AFRL-IF-RS-TR-2004-98
Final Technical Report
April 2004

FACILITATING INTELLIGENT SHARING OF INFORMATION FOR DECISION MAKING OVER DISTRIBUTED HETEROGENEOUS NETWORK-CENTRIC ENVIRONMENTS

University of Texas at Arlington

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-98 has been reviewed and is approved for publication

APPROVED: /s/
RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 2004	**FINAL Jun 02 – Dec 03**

4. TITLE AND SUBTITLE
FACILITATING INTELLIGENT SHARING OF INFORMATION FOR DECISION MAKING OVER DISTRIBUTED HETEROGENEOUS NETWORK-CENTRIC ENVIRONMENTS

5. FUNDING NUMBERS
G - F30602-02-2-0134
PE - 62235N and 62702F
PR - R427
TA - 0P
WU - 10

6. AUTHOR(S)

S. Chakravarthy, S. Varakala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Texas at Arlington
416 Yates Street (P O Box 19015)
Arlington TX 76019-0015

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-98

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577 Raymond.Liuzzi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

This report 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 are not appropriate in many environments that require fine-tuning of rules on the fly. The dynamic programming environment (DPE) presented in this report 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.

14. SUBJECT TERMS
Computers, software, database, architecture, decision making, networks

15. NUMBER OF PAGES 37

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

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, 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 superiors 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 the paper

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 have 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 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 the 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 paper 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.

2. RELATED WORK

This section 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 1. Business rules are organized into 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.

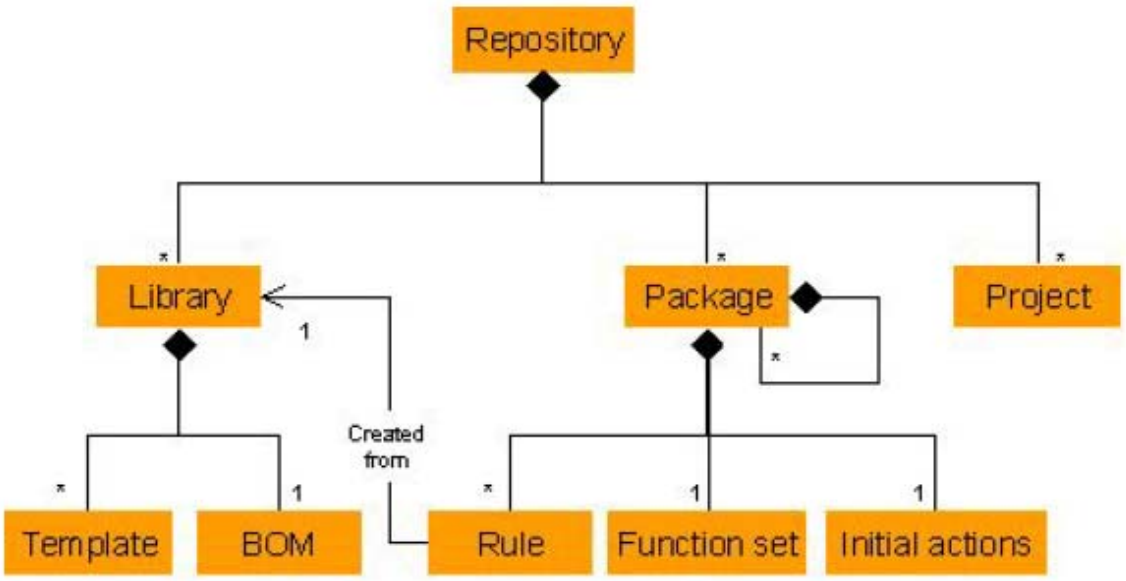

Figure 1. The ILOG Business Rule Repository structure

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.

2.2.6. 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.

