An Interactive Visualization Tool for Managing Active Capability

By

SEOCKWON YANG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Sharma for his support during the research, his innovative ideas, suggestions, and more importantly, for giving me an opportunity to work on this research project.

I would like to thank Dr. Joachim Hammer and Dr Herman Lam for giving me valuable suggestions and feedback. I would also like to thank Dr. Dankel for his suggestions and help.

I would like to express my special thanks to Sharon Grant for maintaining a well-administered research environment and being so helpful in times of need. Sincere appreciation is due to Hyoungjin Kim for his invaluable help during the implementation of this work.

This work was supported in part by the Office of Naval Research and the SPAWAR System Center–San Diego, by the Rome Laboratory, DARPA and NSF grant.

I would like to take this opportunity to acknowledge my gratitude to my parents and my sister for their encouragement and inspiration throughout my academic career.

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

By

Seokwon Yang

August 1999

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

Active feature is typically incorporated into a DBMS by ECA (event-condition-action) rule abstraction. The design of an active DBMS for a particular application entails the design of schema of the database for that application as well as ECA rules that correspond to business rules, constraints, invariance, and situations to be monitored in that application. Although schema design can be done statically based upon the application requirements, the design of ECA rules requires that the rules be validated with respect to transactions and applications, as there is semantic interaction between rules and applications. Furthermore, as the application evolves, there is a need for modifying (inserting, altering, or deleting) the business rules as the policies and other requirements of the application changes.

This thesis continues our previous work on the visualization tool as well as our work on interfaces to ADBMSs along several dimensions. We envision our visualization tool as a general purpose one that is useful not only for the designer of ECA rules but also for visualizing the behavior of the ADBMS from an end users' viewpoint. For the

designer of the rules, the tool will behave as a debugger and a regression analysis tool at a higher level of abstraction as compared to conventional debugging tools (such as dbx). On one hand, the designer is interested in understanding events and rules relevant to a particular transaction/application, order of rule execution, interaction among rules, and potential cycles among the set of rules. The ability to interact with the tool is critical from a designer's viewpoint. On the other hand, for an end user, the actual set of rules executed, the policies enforced for a particular application, and whether policies interact inconsistently are important.

Whether it is a visualization tool or an application interface, multi-platform compatibility as well the ability to use the tool/interface from the web is critical. Also, in our opinion, a general-purpose mechanism for connecting interfaces on the web to applications running on different machines is very important.

This thesis concentrates on two aspects: 1) architecture and development of a general-purpose lightweight proxy that enables us to support interfaces on the web. 2) design and development of an interactive visualization tool for ADBMSs. The tool is intended both as a debugging tool and a visualization tool and is developed in such a way that it can be easily used with other components of the Sentinel system. The tool allows the designer/user to set breakpoints on events occurrences and rule execution. It is also possible to enable or disable events and rules. The priority and other attributes of rules can be changed interactively to study the behavior of rule interaction among themselves and with transactions/applications. The tool also provides a facility to track potential cycles with respect to a rule or an event.

CHAPTER 1
INTRODUCTION

1.1 Active Database Systems and GUI tool

Over the last decade database management systems (DBMSs) have evolved to meet the diverse requirement of the application domains. One of the extensions has been to monitor the user-defined situations specified to the application and notify the changes to the user automatically. Conventional DBMS, which do not have this capability, are considered passive. Commercial database systems currently provide limited active capability, such as the triggering of procedural code when database operations are performed on tables (e.g., insert, delete, and modify). The realization of full-featured active database capability would allow more sophisticated database support of nonstandard database applications, such as computer integrated manufacturing, office workflow control and others.

Active capability in DBMSs can benefit from clever management of Event-Condition-Action (ECA) rule abstraction, which consists of three components: an event, a condition, and an action. This research is based on the ECA rule environment in Sentinel, which is an object-oriented active DBMS.

The introduction of active capability into a database system has added a new dimension to the validation or analysis tool. For the traditional DBMS, we used to have only schema design for the application and the main usage of the validation tool was just

browsing the data returned as a result of user request. In contrast, in active database systems, we have schema design as well as rule design for the application that uses active capability. The schema design for an applications can be done statically with respect to the application semantics; However, the design of ECA rules requires that the rules be validated with respect to transactions and applications, as there is semantic interaction between rules and applications. Besides, regression analysis is important in ADBMS. Addition and deletion of rules is needed as business rules change over a period of time in an application. A tool is needed to help the designer to understand the impact of changes in rules within application. The same tool should also help the end user understand the correction of rule executed for an application.

The additional analysis feature for rule design includes the cycle detection utility. When an application has cyclic rule definitions, any transaction that triggers the cyclic rule set may not terminate in a normal way. This should be captured in the visualization tool so that user can use the additional debugging feature to inspect potential cycles in rule execution.

1.2 Motivation

The previous version [1] of the visualization tool supported only post-execution analysis of an application. There were no mechanisms to allow users to interrupt the rule execution and change the state of business rules at run time. But, for the rule analysis, it is critical that the visualization tool should not only show what is happening in LED(Local Event Detector) in terms of event occurrence and rule firing, but also allow user to enable or disable rule at the various rule execution points.

Another issue of user interfaces in Active DBMS is how we can connect the interfaces to a web-based distributed environment. Among today's information technologies, it is apparent that the Internet has incredible potential to form the basis or foundation of an infrastructure. Internet has greatly expanded with a short time period, and it is still expanding at a high rate today. A more interesting issue of the Internet is the foundation of the client/server infrastructure. Internet can be explored to become a gateway for the applications between the different computing environment. But, the issue of interaction with a remote database application through the Internet has posed interesting problems for many practitioners and researchers, especially in the area of security, reliability, and performance of web-based database systems. The previous works tried to address this problem by introducing a 3-tier architecture and the concept of a proxy [1]. The idea behind a 3-tier architecture is to separate the service logic from the application logic. User application does not include the service logic to operate once it knows how to request the service from the systems, which have all the details of the service logic. The obvious benefit of this approach is that the boundaries of active database system are no longer limited to only one platform. This also saves a lot of time and effort in porting the system from one platform to another platform. But, the implementation could not completely generalize the approach to connect user interfaces to the web. Proxy developed in previous work was not extendible to solve some of the issues, because of the security restriction in web-based applet imposed by Java. In this work, Proxy concept has been extended to generalize GUI architecture for Sentinel on the web.

Another issue addressed in this thesis is the integration of nested transaction model into visualization tool. The previous work does not properly define output scheme of rule execution to reflect what is happening in LED. The output scheme was not extensible to show the nested subtransaction model and help the user easily understand or analyze the state of rule execution. After nested transaction model in LED was integrated in Sentinel, corresponding changes had to be made in LED to present the nested transaction model correctly into debugging environment.

Termination is another issue that must be addressed by any active rule-debugging tool since common symptom of erroneous rules is recursive execution of rules that does not terminate. As rule behavior is usually determined at run time and each event and condition behavior of a rule is not fixed at compile time, it is hard to predict whether the rule execution results in non-terminating cycle. The prediction of rule execution is even more difficult when we take into account different event propagation contexts, and the semantics of event operators. A termination analysis utility has been implemented in this thesis will be discussed in detail.

CHAPTER 2
OVERVIEW OF LED (LOCAL EVENT DETECTOR)

Validation or analysis tools that we intend are closely related to the local event detection module because the tools essentially show what is happening in the LED. Therefore, it is necessary to give an overview of LED before we discuss the design issues of the tool for a better understanding of subsequent chapters. In this chapter, we overview the LED module of Sentinel.

Sentinel is an integrated active DBMS incorporating ECA rules using the Open OODB Toolkit (from Texas Instruments). Sentinel allows users to specify events and rules at an abstract level using the snoop event and rule specification language, which is incorporated into an application written in C++. Any method of an object class can be a potential event generator (in our case, primitive event). To identify when the events are generated inside methods as primitive events, event modifiers were introduced in Snoop [2]. Event graphs are used to detect composite events that are defined by applying a set of operators to primitive events and composite events, recursively.

In Sentinel, Several rules can be defined on the same event in different contexts, rather than duplicating event for each context.

2.1 Types of Event detected by LED

2.1.1 Primitive Events

Events are classified into i) primitive events which are pre-defined in the system and ii) composite events that are formed by applying a set of operators to primitive and composite events. Primitive events are further classified into domain specific (e.g., database), temporal, and explicit events.

Database domain events correspond to database operations, such as retrieve, insert, update and delete in a RDB or a method invocation in an OODB.

Temporal events are classified into absolute and relative events. An absolute temporal event is specified with an absolute value of time using the format *<(hh/mm/ss)mm/dd/yy>*. The field can be filled with wild card(*). A relative event is a unique point of time event which is defined by a reference event and an explicitly specified offset. The syntax for a relative event is event + *<(hh/mm/ss)mm/dd/yy>*.

Explicit events are those events that are detected along with their parameters within application programs. Any method of an object class can be an explicit event. Once registered with the system, they can be used as primitive events.

2.1.2 Composite Events

Composite events that are formed by applying a set of operators to primitive and composite events, recursively. The operators of Snoop are OR, AND, ANY, Seq, Not, Aperiodic, and Periodic. Periodic and aperiodic operators were introduced to meet the requirements of process control, network management, and CIM applications [2].

2.2 Composite Event Detection

A sequence of primitive event occurrences (over a period of time) may make a composite event occur in LED. Hence, the local event detector needs to record the occurrence of each event and save its parameters so that they can be used to compute the parameter set of the composite event.

Sentinel constructs an event tree for each composite event. The trees are merged to form an event graph to detect a set of composite events. Event propagation is performed in a bottom-up fashion. Leaf nodes of an event graph corresponds to primitive or external events. Internal nodes correspond to event sub-expressions. Each node has a list of subscribers to whom it has to notify when the event denoted by that node is detected.

The notion of parameter contexts is introduced to capture application semantics for consuming event occurrences of composite events. The event detection can be varied according to the parameter context of the constituent events and the event. These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event, which can start the detection of the composite event, and a terminator is a constituent event, which completes the detection of an composite event occurrence. We have identified 4 contexts - Recent, Chronicle, Continuous, and Cumulative context, to meet the various characteristics of the applications. Each context [2] is summarized below.

RECENT: In this context, only the recent occurrence of the initiator for any event that has started the detection of that event is used. An initiator of an event

(primitive or composite) will continue to initiate new event occurrence until a new initiator occurs.

CHRONICLE: In this context, the initiator of the given event can be paired with a unique terminator of the event. The parameters are computed by using the oldest initiator and the oldest terminator of the event. This context preserves the chronological order of event pairings.

CONTINUOUS: Each initiator of an event starts the detection of that event. Multiple initiator can be paired with a single terminator.

CUMULATIVE: All occurrence of an event type are accumulated as instances of that event until a terminator is detected. When the terminator event is detected, all the occurrences that are used for detecting the event are packaged and propagated along the event graph.

