DESIGN AND IMPLEMENTATION OF A DYNAMIC PROGRAMMING

ENVIRONMENT FOR ACTIVE RULES

The members of the Committee approve the master's
thesis of Sridhar Reddy Varakala

Sharma Chakravarthy
Supervising Professor   _____

David Kung   _____

Leonidas Fegaras   _____

DESIGN AND IMPLEMENTATION OF A DYANMIC PROGRAMMING

ENVIRONMENT FOR ACTIVE RULES

by

SRIDHAR REDDY VARAKALA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2003

To My Parents, Family and Friends

ACKNOWLEDGMENTS

Oct 28, 2003

ABSTRACT

DESIGN AND IMPLEMENTATION OF A DYANMIC PROGRAMMING

ENVIRONMENT FOR ACTIVE RULES

Publication No.____

Sridhar Reddy Varakala, M.S.

The University of Texas at Arlington, 2003

Supervising Professor: Sharma Chakravarthy

Currently, many applications that require monitoring (e.g., surveillance, track monitoring) use the event-condition-action (or ECA) rule paradigm. The Local Event Detector (LED) that has been developed based on the ECA rule paradigm provides active capability to various applications including relational database systems. LED is capable of handling events that are local to the application, thus providing active capability. Existing LED framework supports creation of events and rules at compile time. Modifications to existing rules or events are difficult once the execution of the application begins. Thus, modifying rules or events requires adding code for new rules and events, recompiling and

executing. This is a limitation for many applications where the rules and events need to be modified dynamically at run time and without bringing the system down.

This thesis provides a novel approach for supporting the creation of rules and events dynamically at run time, which is critical for several classes of monitoring applications. The proposed approach avoids recompilation and restarts which is not appropriate in many environments that require fine-tuning of rules on the fly. The dynamic programming environment (DPE) presented in this thesis provides a generic set of classes that are designed to handle the creation, management, and execution of rules. This set of generic classes is application-independent making the system a general-purpose tool. The user application provides necessary class and event information to the DPE. The DPE uses this information to create, modify and manage the rules. The DPE supports the creation of new composite and temporal events. The system also provides information about rules and events in XML format. A user-friendly interface is provided to perform these tasks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The imminent need for security in the real world has led to the development of many systems. These applications range from Coast Patrolling mechanisms to Network Management protocols. There is a need to continually monitor these applications, which may not be possible at all times. For example, a radar personnel in a military environment has to constantly monitor air traffic and inform superiors if any hostile planes are tracked. Alternately, the Event-Condition-Action paradigm can be applied to monitor the radar and respond to events without any user or application intervention.

The Event-Condition-Action rule consists of three components, namely, an event, a condition and an action. An event is an instantaneous, atomic occurrence of interest at a specific point in time. The condition part of the rule checks the state of application object and is evaluated according to the context of occurrence of the event. The action describes the task to be carried out by the rule. When an event occurs, the corresponding rule is triggered and the condition is evaluated. If the condition evaluates to true, the corresponding action is taken. Considering the radar personnel example again, the event can be defined as the appearance of a new plane and the condition as the plane belonging to a hostile country or unidentified and the action can be that of informing the seniors in a predefined manner. This way, the radar can be constantly monitored using the ECA rules and it responds to the events automatically.

1.1 Background

A Local Event Detector (LED) [1] has been developed which is based on the Event-Condition-Action rule paradigm. It provides active capability to various applications including relational database systems. It uses flexible and expressive event specification along with well-defined semantics provided by the SNOOP event specification language [2, 3]. It is well suited for monitoring complex changes within an application.

LED is an event detector implemented to detect events and execute rules based on those events. It is capable of handling events local to an application. It can detect primitive and composite events and define rules at class and instance levels. The event detector processes the event occurrences asynchronously from the application. It provides an option to create rules with different priorities, and the rule with the highest priority being executed first. Rules with the same priority are executed concurrently. The application uses LED as a library and invokes the LED API's to create, raise events and create rules.

1.2 Focus of this Thesis

The Current LED implementation supports the creation of events and rules at compile time. It uses the conventional Edit/Compile/Execute paradigm. This has several drawbacks: once the application has started execution, modifications to existing rules and events in the application are difficult. In real-life monitoring applications, during execution there may be a need to create new rules and events. At present, the only way to make changes is to rewrite/add code, recompile, and re-execute it. This is not desirable for many applications where the previously defined rules may need to be refined or new rules added without bringing the system down. One application that has been developed for the Navy is

a track monitoring system. This application uses a customized rule editor to create a few rules that has been built into the editor. The rule editor contains all the rules that were known to occur during the lifecycle of the application. For example, one of the rules defined is "When the speed of any track is greater than 700 mph and is present in a specific region termed as Area of interest, update the display of the GUI with the status of the track". The user could create only those rules that are built into the rule editor. As a result the user is restricted in the types of rules that can be created. If the user needs to create new rules then this rule logic has to be added to the rule editor, the application has to be recompiled and reloaded for the new rule to be available. The applications some times require that the conditions be changed/updated due to changes in the run time environment. For example, for the above-mentioned rule, the condition may have to be fine tuned to 600 mph instead of 700mph.If the edit/compile/execute paradigm is used then the code has to be edited, compiled and reloaded. Instead if the interactive/dynamically modify paradigm is used then the as soon as the condition is modified the change is visible to the application and LED as well and as a result the rule is modified. Therefore the interactive/dynamically modify paradigm is better suited for applications that cannot be interrupted to make modifications.

For many applications such as the track monitor for Navy, and others (e.g. Stock, trading application) there is a need to add events and rules without using the edit/compile/execute cycle normally used for software development. It is necessary to change or fine-tune the conditions on the fly. Hence, an interactive, easy-to-use interface was needed to perform the above tasks on the fly and without stopping and/or restarting the application.

This thesis provides a flexible design and implements a tool called "Dynamic Programming Environment" which has the ability to create or modify the rules as well as create new composite or temporal events without interrupting the application execution. This tool provides a generic set of classes to handle the rules and user application has to be only concerned about defining events and raising them. The tool also can provide information about the rules and events in an application in XML format. This provides for persistence, so that the next time around a user application can revert to the same state (events and rules defined) when execution was stopped.

The rest of this thesis is organized as follows. CHAPTER 2 describes the related work in active systems and rule engines. CHAPTER 3 covers the Event Semantics of Snoop and CHAPTER 4 gives an overview of LED and the APIs required for creating temporal, composite events and class or instance level rules. CHAPTER 5 provides details of various design issues involved in building the system. CHAPTER 6 deals with implementation issues encountered during the course of the thesis. CHAPTER 7 contains the conclusions and future work.

CHAPTER 2

RELATED WORK

This chapter reviews the related work in the area of the ECA paradigm and business rule engines that create new rules and events dynamically and allow modifications to the rules.

2.1 WebLogic Events

WebLogic Events [4] is an event management system using a publish/subscribe paradigm from WebLogic. It allows an event of interest (topic) to be registered across the network by any WebLogic application. The WebLogic Server matches the occurrence of events with registered events of interest. When WebLogic receives an event that matches the registration, it executes a user-written evaluate (or condition) method and based on the outcome, will execute a user-written action method.

Event registrations are stored in a Topic Tree, which is central to this architecture. It is a hierarchical, n-ary tree of period-separated words where each node represents a particular level in tree. Each level in the hierarchy represents a greater level of specificity; for example, a topic about the city weather in Dallas is represented using the notation as *"weather.northamerica.us.texas.dallas"*. The whole event service flows through the topic tree. When an event is published by an application, it trickles down the topic tree until it reaches the registration corresponding to the topic. When an event matches a registration, the registration's 'evaluate' method is called and if the results from the evaluate method are

successful then the 'action' method is called. The 'evaluate' and 'action' methods correspond to the condition and action parts of the rule. Since a registration is associated with a single evaluate and action pair only a single rule can be defined where as multiple rules can be defined on an event in LED.

The possible groups of operation on a Topic Tree are:

- EventRegistration: An application registers interest in a particular topic by sending an Event Registration to WebLogic operations

- EventMessage: Any application on the network can generate an event for a particular topic, by sending an Event Message to WebLogic.

The topic tree is dynamically built inside the WebLogic Server as clients subscribe to Event Topics. When a client subscribes to an event topic that does not exist then a new node and new branches to reach that node are constructed in the topic tree. Now, when this event is published, the subscribing clients will receive notifications. The leaf nodes in the topic tree correspond to events and all intermediate nodes are used for branching conditions. All the leaf nodes correspond to primitive events and there is no mechanism to combine the primitive events to form composite events.

The WebLogic Event Server provides communication between applications via subscribe and publish paradigm. When a client subscribes to an event on a WebLogic server, it can specify extra conditions that must be satisfied before an event is forwarded to it. This is achieved via an evaluator that runs on the server. The evaluator checks the conditions specified by the client before forwarding the event to the client. The event action() method executes either at the server or the client. WebLogic also supports event parameters as a set

of name-value pairs that further qualify the event. An event is submitted to the server with a set of event parameters.

In WebLogic server, it is not possible to change the evaluate methods once it is registered. That is one cannot modify the rule once the event, condition and action methods have been registered.

2.2   ILOG JRules

The software ILOG JRules [5] is one of the fastest, compact and robust rule engines in the market. It includes a rule engine for the Java platform, a rule language with support for business rules and the rule kit – a set of tools supporting the development of business rule applications.

2.2.1   Rule Engine:

It is a Java class that can be implemented directly or derived from to add application specific data members and methods. It provides flexible API's for the developer to have complete control over the engine. The rules can be dynamically added on the fly, modified or removed from the engine without shutting down or recompiling the application.

2.2.2   Rule Language:

Rules may be represented in Java-like syntax of ILOG rule language or in XML or in a readable syntax called a business rule language. They provide full support of java operators in expressions. They also support a relation between objects for a rule. It also supports rule creation for events based on time. The rule structure consists of the following three parts: a header, a condition part and an action part.

The rule structure is shown below:

rule ruleName { [priority = value;] [packet = packetName;]

when { conditions …}

then { [actions …] }};

The header defines the name of the rule, its priority and packet name. The condition part begins with keyword when, and is also referred to as the left-hand-side (LHS) of the rule. It defines the conditions that must be met in order for the rule to be eligible for execution. The action part is referred to as the right-hand side (RHS) of the rule. When all of the LHS conditions are met, the RHS of the rule is executed (or fires). A rule's priority controls the sequence of the rule execution. The packet name associates a rule with a rule packet. Packets provide a way of grouping of related rules. The packets can be activated/deactivated through rule-engine API or the Rule Language itself.

2.2.3   Rule Kit:

The rule kit contains a common set of tools delivered with ILOG JRules. It features an integrated development environment, rule-editing capabilities and business rule language support for business level representation of rules. When used with JRules repository, the rule kit provides all features necessary to create a rule management environment, namely mechanism to create, edit, maintain, debug, and deploy business rules and associated data, customize it to needs of the application. The components of the rule kit are briefly described below:

Rule Builder: The Builder is a graphical environment for developing and debugging rule sets. It manages projects, provides a set of sophisticated editors to create and

modify rules, and has integrated debugging, inspection and tracing tools to simplify the process of developing and debugging a business rule application.

Rule Editors: The Rule Kit enables developers as well as end users to modify rules by providing a flexible rule editor component that can add (acts as a plug-in to add) business rule editing capabilities to any Java application. It is a point and click interface that can be used to create or edit syntax that business people can use and understand.

Business Rule Language: It uses a business vocabulary and allows reasoning on an object model that reflects the structure of the business domain, rather than underlying Java implementation. It can be made as simple or complex as necessary, providing business users with the freedom of creating complex rules or constraining them to work within a predefined scope of business domain.

2.2.4 Rule Repository:

ILOG Rules persist business rules and all relevant business data in a structure called "repository". The repository is the central place for an application's business rules data and is much more efficient than traditional file based persistence schemes. The overview of the structure of business rules repository is shown below in Figure 2.1. Business rules are organized in to packages that can be nested to any arbitrary depth, which allows one to structure a project in a way that is closer to the logic of the application. Also, a project is a collection of references to packages and libraries of the repository. The libraries contain the data that is used through out business rules, rule language and rule templates.
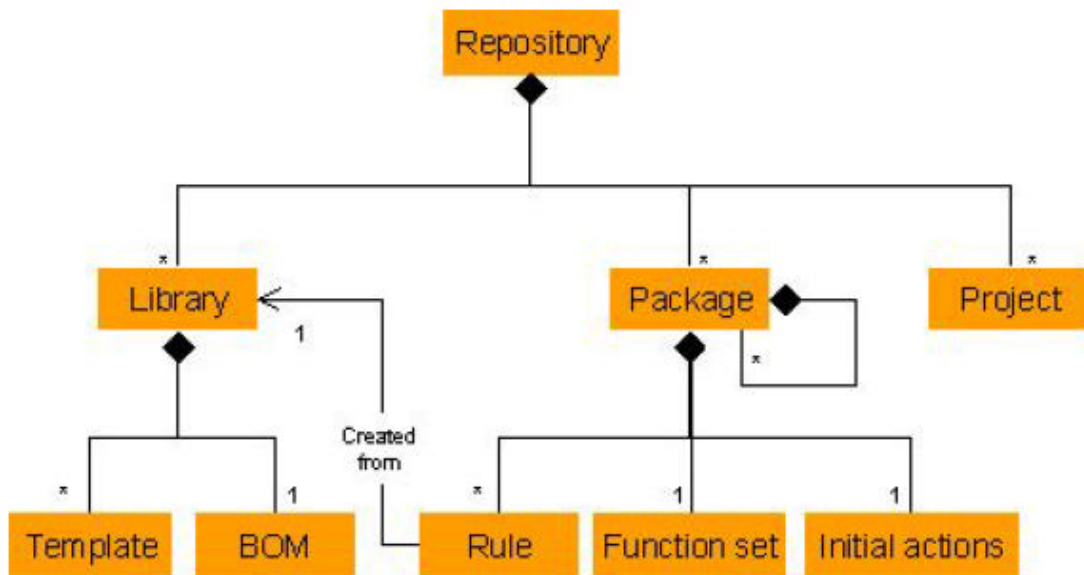
Figure 2.1 The ILOG Business Rule Repository structure

2.2.5   Working Memory and Context

When an ILOG rule engine is integrated into an application, part of its function is to monitor the state of various application objects. This is necessary, as rules will typically reference application objects in the condition part of their logic. Working memory is where ILOG JRules stores references to all of the objects with which it is currently working.

Rules are grouped into sets. Rules in a rule set and the application objects that they reference are associated by what is called a context. That is, a context associates rules with working memory and implements the rule engine that controls the relationship between the two.

ILOG Rule Engine

An ILOG Rule Engine is integrated into an application by simply having the application instantiate an IlrContext object, which creates a rule engine. Formally, this

instantiation creates a context. The context associates a rule set with working memory and implements a rule engine that controls the relationship between the two.
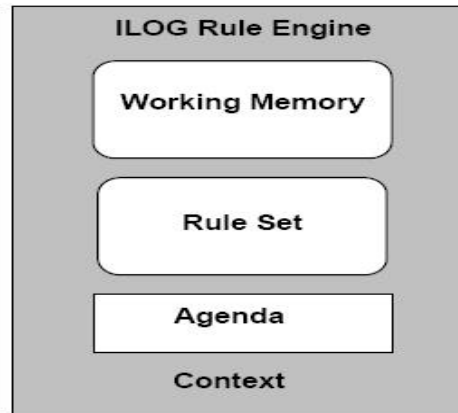
Figure 2.2 Context

The IlrContext constructor creates a rule set container, working memory, a rule engine, and a context to control it all. The final step in setting up the rule engine is adding rules to the rule set container. The rule engine is now ready to examine working memory, matching objects referenced in working memory against patterns found in the rule conditions and placing rule instances in the agenda. The agenda is a container that stores rule instances that are ready to be executed. The rules are fired once the *fireAllRules* method is called on the IlrContext object. Once the rule fires, it will be removed from the agenda and does not fire again.

JRules is not based on active technology; the only event it provides is the temporal event. But it is a powerful business rule engine that is widely used.

