders
                    where pn.id=pnId and pn.actUpon=s
                    and desPrice<=s.price
action    immediate
          from sp in stockAndPendingOrder
          do  sellStockOnNewPO(stockId,  sp.stk.price,
portId, numOfShares, sp.newPo)
```

Figure 4: IRL Example of the clientWantsToSellStockRule rule

An example of IRL is shown in Figure 4. In this rule, the event is signaled after the clientWantsToSellStock application transaction. The condition checks whether the pendingOrder intends to sell stock.. If the rule condition is satisfied, a binding structure is defined for relevant instances of stock and pendingOrder. The binding structure definition uses the extents of the stock and pendingOrder components that are specified in the CDL and the parameters of the event to find relevant stocks and pending orders. The action part iterates through the stockAndPendingOrder structure and executes the sellStockOnNewPO application transaction to perform the functionality of selling stocks.

3.1.5 Putting It All Together

Figure 5 presents an example of how all of the sublanguages of IRDL work together. CDL defines wrapped components and the compilation of CDL generates wrappers. After wrappers are generated [10], distributed components in different containers may have externalized relationships. Four types of events are defined by EDL and CDL. Events can trigger integration rules defined by IRL. The action part of the rule can invoke an application transaction (defined by ISL) or a method on an EJB component. The execution of an application transaction or a method can raise application transaction events, method event, or internal events, which can trigger additional rules. As a result, integration rules can be triggered in a nested structure. We will present an execution scenario of rule nesting in Section 5.3.

3.2 The IRules Architecture

IRules has designed a distributed architecture to support the IRDL language framework. The

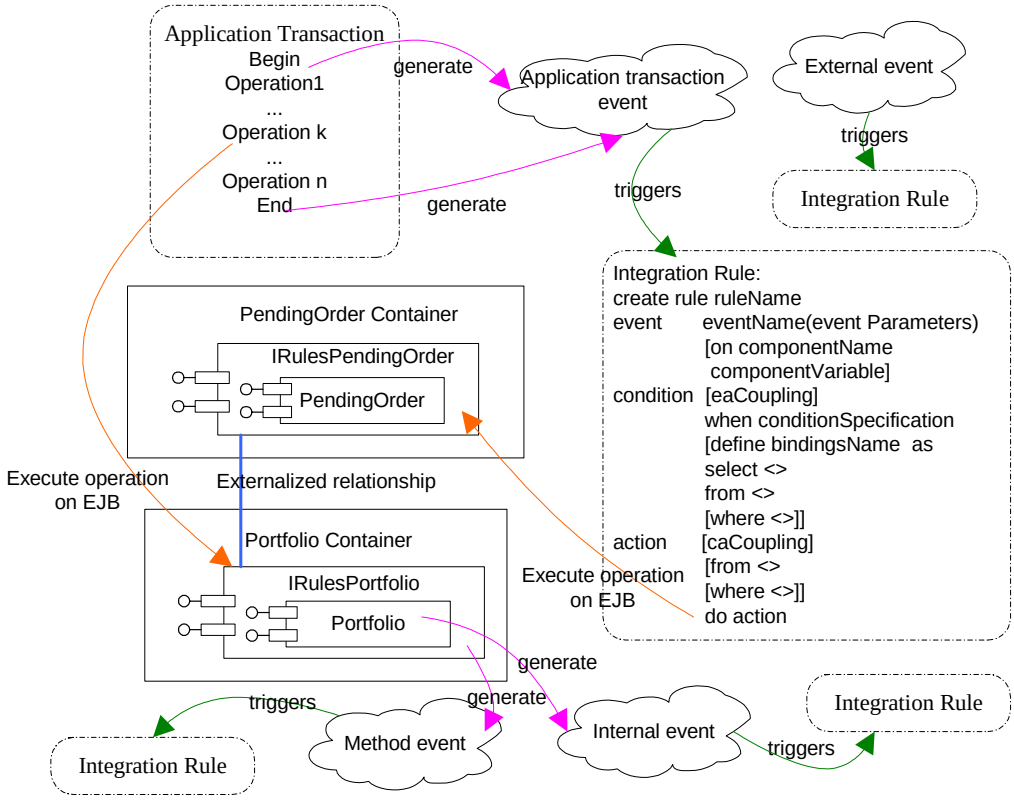


Figure 5 Interaction Between the IRDL Sublanguages

architecture can be abstracted into three layers, as shown in Figure 6. The top and middle layers are the interfaces. The bottom layer is the implementation of the integration system.

IRules provides interfaces for two types of integration users: integrators and end users. Integrators use IRDL to describe integration logic. The compilation of IRDL results in the population of metadata and the automatic generation of wrappers. The interface for end users consists of a list of application transactions, which have been expressed by integrators using ISL and compiled by the IRules compiler. An end user then selects an existing application transaction to express their integration request. For example, if a user wants to sell stock, the user can select the clientWantsToSellStock application transaction from Figure 2, providing values for each parameter of the application transaction. The user request is sent to the application transaction processor component of the IRules system.

The implementation layer consists of architectural components for the IRules system. Figure 6 presents the fundamental components of the architecture. The Jini distributed computing environment is used as the backbone of the system, with IRules architectural components implemented as Jini Services. Upon user request, the application transaction processor processes the ISL script. The processing may invoke wrapped EJB components. The processing of an application transaction or a method call on an EJB component can raise events. The event handler pushes the event to the rule manager, where the rule manager queries the metadata to retrieve rules triggered by this event. During rule processing, the rule manager interfaces with the transaction manager to establish the transaction context for rule execution. The rule manager also interfaces with the object manager for accessing components. The rule manager submits requests to the query processor for rule condition evaluation during rule processing.

Through the IRDL language framework and the architectural design, the IRules integration system allows application integrators to specify the integration logic in a declarative fashion, while the end users can use the defined integration logic to specify their request. In contrast to the traditional integration approach, the IRules approach does not require an integrator's low-level knowledge of distributed programming issues. Integrators can focus on integration logic, rather than the technical details of rule and transaction processing.

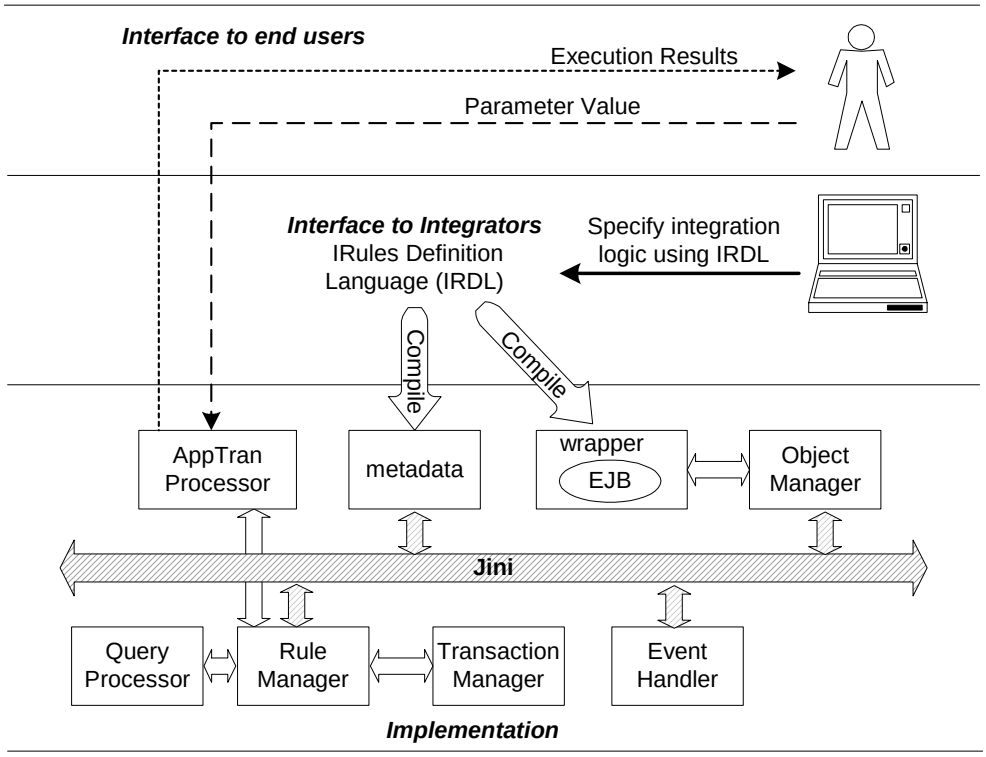


Figure 6: IRules Architecture

4 The IRules Execution and Transaction Model

This section presents the details of the IRules execution and transaction model. A preliminary discussion of the IRules transaction model appears in [6]. The full details of the transaction management system are presented in this section to establish the context for presentation of the rule processing algorithm in Section 5. Section 4.1 defines the coupling modes of the environment, with specific emphasis on the synchronous and asynchronous options of the immediate coupling mode. Section 4.2 elaborates on the transaction model and the manner in which it interacts with the transactional features of EJB containers. Section 4.3 discusses the support that IRules component wrappers provide for the global transaction context as well as the synchronization of rule and method execution.