2.3 Rule Processing

Rules are specified at class definition time as a part of an application. The class-level rule specification is pre-processed into C++ statements and inserted into the application program. Sentinel also supports rule activation and deactivation at run time.

When an event occurs in the database system, the rules that subscribe to that event are triggered. A simplistic approach would be to let all the rules that have been triggered by this event to run one at a time sequentially. The whole process can be done in the same transaction that triggered the event. The disadvantage of this approach is that it cannot support concurrent or parallel rule execution associated with one event in order to maximize the throughput. All the rule executions are serialized and executed sequentially. It also does not support any dynamic rule execution based on the priorities

of each rule. Briefly, in the nested transaction model, when an event occurs, the corresponding condition evaluation and action execution is treated as a separate thread and the thread execution in turn fires other rules as child sub-transaction. If an event is associated with several rules and rules have the same priority, then multiple threads are created and executed concurrently.

Rules are typically specified with a priority. A scheduler, based on the rule priority, controls the execution of the rule thread. If several rules have the same priority, we have concurrent execution of the rules. This may result in conflicting access to the same data item. To deal with this problem, the nested transaction model executes a rule thread as a sub-transaction and uses the transaction synchronization scheme. Sentinel has a lock manager for the purpose of concurrency control. When the scheduler wakes up, it allows the rule with the highest priority to execute first. The order of rule execution can be varied according to the relative priority of its sibling rules.

Coupling mode specifies the point of time after the event occurrence when condition evaluation and action execution begin. Sentinel supports both immediate and deferred coupling modes to specify the semantics of a rule execution. *Immediate coupling mode* rule is executed at the point where the event occurs, while *deferred coupling mode* rule is executed at the end of the transaction.

Once an event is detected, a set of rules may be triggered. For each rule that qualifies, a separate thread is created in suspended state and inserted into the rule list [3]. The placement of rules in the list depends on the coupling mode, its parent and its priority. The rules are inserted in decreasing order of priority. Also the process-rule-list is sub-divided into two sections, one for rules with immediate coupling mode and the other

for rules with deferred coupling mode. All top level deferred rules triggered by rules executing with immediate coupling are put into cycle-1. All rules triggered from a deferred rule are assigned the next cycle number.

After event notification inserts the associated rules into the rule list, it wakes the scheduler and the rules are scheduled based on their priorities and coupling mode. The scheduler traverses the process-rule-list to schedule the threads. Th ‘notify’ function waits until all triggered rules in the immediate coupling mode finish execution.

Because rule execution proceeds in a depth first manner, if a fired rule has an immediate coupling mode, the state of parent transaction will be changed to 'wait' until the child sub-transactions finish their execution. Only after the execution of these rules, the suspended triggering transaction can continue.

If the rules are created in deferred mode, threads for the rules are created but are not scheduled right away. The triggering rule proceeds normally and before the commit of the top-level transaction, all of the deferred rules are scheduled and executed. The triggering or top-level transaction does not commit until all deferred rules are executed.

To process deferred rules, we define a rule which is triggered by the event - commitTransaction. This rule has an immediate coupling mode and has a priority of '-1'. '-1' means that the execution of rule will start only after all immediate rule has been scheduled and finished. The execution sets the 'deferred-flag' flag so that scheduler can start to trigger the appropriate deferred rules in cycle-1 based on the scheduling algorithm. When a rule thread completes its execution, the corresponding rule node in the rule list is deleted. As a result, at the end of the top-level transaction, the rule list becomes empty [3].

CHAPTER 3
DESIGN ISSUES OF TOOL INTERFACE FOR ADBM

3.1 Related Work

In this chapter we provide a summary of the graphical user interfaces for active DBMSs found in current literature. This helps us identify the facilities for user interface or visualization tool for ADBMS and identify the limitations in those tools from our requirements.

3.1.1 DEAR

DEAR [4] keeps track of both rules and events. It automatically detects inconsistencies and potential conflicting interactions among rules. To provide a more focused tracking, DEAR has a "pruning" feature to reduce the scope of tree shown by the debugger. Debugging can be restricted to certain rules and/or events. For detection of inconsistencies and conflict interaction, DEAR can point out potential cycle by highlingting the branch where an event occurs twice.

But, the approach taken by DEAR has a limitation that it works only for primitive events, such as insert, delete, update, and the rules defined over these events. DEAR creates a graph consisting of rules and event nodes, where event nodes alternate with rule nodes. An arc from an event node to a rule node means that the event triggers the rule. And an edge from a rule node to an event node means that the event was produced by the rules. The use of rules on primitive events makes the detection of cycles easier. In case of

composite event expression, such as AND, SEQ, A, A*, P, and P*, it is not always true that the event nodes alternate with rule nodes. Suppose we have e1, e2, and AND event of e1, e2, and we define a rule that triggers e1 when AND event occurs. The graph does not form a cycle with their approach. It seems that their approach does not consider the composite event expressions to keep the problem relatively simple.

Second, they do not support interactive features that we feel is necessary for a debugger of this sort. In DEAR, events and rules are monitored and displayed at run-time without user interruption. The only user-initiated request that may alter the visualization is the “spy” command prior to execution. The extent of implementation is not clear from the literature. We are assuming that all of the features discussed are actually implemented in their system.7

3.1.2 PEARD

The debugging features of PEARD [5] also include detecting potential cycles in rule execution and a utility to examine different rule execution paths from the same point in the rule triggering process. This tool is similar to our visualization tool especially with respect to rule browsing, breakpoint setting, and rule enabling/disabling. It detects the cycle by counting the repeated event occurrences. But, this approach does not differentiate the external events generated by the applications and the internal events coming from the nested rule execution. So, the probability of false cycles is not negligible.

3.1.3 SAMOS

SAMOS [6] has been implemented based on a layered architecture where all the components that implement the active behavior are built as a layer "on top" of a conventional passive database system. For the prototype implementation of SAMOS it

uses the commercial object-oriented database management system ObjectStore. SAMOS has its own rule definition language similar to snoop in sentinel to specify ECA-rules. Event detection & notification is performed using the petri-net mechanism. In SAMOS, coupling modes are specified separately when the condition is evaluated with respect to the trigger transaction and when associated action is executed with respect to the condition evaluation. In Sentinel, condition evaluation and rule execution are treated as one sub-transaction, which is executed in a thread, and the coupling mode is defined between the event and the sub-transaction. SAMOS has several tools, such as a rule analyzer, a rule browser and a rule explanation component. The rule browser retrievers the rulebase by performing queries over the rulebase. It shows individual rules, events, actions and conditions, and allows the selection of items which meet various desired criteria. In addition, for a primitive event the event browser indicates the list of composite events it is participating in. When an event, condition, or action is selected from the rule body description, the appropriate browser is activated. The event browser displays the list of composite events an event participates in and the list of composite events. The condition browser shows the list of all conditions, the list of rules the selected condition belongs to, and also the body of any selected item, i.e., the corresponding source code.

It also supports the detection of cyclic rules as part of utilities. The termination analyzer assists users in checking the termination of rules. It investigates rule definitions at compile time and determines whether rules could potentially trigger each other indefinitely, i.e., it detects and visualizes potential loops that could occur during the execution of an application. This information helps the rule designer to decide whether the rule set must be changed. One limitation they have is that they do not differentiate the

internal event from the external event because they do not support the nested transaction model. Therefore, the probability of false cycles is not negligible. It is also not clear from the literature how much of the visualization tool has been implemented.

3.2 General design requirements of our approach

Below, we enumerate some general design requirement for the visualization tool and Sentinel user interface.

1. The tool should support different user perspective. Rule designers need to understand the details of the system's modules, trace the execution, discover existing or potential errors and correct the errors if necessary. Their interests go well beyond a specific application's running behavior. On the other hand, the end users are more interested in the running behavior of an application. The user interface is meant to highlight information related to an application without irrelevant data.
2. *Portability* has become an important factor in present day's software arena. Users should be able to run the tool in stand-alone on Unix or NT as well as in a web-based environment. In addition, running environment should be transparent to users in terms of usage. User should not be burdened with additional settings for each environment. Scalability of the architecture should be taken into account to make the tool available in multiple settings.
3. The tool should provide multiple modes of usage so that the user can choose the mode that is appropriate for his/her need. The tool at the least needs to support run-time and post-analysis trace modes. In addition, the interactive

mode will be helpful to debug rule set at run time. A detailed description of each mode will be given in Chapter 5.

4. The tool should be able to support visualizations of different applications as well as multiple visualization of the same application in a distributed setting. This will allow multiple users to observe and interactively debug the rules defined for different applications. Multiple visualization on different applications as well as multiple users should be supported by the tool.

3.3 Specific Design Consideration for the Visualization tool

When we consider the debugging context of sentinel, we need to take into account the following specific design consideration.

1. **Facilities for understanding of ECA rule abstraction:** When an event triggers a rule and the rule executes the action, the task of the user interface is to demonstrate the situation changes and convey them to users graphically. It is advantageous to couple action-oriented GUIs with the rule system for better understanding of ECA abstraction.
2. **Attaching graphical objects to Sentinel objects:** Each graphical object corresponds to an event node in the event graph of LED. Users can query or browse the objects simply by selecting the object with the mouse. This feature helps a user to check the component of each event and rule, not in the text form but in the graphic form and decide where to put break points or which event or rule is temporarily disabled in order to eliminate the potential infinite cycle. The process of debugging an active database application is not sequential and each operation is not executed sequentially,

as in a conventional programming language. Hence, an ability to browse is fundamental and important as part of the visualization of an active database.

3. **Rule and event interaction:** The visualization tool is mainly used to help the user to see the interaction among rules and events in an ADMBS. When event(s) occur, the corresponding rules are triggered. And rule may raise an event, which in turn raises the rules to invoke other events. Without visual aid, this interaction is too complicated to analyze and understand. The causal relationship between event occurrence and rule execution should be explicitly shown to the user in order to understand the interaction.
4. **Presenting nested rule execution to user:** In Sentinel, rule execution (i.e., condition and action portions of a rule) is done in a sub-transaction. Sub-transactions can be nested to arbitrary levels and are represented by a n-ary tree. The transaction tree grows in a top-down way. The sibling sub-transaction is positioned side by side and child sub-transaction is located below the parent sub-transaction to show the relationship of each transaction. In addition, visualization tool should show the execution of rules graphically preserving the triggering order and current state (suspend or commit) of each sub-transaction.
5. **Rule debugging scope:** A user can change the scope of rule that he/she wants to monitor. When a user has an interest in only particular rules and events, the tool should be able to eliminate the unnecessary parts from the user view so as not to disturb the interesting part.
6. **User intervention in application:** We need a graphical tool that allows the user to intervene during the execution of application for debugging purpose. Users may want

to change the rule execution path by changing the order of priority, disabling/enabling some rules or events. This facility is similar to changing a variable value in dbx to observe what is going on. After we change a variable value in dbx, we can observe the change in control flow, or debug the specific function call routine at run time. Analogously, the tool should be able to provide mechanisms to change the characteristics of rules and events (enable/disable, change in priority, change in coupling mode etc.) at a level of abstraction that is appropriate for the active database usage.