2.3 Vitria BusinessWare Process Automator

The Process Automator in Vitria BusinessWare [6] provides a modeling environment that captures business objects, events, rules and processes and uses them to build a collection

of business policies. It incorporates four modeling techniques into an integrated process-development environment: business object models, business event models, business process (or state) models and business rules. It uses a graphical modeling language to create process charts that describes the different stages of a business process. Process charts use graphical elements to describe processes in terms of discrete states and the conditions that cause the business object to move through various stages of the process by means of transition between states. Business rules and policies are expressed in event-condition-action sequences. After model states and transitions are defined in the process chart, the business rules for the processes are defined using a form based editor. The rule editor is used to specify the processing condition and action when a transition occurs between process states. Thus, an event is a transition from one state to another. A rule condition compares a process variable with some value and such relational comparisons can be connected through logical operators. The rule action invokes methods on process objects. It has no support for composite events and context based event detection unlike LED. Also, rules cannot be specified with a priority. Rule conditions cannot perform complex computations unlike the Sentinel system. Since conditions and actions are implemented as methods in Sentinel system, they can perform any operations that are permissible in a Java method.

2.4   Sentinel

Sentinel [7] [8-11] is an integrated active DBMS incorporating ECA rules using the Open OODB Toolkit from Texas Instruments. Event and rule specifications are seamlessly incorporated into the C++ language. Any method of an object class is a potential primitive event. The event occurs either at the beginning of the method or at the end of the method.

Composite events are defined by applying a set of operators to primitive events and/or composite events. Events and rules are specified in a class definition. In addition, Sentinel supports events and rules that are specific to object instances. In that case, events and rules are specified within the program where the instance variables are declared. This ability to declare events and rules outside of a class allows for composing events across classes. This provides a major advantage to Sentinel compared to many other active systems.

The parameters of a primitive event correspond to the parameters of the method declared as the primitive event and other attributes, such as the time of occurrence of the event. The processing of a composite event involves not only its detection, but also the computation of the parameters associated with the composite event. The parameters of the event are passed onto condition and action part of a rule. The parameters associated with the detection of an event can be different in different contexts. Sentinel supports four contexts namely recent, chronicle, continuous, and cumulative contexts.

An event can trigger several rules, and rule actions may raise events that can trigger other rules. Sentinel supports multiple rule executions, nested rules executions as well as prioritized rule executions. Sentinel supports immediate and deferred mode of execution.

A Dynamic Rule Editor [12] was built to extend the Sentinel system to support external/dynamic rules. It provides a friendly environment to manipulate the rules without changing, recompiling or relinking any source code of their applications. It has a three-tier architecture and it consists of four modules namely the interface, the Rule Editor Server, the rule database and the Dynamic Rule Loader. The architecture is shown in Figure 2.3 and the functional modules is shown in Figure 2.4

13
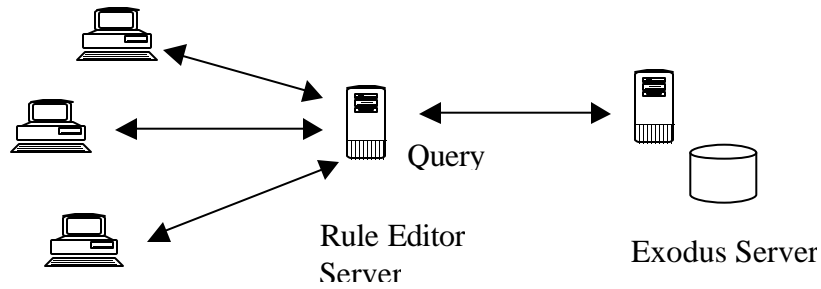
Figure 2.3 The Architecture of the Flexible Dynamic ECA Rule Editor
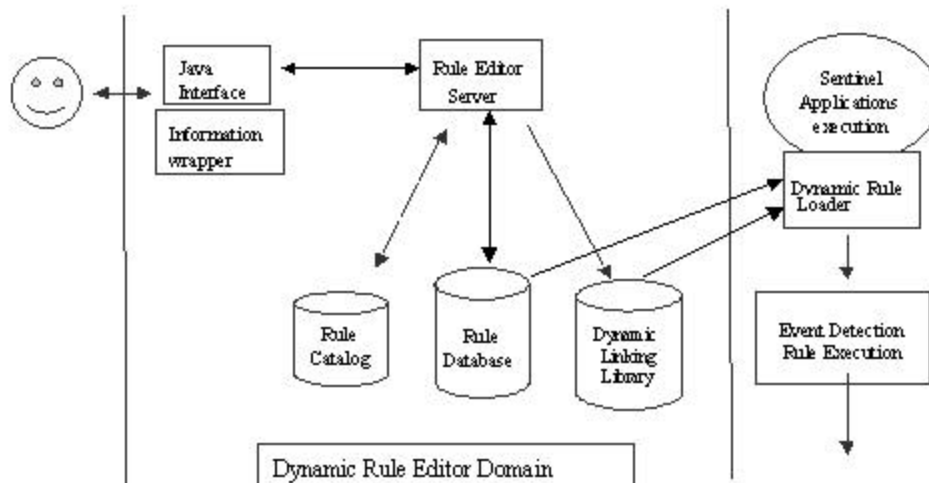
Figure 2.4 The Functional Modules of the Flexible Dynamic ECA Rule Editor

The Java interface provides the users with GUI to collect, validate, wrap, and display information. As a client, it interacts with Rule Editor Server via TCP protocol. The Rule Editor Server has most of the application logic. The database logic from the Rule Editor Server is decoupled using OQL. The rule database is stored in and managed by Exodus Storage System [13]

14

When Rule Editor Server is invoked, the first task is to initialize the rule database. If the rule database does not exist, the Rule Editor Server creates one and initializes it based on the system file named "applications.txt" in the rule catalog directory. The file records all the applications' alias names and their corresponding Exodus storage group numbers. On the other hand, if the rule database already exists, the Rule Editor Server updates it based on the system file, deleting out-of-date applications and adding new applications. After initializing the database, it listens to clients in a loop. Whenever a client initiates a connection to it, it forks a child process to serve the client. Dynamic linking library is used to invoke condition and action functions, which avoids relinking of the applications. The sentinel application invokes the dynamic rule loader at run time based on the rule database and the DLL that creates events and rules by calling the EVENT and RULE class constructors. The event detection logic and the rule execution logic were implemented in these classes.

The Sentinel (C++ version) had several limitations due to the use of C++, which is a compiled language. To overcome some of the limitations, a stand-alone local event detector (or LED) was implemented in Java. Some of the limitations of C++ were that conditions and action could not be modeled as methods associated with a class, the implementation of temporal events was platform dependant, four parameter lists were used at each composite event node to detect the event in four contexts, object data types were not allowed in the parameter lists, instance level had to be defined for each instance level rule and finally the events are detected synchronously in the application. In other words, event detection was processed in the application thread itself.

CHAPTER 3

EVENT OPEARTORS AND THEIR SEMANTICS

This chapter discusses primitive events, various types of composite event operators, the semantics of the operators, and the different parameter contexts using which a composite event can be detected, as applied to the Snoop language [2, 3]. Rules can be defined on both primitive and composite events. Whenever an event (primitive or composite) is detected, the rules associated with that event are executed. Also, rules can be specified with a priority, a coupling mode and a triggering mode. This chapter also discusses the different coupling modes and triggering modes associated with a rule as well as rule priority.

3.1   Primitive Events

An event is an occurrence of interest at a specific point in time. Primitive events are the elementary occurrences and are classified into domain-specific events (e.g., database, oodb, track application), temporal, and explicit events. Domain-specific events are specific to a domain and are associated with the manipulation of data in that domain (such as the creation, deletion, or insertion that are executed over a period of time in an RDBMS). Event modifiers (begin and end) were introduced to transform operations that take an interval into an instantaneous event. In other words, the event modifiers (begin and end) are used to map the logical events at the conceptual level to physical events. The begin event modifier denotes the starting point of an event and the end event modifier denotes the ending point. Temporal events correspond to absolute and relative temporal events. The absolute temporal

event is an event associated with an absolute value of time. For example, 11 A.M. on October 28, 2003 is an absolute event. The relative temporal event corresponds to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event). Explicit (also termed abstract) events are explicitly defined in an application, but their occurrences are either detected outside of the application or conveyed to the application or the application explicitly raises those events. Events can be detected using either a point-based semantics [2, 3] or an interval-based semantics [14] [15]. In this thesis, point-based semantics is used as we use an LED that detects point-based semantics. Nothing in the DPE will change if an LED is replaced with one that detected interval-based semantics.

3.2   Composite Events

A composite event is an event that is composed of primitive events and/or other composite events by applying Snoop [16] [2, 3] event operators such as OR, AND, SEQUENCE, NOT. In other words, the constituent events of a composite event can be primitive events and/or previously defined composite events.

3.3   Snoop Event Operators

The event operators are used to construct composite events. Some of these event operators and its *point* semantics [2, 3] are described briefly in the following section. The upper case letter E, which represents an event type, is a function from the time domain on the Boolean values. The function is given by

E (t) = True if an event type E occurs at time point t

False otherwise

17

- **OR (V):**

    Disjunction of two events E_1 and E_2 denoted by E_1 V E_2 occurs when either E_1 occurs or E_2 occurs. Formally,

    $$(E_1 \text{ V } E_2)(t) = E_1(t) \text{ V } E_2(t)$$

- **Conjunction: AND (?)**

    Conjunction of two events E_1 and E_2, denoted by E_1 ? E_2 is applied when E_1 and E_2 both occurs (in any arbitrary order). Formally,

    $$(E_1 \text{ ? } E_2)(t) = (E_1(t1) \text{ ? } E_2(t)) \text{ ? ?? } ((E_1(t) \text{ ? } E_2(t1)) \text{ and } t1 \le t$$

- **Sequence (;)**

    The sequence of two events E_1 and E_2, denoted by E_1; E_2 occurs when E_1 happens before E_2. That is, the timestamp of occurrence of E_1 is less than the timestamp of occurrence of E_2. Formally,

    $$(E_1; E_1)(t) = E_1(t1) \text{ ? } E_2(t) \text{ and } t1 < t$$

- **NOT (∅):**

    The not operator, denoted by $\neg(E_2)[E_1, E_3]$ detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3.

    $$\neg(E_2)[E_1, E3](t) = (E_1(t^1) \wedge \sim E_2(t^2) \wedge E_3(t))$$

    and $t^1 \le t^2 \le t$

- **Aperiodic Event Operators (A, A*)**

    The Aperiodic operator A is used to express the occurrence of an aperiodic event in the half-open interval formed by E_1 and E_3. There are two variants of this event specification. The non-cumulative variant of an aperiodic event is expressed as $A(E_1, E_2, E_3)$ where E_1, E_2

18

and E_3 are arbitrary events. The event A is signaled each time E_2 occurs during the half-open interval defined by E_1 and E3. A can occur zero or more times. Formally,

$$A\ (E_1, E_2, E_3)\ (t)\ = E_1(t^1)\ \Lambda \sim E_3(t^2)\ \Lambda\ E_2(t))$$

$$\text{and } (t^1 < t^2 \leq t \text{ or } t1 \leq t^2 < t)$$

There are situations when a given event is signaled more than once during a given interval (e.g., within a transaction), but rather than detecting the event and firing the rule every time the event occurs, the rule has to be fired only once. To meet this requirement, there is an operator $A^*\ (E_1, E_2, E_3)$ that occurs only once when E_3 occurs and accumulates the occurrences of E_2 in the half-open interval formed by E_1 and E_3. This constructor is useful for integrity checking in databases and for collecting parameters of an event over an interval for computing aggregates. As an example, the highest or lowest stock price can be computed over an interval using this operator. Formally,

$$A^*(E_1, E_2, E_3)\ (t) = (E_1\ (t^1)\ \Lambda\ E_3\ (t))\text{ and } t^1 < t$$

In this formulation, E_2 is not included because the occurrence of the composite event A^* which coincides with the occurrence of E_3 is not constrained by the occurrence of E_2. However, the parameters of A will contain the parameters of E_2.

- **Periodic Event Operators (P, P*)**

The periodic operator, denoted by $P\ (E_1, [t], E_3)$ is used to express a periodic event that repeats itself within a constant and finite amount of time. The event P is signaled for every amount of time t in the half-open interval $(E_1, E_3]$. Formally,

$$P\ (E_1, [TI], E_3)\ (t)\ =\ (E_1(t1)\ ? \sim E_3(t2))$$

and $t1 < t2$ and $t1 + x * TI = t$ for some $0 < x < t$ and $t2 \leq\ t$

where TI is a time specification..

There is also a cumulative version of the P operator, similar to the aperiodic context, defined as

$$P^*(E_1, [TI], E_3) (t) = (E_1(t^1) \wedge E_3(t)) \qquad \text{and } t^1 \leq t$$

Though TI is not mentioned in this formulation, the parameters specified are collected for each occurrence of [TI] as part of P^*.

- **Plus** (+)

The plus operator denoted by $E_1 + [T]$ is applied when T time units are elapsed after E_1 occurs.

3.4 Parameter Context

Four parameter contexts — recent, chronicle, continuous, and cumulative — were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of composite events using the point semantics described above [2, 3] [16]. The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event that completes the detection of a composite event is denoted as the terminator of the composite event. For example, a composite event (E1; E2; E3) has E1 as initiator and E3 as terminator.

- **Recent**:

In the recent context, the most recent occurrence of the initiator (when there are multiple instances of the same event) for any event that has started the detection of that event is used. When the event occurs, the event is detected and all the occurrences of events that

20

cannot be initiators of that event in the future are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs.

- **Chronicle**:

In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event. When an event occurs, the occurrences of the events used for detecting it are deleted. The event occurrence can be used at most once for computing the parameters of the composite event.

- **Continuous**:

In the continuous context, each initiator of an event starts a separate detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected.

- **Cumulative**:

In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the event occurs, all the occurrences that are used for detecting that event detection are discarded.

3.5 Coupling Modes

In early systems such as POSTGRES [17], condition evaluation and action execution were done immediately after the event was detected. However, in some situations this is too restrictive. For integrity checks, condition evaluation and action execution need to be done at the end of a transaction before it commits. Coupling modes were introduced [18] to specify a relative point in time where condition evaluation and action execution should take place after

21

the event is detected, with the constraint that the action will be performed only when the condition is satisfied. There are three coupling modes:

- **Immediate**:

    When an event is detected, the transaction is suspended, and the condition associated with the event is evaluated immediately. If the condition evaluates to true, the action is executed. The execution of the triggering transaction is suspended while the condition evaluation and action execution are completed.

- **Deferred**:

    The triggering transaction is continued after an event is detected. Condition evaluation and action execution are done at the end of the triggering transaction before it commits.

- **Detached (or decoupled)**:

    Condition evaluation and action execution are done in a separate transaction (or triggered transaction) from the triggering transaction. The detached mode can be classified into two types namely totally independent and causally dependent. When two transactions are totally independent, the triggered transaction is executed regardless of whether the triggering transaction commits or aborts. On the other hand, the triggered transaction can commit only after the triggering transaction commits for the causally dependent mode.

3.6 Rule Priority

In addition to the parameter context, coupling mode associated with a rule, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priority increases with the increase in the numerical value. That is, 2 has a higher priority

than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both the priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution.

CHAPTER 4

LOCAL EVENT DETECTOR

This chapter summarizes the Java Local Event Detector (LED) [1] and its
functionality. Based on ECA rule paradigm, local event detector supports active capability to
various kinds of monitoring applications including relational database systems. It uses
flexible and expressive event semantics provided by SNOOP[2, 3]. The LED has been used
to develop an agent that works with various commercial Relational DBMSs (such as Oracle
[19], DB2 [20] and Sybase[21]). The local event detector is well suited for monitoring
complex changes within an application.

4.1   Event Specification Interfaces and Usage

Local event detector provides active capability to an application by detecting the
occurrence of local primitive and local composite events defined as part of the application.
The application interacts with the local event detector through a set of interfaces shown in
Table 4.1 Common API's used in Event Detectors. Following are the steps for using the
event detector.

1. Application has to initialize an agent by invoking the *initializeECAAgent()* or
   initializeECAAgent (String agentName) method.

When initialized for the first time, the system initiates a new default ECA agent. If it
is not the first time, it will return the existing defaultECAAgent to the application. Multiple
ECAAgents identified with different names can be initialized using *initializeECAAgent*

(String agentName) API. Each agent is responsible for monitoring events defined in that agent and performing appropriate actions.