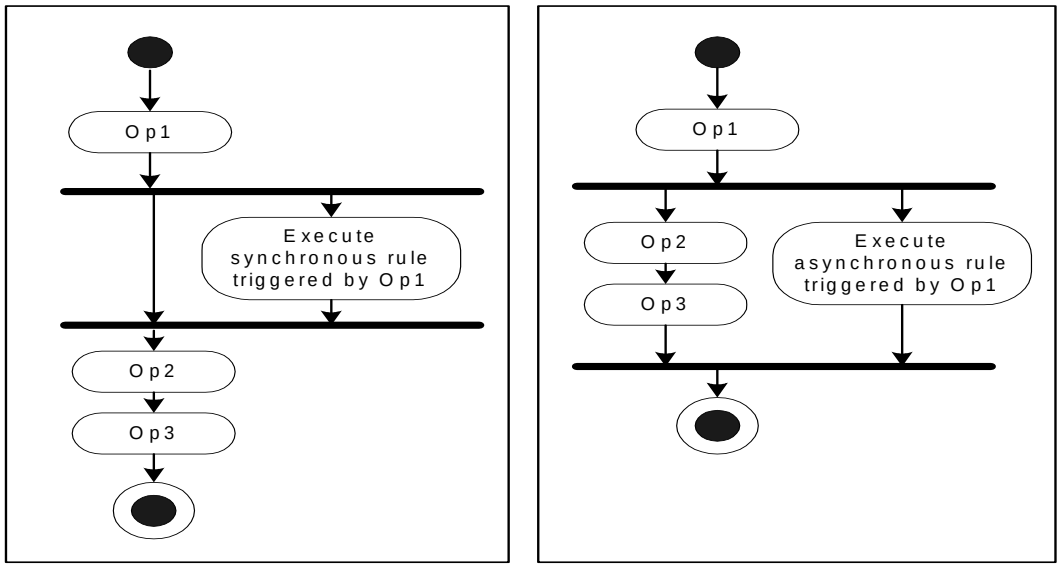
4.1 Integration Rule Coupling Modes

The execution model of an active rule system specifies how to coordinate a set of rules at runtime. The execution model is characterized by several features, such as coupling modes, transition granularity, net-effect policy, cycle policy, priority, and scheduling [7]. IRules follows the definition of the execution model features that are defined in [7]. In this paper, we address the coupling mode feature, since the use of coupling modes in a distributed environment is the primary focus of the IRules execution model.

The coupling modes of an active rule allow rule definers to specify how to execute the rule at run time. A coupling mode can be specified between the event and condition (E-C), between the condition and action (C-A), or between the event and action (E-A). Integration rules support four types of coupling modes: *immediate synchronous*, *immediate asynchronous*, *deferred*, and *decoupled*.

Using the E-C coupling mode as an example, the immediate synchronous E-C coupling mode indicates that the condition of a rule must be evaluated immediately after the event is raised. The immediate asynchronous mode is a new coupling mode that has been defined as part of this research. In an immediate asynchronous E-C coupling mode, the condition is evaluated immediately after the occurrence of an event, but the triggering transaction that raised the event will not be suspended. The execution of the integration rule and the triggering transaction are therefore concurrent.

Figure 7 illustrates the difference between the immediate synchronous and immediate asynchronous modes using a UML activity diagram [9]. In each box of Figure 7, the pair of synchronization bars (heavy black bars) represents the logic to fork and join processes, where the first bar is a fork and the second bar is a join. In Figure 7a, Op_1 is an event that triggers an immediate synchronous rule, so a subtransaction is started to process the rule. The triggering transaction suspends until the rule completes. After the rule joins the triggering transaction, Op_2 and Op_3 can be executed. In contrast, as shown in Figure 7b, Op_1 is an



a. Execution of Immediate Synchronous Rules

b. Execution of Immediate Asynchronous Rules

Figure 7: Synchronous vs. Asynchronous Rule Processing

event that triggers an immediate asynchronous rule. A new subtransaction is started to execute the asynchronous rule, but the triggering transaction does not suspend. As a result, Op_2 and Op_3 are concurrently executed with the asynchronous rules. At the end of the triggering transaction, the asynchronous rule joins the triggering transaction. The use of the immediate asynchronous mode can only be used when the rest the operations of the triggering transaction do not depend on the results of the immediate rule.

The deferred E-C coupling mode postpones rule condition evaluation to the end of the top-level transaction of execution (i.e., the outermost transaction within which the event was raised), based on the use of the deferred coupling mode as defined in [378]. The decoupled mode is only available for E-A and C-A coupling. Using the decoupled C-A coupling mode as an example, the decoupled action of a rule is executed immediately in a new top-level transaction, concurrent with the transaction that triggered the rule.

In addition to coupling modes, an integration rule can be triggered before an event happens or after an event happens. This feature is specified as before and after modifiers in the definition of a rule. Rule execution before an event is reasonable only when an event generator can trap the occurrence of the operation associated with the event.

4.2 Transaction Model of the IRules Environment

In an active system, the execution model relies heavily on the notion of transactions. For example, coupling modes are used to specify the transactional relationships between different parts of an active rule. Rules are also required to execute within appropriate transaction contexts for correct processing logic.

A fundamental issue with respect to transaction processing within IRules is the selection of a transaction model that is appropriate for the nested execution of rules over EJB components. In the nested transaction model [39], a subtransaction cannot release its results until its parent transaction commits. In contrast, the flexible transaction model [40] has a compensating mechanism that allows early commit of subtransactions. The flexible model avoids unnecessary blocking of subtransactions. The compensating mechanism ensures atomicity of a transaction when allowing unilateral commit of subtransactions. Although the flexible transaction model avoids unnecessary waiting time, the flexible transaction model can be more time-consuming than the nested transaction model when compensating work is required in the case of transaction failure.

The underlying component model also constrains the selection of a suitable transaction model. In transaction control for traditional databases, the release of locks and the update of permanent storage can be fully controlled by the transaction manager of the database. It is not possible to control black-box EJB components in such a manner. Each entity bean has an underlying relation in a relational database, and each instance of the bean corresponds to a tuple in that relation. Entity beans must be accessed with a *container-managed transaction*. That is, the transaction for method invocation of an entity bean is totally controlled by the EJB container.

As a more detailed explanation, each container uses ejbload() to refresh an entity bean's state from the database and ejbstore() to save the entity bean's state in the database. Before a method defined in the EJB remote interface is invoked from outside of the EJB component, the container will call ejbload(). After the method finishes execution, the container will call

ejbstore(). If the IRules transaction manager attempts to use the Two Phase Commit (2PC) protocol [41], the permanent storage can only be updated when all subtransactions are ready to commit. However, with no notion of the parent-child hierarchy of the outer transaction semantics for 2PC, the container is independently determining when to retrieve and update the database. In contrast, the flexible transaction model is more suitable for integration rules since it allows unilateral commit of subtransactions. This research has developed techniques for the use of the flexible transaction model to support the nested execution of integration rules. In the scope of this research, we assume a failure semantics where individual rules might abort without affecting the triggering transaction. The design of the compensating mechanism of the IRules system is a research issue that is currently under investigation.