7. **Presenting potential cyclic rule set to user:** When the tool detects a cycle in rule execution, it should give a warning with an appropriate description to the user to help analyze the situation. The warning message includes the scope of rules and events which the user can concentrate on to find the loop. This feature assists users in finding the cyclic rule set.
8. **Using the Tool in a Distributed Environment:** Allowing several tools to visualize and explore the same rule set simultaneously would be another interesting direction. Starting several (at least two) visualization tools or user interfaces, loaded with the same rule sets, but having different set of events to view by pruning operation or different aspect of rule set, would be helpful for debugging rule sets. Each visualization tool would be responsible for the actual visualization of corresponding rule set, but operations such as breakpoint, disable, enable operation or generating reports, etc, would be distributed to all other connected visualization tools or clients. User action would be packaged into message from the tool and sent to LED module.

9. **Just-In-time display:** If we have only one process to receive the run-time information such as triggering event, firing rules, starting a new sub-transaction, the delay due to parsing the input, deciding what it should display, and drawing the result on canvas, may result in loss of some information or yield unwanted effects. There can be visible drawing latency and flicker. This becomes unsatisfactory when there are many images being updated frequently. A good animation should be smooth and flicker-free. To solve these kinds of problems, the tool should separate each job and assign it to a separate thread to work in parallel. The tool should have a receiving thread, a parsing thread, and drawing threads taking charge of each part. To get smoothness and flicker-free graphics, it is important to use the double buffering technique. Doubling buffering technique is frequently used in many 3D image-rendering software. While one buffer is used to display on the screen, the rendering process writes the rendering result into the other buffer. The role of buffer is switched when the rendered buffer completes. In the same way, the tool should use two buffers, one for drawing image, one for showing the result. While drawing thread works on one image buffer, the other buffer is used to display on canvas.
10. **Scrolling:** Sentinel uses the nested transaction model for rule execution. Nested rule execution can happen to any depth and concurrent rule execution can create any number of sibling sub-transactions. Sibling sub-transactions are positioned side by side and child nested transactions are positioned bellow the parent sub-transaction to show the relationship of each transaction graphically. So, it often happens that the currently executing transaction is drawn outside of canvas. Hence, the tool should give control

to the user to shift the view to any point on the virtual (or logical) canvas. Using horizontal and vertical scroll bar, a user can shift the view to anywhere he/she wants.

CHAPTER 4
ISSUES OF EXPLANATION TOOL ON THE WEB

First we discuss the different scenarios where we want to use the user interface for sentinel and identify the requirements. After that, we summarize the issues related to making our interfaces available on the web and explain how the proxy concept was extended to generalize the approach. In the last section, implementation choices and details of proxy will be presented.

4.1 Transparency Requirement for User Interface

Figure 4-1 shows two different environments where Java user interface interacts with sentinel applications. Figure 4-1(a) illustrates the typical environment where each application and GUI creates a socket to listen to the incoming message, communicating in both directions. Provided that the GUI knows where each application is executing and applications know where the GUI waits for notifications in some way, applications and GUI might run on the same machine or on different machine.

Figure 4-1 (b) demonstrates the need for additional requirements for environments other than the one shown in Figure 4-1 (a). In short, a GUI should be able to run on top of a browser and communication channel between GUI and applications goes through web. This thesis defines *'Distributed Web environment*' as follows. GUI element runs on web browser and applications run on remote machines, while bi-directional communication between GUI and sentinel applications is established through the web server.

The difficulty in supporting both scenarios *uniformly* is due to the differences in the communication model. In the first scenario, the communication follows the typical client/server model. GUI creates a socket to wait for incoming messages. The applications can reach the machine and port where GUI binds the socket. However, in the second scenario, a GUI on top of a browser cannot create a socket to wait for incoming message as in the first case. The communication model supports *request/reply paradigm*. A GUI on top of a browser can receive messages only when it requests or initiates an operation. So, applications cannot send asynchronous messages to a GUI as in the first case.

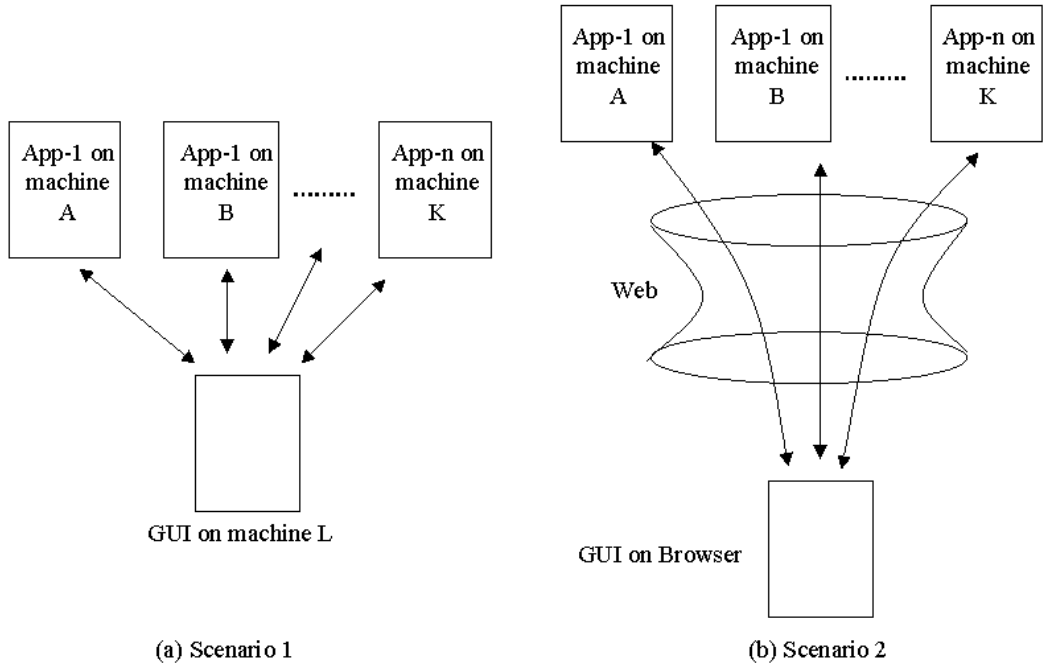

Figure 4-1 Distributed Environment vs Distributed Web environemnt

This thesis proposes a generalized architecture that supports both scenarios and furthermore makes the approach transparent to the user. User interface can be executed on any standalone machine. In addition, when the same user interface is used through web, the same code can be used in the same way, without recompilation. We generalize

the communication channel between a GUI and applications to satisfy both scenarios uniformly.

4.2 Proxy

In the previous section, we discussed the one example scenario related to security restrictions of GUI on top of browser. In short, the restrictions are as follows:

1. Current commercial browsers do not allow the downloaded applet to open a listening port for accepting connections. This restriction comes from the fact that Java applets run on a virtual machine in the browser, which insulates them from direct contact with the host system. This so-called ‘sandbox’ around the applet enforces restrictions that prevent it from interfering with the host.
2. Applets on top of a browser cannot create a server socket as it is normally done and cannot receive messages without initiating a request. Messages can only be received after initiating a connection or interaction with the host web server. It cannot play a role as a server and wait for incoming messages
3. Applets are not allowed to open network connections to any computer, other than the host that provided the class files. This is the host from where the html page and applets came from [7].

We apply a 3-tier architecture to overcome these 3 restrictions which are common in distributed web environment. For example, in Figure 4-1 (b), GUI on top of browser cannot create a server socket to wait for notification, cannot receive asynchronous messages, and directly communicate with applications if the applications execute behind the firewall. These are the reasons why we need to extend the proxy to overcome these restrictions.

The scenario in Figure 4-2 illustrates the need for proxy, which is related to the first and second restriction. Suppose a remote process running on another machine (than a Web-server host) wants to send a message to the applet. How does the process know where the applet is running? Java class files are treated as HTML files in web environment. So, After the Web server transfers the requested Java class file, it does not keep track of who is reading the HTML or executing the applet. This is the reason why applet cannot receive any asynchronous message unless it connects back to the web server host and registers itself. Again, if we have an intermediate server that keeps track of applets between remote processes and applets, then remote process can send asynchronous messages to these applets.

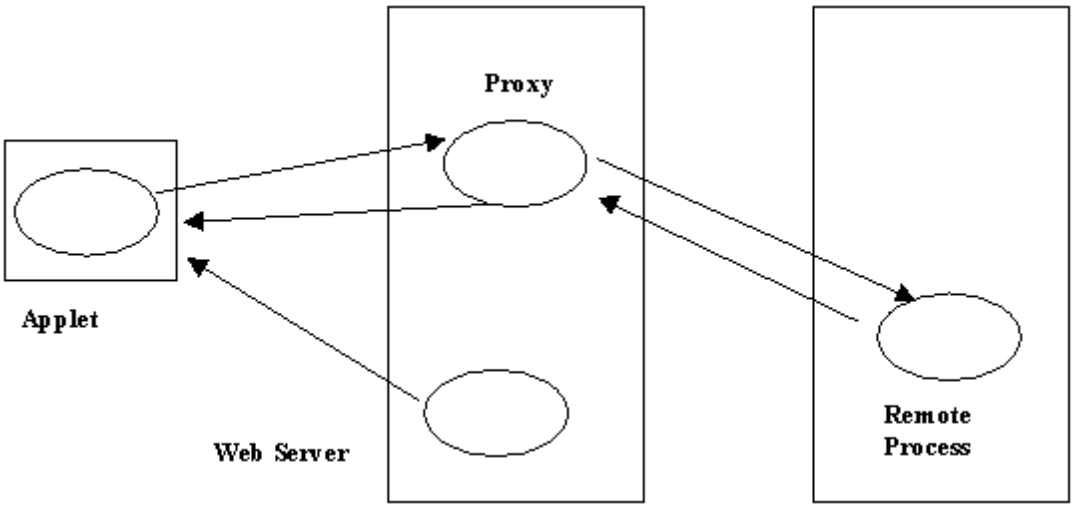

Figure 4-2 Proxy

Here is another scenario to illustrate the need for a proxy server. Suppose an applet GUI wants to send a message to a remote process running on another machine. This scenario is drawn in Figure 4-2. Because of the third restriction, the applet cannot establish a connection to the specific machine directly and cannot send a message. But, If

we have an intermediate proxy server between GUI and the application to which the applet would connect, then messages can be delivered or transferred through the proxy.