2. After initialization, the application can start defining events (both primitive and composite), and rules.

Define primitive events using *createPrimitiveEvent* API.

Define composite events using *createCompositeEvent* API.

Define rules using *createRule* API.

Using the ECAAgent initialized in step 1, the application can define a primitive event by invoking the *createPrimitiveEvent* method. Application has to supply name for the primitive event, name of the class in which the method associated with the event is defined, the event modifier (begin or end), and the complete method signature as parameters to this method. This method returns the event handle associated with the primitive event, which can be used for defining the composite events with this event as a constituent, storing the parameter of the event, and signaling the method invocation of event to the detector.

In some cases, the user may be interested in an event only when a particular instance or instances of the class invoke the method. In this case, the event should be detected only when certain instances invoke the method, but not all instances. To accommodate this, the notion of *"class level"* and *"instance level"* events is introduced. A method can be defined either as a class level or an instance level event. If the method is defined as a class level event, an event is detected when any instance of the class invokes the method. If the method is defined as an instance level event, the event is detected only when a particular instance of the class invokes the method. The instance is specified in the definition of the instance level event.

25

Table 4.1 Common API's used in Event Detectors

Create Primitive Event API	CreatePrimitiveEvent (String eventName, String className, EventModifier modifier, String methodSignature) CreatePrimitiveEvent (String eventName, String className, EventModifier modifier, String methodSignature,java.lang.Object eventInstance)
Create Composite Event API	createCompositeEvent (EventType operator, String eventName EventHandle ehOne, EventHandle ehTwo) createCompositeEvent (EventType operator, String eventName, EventHandle ehOne, EventHandle ehTwo, EventHandle ehThree)
Create Rule API	createRule (String ruleName, EventHandle eh, String condMethod, String actionMethod) createRule (String ruleName, EventHandle eh, String condMethod, String actionMethod, java.lang.Object eventInstance) createRule (String ruleName, EventHandle eh, Object conditionInstance, String condMethod, Object actionInstance, String actionMethod) createRule (String ruleName, EventHandle eh, Object conditionInstance, String condMethod, Object actionInstance, String actionMethod, java.lang.Object eventInstance)
Insert Parameter API	insert (EventHandle[] eventHandleArray, String varName, long longVal) insert (EventHandle [] eventHandleArray, String varName,float floatVal) insert (EventHandle[] eventHandleArray,String varName, Object object)
Raise Event API	raiseBeginEvent(EventHandle[] eventHandleArray, Object instance) raiseEndEvent (EventHandle [] eventHandleArray, Object instance)

The application can define a composite event in a specific ECAAgent by invoking *createCompositeEvent* method on that ECAAgent. To define a composite event, the application specifies the operator type and event handles of constituent events. Event

operators can be classified into two categories, binary and ternary. Two basic API's that can be used to define binary SNOOP operators, such as AND, OR, and SEQUENCE with two constituent events and ternary SNOOP operators, such as NOT, PERODIC, and APERIORDIC are shown in Table 4.1.

To define a rule associated with an event the application should provide the handle corresponding to that event, a condition method name, and an action method name. The condition and action are defined as methods in the associated class. Basic API used for defining a rule is shown in Table 4.1. The rule is fired when the event occurs and the condition is evaluated. If the condition is satisfied, the action is performed

A rule that contains all constituent primitive events as class level primitive events is termed a class level rule. A rule declared on only instance level primitive events is an instance level rule. For an instance level rule, the instance of the class on which event is defined, termed eventInstance, is also specified.

In some cases, the event and rule specification can be defined in separate classes. Further the condition and action can also be defined in separate classes. For this case, the application specifies the class name in condition method and action method accordingly. The format is as follows:

condMethod = "PackageName"+"."+"ClassName"+"."+"MethodName"

actionMethod = "PackageName"+"."+"ClassName"+"."+"MethodName"

To apply the rule on the specific instance's of the classes in which the condition, action methods are defined, the application needs to add two target objects, termed *"conditionInstance"* and *"actionInstance"* accordingly, in the rule.

3. Raise an event.

    1. Insert parameters of different data types using *insert ()* APIs provided by the ECAAgent.

    2. Raise the event.

The application can insert any primitive data type or object through *insert()* APIs shown in Table 4.1. In the method defined as primitive event, arguments of condition and action methods need to be inserted into the parameter list using the event handle before raising the event. These parameters can be used in evaluating the condition and performing actions.

An event can be raised at the beginning or at the end of the method corresponding to that event. The applications are provided with two APIs, *raiseBeginEvent* and *raiseEndEvent* for this purpose. These are also shown in Table 4.1.

LED detects the occurrence of the events through the invocation of *raiseBeginEvent* or *raiseEndEvent* methods. All the associated rules are triggered when the event occurs. The condition is evaluated. If the condition is satisfied, the action will be performed.

4.2 Event Detection

LED uses on an event graph for detecting composite events as shown in Figure 4.1. Each node in the event graph represents either a primitive event or a composite event defined in the application by using above interfaces. Primitive event nodes are leaf nodes from which composite event nodes are constructed. Composite event node can also be constructed from primitive events and/or other composite events. The primitive event node contains an instance-based multiple rule list and an event subscriber list, while the composite event node

28

contains only one rule subscriber list and an event subscriber list. An instance-rule list is a collection of rule subscriber lists for classes and instances. The rule subscriber list and event subscriber list keep the associated rules and composite events respectively. The *eventSignaturesEventNodes* and *eventNamesEventNodes* hash tables provide references to the primitive event nodes. When a primitive event occurs, it is notified to the leaf node and the occurrence is propagated to the internal nodes similar to a data-flow computation. The occurrence is propagated by means of an event table, which is described in the next section.
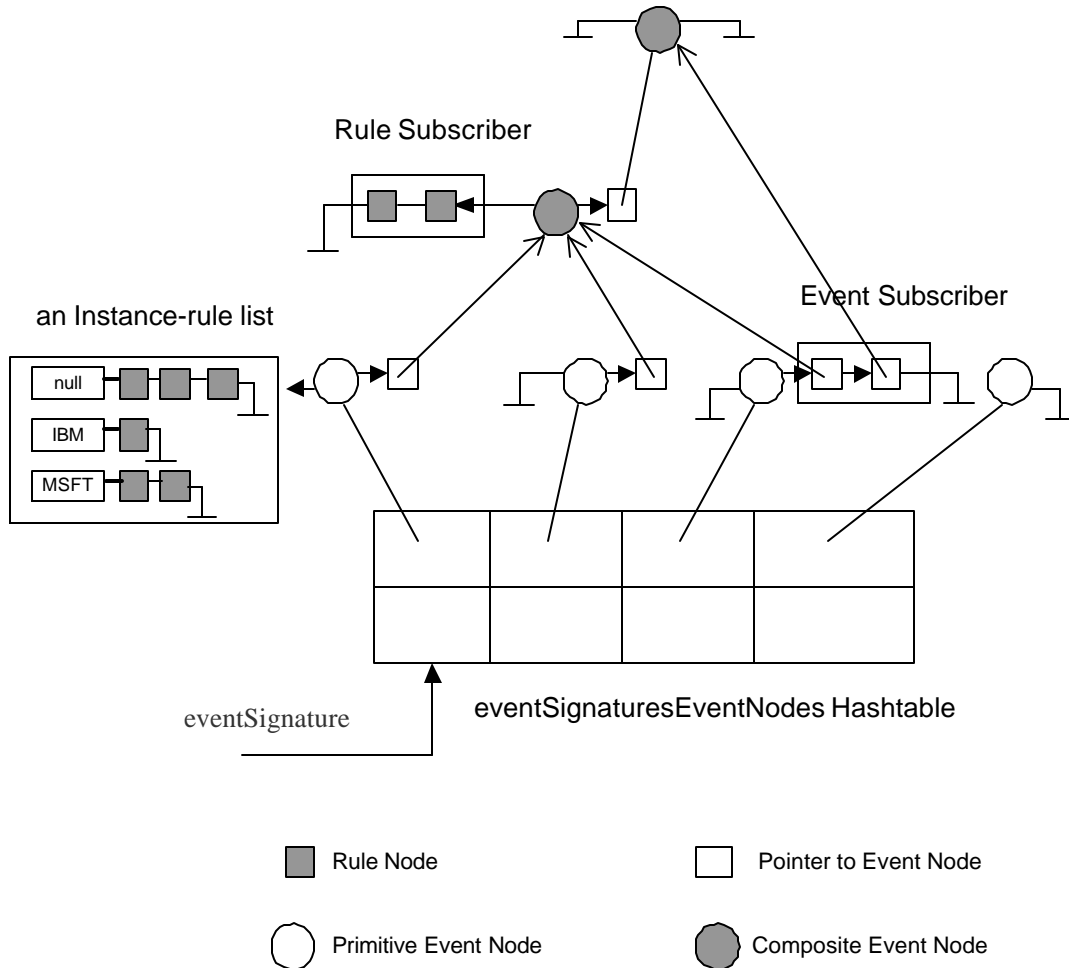
Figure 4.1. Event graph for detecting primitive and composite events.

29

Figure 4.2 shows an event table that stores all the key information including event occurrences, timestamp, a list of parameter lists, and context information used in detecting an event. The event table is also passed to the condition and action methods for extracting appropriate parameters for usage within condition and actions methods. An event table, termed a PCTable, consists of a set of event entries. Each event entry holds four-bit context information (Recent, Chronicle, Continuous, and Cumulative) and a list of parameter lists. The parameter list denotes the occurrence of a primitive event and the list of parameter lists denotes the occurrence of a composite event. The number of parameter lists in the list will be proportional to the number of constituent primitive events. The parameter list is used to store information when the event is raised, and to retrieve the data when the event is detected. The data coming form the parameter list is used for checking condition and performing action according to the rule definition.
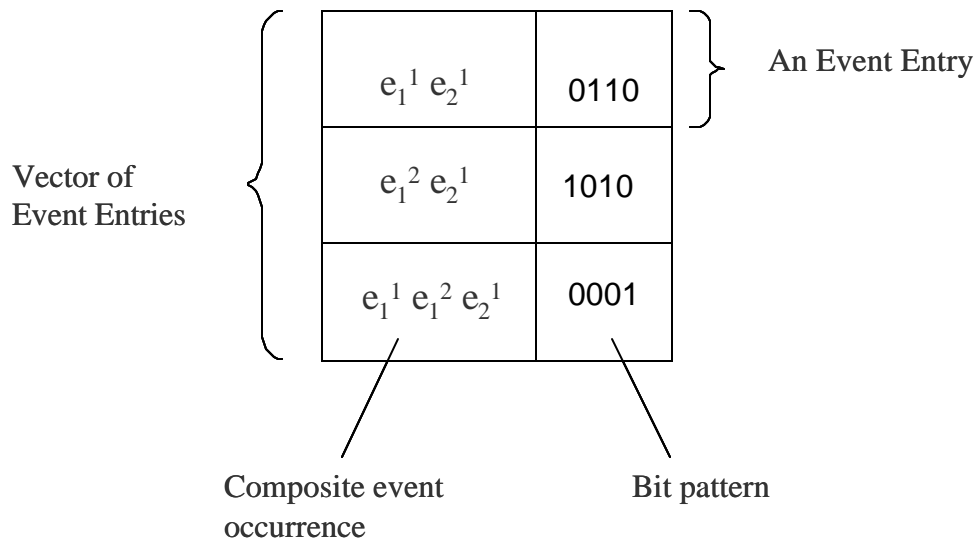
Figure 4.2. An event table represents a set of event occurrences.

The rule subscriber list consists of rule nodes, each of which defines a rule. The rule node stores the rule name, references to context in which the rule should be executed, condition method name and action method name. The rules in the rule subscriber list are triggered when the associated event occurs in that particular context. Whenever the rule is fired, the rule thread that contains the rule definition is instantiated. And it will be inserted into the rule scheduler. Once the rule is executed, the condition will be evaluated to determine whether the action should be performed or not.

4.3   An overview of components in Local Event Detector

The building blocks of local event detector are events, rules, ECAAgent, rule scheduler, and event detector thread. Every event defined in the application has an event handle. EventHandle object stores method signature associated with the event, class name in which the event is defined, and a list of parameter lists associated with that event. Event handle has been introduced to encapsulate several pieces of information pertaining to an event and to reduce the amount of information the user needs to keep track of.

A rule consists of a condition and an action. These are specified in *classname.methodname* format. The conditions and actions are implemented as methods in a Java class. LED uses Java reflection to refer to the classes in which the condition and action methods are implemented. Whenever the rule is triggered, the *RuleThread* object is instantiated and is inserted into the rule queue if rule scheduler is turned on. Rule scheduler later executes it.

The ECAAgent provides interfaces to define events and rules, insert parameters, and signals an event occurrence to the event detector. The ECAAgent class maintains data

structures to refer to event nodes with their names and method signatures associated with them. Application thread is separated from the event detector and works in conjunction with an event detector thread (*LEDThread*) through a buffer (*NotifyBuffer*).

The event detector thread is responsible for detecting events and firing rules. Whenever a primitive event occurs, all the relevant information about its occurrence is wrapped into an object called *NotifyObject*. This *NotifyObject* is put into the buffer, and is processed by the *LEDThread*. Running in an infinite loop, the event detector thread keeps fetching *NotifyObjects*, notifying occurrences of events, propagating the parameters to the internal nodes of the event graph, and firing the associated rules.

4.4   Filter Event in LED

A new event type is defined apart from the class and instance level primitive event, termed as "Filter" Event. This event type allows the user to specify a filter, which has to be satisfied for the event to be detected after it has been raised. If this filter is not satisfied the event will not be detected. The FilterEvent contains two objects of type namely: Event and Filter as data members.

The interface Filter define the following methods

> *abstract public boolean check(Reactive reactive);*
>
> *abstract public boolean check( );*

where Reactive is a class that can evaluate a string which contains public attributes of a class. The "Reactive" uses FESI package [22] to perform the string evaluation. It registers the public data members (attributes) of the class to JavaScript Engine when this object is created. This class takes the comparison string and retrieves the value of the attribute from

32

the object, which is invoking this evaluation. After the values are substituted, JavaScript Engine evaluates the string and it returns a Boolean object as the result.

The class "StringFilter", implements the interface Filter. It contains the following data members:

Reactive context; // The class that evaluates the expression

String expression; //The expression that has to be evaluated by Reactive

The chk() method of the interface is implemented as follows:

public boolean check() {

   if(context != null)

      return context.check(expression);

   else {

      System.out.println("Check   if   you   create   the   filter   with   syntax StringFilter(Reactive context, String expr)'");

      return false;

   }

4.4.1   Usage of a FilterEvent

The following steps have to be followed to use a FilterEvent:

1. First, the class on which the filter is to be defined should extend Reactive class. This class should first call the constructor of Reactive class with name to be associated with, when it is created. When this happens the public attributes of the class are registered with JavaScript Engine

2. Create a filter event with LED using the following API:

EventHandle createPrimitiveEvent(Reactive reactive,

String eventName, String className, EventModifier eventModifier,

String methodSignature, DetectionMode detectMode, String sFilter)

Where reactive corresponds to a instance of class that extends "Reactive", eventName, className, eventModifier ,methodSignature is used as in a normal event, sFilter is a string expression which contains a comparison string which can contain an attribute of the Reactive along with comparison operator and value.

The ECAAgent creates a StringFilter using this sFilter and reactive .

3. Create a rule using the following API:

createRule (String ruleName, Reactive reactive, EventHandle eh,

String condMethod, String actionMethod)

Where reactive is an instance of class extending Reactive, eh is an EventHandle of FilterEvent and the rest of information as in a normal rule

When this event is raised, a notify object is created. When this notifyObject is invoked by LED, it calls chk() method of the StringFilter present in the FilterEvent.This chk() method inturn calls the Reactive.eval(sFilter) ,where sFilter is the expression used while creating the FilterEvent. When this chk() returns true as the result, then the event is detected. The event is propagated and the rules are executedThe string evaluation feature of Reactive class is used by DPE, so that an attribute based comparison string can be evaluated at run time based on the value of the object.