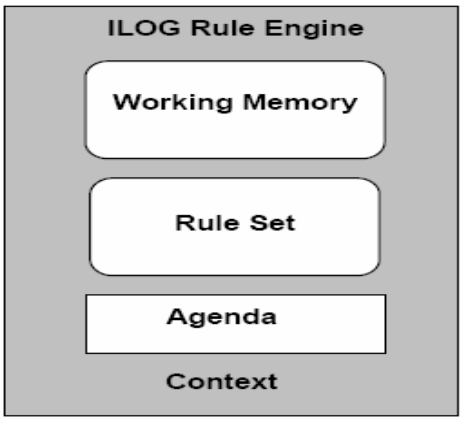

Figure 2 Context

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 [13] 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 database logic was decoupled from the Rule Editor Server by using OQL and separating all the database dependent code as individual functions to facilitate porting. In the Dynamic Rule Loader module, we used dynamic linking library to invoke condition and action functions, which avoids the relinking of applications.

3. 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.

3.1. 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:

3.1.1. 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 another 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.

3.1.2. 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.

3.2. Architecture

One of the issues encountered while designing the DPE was the handling of the various components (Rule Editor, Application, LED) and their integration without modifying their current architecture. The architecture that was initially explored is discussed following the proposed architecture.

3.2.1. Initial Architecture

The first architecture that was considered is the active application consisting of the user application and LED 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 3

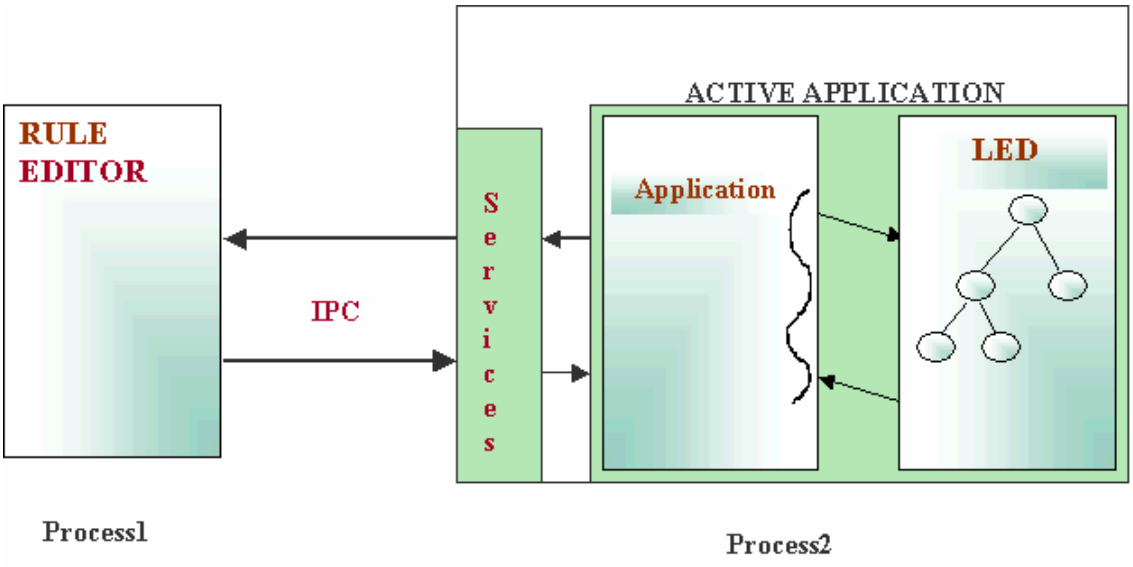

Figure 3 Architecture

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 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.

3.2.2. Proposed Architecture

In this architecture all the three components are in a single address space as shown in Figure 4. DPE is a module that runs in the same address space as that of the application and the LED. DPE is responsible for creating and 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. 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.