In addition to security restrictions, we encountered another proxy issue in the web-based environment. We have Java applets as a front-end interface running on web browser. We want to let this applet launch the sentinel application either on the web server machine or on a remote machine. In other words, we want to have process control mechanisms similar to UNIX terminal on top of a browser. To achieve this goal, a proxy should have the capability to receive remote execution requests on behalf of the user through web channel and launch the specific process on the remote machine. Here, we have to consider security issues. We cannot let anyone access our URL and play with our applet, launching whatever he/she wants to run on our machine. This may damage our systems and lower the system capability. So, Usually in UNIX environments, to launch a process on a remote machine, systems ask user to give user name and password. This means only a legitimate user who has an account on the machine can launch a sentinel application. To circumvent this limitation, we restrict the kind of processes (actually this is just public links to processes), which user can invoke, and the proxy can launch only those processes after it receives the request. In summary, we developed our own proxy, which helps exchange message between applets and sentinel applications. Also, our proxy can launch sentinel applications on behalf of users in a limited way.

4.3 Alternatives

We can make the use of callback mechanism in CORBA [8] to accomplish our requirement. A callback reverses the client and server roles; it allows a client to become a server. Consequently any client can automatically receive callbacks.

Callback is a mechanism that allows one object in class A to call another object in class B that was passed when it was created to accomplish a job of its member function. The original purpose of a callback was to provide flexibility in the behavior of code. For example, you have file operations that list all the files with the extension of *.java, given a directory path. You may create a separate class to perform this specific operation. But, when you need a similar kind of utility that lists all the files with the extension of *.c; it is not a good design method to create another class for it in terms of code reusability. In Java, we can define one file operation class that lists all the files with some extension. When we define the class, the class does not know the filtering details. And we create separate class for each filtering operation that deals with filtering according to requested extension. We pass the instance B of that class to file operation object. Later, the object of file operation will call back the instance B when it needs a filtering method.

This idea was extended in CORBA to make clients play the role of a server. A client would pass the object of its own interface to a server. When the server need to get the service of the client (in our case, receiving asynchronous message), it can get the service because the server has the proxy object to the client's interface. Note that the proxy object in callback is not the same as proxy process in sentinel. A callback is an operation invocation made from a server to an object, which is implemented in a client. Such invocation allows servers to send information to clients without forcing clients to explicitly request the information [8].

The transmission of requests from a server to the client is possible because OrbixWeb maintains an open communications channel between client and server while both processes remain alive.

Many firewalls do not allow an application inside the firewall to receive connections from outside, so client applet downloaded to a machine behind such a firewall cannot use standard IIOP to receive callbacks from a server outside the firewall. OrbixWeb V3.0 introduces an optional extension to IIOP to allow the protocol to use bi-directional connections. Client can receive requests from servers on the connection that the client originated to the server.

It is possible to use OrbixWeb as underling communication infrastructure. But, this would add the system requirement (OrbixWeb product) to Sentinel for user interfaces. The original purpose of callback was to get access to client objects and invoke the methods of those objects. If we can avoid using another system and do similar work, it would be better not to use additional products. Using OrbixWeb (or a CORBA implementation) tools for this purpose would add additional processing overhead as well as significant increase in code size. Client applets only need a communication channel to get messages from Sentinel server side, show the message and send a user input to the server. So, we generalize the idea of how OrbixWeb works for callback service and implement our own callback service with socket communication. Actually, Our proxy can be viewed as a lightweight OrbixWeb daemon.

4.4 Proxy Architecture

We consider two alternative architectures to implement the proxy. The first architecture uses the fork, semaphore, and shared memory. The other one uses threads and one linked-list which stores the open connection. In this section, we present two different architectures and explain why we chose the second.

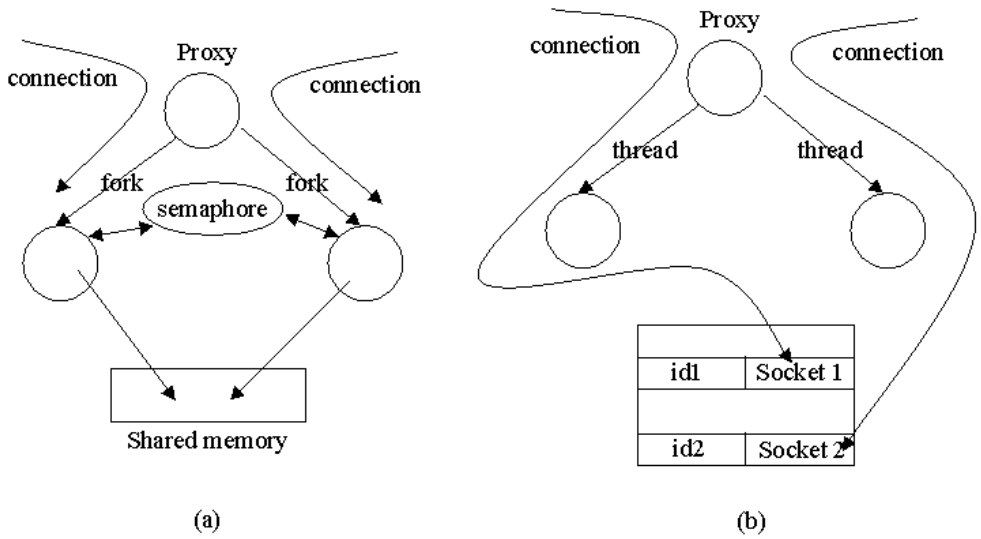

Figure 4-3 Comparison of Proxy Architecture

Figure 4-3 (a) shows the first approach, using fork, semaphore, and shared memory. The proxy creates a child process for each client. It has one global semaphore to coordinate the child processes, and one inter-process shared memory to transfer message received by one child process to another child process. Initially, the child process is in wait state. When any child process receives a message from the client, it first put the message in the shared memory, wakes up the other child process that serves the other client, using the semaphore operation. At the same time, the awakened child process accesses the shared memory, gets the message and sends the message to the client it serves. Again, the state of child process is also changed into wait state after it sends the message to the destination.

The second approach is shown in Figure 4-3(b). When a proxy receives the connection, it creates a daemon thread to serve each client and puts the socket connection with unique id into a global list. The socket connection that is on the list is not closed by the proxy until the client explicitly asks to close it or the client closes the other

end of the socket connection. Each daemon thread is given access privilege to look up the socket connection with a given id. When a proxy receives the request for transferring a message, it first looks up the list to find the destination socket connection with the id, which is contained in the message, and writes the message into that socket connection. This will wake up the client and the client processes the message according to its semantics.

The first approach has been used for a while, but this architecture has several limitations. One is the size for shared memory. Each machine defines a minimum allocation size shared memory. If the size of shared memory requested does not match the requirement, the shared memory would not be granted. For this reason, the same proxy process may not work on other machine, which we experienced in the past. In some case, a machine may not have any more shared memory to grant. Then the first approach would not work at all.

The other limitation is the number of semaphores required. The first approach uses a semaphore to coordinate the child processes. The number of semaphores needed is determined at run time. If we have 2 processes or 2-process groups to coordinate, then one semaphore would be sufficient. But, if we have 3 processes or 3-process groups to coordinate, one more semaphore is needed. In this way, the communication, such as 2-way or 3-way, determines the number of semaphores needed. Therefore, we cannot make a general-purpose proxy using this architecture.

Besides, the first approach cannot port directly to an NT operating system. The resources such as semaphore, shared memory and fork system calls are all specific to

each operating system. Also, the size of a process is also large when forked as compared to a thread and furthermore process switching is more expensive in case of a fork.

All of the above limitations are overcome when we use the second approach. It does not employ semaphores and shared memory, so the size of the semaphore or availability of shared memory would not cause any malfunction. Only necessary function is coordination in accessing the list, but if we implement this approach in Java, we can make use of Java object monitor or synchronized keyword to prevent simultaneous access and manipulation of data structures. In addition, the same code can be used on multiple platforms.

CHAPTER 5
IMPLEMENTATION OF VISUALIZATION TOOL

First we briefly summarize the overall architecture of Sentinel to understand how the Visualization tool works and interacts with other functional modules. After that, this chapter discusses the implementation details of the visualization tool, adapted to the proxy architecture presented in chapter 4 to make the tool available both on a standalone machine as well as on the web. Finally, our approach to termination analysis that includes the composite events will be presented.

5.1 Sentinel Architecture

The sentinel architecture [9] shown in Figure 5-1 extends the passive Open OODB system. The Open OODB toolkit uses Exodus as storage manager and supports persistence of C++ Objects. Concurrency control and recovery for the top-level transaction are provided by the Exodus storage manager. Sentinel has extended the skeletal transaction manager of Open OODB to a full-fledged transaction manager to support the nested transaction model and maintain a separate lock table for providing concurrency control at the sub-transaction level. There is no recovery at the nested sub-transaction level.

Detection of primitive events is achieved in Sentinel by adding Notify (a method call to the event detector class) into the wrapper method generated by the Open OODB. A local event detector has been implemented to detect composite events. The event

detector is implemented as a class and each Open OODB application has a single instance of the local event detector.

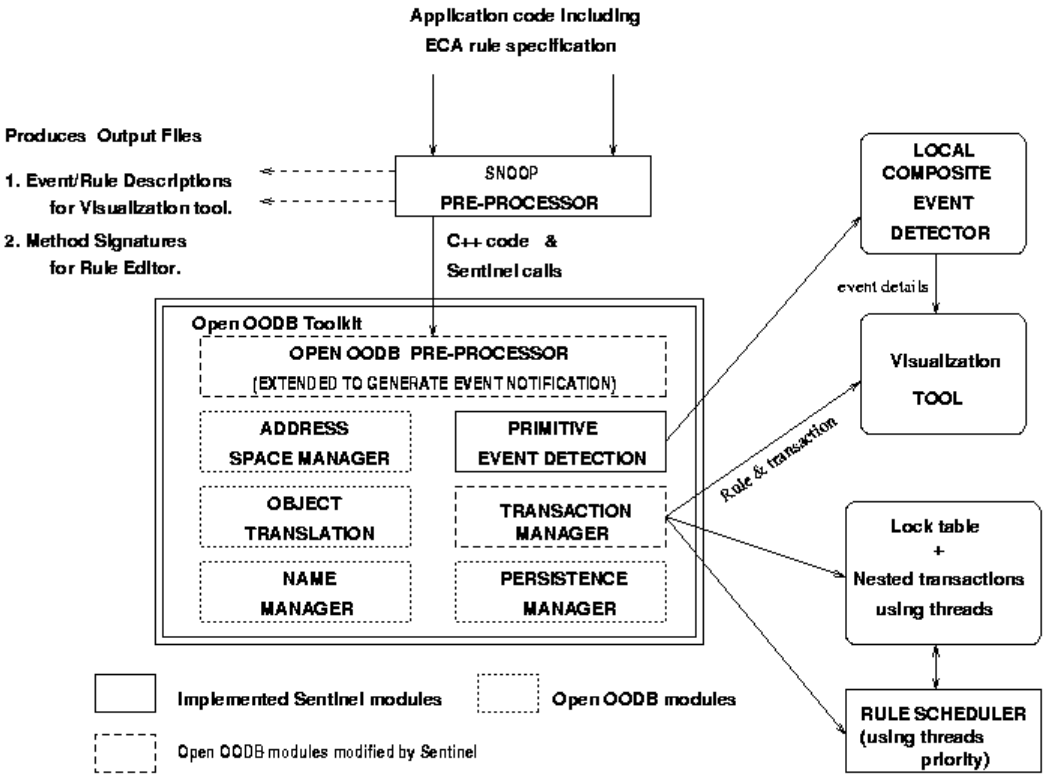

Figure 5-1 Sentinel Architecture

A Snoop pre-processor is used to extend the user class definitions as well as application code. It processes the ECA rules specified in Snoop language as a part of a class definition or as part of an application. It converts the high-level event-rule specification into appropriate code for event detection, parameter computation, and rule execution. Snoop pre-processor also generates two additional files, one for the dynamic rule editor and another for the visualization tool. The file for the dynamic rule editor contains a list of potential events and keeps it current as the application is recompiled. This information is used by the dynamic rule editor for presenting the events and keeps it

current. Another file is generated for the interactive visualization tool. This file contains the static primitive/composite event and rule definitions in one application.