CHAPTER 5

DESIGN OF DYNAMIC PROGRAMMING ENVIRONMENT

This chapter discusses the issues encountered during the design of the architecture of DPE, identifying the information needed for DPE, creation and modification of rules, the creation of temporal and composite events, and finally persisting information for reusability.

5.1   Approaches explored

An interactive, easy to use interface is required that can change or fine-tune conditions, create new events and rules. All these tasks need to be accomplished without recompiling or restarting the application. The various approaches that were explored to accomplish the above tasks are discussed below:

5.1.1   Approach 1- Static rule editor

In this approach, a rule editor is built for each application. Therefore for any other application a new rule editor is required, since this rule editor is specific to the application. Also, the user application can be considered to consist of two sets of classes; one set used to handle the interactive data input, to create and raise events and an other set of classes to maintain the semantics of the rules. In this approach, these two sets of classes are tightly coupled or intertwined, due to which it is difficult to understand the logic of the either set of classes without understanding the other. This also makes the management and maintenance of both these sets of classes difficult.

5.1.2 Approach 2 – General purpose rule editor

A general-purpose environment for creation, modification and management of rules can be built instead of building a specific rule editor for each application, as in the previous approach. Since it is a general rule editor, it is required to customize it so that it can be used with a specific application. To customize it, the following tasks are to be performed. First it has to interface with the user application. It has to acquire information that is specific to the user application. Also it has to handle the interface and domain specific issues of the application.

Using this approach, "*Dynamic programming environment for active rules (DPE)*" was designed. This approach has the following advantages as compared to the other approach, namely: First, it is application independent and hence can be used in conjunction with any application. Second, it has an interactive interface to dynamically edit and execute ECA rules. Most importantly, the application need not be modified/recompiled or restarted. The application developer can focus on the application semantics of handling the data, creating and raising the primitive events. DPE provides the flexibility to predefine some rules or let the DPE create the rules completely. Finally, the rules that are defined by the DPE can be persisted and reused again.

The following sections discuss the various designs issues that were encountered during the design of DPE.

5.2 Architecture

One of the issues encountered while designing the DPE was the handling of the various components (Rule Editor, Application, LED) and their integration them without

modifying their current architecture. The architecture that was initially explored is discussed followed the proposed architecture.

5.2.1   Initial Architecture

The first architecture that was considered is that the active application consisting of the user application and LED was in one address space (or a single process). The rule editor was in another address space (or as a separate process). The architecture is as shown in Figure 5.1
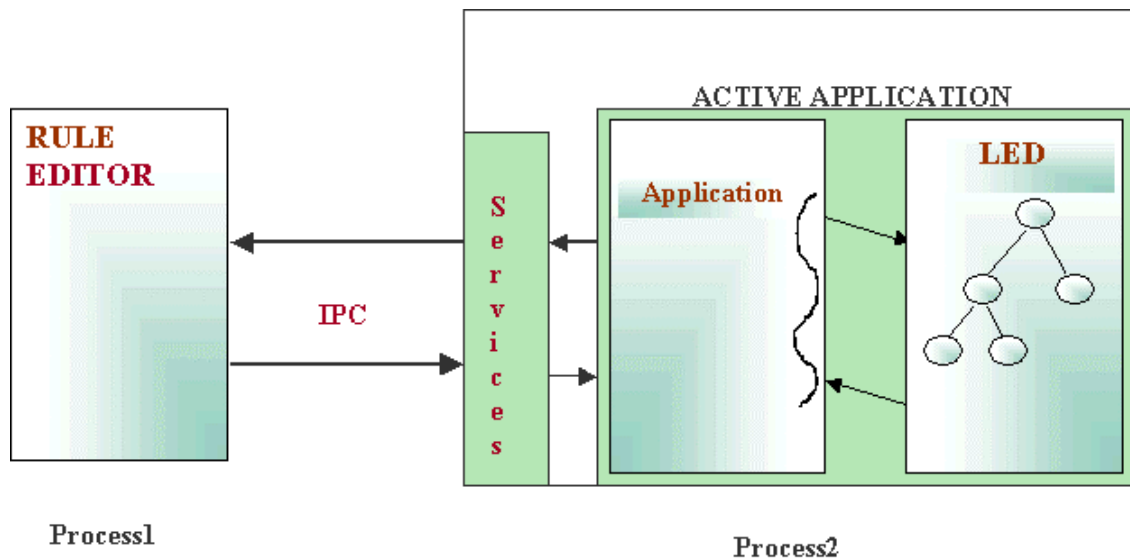
Figure 5.1 Architecture

This architecture is designed based on the general-purpose rule editor. It has the advantage of being application independent. But the two address space architecture has the following disadvantages: Firstly, since the rules have to be created in the LED eventually, but managed outside in the rule editor, Inter-Process communication (IPC) is required to send objects from the application to the rule editor. Also, additional services have to be

37

implemented for each application that can send and receive data in the application. The overhead to create events in this approach is too much. Finally, there is not much gain for the complexity posed by this architecture. Another architecture was therefore proposed due to the disadvantages in this approach.

5.2.2   Proposed Architecture

In this architecture all the three components are in a single address space as shown in Figure 5.2. DPE is a module that runs in the same address space as that of the application and the LED. DPE is responsible for creating & modifying rules and for creating new events at run time without recompiling the code. This is possible as the DPE can now access data structures of LED through the APIs. The application can send needed information to the DPE using a number of APIs that are part of the DPE. DPE is started as a separate thread by the application. The functional modules of DPE are shown in Figure 5.3. LED corresponds to a library that has been developed to declare events and to execute associated rules when the event occurs. The application uses LED API's to create and raise primitive events. The application provides necessary information to the DPE in the form of input configuration file. The DPE uses this information to create new rules and events that are registered with LED. The DPE accepts user request and performs the task.

In this architecture, there is no need for any IPC since all the components are in a single address space. The user application has to only invoke DPE and no other modification in the user application is required. DPE provides the necessary API's to be used by the user application. This architecture provides a clean, logical separation of code required for the Rule Editor and the code developed for the application.

ACTIVE APPLICATION

LED

Create Events/Rules

DPE

Conditions/
Actions

Create Events/Rules

Conditions/ Actions

User

Object References

Application

Output
Config

Input
Config

Single Process
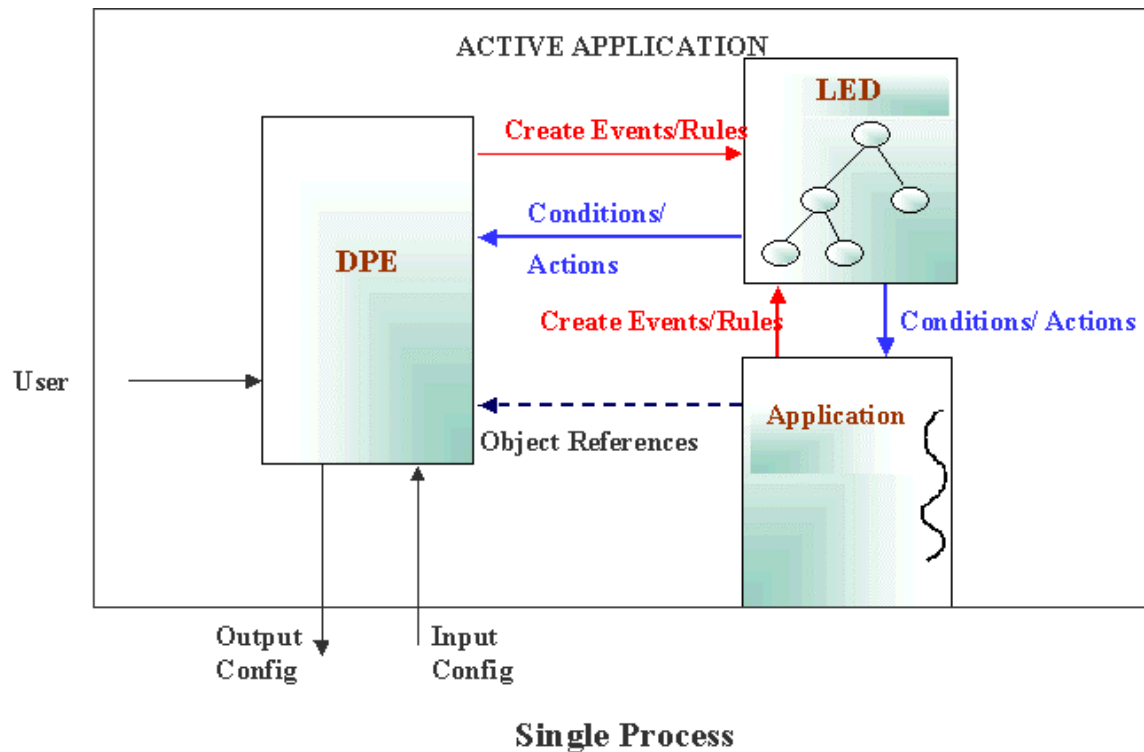
Figure 5.2 Proposed Architecture

5.2.3   Dynamic Programming Environment (DPE) functional modules

The functional modules of DPE are as shown in Figure 5.3. The Configuration

Processor accepts the input configuration file, which can either be in XML format or Text

format. This module parses the input configuration file and initializes all the data structures

that contain information required to create and modify rules, and create events. Once the data

structures are initialized, the GUI is set up to contain information about the available event

domains, event types, condition domains, condition methods, attribute, action domains and

finally the action methods. After the GUI is set up, the user input is accepted. User request is

sent to the module that is responsible to create and modify rules, and create events.
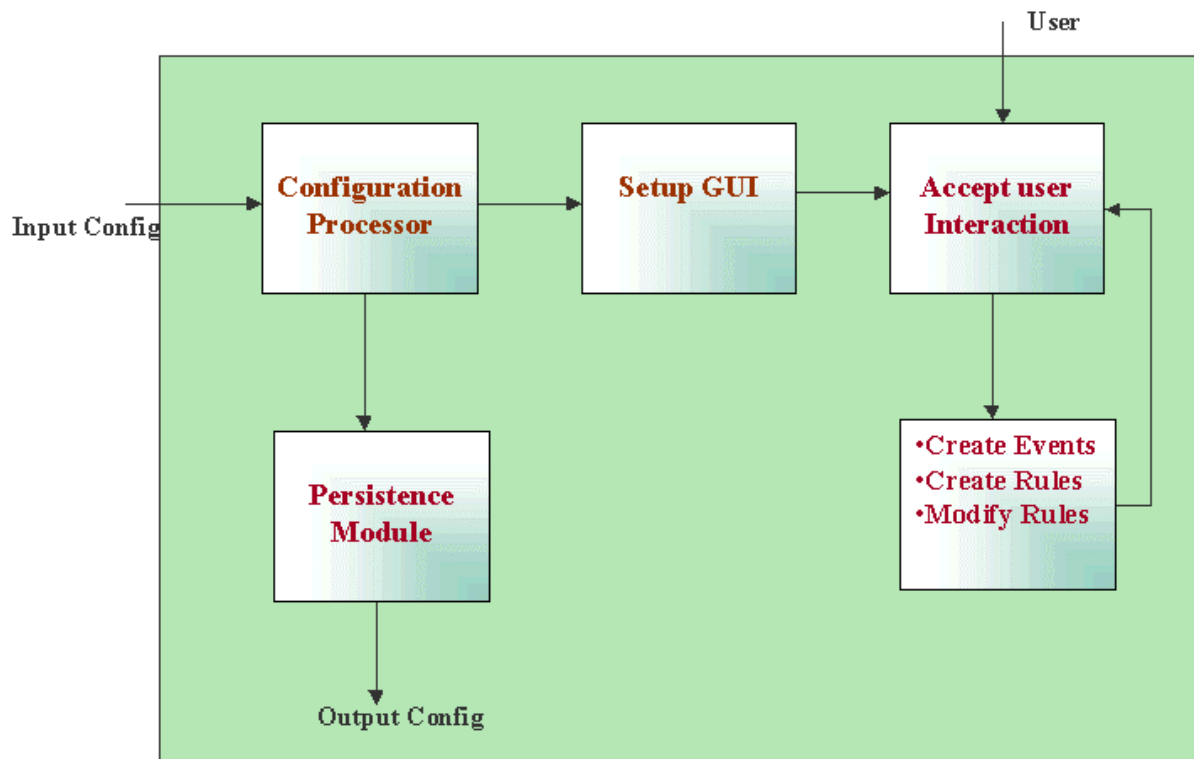
Figure 5.3 DPE Functional modules

If the request is to create a rule, the information corresponding to the new rule is stored internally and a rule is created in LED. This rule is triggered when the corresponding event occurs. The user can modify only the rules that have been created by the DPE. When the request is to modify a rule, the user can only modify the condition part of the rule. When a request to modify a rule is received, the information corresponding to this rule is retrieved. Now the new condition information is updated with the rule information. After the update is done, the new condition is effective. If the user request is to create a new event, then the corresponding temporal or composite event is registered with the LED. This new event created can be used to create new rules.

The DPE contains a module that generates a configuration file, which contains information about the events and rules created by the DPE. It also contains information about

40

the class and events created by the application. This configuration file can be used as input

configuration file to the DPE when it is restarted.

5.3 Rule Editor Configuration file

One of the important issues in the design is how to provide the application specific

information and what is the minimal information that would be required from the user

application so that the rules and events can be created successfully. The configuration file is

used to provide application-specific information to the DPE and the structure of the

configuration file is shown below in Figure 5.4.

```
Class:     Name
           InstanceName:Names
           Attributes :Name,Type,Units,EventAssociated
           ClassLevelPrimitiveEvent:EventName,methodsignature, Parameters to be inserted
           InstanceLevelPimitiveEvent:EventInstanceName,EventName,methodsignature,

                                Parameters to be inserted
           CompositeEvent:EventName,EventType,ConstitutentEvents(left,middle,right)
           Condition: MethodSignature
           Action:MethodSignature

EmailAction:
           EmailServer:
           FromAddress:

EventsCreated:
           ClassLevelPrimitiveEvent:EventName,AbsoluteTimein the format hh:mm:ss/mm/dd/yyyy
           CompositeEvent:EventName,EventType,ConstitutentEvents(left,middle,right)

RulesCreated:
           ClassRule:RuleEventName, ActionObject, RuleActionMethod,ConditionString
           InstanceRule:RuleEventInstanceName,RuleEventName, ActionObject,RuleActionMethod,

                ConditionString

Event Creation Messages:
```

Figure 5.4 Structure of the configuration file

41

The configuration file contains information about classes, new rules to be created and the default action information (e.g., email) The DPE provides a default action namely (Email) so that it can be used for notification when a rule is successfully executed. The class information contains the instance names, the information of events that has been created by the application and the events that have to be created by DPE. Also, it contains information about the attributes that can be used for condition and the list of conditions methods. The information about the available action methods is also specified. After DPE parses this configuration file, it uses this information to create rules and events.

5.4   Dynamic Creation of Events and Rules

The next issue in the design is to interactively add events, rules and modify rules. To add events or rules, the objects are needed, which is provided by the application through the configuration file. The objects provided by the application are just the references to the original objects and no duplicates are maintained.

Applications that are currently using LED have to predefine all the events and rules. Changing the rules at run time would involve changing the classes containing the rule definition (i.e., updating the condition part of the ECA Rule). Therefore, managing the condition part is the core issue of the DPE design since the condition can be changed at run-time. Currently, the condition is represented as a method and another representation is required since any update to the method is reflected only after recompiling the application. Conditions that have attribute comparisons can also be represented as a string instead of a method, so that when the string is evaluated (actually interpreted at runtime) the condition is checked and any update to the string is effective immediately. This condition string is an

42

attribute-based comparison. This representation is flexible and it can also be combined with existing condition methods. Also, the condition methods are not appropriate for interactive use but they can be still used for conditions if desired. The available types of conditions currently supported in the DPE are: an attribute-based condition string (simple condition), execution of an existing condition (simple condition) and finally a combination of the above two types (complex condition).

Consider an example class Track with two attributes: Speed and Altitude and a condition based on these attributes represented as "Speed > 700 && Altitude < 20000". This condition needs to be evaluated at runtime where in the values of the attributes "Speed" and "Altitude" at that point of time has to be substituted and then the resulting string is evaluated. The "Reactive class" in the LED package already provides this capability. This class uses a software package "FESI (Free EcmaScript Interpreter)" [22] which is an interpreter implemented in Java. The description of the Reactive class is already discussed in Section 4.4. Therefore any class that wants to use attribute in the condition needs to extend the Reactive class.