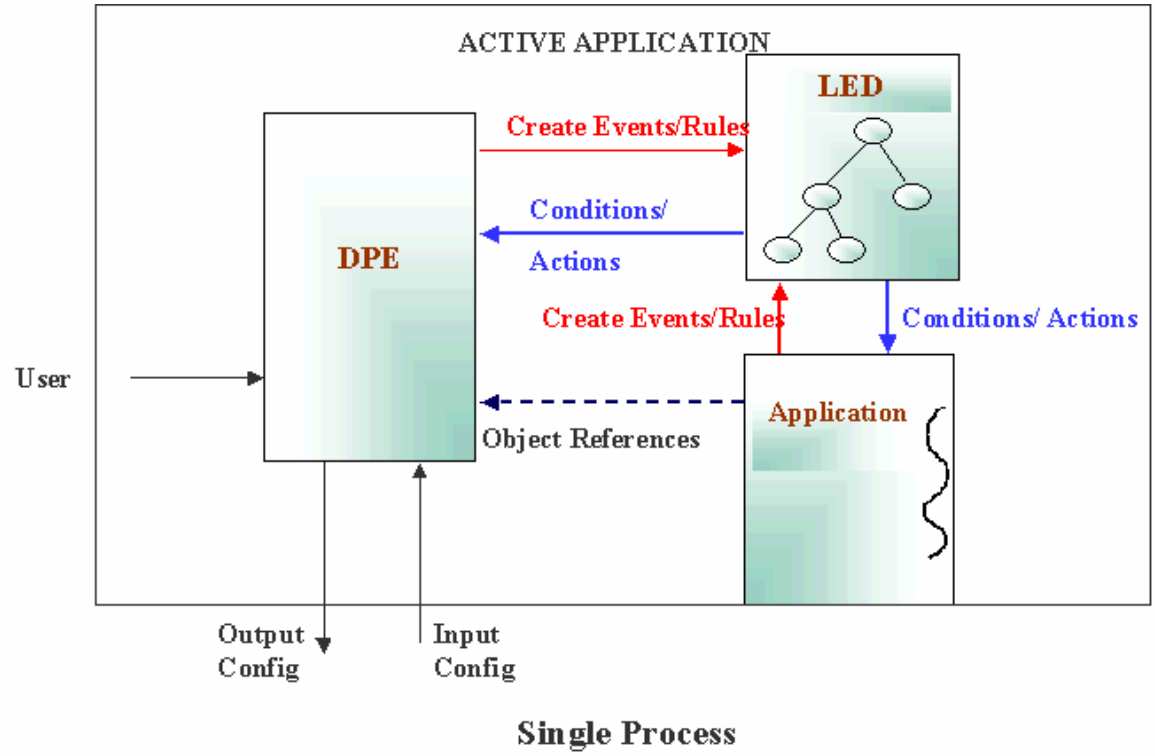

Figure 4 Proposed Architecture

3.2.3. Dynamic Programming Environment (DPE) functional modules

The functional modules of DPE are as shown in Figure 5. 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, attributes, 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.