The visualization tool communicates with the local event detector and the transaction manager. The runtime information about the event occurrence and rule execution is obtained from the local event detector. Also, from the transaction manager, the tool gets the transaction ids of sub-transactions within which the rules are executed.

5.2 Interactive Visualization Tool

The visualization tool has been extended to enable user interaction at run time. Interaction through a two-way communication channel, rather than passively receiving information, allows the user to make changes (enable, disable) on the rule set at run time. The user can set breakpoints during a debugging session so that the state of current rule/event execution can be inspected, and user can enable/disable rules or events at run time, like a conventional programming debugging tool. The user also can graphically contrast event graph constructed for the purpose of detecting the composite events with the state of a transaction at different rule execution points. Other debugging features include a utility for detecting potential cycles in rule execution and a utility to examine different rule execution paths from the same point in the rule triggering process.

To support the various test modes, the tool supports 3 modes of operation: post-analysis, real time analysis, and interactive-analysis. A mode is specified at compilation time with a proper switch to Spp (sentinel preprocessor) in an abstract fashion so that the user does not need to know how it works internally. The difference between the modes is how the rule execution information is supplied to the tool at run time and whether the mode accepts any user interaction during run time.

Post-analysis does not allow any interaction while the user application runs. Instead, the LED and transaction management module writes event occurrence and rules that are fired into a log file as they happen. The visualization tool reads the log file through the proxy server and simulates the event occurrence and rule executions. Besides, the user can select either step mode or continuous mode with post-analysis. Step mode will step through the log, stop at the next the stop and wait for the user to click the next button to continue the simulation. The unit of consecutive execution in the step mode is the interval between an event occurrence and rule execution. When continuous mode is chosen, the unit of trace is the entire application and the simulation goes through the log file consecutively until it finishes. Both options are desirable because the step mode enables the user to watch the system change on a finer scale, while the batch mode suits the situation when the user wishes to visualize and understand the result of an execution as a whole.

Real time analysis does not allow any user interaction while the application executes. The difference between post-analysis and real-time analysis is *when* the tool displays rule execution on canvas. Real-time analysis is used to show the change of active database as it occurs. While the tool and the user application runs at the same time, the information about event occurrence, execution of rule is sent to the tool through socket connection. In other words, LED sends the information about events and rules execution to the proxy as they occur and the proxy in turn routes this information to the registered visualization tool. When the tool receives the message, the tool parses the message to figure out what it should show on canvas in a knowledgeable way and draw the result.

Finally, interactive-analysis allows the user to intervene during application execution. The user, typically, selects this mode to set breakpoints during a debugging session so that the state of current rule/event execution can be inspected, or to enable/disable rules or events at run time. The interactive feature is implemented using a socket connection, which is similar to the real time analysis scenario.

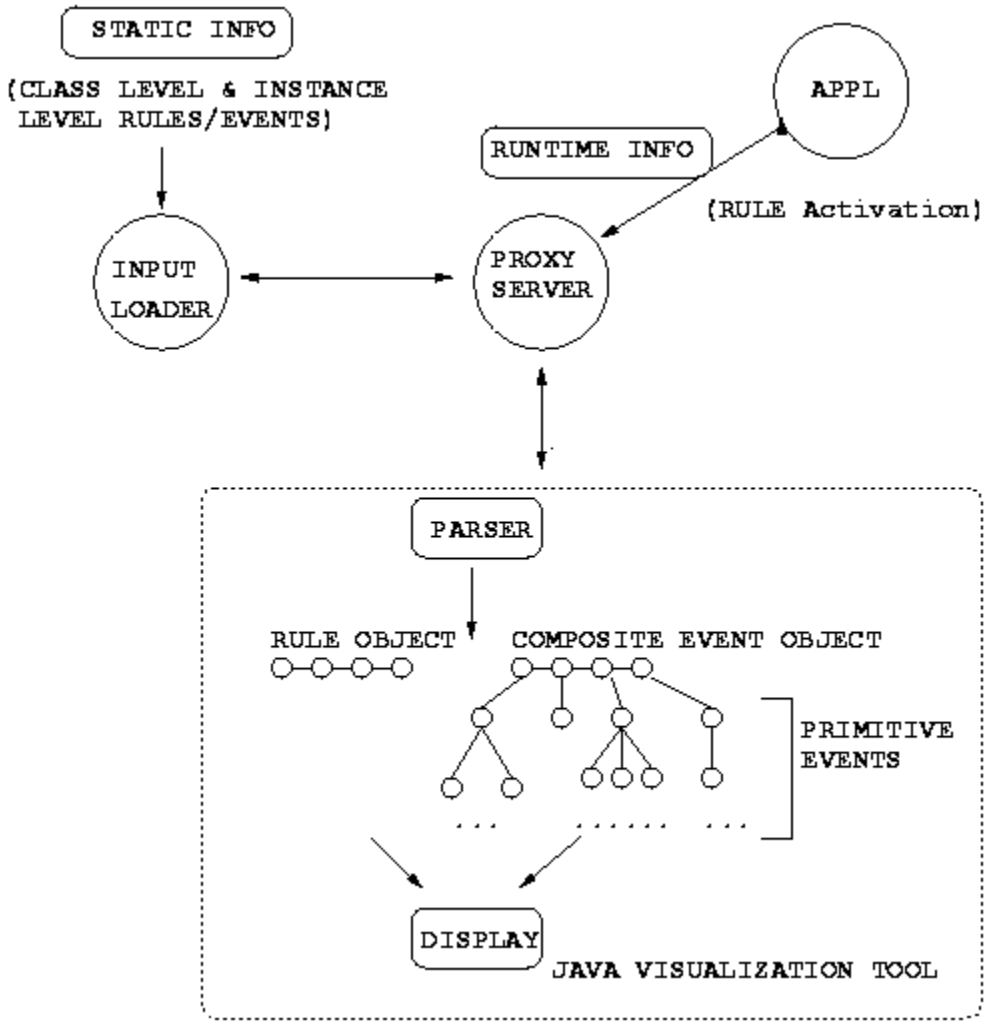

Figure 5-2 Input to the Visualization Tool

The visualization tool reads the static information about event trees and rule definitions, generated by spp (sentinel preprocessor) when the application is compiled, constructs the event graph, and stores the event and rule information in memory. The

runtime information such as rule activation and firing event is supplied in the form of a log file or socket connection according to the 3 different modes we have discussed. The data structure that captures the nested execution of rules is a n-ary tree. A transaction can have any number of sub-transactions and each sub-transaction in turn can have its own child sub-transactions. The root transaction is called the top-level transaction and all others are called sub-transactions. The transaction tree grows in a top-down way: it starts from the top-level transaction and spans to descendents.

The transaction manager also generates the transaction ID, which is used to infer the parent and child relationship among transactions. The naming scheme for transactions is a general-purpose one to accommodate multiple levels and multiple sub-transaction. We support concurrent rule execution using the nested transaction model. The visualization tool should be able to show the concurrent rule execution of an application as it happens. Figure 5-3 shows two sample outputs, comparing serialized execution with concurrent execution. In Figure 5-3, STOCK_e2, STOCK_e3, and STOCK_e_AND which stands for the composite event STOCK_e2 and STOCK_e3 are defined. When STOCK_e3 occurs, the event propagates to STOCK_e_AND, and fires Rule R3, while STOCK_e_AND fires R4 at the same time. In serialized execution output, sub-transactions 1001 and 1002 are serialized in terms of execution and commit. In contrast, sub-transactions 1001 and 1002 in concurrent execution output overlap in terms of execution and commit, which means the sub-transactions are executed at the same time. But, the causal relationship between the event and the sub-transaction is not represented clearly in the output. The tool needs to find out which event is associated with which sub-transaction. For this purpose, we include event name with the sub-transaction ID when

the LED module generates sub-transaction messages to the tool. The final output format is discussed in the next section and the sample output is included in APPENDIX B.

Toplevel 1	**Toplevel 1**
Event STOCK_e2 6372064	**Event STOCK_e2 6372064**
SubTransaction 1000	**SubTransaction 1000**
Rule R2 6374352	**Rule R2 6374352**
SubCommit 1000	**SubCommit 1000**
Event STOCK_e3 6373840	**Event STOCK_e3 6373840**
Event STOCK_e_AND 6374072	**Event STOCK_e_AND 6374072**
SubTransaction 1002	**SubTransaction 1001**
Rule R4 6376200	**SubTransaction 1002**
SubCommit 1002	**Rule R4 6376200**
SubTransaction 1001	**Rule R3 6374504**
Rule R3 6374504	**SubCommit 1001**
SubCommit 1001	**SubCommit 1002**
.....	**.....**

Figure 5-3 Serialized vs Concurrent Rule Execution

5.2.1 Implementation details of the Visualization tool

The previous version of the visualization tool did not support the nested sub-transaction model in the real time as well as the interactive mode. LED just simulated the nested transaction by inserting sub-transaction and sub-commit with the predefined sub-transaction ID when a rule was fired. The transaction ID was not the one generated by the transaction manager. The transaction ID only increased by one. Therefore we could not simulate the application that goes to more than 2 sub-transaction levels. In other words, we could not simulate and show the nested relationship between rules.

Besides, the previous output scheme could not differentiate between the internal events generated by rule execution and external primitive events generated by the application method call. This was one of the reasons why termination analysis was difficult. But, with the nested transaction model, we can differentiate the two kinds of

events and reduce the possibility of false cycle detection. This will be discussed in more detail in termination analysis.