Also, using this approach a complex condition can be created which could contain many simple condition strings (SCS) connected by Boolean operators. Therefore a class, *"RuleConditionObject"*, corresponding to the SCS is constructed to allow the following functions.

- Represent both the predefined condition methods and condition strings constructed from attributes
- Evaluate the attribute comparison string

43

- Execute a condition method as a simple condition

All the *"RuleConditionObjects"* corresponding to the condition are contained in a *"ConditionGeneric"* class. Each rule created by DPE will contain an instance of *"ConditionGeneric"* class as the condition instance. The usage of *"ConditionGeneric"* class provides the flexibility to change the condition of rule later and not requiring the rule to register the updated condition with LED since only the data members are changed and no new instance is created. This class contains a method, *"condition"*, which follows the condition method requirement of the existing LED design. By this, the structure required by the existing LED for condition method of rules is retained and the modification of rules at run time is allowed. The condition method evaluates the complete condition by first getting the results from each of the RuleConditionObject and constructing a string, which is evaluated by the Reactive class to give the result, based on which the action part of the rule will be executed.

The available action types are default action, email and the action methods. To create instance-level rules and conditions, the instances of the classes are required. These instances are provided by the application to the DPE directly and in java objects are passed by reference and therefore no duplicates are maintained. Each instance is named and a mapping of user-defined names to object references is supplied by the application to the DPE by invoking an API provided by the DPE. Each instance is named and a mapping of user-defined names to object references is supplied by the application to the DPE by invoking an API provided by the DPE.

5.5   Rule Creation

Rules can be of two types: class level and instance level rules as described earlier. Rule creation is a three-step process that comprises of event, condition and action selection.

5.5.1   <u>Event Selection</u>

For each event that has been specified in the DPE configuration file, an instance of the class *"EventObject"* is created. This instance will contain all the information corresponding to the event and also the class that defines the event. All the *"EventObjects"* are stored in a data structure. Events are grouped based on the classes they belong to.

In the DPE, events are selected in the following way:

- Initially select a class from the list of event classes

- Select an instance of the class as the event instance if an instance level rule is required

- Select null as the event instance if a class level rule is required

- Finally select the interested event present in the selected class

A sample event selection would be constructed as

"null, Track.SpeedEvent" or

"T7001, Track.LatLongEvent"

Track is the name of the class containing the events. "null" and "T7001" refer to the event instances for a class and instance level rule respectively. SpeedEvent and LatLongEvent are names of the events selected from the class Track.

5.5.2   Condition Selection

The condition methods or attributes are grouped based on the class they belong to. A condition is selected once the event is selected. In the DPE conditions are selected in the following way:

- Select a class from the list of condition classes

- Select an object instance of this class or null

    o When "null" is chosen as condition instance, then the instance is internally created and assigned when the rule is executed

- Selects either a condition method or an attribute

    o If an attribute is chosen then specify the comparison operator and value.

This forms the SCS, and this selection process can be repeated to add additional Boolean "AND" or "OR" operators. The various comparison operators that can be used with the attribute are detailed below:

Normal Comparison Operators used with Attributes: An attribute of a class can be used as a part of the SCS only if the class is derived from the *"Reactive"* class present in the LED package. The attributes should be public members of the class.

Currently only two types of attributes are supported: string or numeric. This is because of the limitation of regular expression evaluator used by the *"eval"* method of the *"Reactive"* class, to compute the result of comparison string. To this method an attribute name along with normal comparison operators and comparison value is passed. This method replaces the attribute with its value at run time and uses the evaluator to compute the result. Hence the restriction on the attribute that it has to be a public member of the class, so that it

can be accessed directly. The result of the evaluator is a Boolean object, whose value is returned by the *"eval"* method.

An attribute of type "string" allows only two comparison operators namely *"=="* and *"! ="*. Examples are shown below

- *"null ,Track.TrackNumber == "T7000" TrackNumber"*

- *"null, TrackTrackNumber != "T7000" TrackNumber"*

An attribute of type "numeric" would include all data types for integers and real numbers. The possible comparison operators are *"==, ! =, <, <=, >, and >="*. Examples are shown below:

- *"null, Track.Speed == 7000 miles"*

- *"null, Track.Speed != 7000 miles"*

- *"null, Track.Speed > 7000 miles"*

- *"null, Track.Speed >= 7000 miles"*

- *"null, Track.Speed < 7000 miles"*

- *"null, Track.Speed <= 7000 miles"*

Special Comparison Operators with Attributes: There are some circumstances wherein the user would like to do special types of comparison, such as to check if the speed of the aircraft has increased compared to the previous value or to check if the altitude is decreasing compared to the previous value or there is no change in the value of certain attributes. Therefore, some additional comparison operators for numeric attributes were introduced. These operators are described below.

- **Nochange operator**

This operator would be used when the user wants to perform some action when the value of a certain attribute has not changed compared to its previous value. Example of a SCS using this operator is shown below:

"null, Track.Speed nochange"

- **ChangesBy operator**

There are two ways this operator could be used and they are:

- Changesby value unit

Example: *"null, Track.Speed changesby 10 miles"*.

This denotes that the user wants to check if the speed has changed by 10 miles compared to its previous value.

- Changesby value %

Example: *"null, Track.Speed changesby 10 %"*.

This denotes that the user wants to perform an action when the attribute speed has changed by 10% compared to its previous value

- **IncreasesBy operator**

The two possible types are specified below:

- Increasesby value unit

Example: *"null, Track. Speed increaseby 10 miles"*

This denotes that the user wants to perform an action when the value of attribute Speed has increased by 10 miles compared to its previous value.

- Increasesby value %

Example: *"null, Track.Speed increases by 10 %"*

This denotes that the user wants to perform an action when the value of attribute

Speed has increased by 10 % compared to its previous value.

- **DecreasesBy operator**

The two possible types of this operator are

- o Decreasesby value unit

Example: *"null, Track.Speed decreasesby 15 miles"*

This denotes that the user wants to perform an action when the value of the

attribute Speed has decreased by 15 miles compared to its previous value.

- o DecreasesBy value unit

Example: *"null, Track. Speed decreasesby 15 %"*

This denotes the user wants to perform an action when the value of the attribute

Speed has decreased by 15% compared to its previous value.

A complete condition selection connected by Boolean "&&" operator is as shown:

"null, Track.conditionAltitude" && "T7000, Track.speed > 7000 miles"

5.5.3 Action Selection

Two kinds of action are available for selection. An action method and an email

option can be selected for the action part of the Rule. In the DPE actions are selected in the

following way:

- Send an email using DPE

  - o Recipient's email address should be provided

o Other information such as sender's address and mail server address necessary to send an email is provided in the configuration file.

- Select an action method

    o Select an action class from the list of action classes

    o Select an instance belonging to that class

    o Finally select one of available action methods from the selected class

The information pertaining to each rule needs to be stored because there is no mechanism in the LED to retrieve this information later for modifying the rule. Therefore, a class *"GeneralRuleObject"* is used to contain the details of a rule. An instance of *"GeneralRuleObject"* is created from this information and it internally contains a *"ConditionGeneric"* object corresponding to the condition. The *"ConditionGeneric"* object in turn creates the necessary *"RuleConditionObjects"* for each SCS. This *"GeneralRuleObject"* is stored in a data structure that contains all the *"GeneralRuleObjects"*. The *"createRule"* API of LED is invoked to register the rule with the corresponding event.

5.6 Rule Modification

Modifying the existing rules without having to recompile the code or to register again with LED was the main driving force behind separating the condition into SCS, which are represented as *"RuleConditionObjects"*.

A rule to be modified is selected from the list of rules created. The list of rules consists of all the *"GeneralRuleObjects"* present in a data structure that contains rules. Currently only modification of the condition is supported. After a rule is selected then the

condition of the rule is fetched and is presented to the user for modification. The following are the possible ways of changing the condition part of the rule

- Changing the condition method

- Changing the comparison operator

- Changing the value in an attribute comparison condition string

- Changing the attribute comparison string

- Changing a condition method to attribute comparison or vice-versa

Initially, different classes were created to handle the different types of SCS; one class to handle attribute based SCS along with a condition method and another class to handle special comparisons operators in attribute based SCS. But this implementation had a problem during rule modification. In rule modification, you can modify a SCS that contained simple comparison operator to a method containing a special comparison operator or vice-versa. To handle this case, different objects need to be created when conditions changed and update the *"ConditionGeneric"* class. This was not possible easily at run time since the object for the old SCS had to deleted and a new object for the modified SCS has to be created. A better approach was needed where there is no overhead due to objects being deleted and/or created. Therefore, only one class is used instead of two classes, and information of the class handling special comparison operator were added to the other class. In this class, for attribute based comparison strings containing special operators, instead of directly sending the string to eval method of the Reactive class, the SCS is evaluated, to reduce the special operators to normal operators and is then sent to the eval method so that it can evaluate the string properly. This approach would prevent the creation of new objects at run time when condition is modified

51

and the task of modifying the condition can be accomplished by just setting the data members with updated values. There is an additional flag present in *"RuleConditionObject"* to differentiate if it is a class or instance level attribute condition string or it is a condition method in the SCS. This flag needs to be updated accordingly if the condition object is changed between null and a specific instance or if the condition changes from an attribute-based to method-based or vice versa. After all these updating changes are done, the condition is saved and the rule has been modified successfully.

5.7   Event Creation

When an application is executing, apart from the need to modify or create rules, the user may also want to create new events. Hence the ability to create new temporal and composite events is provided. New primitive events are not provided as it corresponds to creation of method code, compiling it programmatically, and loading the class at run time. The primary goal of this editor is to support creation of rules by changing conditions at run time. Typically, we are trying to avoid accepting code as part of the interactive environment. although it is possible to do so

To create a temporal event, the user should specify a class to associate the event to. After selecting the class the absolute time when the event should be raised need to be specified. After this information is collected, this event is registered with the LED by making a call to createPrimitiveEvent of ECAAgent. Once this event is created, it is available to the user for creating the rules.

Composite events are created by initially selecting an event operator along with the required number of constituent events from the list of available events. The list of event

operators consist of "AND, OR, SEQ, PLUS, PERIODIC, PERIODIC STAR, APERIODIC, APERIODIC STAR, NOT ". Once the event type is selected, they are classified as either binary or ternary operators, and then the user chooses the corresponding events. For PLUS, PERIODIC, PERIODIC STAR operators one of the events is the absolute time string. Once all the information required for creating a composite event is collected, the event is registered with LED by invoking the *"createCompositeEvent"* API. The composite events created are associated with the class called "Composite" so that the user can look for the composite events created under this class. Therefore, the user is provided with the ability to create new events at run time.

5.8   Persist created events and rules to support reusability

When the DPE is used with an application, a number of events, conditions and rules are created interactively. A mechanism is required to input these events, conditions and rules into the system if it were to be restarted. Otherwise, the process has to be repeated all over again. To accomplish this, the system can generate a configuration file that can be used as input to the system. The configuration file contains information about the events and rules created by the DPE. It also contains information about the class, events created by the user application. This output file can be generated either in XML or text format.

CHAPTER 6

IMPLEMENTATION OF THE DYNAMIC PROGRAMMING ENVIRONMENT

This chapter discusses the implementation details of the DPE. It is organized as follows: First, the data structures required to create, modify rules and create events are discussed. Second, parsing of input files in both formats XML and TEXT to store the information in the data structures is discussed. Finally, the objects created to represent the condition part of the rule and to handle the complete information about the rule are described.

6.1 Data structures used in DPE

The information from the configuration files is stored in the following data structures. Some of these data structures store the data in DPE-defined objects. These objects belong to either *"EventObject"* or *"AttributeObject"* class.

EventObject

The information corresponding to each event is stored in an instance of class *"EventObject"*. The various data members of *"EventObject"* are shown in Table 6.1. This class is used to store information regarding the various events types. The possible event types are:

- Class-Level primitive event

- Instance-Level primitive event

- Composite Event.

Table 6.1 : Data members of EventObject class

Data member	Description
SeventType	Used to store the event type
SeventName	Used to store the event name
SCreatedBy	Used to differentiate if the event was created by DPE or user application
SEventSignature	Used for primitive events only .It is the method signature of the event
VecParamters	Used to store the names of parameters that will be inserted into the parameterlist when the primitive event is raised
SEventInstanceName	Used for instance-level primitive event only. It is used to store the name of EventInstance
CompositeEventType	Used to store the composite event operator.
VecEvents	Used to store the constituent events of a composite event

The various composite event operators permitted are: AND, OR, NOT, SEQ, PLUS, PERIODIC, PERIODIC STAR, APERIODIC, APERIODIC STAR. Since different types of events are represented in this class, all the data members are not intitalized intially. For each event, sEventType, sEventName, sCreatedBy, are initialized. When the sEventType is a class-level primitive event then the following data members are used:

- sEventSignature,
- vecParameters

When the sEventType is an instance-level primitive event; the following data members are used:

- sEventSignature,
- vecParameters
- sEventInstanceName

When the sEventType is composite event then following data members are used:

- CompositeEventType

- vecEvents

AttributeObject

Complete information corresponding to each attribute is stored in an instance of "*AttributeObject*" class. The data members of the "*AttributeObject*" are shown in Table 6.2. The various types of attribute that are currently supported are: String and Numeric. The sEventName specified would correspond to name of an event that is associated with the attribute and an instance of "*EventObject*" corresponding to the event name should exist.

Table 6.2   Data members of AttributeObject

Name	Description
sAttributeName	Specifies the name of the attribute
sAttributeType	It specifies if the type of the attribute
sAttributeUnits	It specifies the units used for attribute
sEventName	It specifies the name of event associated with attribute

Data Structures

The list of data structures present in DPE is shown below in Table 6.3. The data structures are described below:

HtInstanceNames_InstanceObject: This is used to store the list of names of instances of each class specified in the configuration file and their corresponding objects. The object stored is the reference to the corresponding object used in the user application. In java,

the objects are passed by reference, and hence there is no need to synchronize with this object when the object in the user application is updated.