In the IRules environment, transaction entities are execution objects that encapsulate the transactional control for rules and application transactions. All transaction classes inherit from an abstract class IRulesTransaction. The superclass IRulesTransaction encapsulates the generic logic of transaction execution. There are four sub-classes of IRulesTransaction: ISLTransaction, TopLevelTransactionForEvent, TopLevelTransactionForMethod, and NestedTransaction. These four sub-classes are responsible for capturing the execution-time behavior of transaction processing under different circumstances within the IRules system. ISLTransaction implements the transactional behavior of an application transaction. The TopLevelTransactionForEvent is used to offer transactional context to handle responses to internal and external events. The TopLevelTransactionForMethod applies to the specific case of a decoupled coupling mode when the action of the rule invokes a method of an EJB component. The NestedTransaction class encapsulates the execution of a subtransaction created as the context of a nested rule. Since the processing of rules is wrapped by transactions, rule nesting behavior can be controlled by the execution of parent and child transactions. For example, suppose rule R_x triggers immediate synchronous rule R_y, and the transaction contexts of R_x and R_y are T_x and T_y, respectively. Since we want to let R_x suspend until R_y finishes (according to the immediate synchronous E-C coupling mode), we control this behavior by suspending T_x until T_y commits.

4.3 Transactional Support for Wrappers and Rule/Method Synchronization

In the immediate synchronous coupling mode, the triggering transaction suspends while the triggered rule executes as a subtransaction, so this coupling mode results in rule nesting of rules. In this research, the suspension of transactions in a distributed environment is supported by the design of the IRules wrapper.

Figure 8 shows the structure of an IRules wrapper. A black-box bean may have a specific method, such as $m_1(param_1,param_2)$. IRules builds a property bean for each black-box bean to store external relationships between distributed components, as well as extents, derived attributes, and stored attributes for each component. A property bean has methods to provide the above functionality. A detail description of the property wrapper can be found in [10].

There is a proxy bean in the IRules wrapper structure that interfaces with clients. A proxy bean is responsible for generating method events, passing transaction contexts, and handling the suspension of current execution. A proxy bean has the same methods as the black-box bean, as well as a corresponding method for every method provided by the property bean and the black-box bean. For example, as shown in Figure 8, the proxy bean has a method $m_1(param_1,param_2)$ that is the same as the black-box bean. So any client that is unaware of the IRules system can still use the black-box API to access the purchased component. The proxy bean also has the $m_1(param_1,param_2,transactionId)$ method, which has the same name as the corresponding method in the black-box bean. In addition to the method parameters of the black-box bean, every method in the proxy bean has a parameter named transactionId. The transactionId parameter is used to pass transaction contexts during execution time in the IRules system. Similarly, the proxy bean has the $m_2(para_1,transactionId)$ method responding to $m_2(para_1)$ of the property bean to pass transaction contexts. When there is method invocation from clients to a proxy bean, the proxy bean will delegate the invocation to the corresponding method of the property bean or the black-box bean.

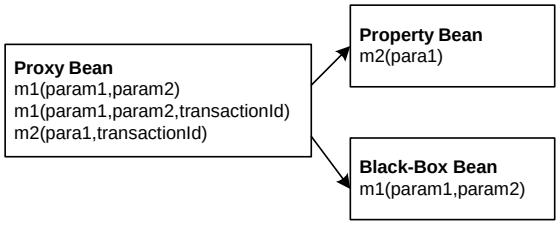


Figure 8: IRrules Wrapper Structure for EJB Components

A proxy bean has a three-step logic for event generation. The first step is to generate before method events, if any such events exist. When there is a method call to the proxy bean, the proxy bean will contact the metadata manager to determine whether this method call can generate a before method event. If it can, the proxy bean generates a method event through the Java Message Service (JMS) [42]. Then the proxy bean will try to read a semaphore object from the synchronization space in JavaSpaces [43]. JavaSpaces is a Jini Service that supports the storage and retrieval of objects in a distributed environment.

The blocking call mechanism of JavaSpaces is used to synchronize the execution of a transaction and its triggered immediate rules, releasing the suspension of the transaction upon the completion of the rule processing. Initially, the semaphore object does not exist. Since the read operation to JavaSpaces is blocking, the current transaction suspends at the proxy bean. During this suspension period, the event is propagated to the rule manager and the rule manager begins to process any rules triggered by the event. After processing rules triggered by a before method event, the rule manager will put the semaphore object into the synchronization space. Once the semaphore object is in the synchronization space, the proxy bean can successfully read the object to release the suspension of the wrapper.

The second step of the logic of the proxy bean is to call the property layer or the black-box bean to execute the method call. The third step is to generate after method events so that rules triggered after the execution of the method can be executed. The logic of generating an after event and the suspension for immediate synchronous rules is the same as in the first step. A more detailed description of the synchronization algorithm appears in [10].

5 Integration Rule Processing Algorithm

The rule processing algorithm is the logical circuit through which integration rules are processed. The algorithm instructs the rule manager in the processing of rules at execution time, depending on the transactional framework described in Section 4 for interaction with EJB components and coordination of rule execution with method execution. In Section 5.1, we specify the behavior of integration rule coupling modes in the context of cycles and levels of rule execution. In Section 5.2, we present the logic of the rule processing algorithm. A specific example of rule execution in the IRules environment is presented in Section 5.3. A brief summary of a performance analysis of the IRules environment appears in Section 5.4.

5.1 Specification of Coupling Mode Behavior

The Integration Rule Processing (IRP) algorithm is based on the algorithm of the ADOOD RANCH project [38], using *cycles* to control the nested execution of active rules. This research has re-designed the rule execution algorithm for a distributed environment, fully supporting the IRules coupling modes and transaction processing model.

Within the IRules environment, integration rules are processed according to coupling modes. A rule with an immediate E-C mode (either immediate synchronous mode or immediate asynchronous mode) is scheduled to execute as soon as it is triggered, while a rule with a deferred E-C mode is added to the deferred rule list that will be scheduled to execute at the commit time of the top-level transaction. A decoupled rule is executed immediately in a new top-level transaction, while the transaction of the triggering event execution resumes.

As shown in Figure 9, rule execution occurs in a coordinate system in two dimensions: the *Cycle* dimension and *Level* dimension. *Cycle* represents the logic of deferred rule processing, while *Level* represents the logic of nested rule execution. In Figure 9, dashed arrows represent immediate rule triggering in *Levels*, while solid arrows represent deferred rule scheduling in *Cycles*.

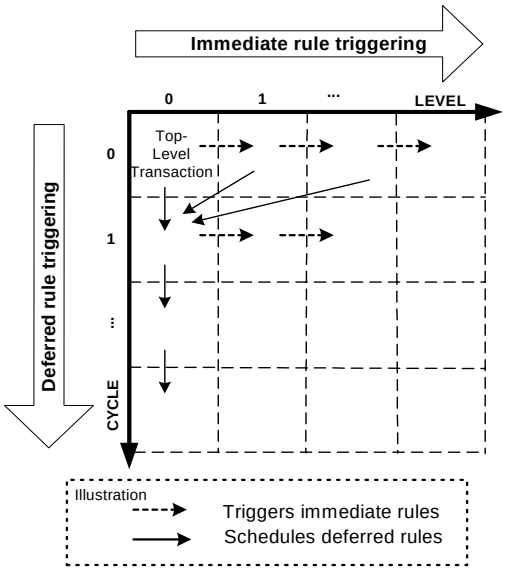


Figure 9: Cycles and Levels of Rule Execution

Each top-level transaction and its subtransactions are represented as a coordinate system, formed as a Cartesian product of *Cycle* and *Level*. A top-level transaction t^k, as the root of a transaction, is executed at $Cycle^k_0$ and $Level^k_0$ in coordinate system G^k. If an event in ($Cycle^k_0$, $Level^k_j$) triggers an immediate rule, the rule will be executed in the same cycle but in $Level_{j+1}$ as a new subtransaction. As an example, if an event e_1 in ($Cycle^k_0$, $Level^k_0$) triggers immediate rules r_1, r_2, and r_3, then r_1, r_2, and r_3 execute at ($Cycle^k_0$, $Level^k_1$). Within the same level, rules r_1, r_2, r_3 can execute sequentially or concurrently. The algorithm for determining sequential or concurrent rule execution is presented in [44].

If an operation of a top-level transaction ($Cycle^k_0$, $Level^k_0$) triggers a deferred rule, the rule will be scheduled to execute at the end of the top-level transaction in ($Cycle^k_1$, $Level^k_0$). In general, if an event in $Cycle^k_i$ triggers a deferred rule, the rule will be executed in $Cycle^k_{i+1}$. If there is more than one deferred rule at the end of the transaction, the rules are executed in sequence. If an event in $Level^k_j$ triggers a deferred rule, the rule will always be executed in $Level^k_0$ of the next cycle.