The previous output scheme to static file or socket communication was somewhat ambiguous with respect to the relationship between triggering events and rules. Suppose an event triggers several rules and in turn these rules trigger new events. The interpretation and the relationship would be ambiguous in the trace generated earlier. The previous trace was not well defined to interpret and analyze the rule execution behavior for the user. We have reorganized the trace such that it is easier to interpret the relationship. The following output formats are generated from LED at run time and are acceptable to the visualization tool.

1. **Event EVENT_NAME O_ID**
2. **SubTransaction T_ID**
3. **Rule RULE_NAME O_ID EVENT_NAME**
4. **SubCommit T_ID**

The first format indicates that EVENT_NAME event occurs at this moment. The second format means that a new sub-transaction has started. From T_ID, we can compute the current level of transaction. Usually this format is followed by a third format of output because a sub-transaction always is associated with a rule. The third format indicates the EVENT_NAME event that triggers a new rule. The fourth is to indicate the end of the current transaction. Between the second and the fourth, we can have any number of events, rules and child sub-transactions. The visualization tool shows the relationship between rules and events by drawing a line. Parent and child relationship between transactions is shown in tree forms and relative positions of the sub-transactions.

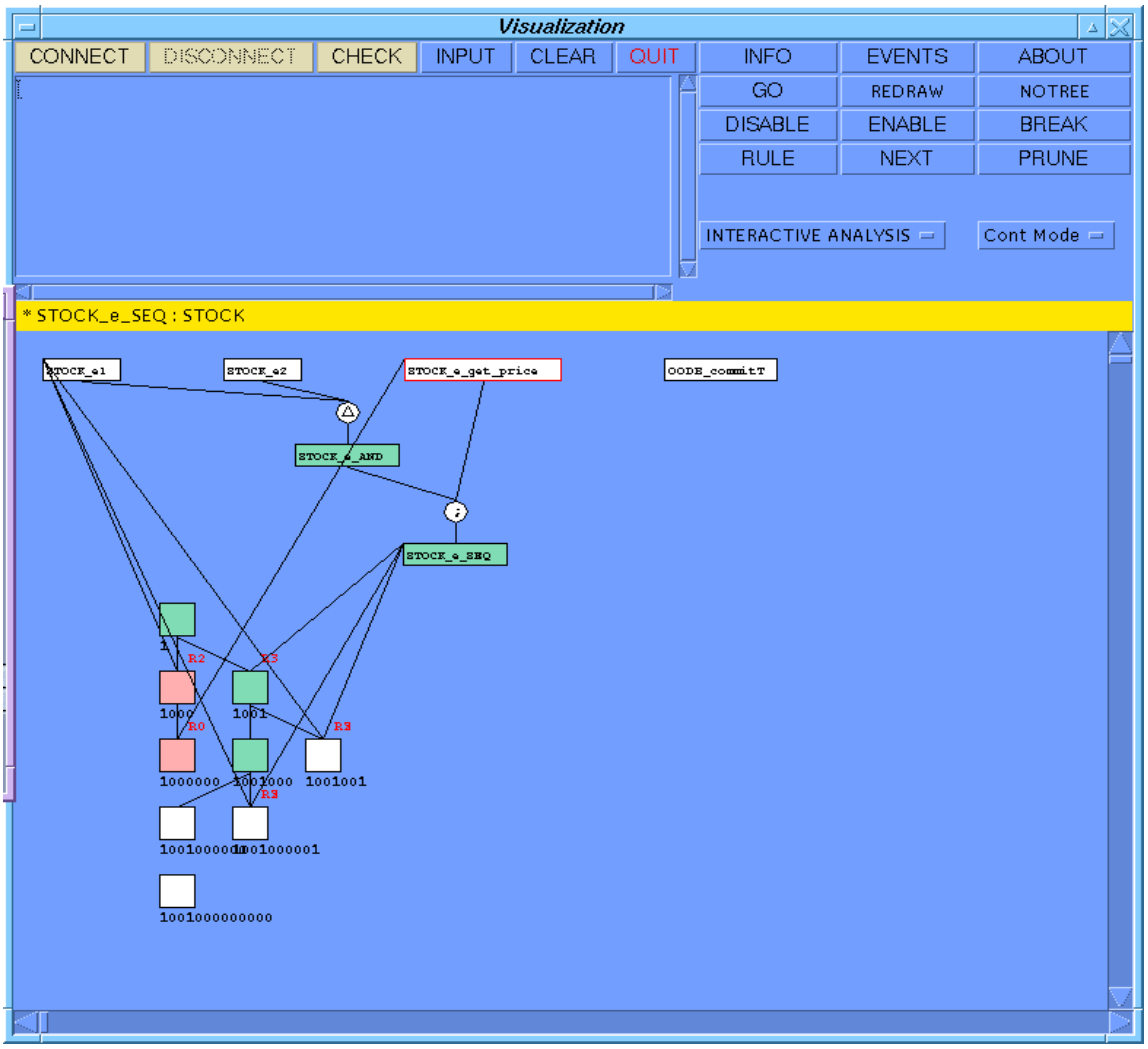

Figure 5-4 Execution of Visualization Tool

When an event occurs, the visualization tool only changes the color of the event node on the canvas. The user will know from this color change that the event happens. When a rule is actually scheduled, LED sends a message with sub-transaction ID, and the tool determines the position where the sub-transaction will be drawn from the ID. LED will also include the event ID or name of the event in the Rule execution output so that it can display the relationship between the sub-transaction and the triggering event. This is needed because it is not always true that the sub-transaction is triggered by the previously

received event right before the sub-transaction. In other words, it can be triggered by any one of the previous events.

To visualize a deferred rule execution separately from the immediate rule execution, the OODB_commitT primitive event is shown as an event in the visualization tool. The deferred rules are connected to the OODB_commitT primitive event when they are scheduled. In this way, the user can distinguish the rule executed in deferred mode from that of immediate mode.

5.2.2 Changes to LED for visualization

When an application is complied for run-time analysis or interactive analysis, the LED which is linked to the application registers itself to proxy at initialization time. We could have socket connection between application and visualization tool, without proxy. But, to be able to execute the tool in a distributed web environment easily as explained earlier, we chose the proxy architecture. This architecture can also support multiple views by multiple visualization tools. For one application, we can hook up applications to multiple visualization tools, each having a different subset of rules, with proxy acting as a coordinator among them.

Also, we define the semantics of disabling a rule as follows. When all the rules subscribed to a particular event are disabled, then LED will stop detecting the event. We extend the global hash table to implement this semantics. LED has one global hash table to get access to each event node in the event graph. Originally the hash table was used to map to only primitive event nodes, which are leaf nodes in event tree. We have extended the hash table to map both primitive events and composite events to the corresponding event nodes in the event graph. For the primitive event, we construct a key to the hash

table uniquely from class name, signature, and modifier for primitive event. For the composite event, we use the event name to map to an event node. Note that only the composite event with event name specified in user application can be inserted into global hash table in order to facilitate the disable or enable by visualization tool. The intermediate composite event node for which the user does not give name in user application is not accessible through the hash table, in the current implementation. If the user wants to access the intermediate composite event node, the user has to give an explicit name for that composite event node.

key	**#of rules**
e1	**2 -> 1 -> 0**
e2	**3 -> 2 -> 1**
e1_and_e2	**2-> 1 -> 0**

Figure 5-5 Change of counter in hashtable

Each entry in the hash table keeps track of the number of rules that subscribe to an event node. When a rule is associated with an event, the counter will be increased by 1 if the event is a primitive event, or the counter for all constituent events will be increased by 1 if the event is a composite event. If the visualization tool disables a primitive event, the enable flag of the event is reset so that the LED will not detect this primitive event. If a composite event is disabled, the event tree is traversed down to the leaf and decrement the counter along the path. If a rule is disabled, the event tree is traversed down to the leaf and the counters along the path are decremented. This was implemented by calling decreament_hash_entry for each child node which it subscribes to, which in turn calls decreament_has_entry of that node's child if it is a composite node or decreases the counter if that child is a primitive event. When the counter of the primitive event becomes 0 after the associated rule is disabled (which means no associated rule is

enabled), LED stops detecting the primitive event. This is done by is_this_subscribe function module. Whenever the primitive event is raised, it first checks this function. This function will return true when enabled associated rules exist so the above semantics can work correctly. For example, suppose we have an event graph which consists of e1, e2, e1_and_e2, and R1, R2 is defined for different contexts of event e1_and_e2. And R3 is defined on e2. Figure 5-5 shows the result of the hash table after we sequentially disable R1 and R2. The primitive event e1 and the composite event e1_and_e2 will not be detected any more. However, e2 is detected, as there is still a subscribed rule on that.

The reverse operation will be performed when a rule is enabled again by the visualization tool on behalf of the user. If the rule is associated with a primitive event, then the counter is finally increased by one. If the rule is associated with a composite event, the counter of all the constituents of the given composite event is increased by one. As a result, the primitive event starts detecting and propagating events.

The increment and decrement of counters are performed in a recursive fashion to eliminate the need for additional data structure. It is done in a top down manner, while event propagation takes place in a bottom-up approach. As the Led is multi-threaded, access and updates to shared data structures are properly synchronized.

Variables used by the visualization tool are set using environment variables, removing all the hard coding of port and host name from the application. This avoids recompilation of code when the machine on which the tool or the application is executed is changed. For a Sentinel application, the environment variables VHOST, VPORT, and CONCURRENCY are used to register with the proxy and interact with the visualization tool. VHOST is the machine where the proxy is running, VPORT is the port used by the

proxy. CONCURRENCY specifies the number of threads that can be scheduled at the same time. Users should be careful to set this variable correctly according the application semantics. For example, when the application has nested transactions and concurrent execution of rules, and the user sets the concurrency level to 1, then the application may not be executed according to our expectation. For a Sentinel application to register its identity to proxy and wait for a signal from the Visualization tool, the proper environment variables should be set.

5.2.3 Java Proxy implementation

As we explained earlier, we have identified the general-purpose proxy as an intermediate server, which can invoke the remote processes, and transmit asynchronous message across the platform boundary. These two utilities are integrated and combined into one proxy process written in Java.

To get asynchronous messages from outside, the user interface (in this case, the visualization tool) should register itself with proxy. Also, in order to interact and exchange run time information with the visualization tool, the Sentinel application compiled with a proper spp switch also is required to register itself to proxy, as we explained earlier. In addition, to load the static information containing event and rule definition, the Input-loader also joins the communication channel. The request for static information from visualization tool will relay to input-loader through proxy, and after input loader reads the static file, it sends the event and rule definition back to visualization tool.

Loading the static information could have been simply done by file read operation in the visualization tool, without the help of proxy. But, Accessing the static information

by file operation is not extendible. For example, suppose that the tool is executed in a distributed environment and the file containing the static information is not accessible because the file is on a remote site and the remote site is down or is not accessible temporarily. In this case, the tool could not load the static information about the application and could not continue the additional operation. Therefore, Adapting to the general proxy architecture described in Chapter 4, we separate the application logic from the service logic.