Table 6.3 : Data structures present in DPE

Data structure	Description
htClass_InstanceNames	This is used to store the Instance names corresponding to each class
htInstanceNames_InstanceObject	This is used to store the object provided by the user application corresponding to its name
htEvents	This is used to store an event name and the corresponding *"EventObject"*
htEvent_Domains_Types	This is used to store every event class and list of available events types in the corresponding class.
vecConditions	This is used to store all the available conditions methods
htAttributes	This is used to store all the available attribute of all classes and the corresponding Attribute Object
htCondition_Domains_Types	This is used to store every condition class and the list of all available condition methods and attributes for the class
vecActions	This is the list of all action methods
sMailServer	This corresponds to the name of Mail Server used to send the e-mail
sFromAddress	This corresponds to the From address used to send the e-mail

HtClass_InstanceNames: This is used to store the list of available instances for each class specified in the configuration file. It is implemented as a hashtable with the key as name of a class and the value as a vector of names of instances of this class. For example, the htClass_InstanceNames is shown in   Figure 6.1. The class *"Track"* has two instances whose names are *"T7000"* and *"T7001"*.
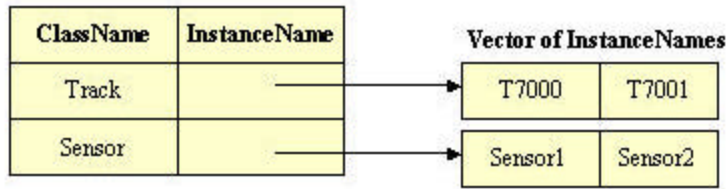
Figure 6.1 : HtClass_InstanceNames

HtEvents: This is used to store information about each event defined in LED. It is implemented as hashtable with the key as "event name" and the value as the corresponding "EventObject". For example, for the sample configuration file it as shown in Figure 6.2. It contains two events whose names are *"SpeedEvent"* and *"SensorEvent"* and corresponding *"EventObjects"* .
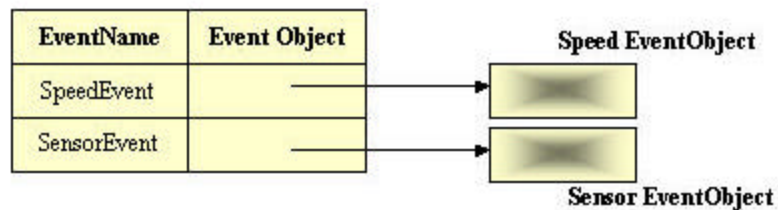
Figure 6.2: HtEvents

HtEvent_Domains_Types: The time required for finding the event classes available and the list of events available for each class from htEvents would be considerable for large number of events. Therefore to reduce the time taken, htEvent_Domains_Types was created. It is implemented as a hashtable with the key as the "class name" and the value as a vector of available events for that class. As a result, the event classes and the list of events available for that class can be easily referenced with out parsing the htEvents. For example, for the sample configuration file, htEvent_Domains_Types is as shown below in Figure 6.3.For the class *"Track"*, the available events are *"SpeedEvent"* and *"AltitudeEvent"*

58
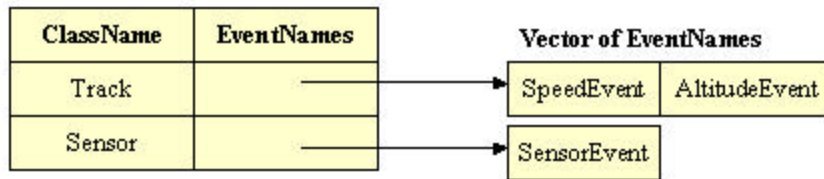
Figure 6.3 HtEvents_Domains_Types

VecConditions: This is used to store all the available condition methods present in all the classes. This is implemented as a Vector with each element stored as a combination of class name and the method name. For example, for the sample configuration file, the content of "vecConditions" is shown in Figure 6.4. It contains two elements namely *"Track.conditionTrackNumber"* and *"Condition.conditionTrue"*.
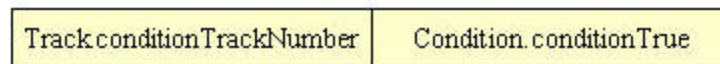
Figure 6.4 VecConditions

HtAttributes: This is used to store the information about each attribute present in each class. This is implemented as a hashtable with key as a combination of class name and attribute name and the value as the corresponding attribute object. For example, for the sample configuration file the htAttributes would contain data as shown in Figure 6.5.
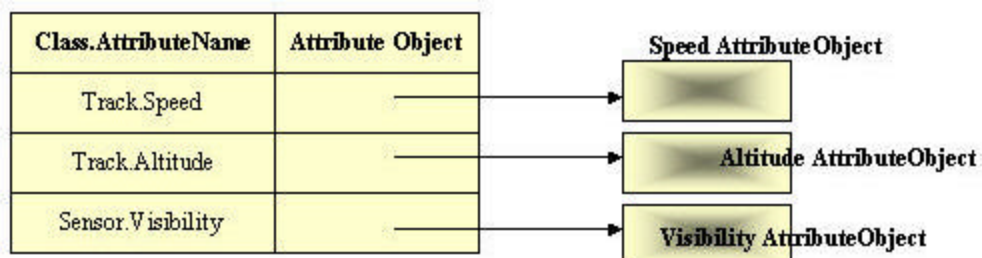
Figure 6.5: HtAttributes

59

HtCondition_Domains_Types: The time required to find the available condition classes and the list of condition methods and attributes present in each condition class would be significant since this information has to be gathered from *"htAttributes"* and *"vecConditions"*. Therefore, *"htCondition_Domains_Types"* has been implemented. It is a hashtable with the key as the name of class and value as a vector of the condition methods and attributes of the class. For example, for the sample configuration file, the htCondition_Domains_Types contains data as shown in Figure 6.6.The class *"Track"* contains *"Track.Speed","Track.Altitude"* and *"Track.conditionTrackNumber"* as the available condition types.
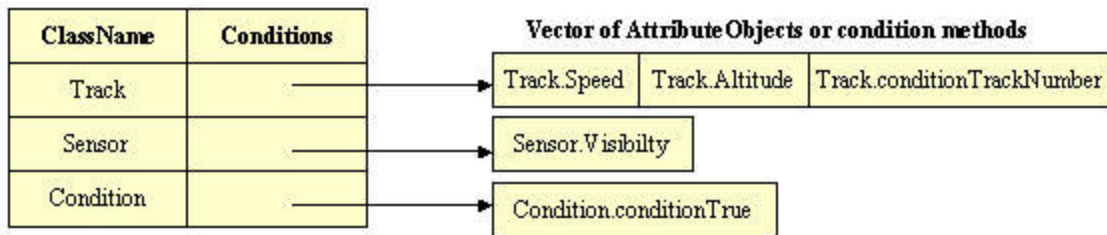
Figure 6.6: HtCondition_Domains_Types

VecActions: This contains the available action methods of all classes. It is implemented as vector, with each element as a combination of the class name and method name. For example, for the sample configuration file it is as shown in Figure 6.7
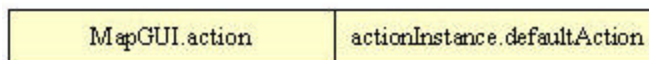
Figure 6.7 vecActions

6.2   Implementation of input configuration file

The input configuration file is required by the DPE to provide information about the user application to create rules and events. Therefore when the DPE is started, the location of

the input configuration file is specified. The file can be specified in either in Text/XML

format. The structure of the input file is shown in the Figure 5.4**.**

Table 6.4: Events defined in SAX Parser

Event	Description
Start Document	Invoked when the beginning of the document, it is found by the parser
End Document	Invoked when the end of the document, it is found by the parser
Start Element	The SAX parser will signal the start of element by invoking startElement() along with some information
End Element	The SAX parser will signal the start of element by invoking endElement() along with some information

Parsing of XML document

There are two types of XML interfaces: tree-based (DOM) and event-based (SAX.).

Event-Based API provides a simpler, lower-level access to an XML document. It is possible

to parse large documents, with sizes greater than the system memory size. Event-Based APIs

report parsing events directly to the application through callbacks. Application implements

handlers to deal with events and constructs a data structure using callback event handlers.

The various events are specified in Table 6.4.

The "*ReadXMLConfig*" class handles parsing of the XML file. This class uses a Xerces

SAX parser to parse the file and extends the DefaultHandler class. The DefaultHandler Class

provides the default event handler methods.

import org.apache.xerces.parsers.SAXParser;

import org.xml.sax.*;

   To use the Xerces SAX interface, a SAXParser object must be instantiated

       XMLReader xr = new SAXParser();

61

You then need to register a SAX event handler. The SAX2 API now includes a ContentHandler interface (org.xml.sax.ContentHandler.) This interface includes callback methods for all significant parsing events, including: start and end of document, start and end of elements, and character data.

xr.setContentHandler(this); // Register the event handler,

xr.setErrorHandler(this); // Register an error event handler

xr.parse(new InputSource(sFileName)); // Parse any XML file

The endElement event is of interest to gather the information from the configuration file. The task performed by the parser when each of the following tags of the configuration file are encountered are specified below:

ClassName: When this tag is encountered, it specifies the name of the class whose properties are provided in the configuration file. The value is stored temporarily in a String since it is required for most of initialization. "Track" is an example of a class name.

InstanceName: This specifies one of the instances available for this class. It is stored in htClass_InstanceNames as a part of the value corresponding to a key corresponding to the class name. The value is a vector of instance names. For example, for the class Track, the names of existing instances name in htClass_InstanceNames are "T7000", to this vector "T7001" is added. Now there are two instances "T7000", "T7001" for the class "Track"

Attribute: Information corresponding to the attribute namely its name, type units and the event associated is collected and stored in an attribute object. This attribute object is added to htAttributes as a value corresponding to a key formed by the combination of the class name and the attribute name. For example, in the sample configuration file, for an

62

attribute, the following information *"Speed, Numeric, mph, SpeedEvent"* is stored in an attribute object and is stored in htAttributes as a value for the key *"Track. Speed"*.

ClassLevelPrimitiveEvent: Consider an example where the following information *"SpeedEvent, void setSpeed(), NewSpeed"* is collected for a class-level event. The information is parsed and the corresponding information about the event name, event signature and new parameter name are initialized and an instance of *"EventObject"* is created using this information. The event type is assigned the value *"ClassLevelPrimitiveEvent"* and "sCreatedBy" parameter is assigned the value "User". If eventType contains a value *"ClassLevelPrimitiveEvent"*, it specifies that the information is about a class-level primitive event. The value "User" for "sCreatedBy" indicates that the user created the event. This instance of *"EventObject"* created is stored as a value in htEvents with event name *"SpeedEvent"* as the key. Also, from the htEvents_Domain_Types, the value of the key corresponding to this class "Track" is retrieved. The value is a vector of all events contained in this class that is updated with this event name *"SpeedEvent"*.

InstanceLevelPrimtiveEvent: The information provided contains the same set of information as in a class-level primitive event, but also has an additional value that specifies the name of event instance. For example, consider *"T7001, T7001TrackEvent, void setTrackNumber(), NewTrackNumber"* as the data, this information corresponding to the event instance, event name , event signature and new parameter name is used to create an *"EventObject"*. The event type is assigned *"InstanceLevelPrimtiiveEvent"* and "sCreatedBy" parameter is assigned the value "User". This value of event type would indicate that the *"EventObject"* specifies the information about an instance level event. This

information is updated in the htEvents as well as in the htEvents_Domains_Type as mentioned before.

CompositeEvent: The information specified for a composite event is: the composite event name, the composite event operator and the list of constituent events. For example *"TrackNumberSpeedEvents, AND, TrackNumberEvent, SpeedEvent"*, there are two constituent events namely *"TrackNumberEvent"* and *"SpeedEvent"*. This information is used to create an *"EventObject"* and the event type is assigned a value "CompositeEvent". The parameter "sCreatedBy" is assigned a value "User". This *"EventObject"* is updated in the hashtables- htEvents and the htEvents_Domains_Type.

Condition: The information retrieved gives the details about the method signature. This is appended to the class name and is stored in the vector "vecConditions". After storing in "vecConditions", it is stored in htCondition_Domains_types corresponding to the class. For example, *"conditionTrackNumber"* is the method signature; it is updated with a value *"Track.conditionTrackNumber"* in vecConditions and for a key "Track" in htCondition_Domains_types as specified.

EventsCreated: When this tag is encountered, a flag is set indicating that any further ClassLevelPrimitiveEvent or CompositeEvent tags encountered would have the *"sCreatedBy"* parameter of *"EventObject"* set to "DPE". This would indicate that the event is created by DPE and define the events in LED.

MailServer: This specifies the address of the mail server used to send an email and it is stored in Constants.sMailServer. For example, mail.uta.edu, would be the address of mail server.

FromAddress: To send an email, the sender-address needs to be specified. This information is specified as FromAddress. This value is then stored in Constants.sFromAddress. For example, *"calvin@calvin_hobbes.com"*, would be the email address used to specify the sender email-address

RulesCreated: This tag is used to indicate to the DPE to create rules. A flag corresponding to this tag is set to true when this tag is encountered.

ClassRule: The information required to create a class level rule, the event name, ActionObject name, Action String, ConditionObject name, ConditionString, is speciifed. For example, if *"NewSensorEvent, null, varakala@omega.uta.edu, null, Sensor.conditionSensorNumber"* is given as the information of a required class rule. The Condition object can be null. Here the name of ActionObject specified, as null would indicate to the DPE to use the default action provided (i.e., email as the action) and the condition string would provide the destination email address. If the action object name is specified then it is used to create the rule. After all the information is collected, it is stored in an instance of *"GeneralRuleObject"* and then a call to LED is made to create the rule.

InstanceRule: For an instance rule, the only difference from class level rule is that an Event instance is specified as part of the rule. For example, consider *"Track.T7001, SpeedEvent, T4001, Track.action., null, Track.dSpeed increases 0 mph && null, Track.sTrackNumber == T7001 TrackNumber"*. Here, Track.T7001 corresponds to an event instance, T4001 corresponds to an action instance name, Track.action is the action method. The condition string is *"Track.dSpeed increases 0 mph && null, Track.sTrackNumber == T7001 TrackNumber"* and the condition object is null. The rule uses an action method for the

action part of the rule. Once all this information is collected, it is stored in an instance of *"GeneralRuleObject"*. A call is made to the LED to create the instance rule.

The text Configuration file is parsed and the same set of steps is performed as mentioned for a XML file.

6.3   Implementation of Email feature as default action

Email is provided as the default action for a rule. This feature is implemented using Java Mail API provided by Sun Microsystems. Java Mail API is platform and protocol independent mail/messaging solution [23]. A simple java program to send an email is as shown in Figure 6.8. In DPE, the information about MailServer and sender address is stored in Constants.sMailServer and Constants.sFromAddress. The action part of the rule would provide the "recipient address" for the email and the message would correspond to information about this Rule and ListOfParameterLists passed to action by the LED during rule execution.

A class *"EmailClient"* is used to implement the feature of sending email and is similar to the sample program shown. An instance of this class is created for action part of every rule that chooses email as the action and is also assigned the recipient address. After the condition of Rule evaluates to true, the sendEmail method is invoked. The LED passes information about the *"ListsOfParameterLists"*. "ListOfParameterLists" content is printed as the message along with the information about the rule.

```java
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class MailExample {

    private String sMailServer;
    private String from;

    MailExample(String sMailServer,String from){
        this.sMailServer = sMailServer;
        this.from = from;
    }

    public void sendEmail(String recipient,sSubject,sMessage) throws Exception {
        String host = sMailServer;
        InternetAddress from = new InternetAddress(sFromAddress);
        InternetAddress recipient = new InternetAddress(strToAddress);

        // Get the system properties
        Properties props = new Properties();

        // Setup default parameters - the protocol, the host, and port #
        props.put("mail.transport.protocol","smtp");
        props.put("mail.smtp.host",host);
        props.put("mail.smtp.port","25");

        // Create session object based on the Properties
        Session mySession = Session.getInstance(props);

        // Create a message from the session
        MimeMessage myMessage = new MimeMessage(mySession);

        //Set the from field of the message.
        myMessage.setFrom(from);

        //Set the to field of the message.
        myMessage.addRecipient(Message.RecipientType.TO,recipient);

        //Set the subject of the message.
        myMessage.setSubject(this.sSubject);

        //Set the content of the message.
        myMessage.setText(this.sMessage);

        // Use a Transport to send the message.
        javax.mail.Transport.send(myMessage);

    }
}
```

Figure 6.8 Simple Program to send an Email using Java Mail API

6.4   Implementation of Condition

When the condition is selected for the rule, it can consist of many simple condition strings connected together by Boolean operators. "ConditionActionGeneric" class implements the complete condition string and internally contains RuleConditionObject that implements the simple condition string

6.4.1   RuleConditionObject

The complete condition can contain either one *"RuleConditionObject"* or many *"RuleConditionObjects"* connected by operators && and ||. This *"RuleConditionObject"* class contains information about the class, name of method to be executed and the actual parameter to pass to the method. The data members are mentioned in the following Table 6.5. The data members are designed in such a manner that the modification of simple condition string can be easily implemented. The important member functions are specified in Table 6.6. The RuleConditionObject uses reflection method in Java to invoke a method for a specified object along with the actual parameters.

6.4.2   ConditionActionGeneric class

The "ConditionActionGeneric" class handles the condition of the rule. The complete condition string is broken into a vector of *"RuleCondtionObjects"* and a vector of Boolean operators. The data members of the class are represented in Table 6.7. The important member functions are represented in Table 6.8. The "ConditionActionGeneric" class has a method *"condition"* which is specified as the name of the condition method while the rule is being created. This method computes the final Boolean result after it evaluates the result from each of *"RuleConditionObjects"* invoked and applying the corresponding binary operators.