When an event in any ($Cycle^k_i$, $Level^k_j$) of a coordinate system G^k triggers a rule that contains a

decoupled action, the decoupled action is executed as a new top-level transaction at ($Cycle^n_0$, $Level^n_0$) of a new coordinate system G^n, where G^n and G^k are two distinct coordinate systems for rule execution.

Additional execution procedures apply for the execution of immediate asynchronous rules. If an event in ($Cycle^k_i$, $Level^k_j$) triggers a rule with an immediate asynchronous E-C mode, the rule will be executed in $Level^k_{j+1}$ without suspending the execution of the operations in $Level^k_j$. A *synchronization point* that requires the commit of asynchronous rule execution exists at this point. The *synchronization point* is after the last operation of $Level^k_j$ and before the commitment of a transaction in $Level^k_j$, which is called the *end-Proc* stage. This point allows the maximum time interval for the execution of the asynchronous rules without delaying the processing of the triggering transaction. In the *end-Proc* state of $Level^k_j$ within $Cycle^k_i$, all of the immediate asynchronous rules that executed at ($Cycle^k_i$, $Level^k_{j+1}$) are required to commit for the triggering transaction to continue. At the end of a top-level transaction, after all of the asynchronous rules commit, deferred rules can be processed.

The above presentation of IRP describes rule execution in a two dimensional coordinate system, focusing on the logic of rule execution. At run time, rules are executed in a distributed environment, which is related to a third dimension – *Location* of the objects accessed by a rule. An event is generated from one location, while the data accessed by a triggered rule can exist in multiple locations. The value of the *Location* depends on which software components are involved in the condition and action part of the rule. The IRP algorithm instructs the rule manager to invoke the IRules object manager to locate the position of a component. In a more complicated case, the objects accessed by a rule can require the use of multiple locations for evaluating the condition and performing the action of the rule.

5.2 Execution Logic of the IRP Algorithm

The IRP algorithm is the core of the rule manager. IRP instructs and regulates the execution behavior of the rule manager for the processing of application transactions and integration rules. In this section, the logic of the IRP algorithm is presented using Unified Modelling Language (UML) activity diagrams [9].

When a user makes a request to the rule manager to process an application transaction, the rule manager will start an application transaction processor to process the request. The processing logic is illustrated in Figure 10 for the PROCESS TOP-LEVEL TRANSACTION module. There are two sub-component modules of the processing: EXECUTE AN APPLICATION TRANSACTION and PROCESS DEFERRED RULES, which are detailed in Figures 11 and 13, respectively. As shown in Figure 10, after EXECUTE AN APPLICATION TRANSACTION, the execution arrives at the pre-commit state, which is the time for deferred rule processing. Then PROCESS DEFERRED RULES is executed and the transaction commits.

The EXECUTE AN APPLICATION TRANSACTION module is presented in Figure 11. Before execution of the application transaction, the algorithm checks for the existence of a before application transaction event. If a before event is raised, rules triggered by the before event are processed. Next, the algorithm will execute the application transaction. Since an application transaction consists of a set of operations, the algorithm calls the EXECUTE OPERATIONS module to execute all the operations of the application transaction. After all operations of the application transaction have been executed, the algorithm checks for an after application transaction event. If an after event exists, rules triggered by the event are executed according to different coupling modes. The EXECUTE AN APPLICATION TRANSACTION module uses the same algorithm as the EXECUTE OPERATIONS module (Figure 12) with respect to rule processing according to different coupling modes. The following paragraph provides an explanation of rule processing for different coupling modes in the context of the EXECUTE OPERATIONS module.

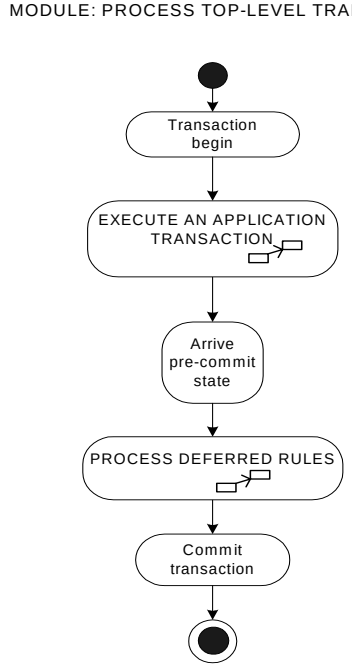


Figure 10: Process Top-Level Transaction

In Figure 12, the EXECUTE OPERATIONS module presents the logic of executing a sequence of operations for a transaction by iterating through all operations. Before execution of any operation, the algorithm will check for the existence of a before method event. If a before event has been raised, all rules triggered by this event will be obtained. The current transaction is suspended until the completion

of all triggered rules. The "*[For each rule]" notation indicates that the PROCESS A RULE module will be started for each rule. The PROCESS A RULE module is illustrated in Figure 14. These rules can be executed sequentially or concurrently.

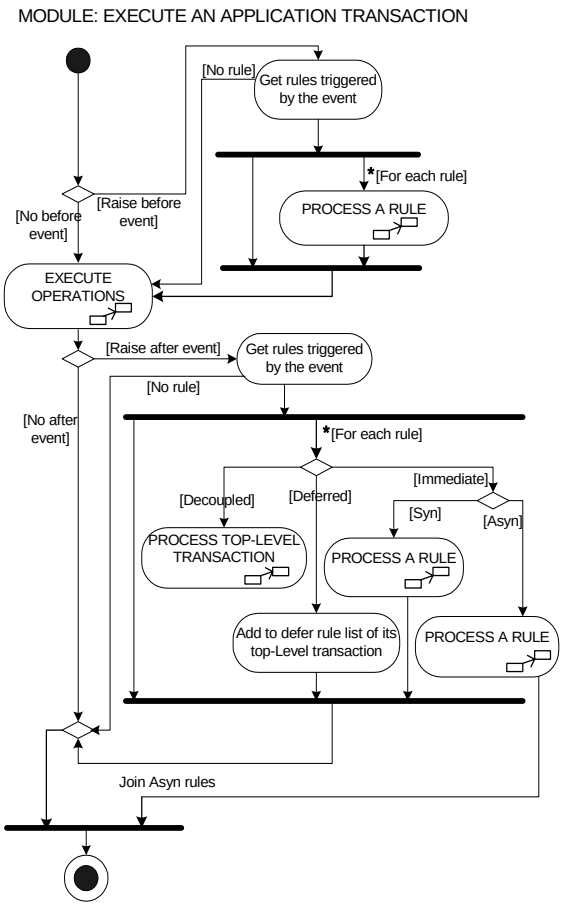


Figure 11: Execute An Application Transaction

Recall that only immediate synchronous rules are allowed for before events, so those rules will be processed immediately. After the rules with a before modifier are processed, the operation is executed as shown in Figure 12. The algorithm in Figure 12 checks for method events raised after the execution of the operation. Rules triggered by the after method event are obtained. If a rule is decoupled, a new top-level transaction is started that follows the logic of the EXECUTE TOP-LEVEL TRANSACTION module. If a rule is deferred, the rule is added to the deferred list of its top-level transaction. If the rule is immediate, the rule will be started immediately as a subtransaction by invoking the PROCESS A RULE module. In the case of an immediate synchronous rule, the current transaction cannot continue until the subtransaction joins the current transaction. In contrast, the current transaction can continue execution in parallel with the execution of its immediate asynchronous rules. At the *end-proc* state, the current transaction will suspend until all immediate asynchronous rules finish execution and join the current transaction.