When the proxy receives the registration information from an application or the visualization tool, it stores the connection with the registering-id into socket connection list. It also spawns a daemon thread to wait for incoming messages from each connection. Each daemon has its own identifier, which is the same as the registration identification, and socket connection as its member. The daemon is implemented as a thread so that it independently transfers the message. When the daemon (or Proxy) receive the message, the daemon will wake up, read the message, and find the socket connection where the message should be placed from the list, finally writing message to the connection or connections.

In summary, Our proxy does not include any functional logic in its implementation. It just receive message from outside, find the destination group of processes or a process from the list, and finally transfers the message to that group of processes or a process. It also includes a utility to invoke the remote process in a remote machine to set up communication channel.

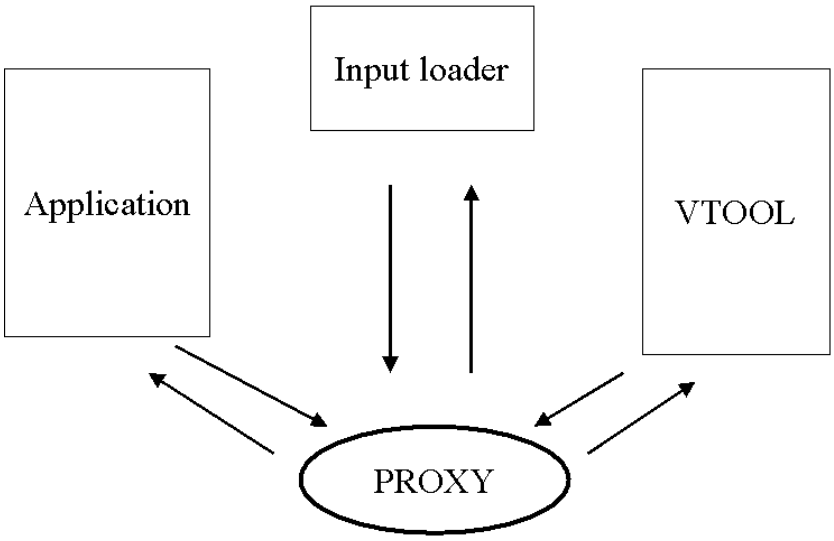

Figure 5-6 Communication channel in the Visualization Tool

5.3 Termination Analysis

The visualization tool is also extended to find potential cycles that could occur during the execution of an application and point out which rule subset is potentially leading to an infinite loop. This functionality is implemented as a sub-package of the visualization tool. This information helps the rule designer to find the cycle within the narrow scope of rules, instead of going over all defined rules. We will summarize why it is difficult to determine the cyclic rule set and show our approach to this issue.

5.3.1 Non-deterministic behavior in rule execution

One of the difficulties in performing termination analysis is the dynamic behavior of a rule, because its condition evaluation can vary according to the Active DB states, and its execution may change the database state. The change may cause some other rule's condition to become true. In some cases, it can be the opposite, which means that the rule execution may change the state such that other triggered rules' condition become false. This dynamic behavior becomes more complicated when we consider the priority and

rule-scheduling schemes. Rules may behave differently when scheduled in different orders, yielding unexpected results. Suppose we have two rules when an event occurs. We assume that the execution of the first rule will change the database state into a state, which will make the second rule's condition true. If we execute the second rule first, both rules may not get executed. But, in the other order, both rules would be triggered. The same discussion applies to the priorities among rules.

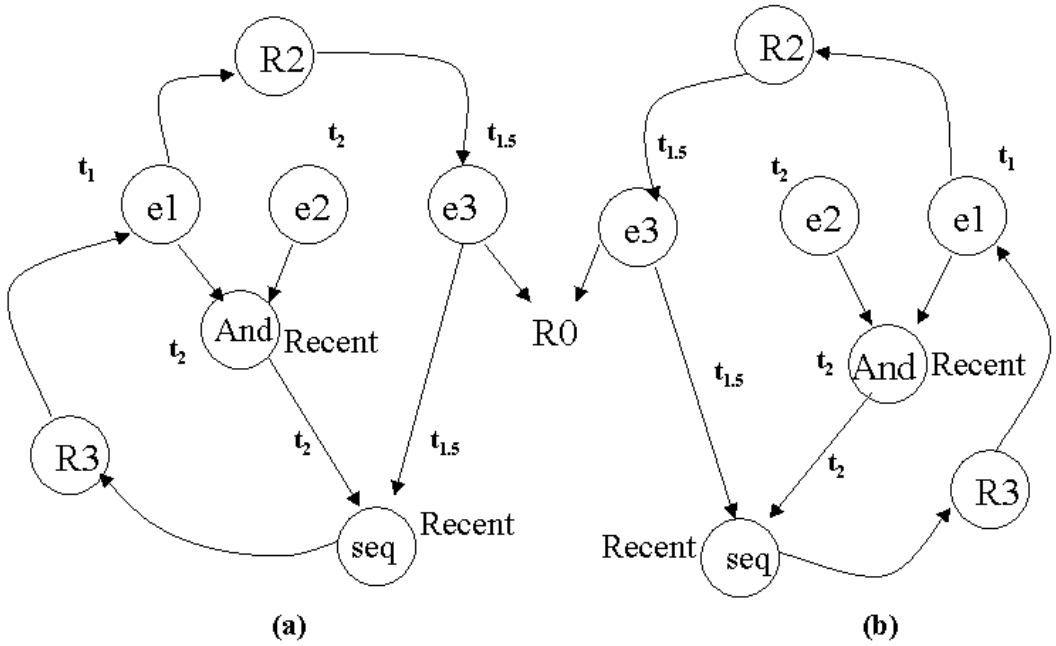

Figure 5-7 Cyclic Rule Set Example

In addition, the rule behavior is very closely related to the semantics of the associated event, and the behavior of event propagation. Figure 5-7 shows an example of how these factors affect the termination analysis. Event tree shown in both diagrams consists of e1, e2, AND, and sequence events. Event e3 occurs by the execution of rule R2. The time when each event occurs is also shown as $\mathbf{t_i}$. Also, $\mathbf{t_k}$ is assumed to be greater than $\mathbf{t_j}$ if is greater than j. As the semantics of sequence event is to detect the event when left child event occurs before the right child event, LED will not detect the sequence

event in Figure 5-7 (a). As a result, R3 will not be fired in Figure 5-7 (a). As a result, there is no cycle in the graph in the first diagram. But, in the similar event tree in Figure 5-7 (b), the sequence event occurs repeatedly and fires rule R3. If we consider the dynamic behavior according to context, the problem becomes more complicated. For instance, we may have a different result if R3 is created in a context other than RECENT.

Another difficulty is that the rule execution may change the database state in such a way that some cyclic rules may stop at some point. This cannot be determined before we actually run the application and find out these facts. For example, suppose we have a rule whose action is to deduct $100 from saving account only when saving account has more than $100. If this rule is part of a cycle, then the cycle eventually stops at some point.

In some applications, the steady arrival of outside events may form a false triggering edge between events so we may end up with false cycle detection. All these discussion supports that we may end up with a false cycle detection in many cases.

5.3.2 Our approach

We consider two approaches to this problem, in our case. One is a passive approach, similar to a conventional programming debugger. The debugger just displays the execution trace and lets the user infer the details of the problem. The debugger does not find rule set that is potentially cyclic. Suppose an application in a conventional language has recursive function calls, for example, function A calls function B and function B in turn calls function A. A traditional debugger will trap the application execution when it reaches the maximum depth of execution set for the systems. The user will try to find the source of the problem from the change of stacks. From the stack

output, the user needs to get a better understanding of the problem. In the same way, the visualization tool just displays the stack frame change and the user himself/herself needs to infer the source of the problem and modify the rule definitions accordingly.

The second one is an active approach, which displays the cyclic rules graphically to users at run time to help users visualize the nature of the problem.

We chose the first approach for implementation on account of its simplicity. We will present our stack frame approach first and discuss the integration of the second approach within the tool in the next section. Our approach forms an event triggering graph using event/rule definition other than the condition evaluation, but assumes the condition evaluation information can be added by the run time trace, later. We use the event trees to form an event triggering graph and add the edges (or relationship) between events with event relationship from rule definition.

Definitions 1 ***The event triggering graph*** (ETG) is a directed graph {V, E}, where each node v_i in V corresponds to an event e_i and E to the directed arcs $<e_i, e_j>$, which means that the relationship between e_i and e_j exists through composite event definition or rule execution.

Even if we find a cycle in the graph, we cannot conclude that the set of rules will not terminate at run time. This is because we did not include the dynamic information, that is, condition evaluation. The condition evaluation is a computation that returns a true or a false according to the database state. Generally, the condition cannot add any edge to the event graph, statically. For example, suppose we have a condition for a rule R that will return false if the price is below $10 dollars. This information cannot be added in the event triggering graph statically.

Commercial systems typically use a counter to detect potential cycles. They detect loops by keeping counters on the number and depth of cascading rules, and suspending rule execution when the counters exceed given thresholds. This approach suffers from the following limitation. Setting a value for the counters is quite critical and difficult: If the threshold is too low, rule processing will stop and give a false cycle warning. If the threshold is too high, a loop may be detected only after expensive processing [10].

When the application shows a cyclic execution, the tool will have the repeated stack frames in memory. This situation is similar to recursive functional calls in traditional programming languages. The number of frames on the stack will increase indefinitely until there is no more memory available. In the same way, analyzing the stack frames and comparing with ETG, the user can easily identify cyclic rules.

Definition 2 A ***stack frame*** contains all events triggered, and active rules at the same level (i.e., current sub-transaction and its sibling sub-transactions). The stack frame will be freed when there are no more active rules.

When the visualization tool reaches the maximum level that is set in the system, the kernel module will trap the application execution and stop it. The tool will dump the static cyclic rule set found from ETG and stack frames to users. From the output, the user can infer the details of the problem. The output generated by the utility significantly narrows the scope of rules, which the user should inspect to find cyclic rules when the application does not terminate normally. This is compared to backward reasoning rather than forward reasoning. Forward reasoning is to inspect all the facts and arrive at a conclusion while backward reasoning comes to a conclusion and finds out the supporting

facts. Forward reasoning can be very expensive if there are many rules defined in the system. In our case, we make a conclusion that there are cyclic rules in the output once the application does not terminate in a normal way, and then identify the cycle from the output generated by the utility. Instead of inspecting each event and rule to find cyclic rules, the user needs to analyze only the portion of rules and events pointed to by the utility. This approach saves time and effort especially when there are many rules and events to inspect.