Table 6.5: Data members of RuleConditionObject

Data Member	Description
SclassName	Name of the class containing the attribute or condition method
IobjectType	Type of condition: a condition method or a attribute
sInstanceName	Name of object corresponding to this condition part, can be null
Oinstance	Reference to condition object
sMethodName	Name of the method if a condition method is specified else it is equal to "check"
sAttributeName	Name of attribute used in the simple condition string
sAttributeType	Type of the attribute, can be either NUMERIC or STRING
sComparisionOperator	Stores the comparison operator used in attribute based condition
sComparisionUnit	Stores the comparison unit either % or the attribute unit. Used in attribute condition
sComparisionValue	Stores the comparison value used in attribute condition
sEventSignature	The event signature of the event associated with the attribute used
sNewParameterName	The name of parameter inserted by the primitive event when raised

Table 6.6: Member functions of RuleConditionObject

Method	Description
setClassMethodNames	Initializes the class name, method and parameters required for attribute condition
setEventSignature_NewParameterValue	Used to initialize the event signature of event associated with the attribute, new Parameter name and the attribute type
setMethodParams	Intialize the object instance, actual parameter for the method "sMethodName", construct the compare string
invokeMethod	Used to invoke the method corresponding to sMethodName of the instance object using Java Reflection
compareValueString	Used to simplify the special comparison condition string into one using the normal comparison
ResetInstance_ConditionString	Used to modify the simple condition string, it compares the new class, instance, method or attribute comparison string and updates all the modified parameters and object type if necessary

Table 6.7: Data members of ConditionActionGeneric class

Data members	Description
vecConditionObjects	This is a vector of RuleConditionObjects. The order of elements stored in the vector corresponds to the order as in the condition.
vecOperators	This is a vector of boolean operators. They are stored in same order as present in the condition.

Table 6.8: Member function of ConditionActionGeneric class

Method	Description
initializeVectors	This method is invoked when an instance is created. A condition string provided is separated into two sets. One set contains the *"RuleConditionObjects"* and other set is boolean operator.
addTovecConditionObjects	This is invoked to by initializeVectors, to create RuleConditionObject corresponding to each simple condition
condition(ListOfParameterLists paramlists)	This method is specified as the condition method for every rule created by DPE. LED invokes it when this rule is fired.
modifyCondition (Vector vecInstanceObjects, String sConditionString)	This method is used to modify the existing condition of a rule. It updates the two vectors with the new values from the modified condition.

6.5   Implementation of GeneralRuleObject

"GeneralRuleObject" class is used to store the information of a rule (i.e., Event, Condition and Action). The event part of rule consists of the event name and event instance. The condition part of the rule is instance of *"ConditionActionGeneric"* class and the action part of the rule consists of the action method and the object corresponding to that action method. Each *"GeneralRuleObject"* is stored as the value corresponding to the name of the

rule, which is a key for Constants.htRuleName_GeneralRuleObject. The data members of the

GeneralRuleObject are as shown in Table 6.9.

Table 6.9: Data members of GeneralRuleObject

Data members	Description
sEventName	The name of event specified
sEventInstanceName	The name of event instance specified
conditionActionGenericInstance	Condition of the rule
sAction	The action method
sActionInstance	Name of action instance specified

The information about a rule is stored in an object so that the condition part of the

rule can be accessed later for modification, and also to keep track of the information of the

various rules created. After the *"GeneralRuleObject"* is stored in the hashtable, the

"createRule" API of LED is called to create the rule.

6.6   Mechanism in DPE that responds to when a rule is triggered in LED

When an event is raised, the rules associated with it are triggered in the LED. When a

rule is triggered, the condition is evaluated. Here the condition is an instance of

"ConditionGeneric" class. The *"condition"* method of this object is invoked and the *LED*

passes "ListOfParameterLists" as the argument. This method calls the *"invoke"* method of

constituent *"RuleConditionObjects"*. When the *"invoke"* method is executed then the object

instance within the *"RuleConditionObject"* invokes either the condition method or evaluates

the attribute comparison.

6.6.1   Assigning the Object Instance in *"RuleConditionObject"*

The "objectInstance" data member in *"RuleConditionObject"* can either be a null

object or a specific object. When a null instance is specified in the SCS as the objectInstance,

71

an object instance needs to be assigned. When the event is raised, the object that raises the event is passed as a part of the parameter list. The number of parameter lists passed, in *"ListOfParameterLists"* would depend on the number constituent events. The *"ListOfParameterLists"* is passed onto to *"RuleConditionObject"* from the *"ConditionGeneric"* class. Now depending on the condition type i.e. a method or an attribute, it is handled differently.

The pseudocode is shown below for assigning the "objectInstance" in the Rule ConditionObject

*Line 1 : **RuleConditionObject.setMethodParams(ListOfParameterLists pLists)***

Line 2 :{   If(ConditionType == ConditionMethod){

Line 3 :   methodParams = pLists

//Assigning the formal parameter required by the condition method

}

Line 4 :   If(instance == null){

Line 5 :   if(pLists.size == 1)

Line 6 :   instance = pLists.getFirst().getEventInstance();

Line 7 :   else{

Line 8 :   If(ConditionType == ConditionMethod){

Line 9 :   instance = pLists.getFirst().getEventInstance();

Line 10 :   }else{

Line 11 :   Get the event signature of the event associated with

attribute

Line 12 :   *if(eventsignature exists in parameterList2 of pLists)*

Line 13 :   *instance = parameterList. getEventInstance();*

Line 14 :   *else*

Line 15 :   *instance = pLists.getFirst().getEventInstance();*

Line 16 :   *create the CompareString()*

     } // end of ConditionType == attribute

    }// end of outer else

   }

This method is invoked by the condition method of *"ConditionGeneric"*. The method signature is shown in Line 1, containing an argument *"ListOfParameterLists"* whose value is set by the LED. If the conditionType is equal to condition method, then the argument of condition method is assigned the value *"i"* as *"pLists"*. When the objectInstance is null, then it has to be assigned a value. For a condition method, if the number of parameter lists is equal to one, then the *"EventObject"* from that *"ParameterList"* is assigned as the object instance of the *"RuleConditionObject"* as shown in Line 6. If the number of parameter list is more than one, then the first parameter list from the list of parameter lists is chosen and *"EventObject"* present in it is assigned as the object instance.

For an attribute condition, if the number of parameterlists is equal to one then the *"EventObject"* present in the *"ParameterList"* is assigned as the object instance. If the number of the *"ParameterLists"* is greater than one, then the event associated with this attribute is retrieved from the corresponding *"AttributeObject"*. All *"AttributeObjects"* are

73

stored in a hashtable corresponding to their class name and attribute name. The corresponding eventobject is found using the event name from a hashtable, where all *"EventObjects"* are stored... The method signature of the event is extracted from the *"EventObjects"*. This method signature is matched with the method signature present in each *"ParameterList"*. When a match is found, the eventInstance present in that *"ParameterList"* is assigned as the object instance. Otherwise, when no match is found, the *"EventObject"* from the first *"ParameterList"* is assigned as the object instance.

6.6.2 Condition Method used as Condition String

A method of a class to be used as a condition method of a SCS should have an argument of type "ListOfParameterList" and return a Boolean value according to the design specified by LED for condition part of the rule. When the "invoke" method of a constituent *"RuleConditionObject"* is executed and this would in turn execute the selected condition method on the *"objectInstance"* in *"RuleConditionObject"*. The result from this method is passed on to corresponding *"ConditionGeneric"* object's condition method.

6.6.3 Normal Comparison Operators used with Attributes

An attribute of a class can be used as a part of the SCS only if the class is derived from the *"Reactive"* class present in the LED package. The attribute should be a public member of the class.

Currently only two types of attributes are supported: string or numeric. This is because of the limitation of regular expression evaluator used by the *"eval"* method of the *"Reactive"* class, to compute the result of comparison string. To this method an attribute name along with normal comparison operators and comparison value is passed. This method

74

replaces the attribute with its value at run time and uses the evaluator to compute the result. Hence the restriction on the attribute that it has to be a public member of the class, so that it can be accessed directly. The result of the evaluator is a Boolean object, whose value is returned by the *"eval"* method. Examples of some of the operators supported normally by the regular expression evaluator in the eval method are:

- *"null ,Track.TrackNumber == "T7000" TrackNumber"*

- *"null, Track.Speed >= 7000 miles"*

In the above examples, only the string containing the attribute name, comparison operator and value are passed as the argument to the *"eval"* method of *"RuleConditionObject"*. The string passed for some of the above examples would be

- *TrackNumber == "T7000"*

- *Speed >= 7000*

6.6.4   Special Comparison Operators with Attributes

When the special comparison operators are used, the SCS has to be altered in such a manner that the same *"eval"* method could be used as in the case of normal condition comparison operators. But before changing, reference to the new attribute value of the object is required.

To achieve this, *"EventObject"* corresponding to this attribute is retrieved. After the corresponding *"EventObject"* is retrieved, the name of the parameter that has been inserted into the *"ParameterList"* when the event was raised is extracted. The value of the parameter would be the new value and the old value of the attribute is obtained from the object at run time from *"ParameterList"*. The value corresponding to the assigned parameter name is

75

extracted from the *"ParameterList"*. This value is substituted in the SCS and the corresponding arithmetic operations are performed and the result is computed and compared to the attribute. This SCS is now in a format similar to the SCS containing normal condition operator. This new SCS is used when the *"eval"* method of the *"EventObject"* is invoked

6.6.5   Computing the result of condition after evaluation of constituent SCS

The *"ConditionGeneric"* object needs to compute the result of the condition from the constituent SCS and needs to apply the Boolean operators in the same order as specified in the condition. The pseudocode is shown below:

*Line 1 : **boolean condition(ListOfParameterLists pLists)***

Line 2 :{   result = "";

Line 3 :   j = 0;i =0;

Line 4 :   for(i < vecRuleObjects.size()){

Line 5 :   object = vecRuleObject.get(i) ;

*   //Get the ith RuleConditionObject from vecRuleObject*

Line 6 :   object. setMethodParams (pLists);

*   //Execute the method and pass pLists as the actual parameter*

Line 7 :   result = result + object.invoke();

*   //Append the result from invoke method when it is executed*

Line 8 :   if(j < vecOperatorss.size()){

*   result = result + vecOperators.get(j);*

*   //Get the jth Boolean operator from vecoperators and*

*   append it to result*

76

j++;   }

}//Evaluate the string and return Boolean result

Line 9 :   return eval(result);   }

Line 1 in the pseudocode shows the method signature that has to be followed by any condition method in LED. When the control is transferred to this method, a variable to store the result is initialized as shown in Line 2. In Line 4, a for loop with a variable "i" is initialized to execute through the complete list of RuleConditionObjects. The ith element, a *"RuleConditionObject"*, is retrieved from the vector as shown in Line5. The "*setMethodParams*" method of *"RuleConditionObject"* is invoked passing the parameter *"pLists"*. This method is responsible for assigning the *"objectInstance"* in that *"RuleConditionObject"* and also if a special comparison operator is used, the to convert it to a string which would be in the form of a SCS consisting of simple comparisons operators. This is represented in Line 6. After the setting the parameters,Line 7 shows that *"invoke"* method is executed . This method is responsible for executing the corresponding method and the result returned is appended to result string. If they are more elements present in the vecOperators, then the jth element is retrieved and it is appended to the result string. This is represented in Line7. If there were no more elements present in the *"vecRuleObjects"*, then this result string is evaluated and its result is returned as shown in Line8. If this result is true then the LED executes the action corresponding to the rule.

6.7   Rule Modification

When the new condition string is received from the user, it is passed to the corresponding instance of *"ConditionGeneric"* class, which in turns sends it the

"RuleConditionObject". The pseudocode for handle the modification of SCS in a *"RuleConditionObject"* is shown below:

*Line 1: **void resetInstance_ConditionString (sNewClassName, sNewInstanceName ,***

sNewMethod_AttributeName)

Line 2:{   Find the type of new condition string and assign it newConditionType

Line 3:   if (oldConditionType == new ConditionType){

Line 4:   if(oldConditionType == Attribute)

Line 5:   {   Compare and Update values that have changed

Line 6:   if(attribute changed)

Line 7:   { update the newParameterName and eventSignature}

Line8:   }else{ //ConditionType == Method

Line 9:   update the method name

}//end of inner else

Line 10:   }else{//condition has modified from an attribute to method or vice versa

Line11:   Update classname Method or Attribute,ConditionType, InstanceName

}//end of else

Line12:   if (instance!= newInstanceName)

Line13:   update the instance name

}

In Line 1, the method signature is shown with the arguments sNewClassName, sNewInstanceName , sNewMethod_AttributeName. The three arguments correspond to the name of class, the instance name and the SCS. The SCS could either be attribute- based

comparison string or a condition method. Once the control has been passed to this method, the condition type of new SCS is found and assigned to "newConditionType", as shown in Line 2. The "newConditionType" is compared to "oldConditionType". If they are same, then the "oldConditionType" is compared to check if it is attribute based, as shown in Line 4. If the condition string is attribute based, then in Line 5, the parameters that have been changed are found by comparison and the changed values are updated. In Line 6,it is checked if the attribute has been changed. If it has changed, then in Line 7 the newParameterName, the name of the parameter that would be inserted when the event is raised and the event signature parameters of *"RuleConditionObject"* are updated. In Line 8, if the oldConditionType is a condition method and oldConditionType is same of newConditionType then the method name is updated as shown in Line 9.

If the oldConditionType is not equal to newConditionType as represented in Line 10, then based on the type of condition the conditionType is updated and the method or attribute parameters are initialized. If the instance name associated with SCS has changed then the instance name is updated.

6.8   Generation of XML file as output

The DPE provides an option of generating an XML file which contains information about the events and rules created by the DPE in addition to the information gathered from the input configuration file, that contained data of the input configuration file specified. The XML file generated is validated so that XML file when parsed follows the syntax specified.

The important steps involved in generation of an XML file is specified below:

```
import java.io.*;
import org.apache.xml.serialize.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

//To generate an XML file, first create an output file
FileOutputStream fileOutputStream = new FileOutputStream(sFileName);

//Create an instance of "OutputFormat" which is used to select the suitable //serializer and
determine how the document should be formatted on output.
OutputFormat outputFormat = new OutputFormat("XML"," UTF-8",true);

//Set the DTD file that will be used for validating the XML generated.
 outputFormat.setDoctype(null," RuleEditorDtd.dtd");

// Instantiate an XML serializer for SAX serializing.
//Since an output stream is used, the encoding is taken from the output format (defaults to
UTF-8)

XMLSerializer serializer = new XMLSerializer(fileOutputStream,outputFormat);

//Get the content handler from XMLSerializer
ContentHandler  contentHandler = serializer.asContentHandler();

//Include a start Document tag in the output file
contentHandler.startDocument();

//For each element to be included in the output file, include the start element tag    //followed
by the text and ending with the end element tag
contentHandler.startElement("","","ClassName",new AttributesImpl());

// AttributesImpl is used to include attributes in the element,there is no attributes
//so a new instance is created and added for element added to the xml file
contentHandler.characters(cTempInstanceName,0,cTempInstanceName.length);
contentHandler.endElement("","","ClassName");

//output the end document tag
contentHandler.endDocument();

//Finally close output file
fileOutputStream.close().
```

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1   Conclusion

This thesis proposes an interactive approach for supporting the creation of rules and events dynamically at run time, which is critical for several monitoring application using LED. It achieves this without recompiling and restarting the application. The DPE provides a generic set of classes to handle creation, management, and execution of rules. This set of classes is application-independent making the system useful for any application. The user application provides the necessary class and event information to the DPE through a configuration file. The DPE uses this information to create, modify and manage rules. The DPE supports creation of new composite and temporal events. Also, a default action- sending email is provided. A user-friendly interface is provided to perform these tasks.

Chapter 1 defines the problem and explains the motivation for this thesis. Chapter 2 describes the related work in active systems and rule engines. Chapter 3 covers the Event Semantics and Chapter 4 gives an overview of the LED and the important API required for creating temporal, composite events and class or instance level rules. Chapter 5 provides details of various design issues involved in in the development of the dynamic programming environment. Chapter 6 deals with implementation issues encountered during the course of the thesis.

7.2   Limitation and Future Work

The DPE proposed in this thesis only supports creation of new temporal and composite events. It does not support the creation of new primitive events. It is possible to accept a method that corresponds to LED specifications, compile the code programmatically and load it at runtime. Modification of an existing event may require a custom class loader, as the default Java class loader does not seem to reload an existing method.

Current configuration input does not specify details of the GUI. One extension would be to generalize the GUI and make it customizable for each application by accepting some layout and form information. It would also be useful to support application specific graphics in addition to the rule editor on the screen.