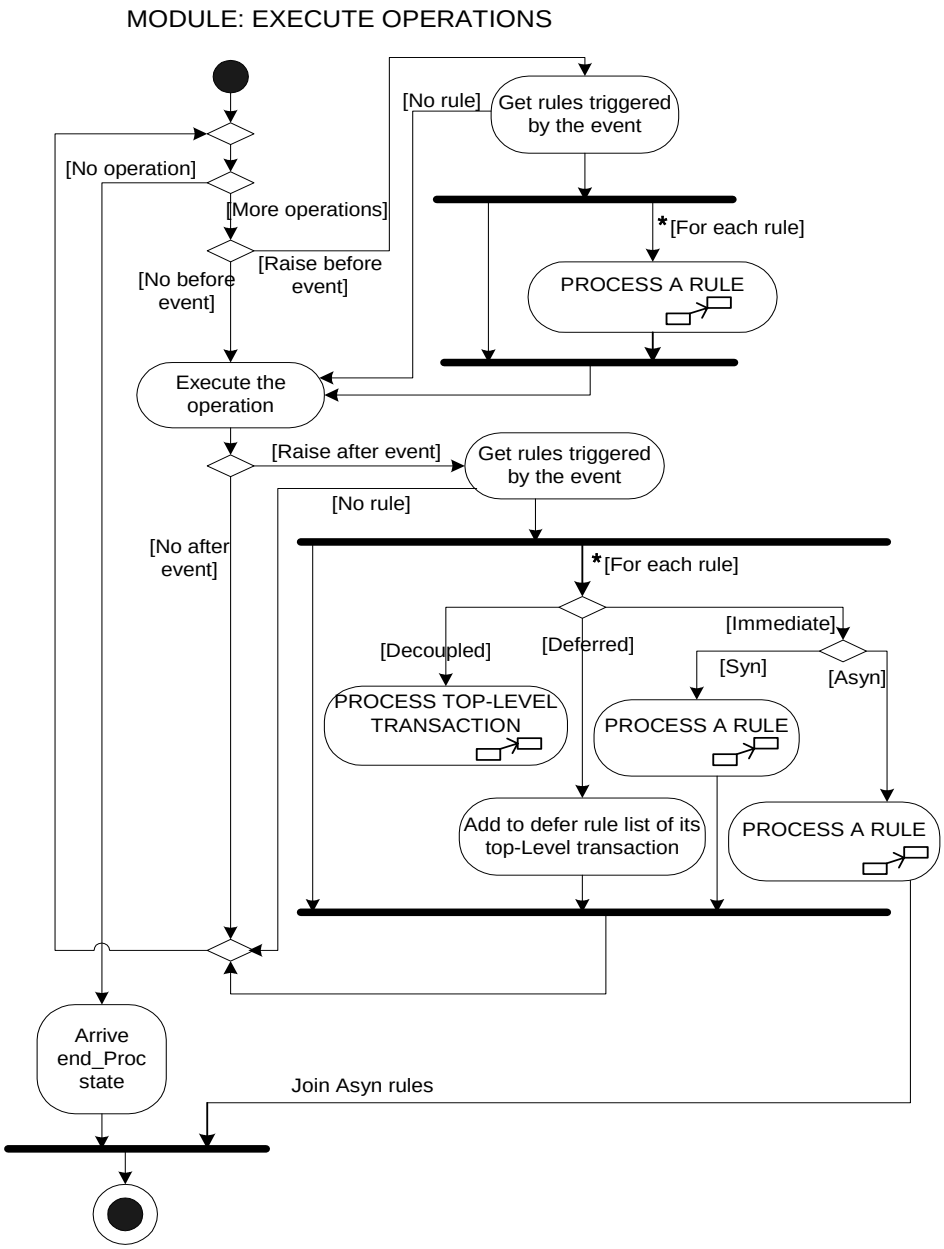


Figure 12: Execute Operations

The PROCESS DEFERRED RULES module is presented in Figure 13. Recall from Section 5.1 that deferred rules are executed in cycles. For each cycle, deferred rules are executed in sequence as described in the PROCESS A RULE module of Figure 14. A subtransaction will be created as the child of the triggering transaction. For an EA rule, the action is executed immediately. The execution of an action occurs in the EXECUTE ACTION module. For an ECA rule, the condition is evaluated first. If the C-A coupling mode is immediate synchronous, the action will be executed immediately. In the case of a decoupled C-A mode, a new top-level transaction is started immediately.

Recall that before calling the PROCESS A RULE module, the rule was already scheduled according to the E-C coupling mode for an ECA rule and the E-A coupling mode for an EA rule. So in the PROCESS A RULE module, the condition of an ECA rule and the action of an EA rule are always executed immediately.

Figure 15 illustrates the logic of the EXECUTE ACTION module. The execution of an action invokes the EXECUTE OPERATIONS module when the action is in the format of a method call. If the action is in the format of an application transaction, the algorithm will call the EXECUTE AN APPLICATION TRANSACTION module.

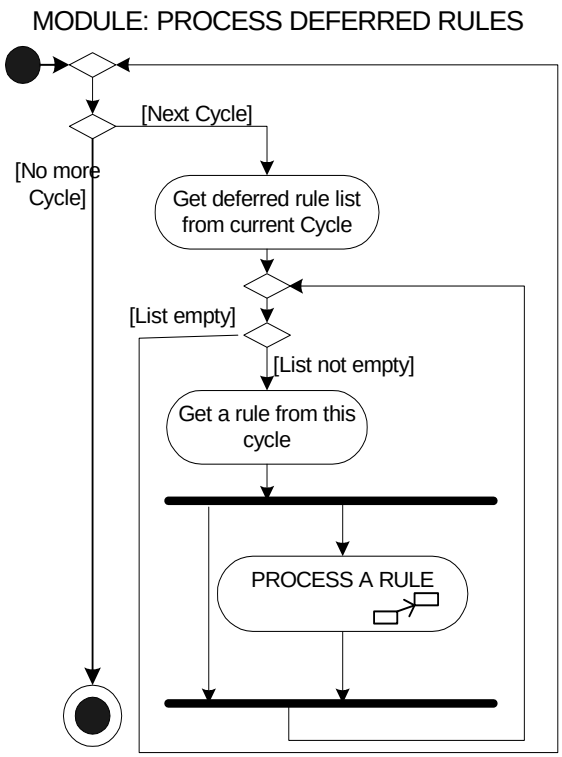


Figure 13: Process Deferred Rules

The algorithm has so far illustrated the rule processing logic for user requests as a top-level transaction. Recall that the other architectural component that can cause the rule manager to start top-level transaction processing is the event handler. For any internal or external event pushed by the event handler, the rule manager will handle the event according to the logic in Figure 16. The rule manager gets rules triggered by the event, and then processes each rule according to different coupling modes. Recall that no rule with a before modifier can be raised by an internal or an external event, because it is impossible for an active system to control when an internal or an external event occurs. Similar to the processing of a top-level application transaction, once the processing arrives at the *end-proc* stage, asynchronous rules must join the triggering transaction. After the *pre-commit* state, all deferred rules are processed.

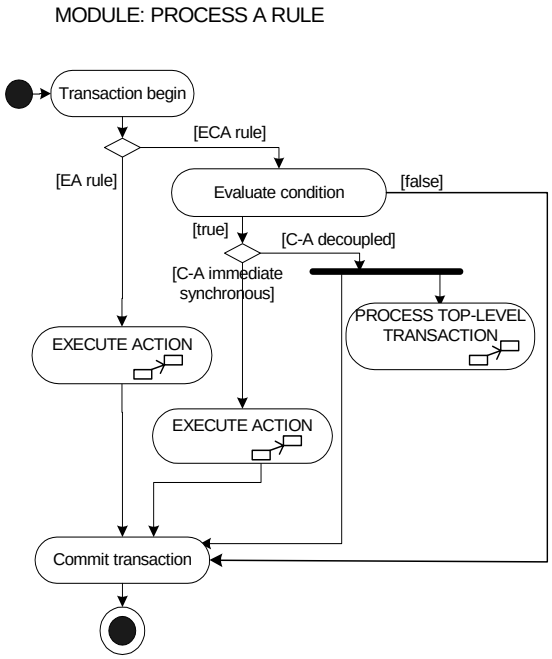


Figure 14: Process A Rule

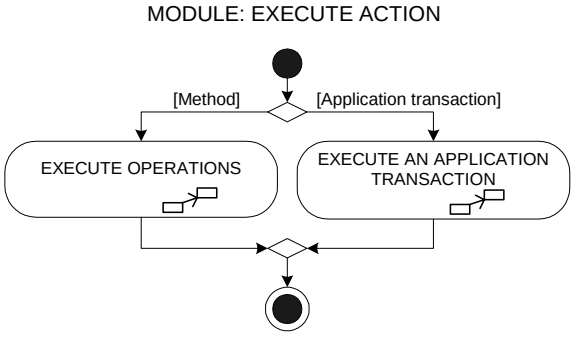


Figure 15: Execute Action

5.3 Execution Scenario of an Investment Application

To illustrate the rule processing algorithm, this section presents an execution scenario for selling stocks using the investment example presented in Section 3. A preliminary version of the scenario from Figures 17-19 appears in [6] without the notion of cycles and levels.