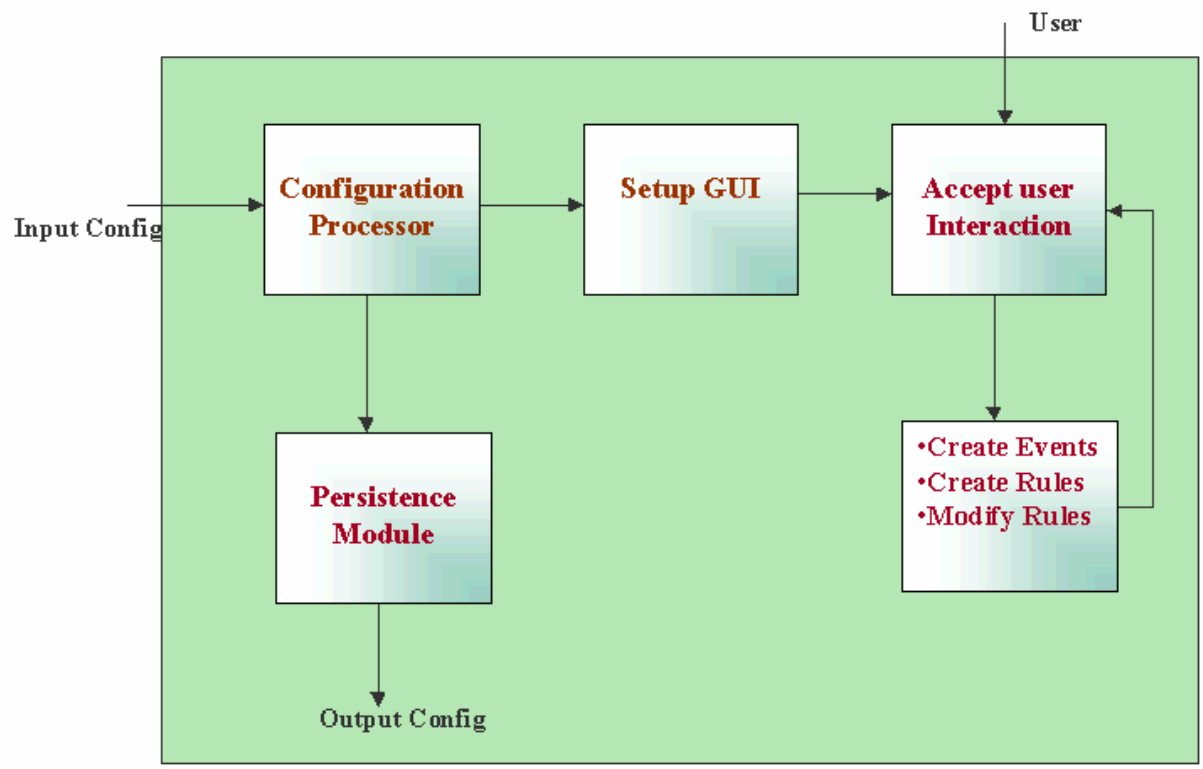

Figure 5 DPE Functional modules

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 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.

3.3. 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 6.

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,Absolute Timein 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 6 Structure of the configuration file

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.

3.4. 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 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 have 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)" [15] which is an interpreter implemented in Java. Therefore any class that wants to use attributes 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
- 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.

3.5. 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.

3.5.1. Event Selection

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.

3.5.2. 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

- 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

- 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:

1. 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.

2. 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:

1. 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.

2. 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:

1. 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.

2. 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”

3.5.3. 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
- Recipient’s email address should be provided
- 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
- Select an action class from the list of action classes
- Select an instance belonging to that class
    - 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.

3.6. 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 be 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 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.

3.7. 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.

3.8. 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.

4. 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.

4.1. 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 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 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.

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 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 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 3. The data structures are described below:

Table 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

4.2. 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 6**.**

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 4.

Table 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

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();
```

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 is 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 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 specified. 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.

4.3. 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 [16]. 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.

4.4. 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.

Table 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	Event signature of the event associated with the attribute used
sNewParameterName	The name of parameter inserted by the primitive event when raised

4.4.1. 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 Table 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. The RuleConditionObject uses reflection method in Java to invoke a method for a specified object along with the actual parameters.

4.4.2. 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 7. The important member functions are represented in Table 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: 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	Initialize 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 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 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.

4.4.3. 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 9.

Table 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.

4.5. 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.

4.5.1. 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, 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       }
```

4.5.2. 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.

4.5.3. 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 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

4.5.4. 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 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

4.5.5. 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
j++;    }
}//Evaluate the string and return Boolean result
Line 9 :        return eval(result);    }
```

4.6. 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
}

5. CONCLUSION AND FUTURE WORK

5.1. Conclusion

This paper proposes an interactive approach for supporting the creation of rules and events dynamically at run time, which is critical for several monitoring applications 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.

5.2. Limitation and Future Work

The DPE proposed in this paper 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, compiles the code programmatically and loads 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) [17] 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.

6. 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 Paper*. 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 Paper*. 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. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Paper*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.
15. Lugrin, J.-M., *Free EcmaScript Interpreter:A JavaScript interpreter written in Java*. 2000, http://www.lugrin.ch/fesi/interp.html.
16. SunMicrosystems, *JavaMail API Specification v 1.3.1*. 2003.
17. Tanpisut, W., *Design and Implementation of Event based subscription/notification paradigm for distributed environments*. 2001, The University of Texas at Arlington.