In addition to LED, we have a global event detector (or GED) [24] that can be used for distributed applications. Rules can be defined using events in one or more applications. It is also possible to combined local and global events into composite events. A similar interactive rule editor for supporting distributed environment would be useful.

APPENDIX A

WRITING SAMPLE APPLICATION TO USE DPE

First, to write any application that uses LED and DPE the following jar files are required namely:

Sentinel.jar ;// The jar that contains the API's of LED

FESI.jar ;//The jar file used by Reactive class in LED for string evaluation

DPE.jar;// this jar file consists of the API's of DPE;

Xerces.jar; //This provides the xml parser

mail.jar;// The following jar file and activation.jar file is required for Java Mail API

activation.jar;//

Set the classpath to include the above mentioned jar files.

A sample application, that uses LED and a class that has been used is shown below with the attributes that would be used for a condition, the important methods, which include some event methods, and a condition method. The configuration file written for this application is also shown. Also a class is shown that defines some events for this application.

```
//The necessary packages imported
import sentinel.led.Reactive;
import sentinel.led.*;
import java.util.*;

//The class has to extend Reactive class of the Sentinel package, if it wants to use it data members in condition
public class Track extends Reactive  {

   public String sTrackNumber = "";
   public String sLatitude = "0";
   public String sLongitude = "0";
   public double dSpeed =0;

   public Track(){
      super("");
   }
   public Track(String sNewTrackNumberIn) {
      super(sNewTrackNumberIn);
      sTrackNumber = sNewTrackNumberIn;
      setNewTrackNumber(sNewTrackNumberIn);
   }

   public void setNewTrackNumber(String sNewTrackNumberIn){
         EventHandle[] theHandle =
                         ECAAgent.getDefaultECAAgent().getEventHandles("NewTrackNumberEvent");
         ECAAgent.insert(theHandle,"NewTrackNumber", sNewTrackNumberIn);
         System.out.println("NewTrackNumberEvent Raised for "+sTrackNumber);
         ECAAgent.getDefaultECAAgent().raiseBeginEvent(theHandle,this);
   }
// An event method, which raises the event "SpeedEvent"
 public void setSpeed(double dSpeedIn)
 {
      EventHandle[] theHandle = ECAAgent.getDefaultECAAgent().getEventHandles("SpeedEvent");
      ECAAgent.insert(theHandle,"NewSpeed", new Double(dSpeedIn));
      ECAAgent.getDefaultECAAgent().raiseBeginEvent(theHandle,this);
      System.out.println("SpeedEvent Raised for "+sTrackNumber);
      this.dSpeed = dSpeedIn;
 }

//This is a overloaded method of the Reactive class
public boolean check(String sExp){
      return super.check(sExp);
}
//A condition method
public boolean conditionTrackNumber(ListOfParameterLists parameterLists){
      System.out.println("In conditionTrackNumber method of Track");
      Enumeration enum = parameterLists.elements();
      while(enum.hasMoreElements()){
        ParameterList pList = (ParameterList)enum.nextElement();
        pList.print();
      }
     return true;
   } }
```

```
import sentinel.led.*;
// A class that defines the events with the LED
public class PE {

    public PE() {
    }

    public void defineEvents(){
        ECAAgent.getDefaultECAAgent().createPrimitiveEvent("NewTrackNumberEvent", "Track",
        EventModifier.BEGIN, "void setNewTrackNumber()");

        EventHandle trackNumberEventHandle = ECAAgent.getDefaultECAAgent().createPrimitiveEvent(
        "TrackNumberEvent",  "Track", EventModifier.BEGIN," void void setTrackNumber ()");

        EventHandle speedEventHandle = ECAAgent.getDefaultECAAgent().createPrimitiveEvent(
        "SpeedEvent",  "Track", EventModifier.BEGIN," void setSpeed()");

        ECAAgent.getDefaultECAAgent().createCompositeEvent(EventType.AND,
                    "TrackNumberSpeedEvents",trackNumberEventHandle,speedEventHandle);

    }
}
```

Configuration file: The XML configuration file written for this application that contains

information about the Track class and the events defined by the class Track is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE RuleEditor_Config SYSTEM "RuleEditorDtd.dtd">
<RuleEditor_Config>
        <Class>
                <ClassName>Track</ClassName>
                <InstanceName>T7000</InstanceName>
                <InstanceName>T7001</InstanceName>
                <Attribute>
                        <AttributeName>sTrackNumber</AttributeName>
                        <AttributeType>String</AttributeType>
                        <AttributeUnits>TrackNumber</AttributeUnits>
                        <EventAssociated>TrackNumberEvent</EventAssociated>
                </Attribute>
                <Attribute>
                        <AttributeName>sLatitude</AttributeName>
                        <AttributeType>String</AttributeType>
                        <AttributeUnits>Latitude</AttributeUnits>
                        <EventAssociated>LatLongEvent</EventAssociated>
                </Attribute>
                <Attribute>
                        <AttributeName>sLongitude</AttributeName>
                        <AttributeType>String</AttributeType>
                        <AttributeUnits>Longitude</AttributeUnits>
                        <EventAssociated>LatLongEvent</EventAssociated>
                </Attribute>
```

```xml
<Attribute>
        <AttributeName>dSpeed</AttributeName>
        <AttributeType>Numeric</AttributeType>
        <AttributeUnits>mph</AttributeUnits>
        <EventAssociated>SpeedEvent</EventAssociated>
</Attribute>
<ClassLevelPrimtiveEvent>
        <PrimitiveEventInfo>
                <EventName>NewTrackEvent</EventName>
                <EventSignature>void setNewTrackNumber()</EventSignature>
                <ParameterName>NewTrackNumber</ParameterName>
        </PrimitiveEventInfo>
</ClassLevelPrimtiveEvent>
<ClassLevelPrimtiveEvent>
        <PrimitiveEventInfo>
                <EventName>TrackNumberEvent</EventName>
                <EventSignature>void setTrackNumber()</EventSignature>
                <ParameterName>NewTrackNumber</ParameterName>
        </PrimitiveEventInfo>
</ClassLevelPrimtiveEvent>
<ClassLevelPrimtiveEvent>
        <PrimitiveEventInfo>
                <EventName>LatLongEvent</EventName>
                <EventSignature>void setLatLong()</EventSignature>
                <ParameterName>NewLatitude</ParameterName>
                <ParameterName>NewLongitude</ParameterName>
        </PrimitiveEventInfo>
</ClassLevelPrimtiveEvent>
<ClassLevelPrimtiveEvent>
        <PrimitiveEventInfo>
                <EventName>SpeedEvent</EventName>
                <EventSignature>void setSpeed()</EventSignature>
                <ParameterName>NewSpeed</ParameterName>
        </PrimitiveEventInfo>
</ClassLevelPrimtiveEvent>
<CompositeEvent>
        <CompositeEventName>TrackNumberSpeedEvents</CompositeEventName>
        <EventType>AND</EventType>
        <ConstitutentEventName>TrackNumberEvent</ConstitutentEventName>
        <ConstitutentEventName>SpeedEvent</ConstitutentEventName>
</CompositeEvent>
<ConditionMethod>conditionTrackNumber</ConditionMethod>
</Class>
<Class>
        <ClassName>Action</ClassName>
        <ActionMethod>actionMethod</ActionMethod>
</Class>
<MailServer>mail.uta.edu</MailServer>
<FromAddress>varakala@uta.edu</FromAddress>
<WhenEventOccurs>When new event occurs</WhenEventOccurs>
</RuleEditor_Config>
```

When the DPE is invoked, this input configuration file needs to be specified. Once this file is parsed by the DPE, it is ready to create rules, events and modify the rules created. Use of the DPE is used to create a rule is elaborated below.
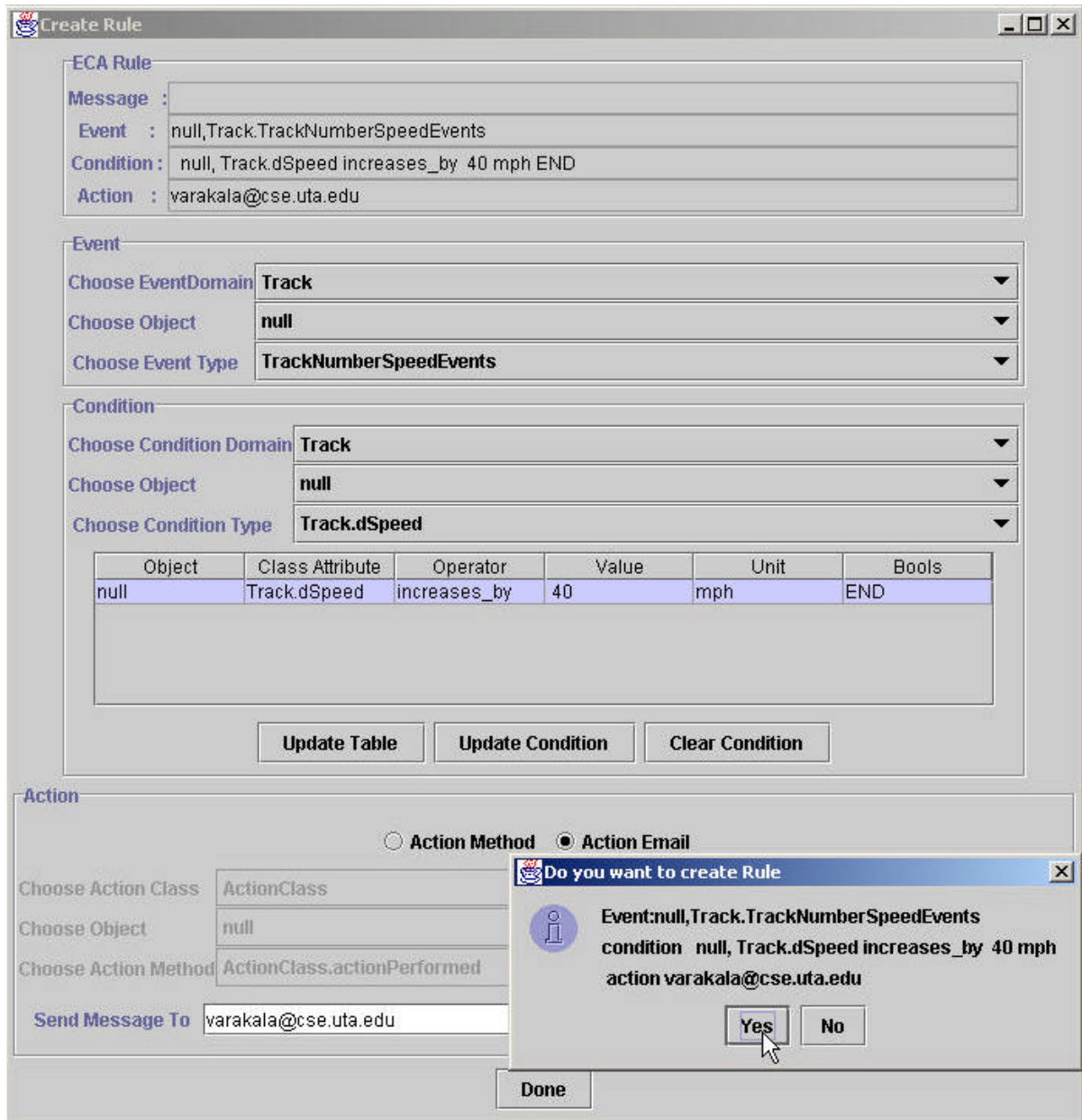

Figure A.1 Create Rule

The steps involved in using the DPE to create a rule are shown below:

- ❖ Event Selection:

  - ➢ Select an event class

  - ➢ Select an event instance

    - ▪ Select null for a class-level rule

  - ➢ Select the event type

- ❖ Condition Selection

  - ➢ Selection of an simple condition

    - ▪ Select an condition class

    - ▪ Select an condition instance

      - • Select null, if you want to use the event object (The condition class and event class should be same)

    - ▪ Select an condition

    - ▪ Click on "Update Table" button, this updates the table with the attribute.

    - ▪ If the condition type chosen is an attribute type, then

      - • Select a comparison operator

      - • Enter the comparison value

      - • Select the comparison unit

      - • Select an boolean operator, choose End to indicate the condition is complete

  - ➢ After the simple condition is selected is done, click on "Update Condition". This updates the Condition on the Rule Panel

- ❖ Action Selection

- ➢ Choose the Action type

  - ▪ If ActionMethod is chosen

    - • Select the action class

    - • Select the action instance

    - • Select the action method

  - ▪ If Action Email is chosen

    - • Enter a valid receipent address who wants to receive the notification of the rule

- ➢ Click on Done button. This displays a confirm message, asking the user that does he want to create the rule. If yes is chosen, the rule is created, else he goes the Create Rule screen and he can select the rule again.

REFERENCES

1. Dasari, R., *Design and Implementation of a Local Event Detector in Java*, in *CISE*. 1999, Univ. of Florida: Gainesville.

2. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

3. Mishra, D., *SNOOP: An Event Specification Language for Active Databases*, in *MS Thesis*. 1991, Database Systems R&D Center CIS Department University of Florida, E470-CSE, Gainesville, FL 32611.

4. http://www.weblogic.com/docs/techoverview/em.html, *WebLogic Events Architecture*. 1999.

5. http://www.ilog.com/products/jrules/whitepapers/index.cfm?filename=WPJRules4.0 .pdf, *ILOG JRules*. 2002.

6. http://www.vitria.com, *Vitria BusinessWare*. 1999.

7. Chakravarthy, S., E Anwar, and L. Maugis, *Design and Implementation of Active Capability for an Object-Oriented Database*. 1993, Tech. Report, University of Florida: Gainesville.

8. Chakravarthy, S. and D. Mishra, *An Event Specification Language (Snoop) for Active Databases and its Detection*. 1991, Tech. Report, University of Florida: Gainesville.

9.   Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB*. 1994: Santiago, Chile. p. 606--617.

10.   Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules.* Information and Software Technology, 1994. **36**(9): p. 559--568.

11.   Lee, H., *Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing*, in *MS Thesis*. 1996, Database Systems R&D Center CISE University of Florida, Gainesville, FL 32611.

12.   Chu, H.-J., *A FLEXIBLE DYNAMIC ECA RULE EDITOR FOR SENTINEL:DESIGN AND IMPLEMENTATION*, in *Computer and Information Science and Engineering*. 1998, University of Florida: Gainesville.

13.   *OpenOODB 1.0 C++ API User Manual*. Texas Instruments. September 1995, Dallas.

14.   Adaikkalavan, R., *Snoop Event Specification: Formalization, Algorithms, and Implementation using Interval-based Semantics*. 2002, The University of Texas at Arlington (MS Thesis): Arlington.

15.   Adaikkalavan, R. and S. Chakravarthy. *SnoopIB: Interval-Based Event Specification and Detection for Active Databases*. in *ADBIS*. 2003:190-204.

16.   Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.

17.   Stonebraker, M. and G. Kemnitz, *The Postgres Next-Generation Database Management System*. Communications of the ACM, 1991. **34**(10): p. 78--92.

18. Chakravarthy, S., et al., *HiPAC: A research project in active, time-constrained database management*. 1989, Tech. Report (89-02), Xerox Advanced Information Technology: Cambridge.

19. Mysore Ganesha Rao, Y., *An Agent based approach for extending the Trigger capability of Oracle*, in *ITLAB, CSE department*. 2002, University of Texas at Arlington: Arlington.

20. Subramaniam, N., *A Mediator-based approach to support ECA rules in DB2*. 2002, The University of Texas at Arlington: Arlington.

21. Gopalakrishnan, G., *Making Sybase fully Active: Supporting Composite events and Prioritized rules*, in *ITLAB, CSE Department*. 2002, University of Texas at Arlington: Arlington.

22. Lugrin, J.-M., *Free EcmaScript Interpreter:A JavaScript interpreter written in Java.* 2000, http://www.lugrin.ch/fesi/interp.html.

23. SunMicrosystems, *JavaMail API Specification v 1.3.1.* 2003.

24. Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments.* 2001, The University of Texas at Arlington.

BIOGRAPHICAL INFORMATION

Sridhar Reddy Varakala was born on March 2, 1979 in Bangalore, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Bangalore University, Karnataka, India in September 2000. In the Fall of 2001, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in December 2003. His research interests include Active Databases and Web Technologies.