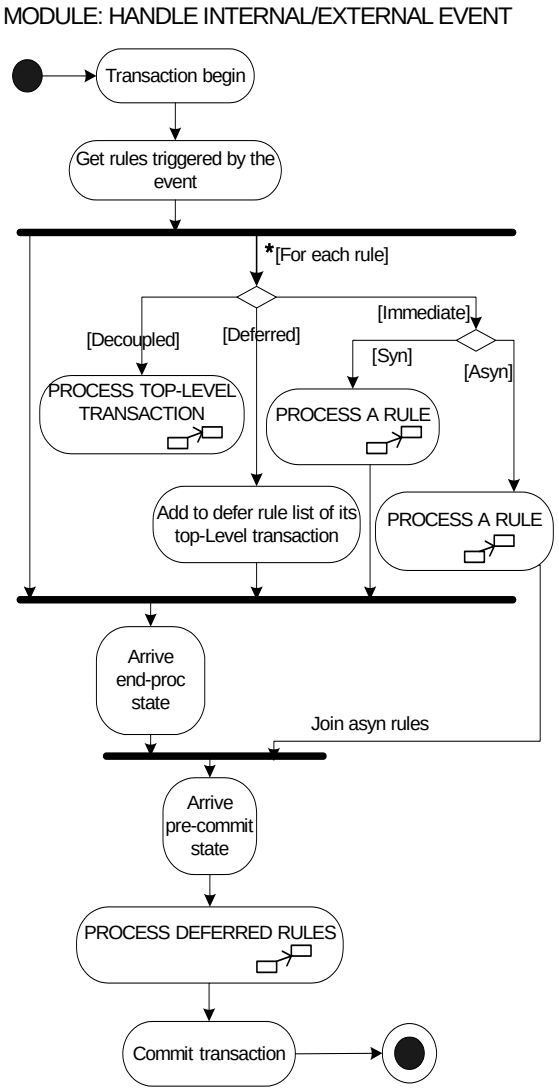


Figure 16: Handle internal/external event

As shown in Figure 17, the application transaction clientWantsToSellStock is the request from a user to perform the function of placing an order to sell a stock. The transaction creates an order to sell a stock at a desired price, and then prints a report. An event is generated after this application transaction, triggering the integration rule clientWantsToSellStockRule in Figure 18. The integration rule examines the desired prices for the stock to be sold and compares it with the current price, selling the stock if the condition is satisfied. The action part of the rule executes the sellStockOnNewPO application transaction in Figure 17, which raises two events. The first event (afterSellStock) is generated after the stock is sold but before the end of the transaction, allowing rules to be triggered in reaction to each sell operation. In particular, the scenario has an active rule stockBuyOnUpdateCash that allows a portfolio to exercise pending purchases when sufficient funds are available in the user's account. The afterSellStockOnNewPO event is signaled after the action of sellStockOnNewPO is complete to trigger a rule billingToAccountOnSell. This rule sends billing information to the user. Both of the stockBuyOnUpdateCash and billingToAccountOnSell rules are shown in Figure 18.

```
application transaction clientWantsToSellStock(String pnId,
String portfolioId, String stockId, int numOfShares,float
desiredPrice, String action, Stock actUpon, Portfolio
orderedBy)
tcl newInstance
{
set pn [newInstance PendingOrder $pnId $portfolioId
$stockId $numOfShares $desiredPrice $action $actUpon
$orderedBy $irulesId];
printPendingOrderInfo $pn $irulesId;
}

application transaction sellStockOnNewPO(String stockId,
float price, String portfolioId, int numOfShares,
StockBroker.PendingOrderComponent.PendingOrder pn)
tcl printSellInfo
{
set session [newInstance PortfolioSessionBean $irulesId];
$session sellStock $stockId $price $portfolioId
$numOfShares $irulesId;
$pn setStatus "executed" $irulesId;
printSellInfo $pn $irulesId;
}
```

Figure 17: Examples of Application Transactions for the Investment Scenario

Figure 19 illustrates the execution scenario that occurs as a result of the IRP algorithm. Figure 19 uses a notation that is based on UML activity diagrams. There are four transactions represented by four different swimlanes [45], one for each transaction context of the application transactions and rules. We use notation such as T1, e1, R11, as the abbreviated names of transactions, events, and rules, respectively.

When a user invokes the clientWantsToSellStock application transaction, the transaction manager creates a top-level transaction (T1) to process the application transaction. The top-level transaction T1 executes at ($Cycle^1_0$, $Level^1_0$) in coordinate system G^1. The clientWantsToSellStock application transaction generates an event named afterClientWantsToSellStock (e1). The event e1 triggers the rule clientWantsToSellStockRule (R11) presented in the second column of Figure 19. Because the E-C coupling mode of R11 is immediate synchronous, the condition of R11 is evaluated immediately. T1 suspends until R11 completes. The execution of immediate rule R11 is within the context of subtransaction T11. Because R11 is an immediate rule triggered by an event at ($Cycle^1_0$, $Level^1_0$) , the rule is executed at ($Cycle^1_0$, $Level^1_1$).

```
create rule clientWantsToSellStockRule
event     afterClientWantsToSellStock(pnId,
          portId, stockId, numOfShares,
```

```
            desPrice, pnaction, actUpon,
            orderedBy)
condition immediate
            when pnaction = "sell"
            define stockAndPendingOrder as
            select struct ( stk: s, newPo: pn )
            from s in stocks, pn in pendingOrders
            where pn.id=pnId and pn.actUpon=s
            and desPrice<=s.price
action      immediate
            from sp in stockAndPendingOrder
            do sellStockOnNewPO(stockId,
            sp.stk.price, portId, numOfShares,
            sp.newPo)

create rule stockBuyOnUpdateCash
event       afterSellStock(stockId, price, portfolioId,
            numOfShares)
condition asynchronous
            define portfolioOnUpdate as
            select p
            from p in portfolios
            where p.portfolioId=portfolioId and p.cash >
            p.buyThreshold
action      decoupled
            from p in portfolioOnUpdate
            do buyStockOnUpdateCash(p)

create rule billingToAccountOnSell
event       afterSellStockOnNewPO(stockId, price,
            portfolioId, numOfShares, pn)
action      deferred
            from p in portfolios
            where p.portfolioId= portfolioId
            do setAccountBillingOnSell(stockId, price,
            portfolioId, numOfShares, p.accountId)
```

Figure 18: Examples of Integration Rules for the Investment Scenario

The condition of R11 is evaluated in the second column. If the condition evaluation returns a non-null structure containing stocks and pending orders, then the action of R11 is performed using the structure as input bindings. Because of the immediate synchronous C-A coupling mode, the action is executed immediately. The action part of R11 is wrapped in a subtransaction named sellStockOnNewPO (T11a), which has four operations. The second operation sellStock is a method that generates a method event afterSellStock (e2). The event e2 triggers the rule stockBuyOnUpdateCash (R111) in the third column. Since the E-C coupling mode of R111 is immediate asynchronous, the condition of R111 is evaluated immediately. Moreover, the condition evaluation of R111 is concurrent with the execution of the triggering transaction T11a. Because R111 is an immediate rule triggered by an event at ($Cycle^1_0$, $Level^1_1$), R111 is executed at ($Cycle^1_0$, $Level^1_2$).

If the condition evaluation of R111 returns a non-null set of portfolios, the action of R111 will be performed upon the set. Due to the decoupled C-A coupling mode, the action of R111 becomes a new top-level transaction named buyStockOnUpdateCash (T2) since the action is an application transaction. Since the C-A coupling mode is decoupled, the action part of R111 will be executed in a different coordinate execution system G^2 at ($Cycle^2_0$, $Level^2_0$).

Once T2 is started in the fourth column, T111 resumes and commits. As shown in the second column, when the set status and printInfo operations of T11a finish executing, T11a is at the end of execution. At this time T11a waits until all the triggered asynchronous rules join. In this example, T111 joins T11.

As shown in the second column, the completion of sellStockOnNewPO generates an event (e3) that triggers an EA Rule named billingToAccountOnSell (R112). Because the E-A coupling mode of R112 is deferred, R112 is scheduled to the end of the top-level transaction (T1). Subtransaction T11 finishes execution and commits. Because R112 is a deferred rule triggered by an event at $Cycle^1_0$, R112 will be executed at ($Cycle^1_1$, $Level^1_0$).

In the first column, the commit of T11 releases the suspension of T1. Just before T1 commits, deferred rule R112 is processed. After R112 finishes executing, T1 commits.

5.4 Performance Analysis of the IRules Environment

The IRules system is a Java implementation that uses the BEA Weblogic Server [46] to provide EJB components. The Jini distributed computing environment is used as the backbone of the system, with IRules architectural components implemented as Jini Services. Java Message Service (JMS) [42] provides asynchronous event notification for communication between the event-signaling components and the event-handling components. JavaSpaces [43] is used for the storage of metadata. The blocking call mechanism of JavaSpaces is also used in the synchronization space to synchronize the execution of a transaction and its triggered immediate rules, releasing the suspension of the transaction upon the completion of the rule processing [11].

We have evaluated the performance of the IRules environment using the OBJECTIVE benchmark [47] as the basis for the evaluation. The OBJECTIVE Benchmark was originally designed to identify bottlenecks and to evaluate the functionality of an active object-oriented database. The OBJECTIVE benchmark was adjusted and extended as part of this research to apply the benchmark to a distributed component integration environment. The full details of the performance analysis and how the benchmark was adapted to the IRules environment is beyond the scope

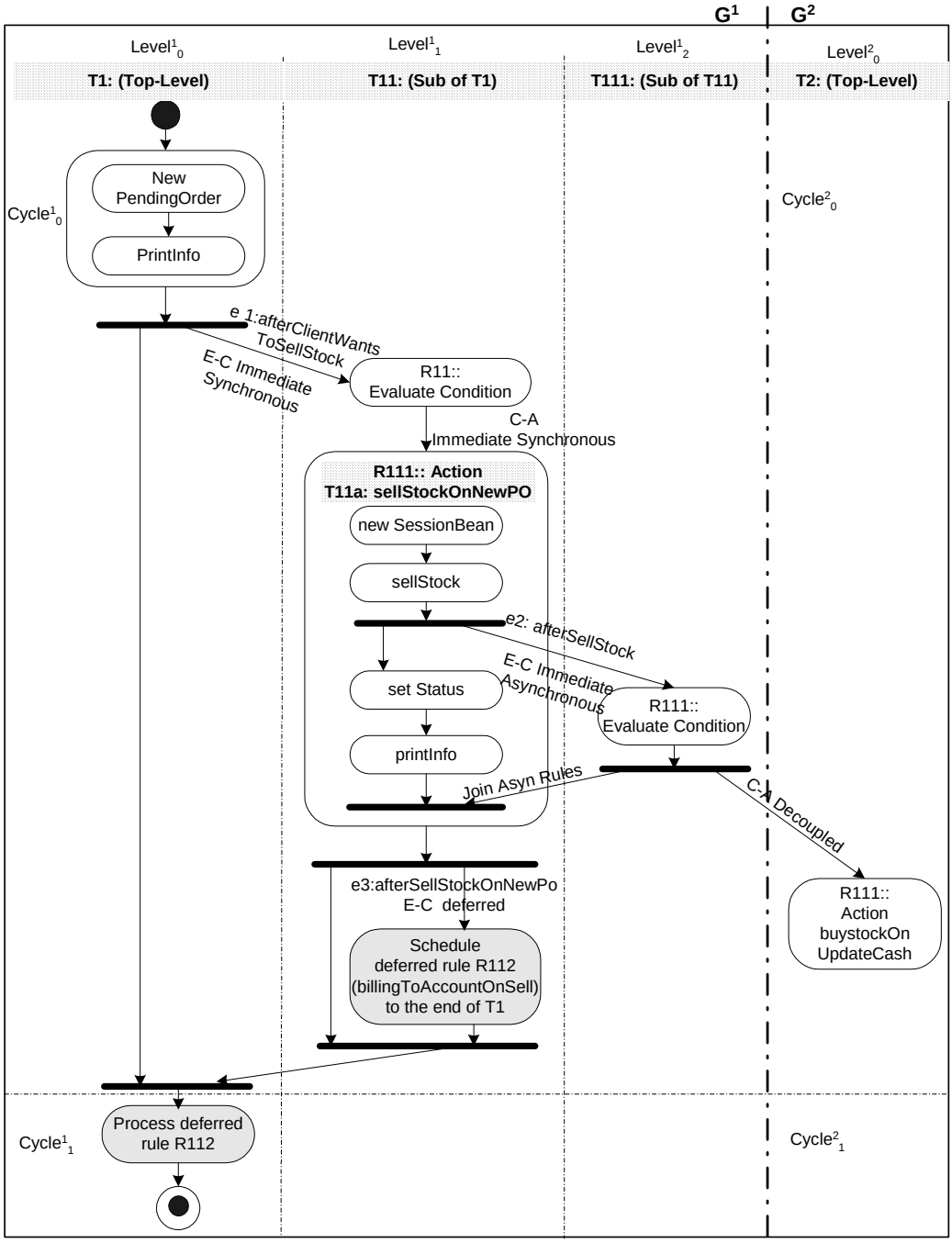


Figure 19: Execution Scenario of the Investment Application

of this paper and can be found in [44, 48]. The evaluation was conducted within the IRules environment and has not been applied to any industrial environment.

As a brief summary of the performance evaluation process, the system was implemented and evaluated using four Windows NT 4.0 computers. The metadata and object manager illustrated in Figure 6 were co-located on one physical machine, while in two different Java virtual machines. The rule manager, event handler, and the Weblogic EJB server, also illustrated in Figure 6, were each physically located on one machines. The evaluation was conducted using four parameters of configuration: 1) the number of events, 2) the number of rules, 3) the number of application transactions, and 4) the number of component instances. The primary focus of the evaluation was on four different aspects of the execution environment: 1) three phases of active behavior (event detection, rule retrieval, and rule execution), 2) performance of different coupling modes, 3) performance of the rule processor under a heavy event load, and 4) the time for event detection

for the different types of IRules events. The primary results of the evaluation indicate that:

1) The decoupled and immediate asynchronous modes provide the best performance since they allow concurrent execution. The deferred mode is the slowest due to the need to schedule rules for execution at the end of the top-level transaction.
2) Execution time is somewhat affected by a large number of rules and transactions due to the larger amount of metadata that must be searched during rule and transaction retrieval.
3) A heavy event load can cause the rule processor to be interrupted to queue events, thus slowing down the performance of the rule processor, but the performance eventually levels off to a consistent execution speed regardless of the event arrival load.
4) Access to EJB components is the primary point of slow performance, affecting the time for method execution as well as the time for method event generation. A future improvement of the IRules system should involve re-design of the wrapper structure to reduce the EJB component layers, thus reducing the time associated with method invocation.

6 Summary and Future Directions

This paper has presented the integration rule processing algorithm of the IRules environment, with supporting descriptions of the rule execution model and transaction model. The IRP algorithm illustrates an approach for active rule processing in the context of distributed component integration, where events are used to trigger rules that invoke application transactions and methods on components. The IRP algorithm is presented in a form that can be reused in other environments for a rule-based approach to integration logic, defining the manner in which immediate coupling modes can be used together with nested rule execution in a distributed environment. The IRules integration system allows application integrators to specify the integration logic in a declarative fashion, which does not require an integrator's low-level knowledge of programming and transaction management. Integrators can focus on mediating the interaction between components, rather than the technical details of event handling and transaction processing.

It has been a challenging effort to develop a distributed rule and transaction processing environment such as IRules, since it involves the combination of issues such as component autonomy, rule distribution, cascaded rule triggering, and distributed synchronization. The implementation of the execution environment presented in the paper has been completed. One future direction is to expand the environment to support multiple component models. The transaction model also needs further investigation to address failure in the execution process, especially when global transactions execute over different component models with heterogeneous transaction processing semantics. These future research directions will be explored in the context of Grid services for virtual organizations, where a Grid service provides a service-oriented view of a component [49, 50] and the Grid environment forms the foundation of the underlying architecture.

References

[1] Object Management Group: The Common Object Request Broker, Architecture and Specification. (1999) John Wiley Publishing.

[2] Enterprise Java Beans Specification (2000) Sun Microsystems, version 2.0.

[3] S. D. Urban, S. W. Dietrich, Y. Na, Y. Jin, and A. Sundermier (2001) The IRules Project: Using Active Rules for the Integration of Distributed Software Components, Proc. of the 9th IFIP 2.6 Working Conf. on Database Semantics: Semantic Issues in E-Commerce System, Hong Kong, April 2001, pp. 265-286.

[4] S. D. Urban, S. W. Dietrich, Y. Jin, S. Kambhampati, and Y. Na (2002) Distributed Software Component Integration: A Framework for a Rule-Based Approach, Handbook of Electronic Commerce in Business and Society, Watson, R., Lowery, P. and Cherrington, J. Ed.

[5] S. W. Dietrich, S. D. Urban, A. Sundermier, Y. Na, Y. Jin, and S. Kambhampati (2001) A Language and Framework for Supporting an Active Approach to Component-Based Software Integration, Informatica, Vol. 25, No. 4, pp. 443-454.

[6] Y. Jin, S. D. Urban, S. W. Dietrich, and A. Sundermier (2002) An Execution and Transaction Model for Active, Rule-Based Component Integration Middleware, Proceedings of the Engineering and Deployment of Cooperative System, Beijing, China, pp. 403-417.

[7] N. W. Paton, O. Diaz (1999) Active Database Systems, ACM Computing Surveys, Vol. 31, No. 1, pp. 3-27.

[8] J. Widom and S. Ceri (Eds.) (1996) *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publisher.

[9] Unified Modeling Language (UML) Specification, version 2.0. http://www.uml.org/#UML2.0

[10] R. Patil (2003) A Framework Supporting an Active Approach to Component-Based Software Integration, M.S. Thesis, Arizona State University, Department of Computer Science and Engineering.

[11] S. Urban, S. Kambhampati, S. Dietrich, Y. Jin, and A. Sundermier (2004) An Event Processing System for Rule-Based Component Integration, Proceedings of the International Conference on Enterprise Information Systems, Porto, Portugal, pp. 312-319.

[12] M. Stonebraker, E. N. Hanson, and S. Potamianos (1998) The POSTGRES Rule Manager, IEEE Transactions on Software Engineering, Vol. 14, No. 7, pp. 897-907.

[13] J. Widom (1992) The Starburst Rule System: Language Design, Implementation and Application, IEEE Data Engineering Bulletin, December, pp. 15-18.

[14] U. Dayal, B. Blaustein, A. Buchmann, and S. Chakravarthy (1998) The HiPAC Project: Combining Active Databases and Timing Constraints, *ACM SIGMOD Record*, Vol. 17, No. 1, pp. 51-70.

[15] S. Gatziu, K. R. Dittrich (1992) SAMOS: An Active Objective-Oriented Database System, Data Engineering bulletin, pp. 23-26.

[16] S. W. Dietrich, S. D. Urban, J. V. Harrison and A. Karadimce (1992) A DOOD RANCH at ASU: Integrating Active, Deductive and Object-Oriented Databases, IEEE Data Engineering Bulletin: Special Issue on Active Database Systems, Vol. 15, No. 1-4, pp. 40-43.

[17] H. Branding, A. P. Buchmann, T. Kudrass, and J. Zimmermann (1993) Rules in an Open System: The REACH Rule System, Rules in Database Systems, pp. 111-126.

[18] P. Gulutzan and T. Pelzer (1999) SQL-99 Complete Really, Miller Freeman Publishing.

[19] P. Fraternali and L. Tanca (1995) A Structured Approach for the Definition of the Semantics of Active Databases, ACM Transactions on Database Systems, Vol. 20, No. 4.

[20] L. B. Warshaw (2001) Facilitating Hard Active Database Applications, Ph.D. Dissertation, The University of Texas at Austin, Department of Computer Science.

[21] S. Chakravarthy, and R. Le (1998) ECA Rule Support for Distributed Heterogeneous Environments, International Conference on Data Engineering, pp. 601.

[22] Object Management Group (OMG) Interface Definition Language (IDL), International Organization for Standardization (ISO) International Standard, number 14750. http://www.omg.org/gettingstarted/omg_idl.htm

[23] A. Koschel, and P. C. Lockemann (1998) Distributed Events in Active Database Systems - Letting the Genie out of the Bottle, Journal of Data and Knowledge Engineering, Vol. 25, pp. 11-28

[24] H. Fritschi, S. Gatziu, K. and R. Dittrich (1998) FRAMBOISE – an Approach to Framework-Based Active Database Management System Construction, Proceedings of the 7th ACM International Conference on Information and Knowledge management, pp. 364-370.

[25] M. Cilia, C. Bornhovd, and A. Buchmann (2001) Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments, Proceedings of 9th International Conference on Cooperative Information Systems (CoopIS'01), Trento, Italy, pp. 195-210.

[26] C. Liebig, M. Malva, A. Buchmann (2000) Integrating Notfications and Transactions: Concepts and X2TS Prototype, Proceedings of the 2nd International Workshop on Engineering Distributed Objects, University of California, Davis, USA, pp.194-214.

[27] W. K. Edwards (2000) Core Jini, Prentice-Hall PTR, Second Edition.

[28] C. Bussler, and S. Jablonski (1994) Implementing Agent Coordination For Workflow Management Systems Using Active Database Systems, The International Workshop On Active Database Systems, Houston TX, pp. 53-59.

[29] F. Casati, S.Ceri, B. Pernici, and G. Pozzi (1996) Deriving Active Rules for Workflow Enactment, DEXA, Switzerland, pp. 94-115.

[30] K. Karlapalem and P. C. K. Hung (1998) Security Enforcement in Activity Management Systems, Workflow Management Systems and Interoperability, Ed. Dogac, etc., Springer-Verlag publisher, pp. 165-193.

[31] G. Kappel, and W. Retschitzegger (1998) The TriGS Active Object-Oriented Database System - An Overview," SIGMOD Record, 27(3), pp.36-41.

[32] Workflow on Intelligent Distributed database Environment. http://dis.sema.es/projects/WIDE/

[33] B. Benatallah, M. Dumas, Q. Sheng, and H. Ngu (2002) Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services, Proceedings of the 18th International. Conference on Data Engineering, San Jose, CA, pp. 297-308.

[34] R. Peri (2002) Compilation of the Integration Rule Language, M.C.S. Report, Arizona State University, Department of Computer Science and Engineering.

[35] K. Marimuthu (2003) An Object-Oriented Query Processor Based on an Extended Monoid Algebra, M.S. Thesis, Arizona State University, Department of Computer Science and Engineering.

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

[37] J. Ousterhout (1994) TCL and the TK Toolkit, Addison-Wesley Publishing.

[38] T. Abdellatif (1999) An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States, Ph.D. Dissertation, Arizona State University, Department of Computer Science and Engineering.

[39] J. Gray and A. Reuter (1994) Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers.

[40] S. Jajodia and L. Kerschberg (1997) Advanced Transaction Models and Architectures, Kluwer Academic Publishers.

[41] M. T. Ozsu and P. Valduriez (1999) Principles of Distributed Database Systems, Prentice Hall Publishing.
[42] JMS 2002. Java Messaging Service. Version 1.1. http://java.sun.com/products/jms/docs.html
[43] E. Freeman, S. Hupfer, K. Arnold (1999) JavaSpace: Principles, Patterns, and Practice, Addison-Wesley Publisher.
[44] Y. Jin (2004) An Architecture and Execution Model for Component Integration Rules, Ph.D. Dissertation, Arizona State University, Department of Computer Science and Engineering.
[45] G. Booch, J. Rumbaugh, I. Jacobson (1999) The Unified Modeling Language User Guide. Addison-Wesley Publisher.
[46] BEA Systems Weblogic Server (2003). http://www.bea.com
[47] U. Cetintemel (1995) OBJECTIVE: A Benchmark for Object-Oriented Active Database Systems, M.S. Thesis, Bikent University, Turkey.
[48] Y. Jin, S. D. Urban, D. W. Dietrich (2005) Extending the OBJECTIVE Benchmark for Evaluation of Active Rules in a Distributed Component Integration Environment, submitted for journal publication, 2005.
[49] S. D. Urban, V. Kumar, and S. W. Dietrich (2005) A Prototype for Integration of Web Services into the IRules Approach to Component Integration, Proceedings of the International Conference on Enterprise Information Systems, Miami, FL., pp.3-10.
[50] H. Ma, S. D. Urban, Y. Xiao, and S. W. Dietrich (2005) GridPML: A Process Modeling Language and History Capture Systems for Grid Service Composition, Proceedings of the International Conference on e-Business Engineering, Beijing, China.