Algorithm for Cycle Detection

```
make a event triggering graph G from the modified static file;
load the dynamic file into buffer B.
set clevel = 0; // clevel = current_frame_level`

For(each line i from the dynamic file which is a snapshot of user application) {
switch(i) {
case (i is a kind of Event) :
put that event into the current frame;
break;

case (i is an indication of a new subtransaction) :
calcuate the level from the subransaction;
Let k is the result of the previous calculation;
change the current_frame_level to k;
break;

case (i is an indication of a subtransaction commit) :
calcuate the level from the subcommit;
Let k is the result of the previous calculation;
distroy the frame of k level;
change the current_frame_level to k-1;
break;

case (i is a kind of Rule) :
put that rule into the current_frame_level;
for(i = 0, COUNTER=0; i < clevel; i++) {
j = i+1;
while(j <= clevel) {
if(frame[i] == frame[j]) COUNTER++;
if((COUNTER = THRESHOLD)
```

```
&&(the events in each frame are subset of cyclic set in static analysis)){
We detect the cyclic rules;
Give warning the user to check the rule sets; break
     }
j++;
}// end of while
   }
}
```

An Example

When the user runs the application that has the events and rules shown in Figure 5-7 (b), the application will not terminate in a normal way because it reaches the maximum depth defined in sentinel and the kernel module in sentinel sends a terminate signal to the application. This fact is visualized and shown to the user as shown in Figure 5-4. Note that sub-transaction 1000 and 1000000 are committed (this is shown as red boxes in the visualization tool), and the other sub-transactions are terminated without committing.

Then, the user runs the cycle detection utility to find the potential cycle rules. The utility first constructs the event triggering graph (the graph is the same as Figure 5-7(b)) from static information of the application and finds cycles (one of them consists of e1, AND, SEQ) in the graph. It also finds repeated stack patterns as shown in Figure 5-8(c) when it analyzes the run time information. Figure 5-8 shows the change of stack frames in the utility, which uses a dynamic run time information (This is included in the APPENDIX B as a sample trace) of the same application. In the Figure 5-8, (a), (b), and (c) corresponds to the stack frmes contents when the trace has been processed up to (a), (b), and (c) respectively in the trace shown in APPENDIX B. As a result, the tool gives those patterns and cycles in the graph to the user as a possible cyclic rule set. Note that

the events in a cycle are the subset of events in the repeated pattern (e.g., e1, AND, and SEQ).

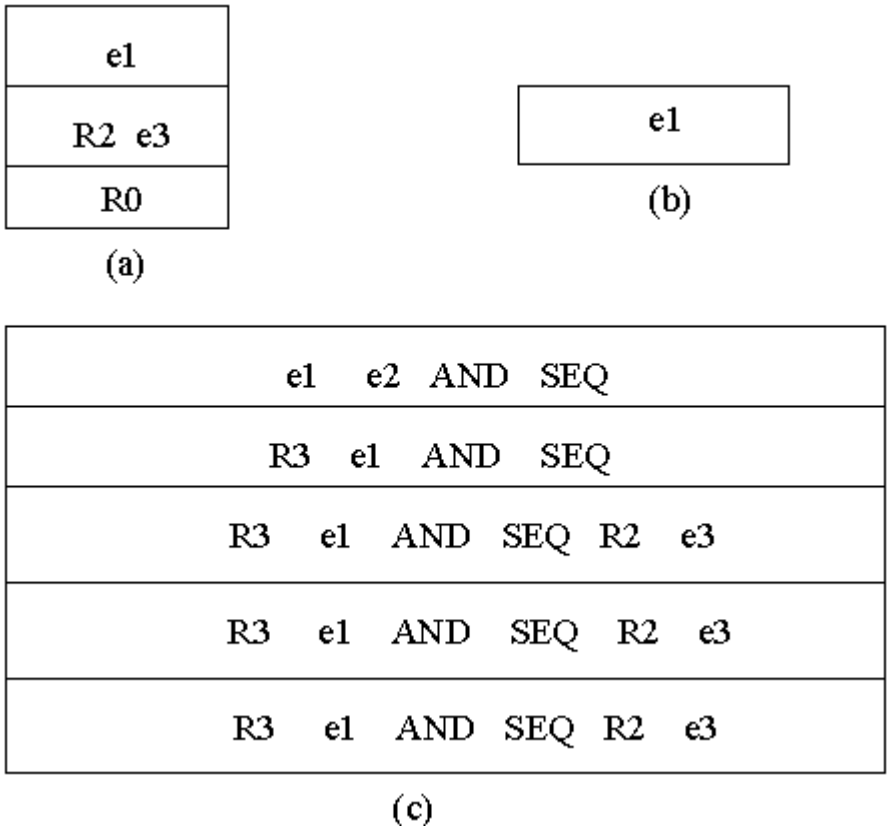

Figure 5-8 Changes of Stack Frames with APPENDIX B as Run Time information

5.3.3 Visualization of the Cyclic Rules

The previous stack frame approach is passive in terms of supporting the analysis of cyclic rules because the tool needs user's reasoning processing with ETG and the stack output to find the cycles. The tool would be user-friendly if it is able to show the cyclic behavior to users at run time, in the same way it shows the rule execution and event occurrence. Besides, the stack frame approach has a limitation. It needs an arbitrary numbers of stack frames to be grouped to detect repeating patterns. For example, Figure 5-9 shows 3 simple different cyclic rules and the change of their stack frames at run time.

Note that the cyclic rule path length determines the number of stack frames to identify the repeated pattern. Of course, the examples in Figure 5-9 are simple cases so it is easy to identify the groups. But, in the real applications, which also have composite events and concurrent rule execution, this grouping is not easy for users.

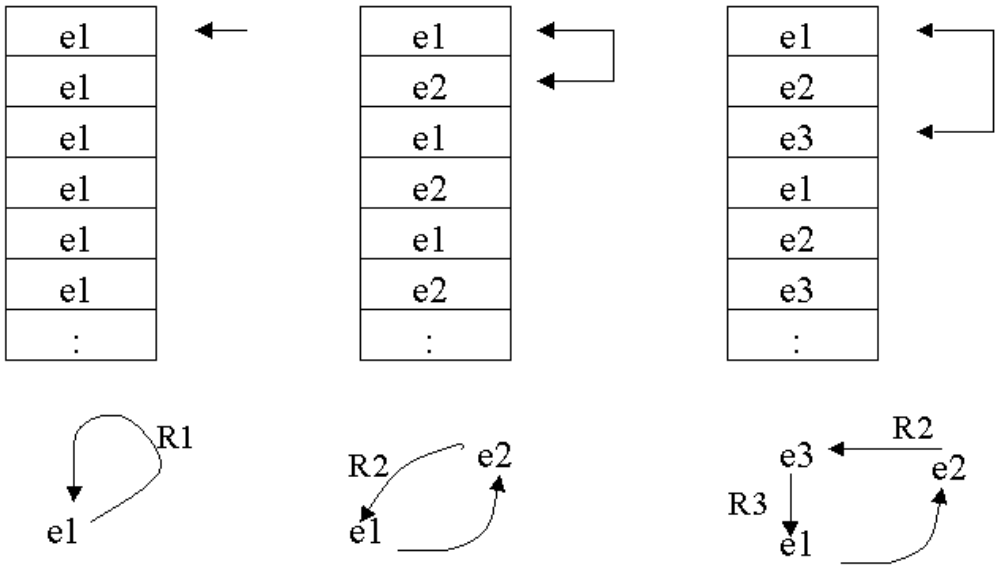

Figure 5-9 Grouping stacks to identify the repeated patterns

Figure 5-10 shows one of ways visualizing cyclic rules to users. Each event and rule is treated as a type. It does not show the instance level of event occurrence and rule firing.

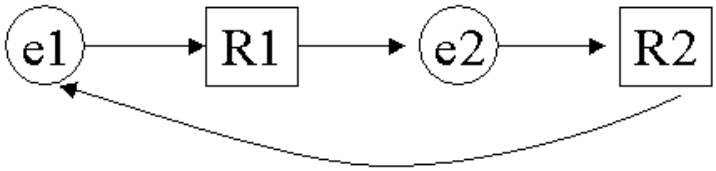

Figure 5-10 Visualization of Cyclic Rules as Types

The graph can be generated statically, but needs to be verified at run time. The visualization tool currently generates event graph statically from event definition and

displays the rule execution at the instance level to visualize the nested execution of rules. The edges between events and rules are generated at run time.

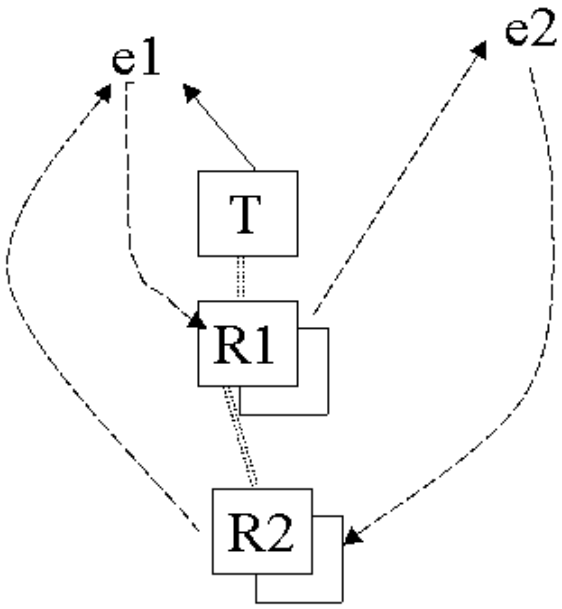

Figure 5-11 Visualization of cyclic rules in instance level

Figure 5-11 shows the visualization of cyclic rules at the instance level, which overcomes the disadvantages of the display scheme in Figure 5-10. Instead of displaying the n-ary tree, the tool positions the same rules at different transaction level side by side. In other words, when the tool detects the same rule execution at different transaction level, the tool assumes a cyclic mode, which is differentiated from display mode of rule execution as a transaction hierarchy explained earlier. To visualize cyclic rules such as the one shown in Figure 5-11, the tool should be able to identify the rule that generates a certain event and to identify the event that triggers certain rules. In other words, to draw the edge between R1 and e2 and edge between e2 and R2 in Figure 5-11, the tool should be able to identify, from the run time trace, which event is generated by the execution of a rule and which rule is triggered by an event. The tool would obtain this information from LED and draw direct edges between events and rules in the above graph at run time.

Current implementation generates messages for the second case. From the message "**Rule RULE_NAME O_ID EVENT_NAME",** the tool can find the identity of the event that triggers a rule. In addition, the information to draw an edge from a rule to an event also can be easily obtained from LED by generating the current transaction ID with event occurrence. From the transaction ID, the tool is able to know which rule generates an event and show it graphically.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This thesis significantly extends the previous visualization tool developed for Sentinel. The Previous work was primarily concerned with the static analysis and did not incorporate the nested sub-transaction model in the tool. There was no cycle detection mechanism either. The visualization tool has been changed to incorporate the nested sub-transaction model, and extended to have user interaction at run time. The functionality includes setting breakpoints, disabling events and rules, and enabling events & rules at run time. The work encompasses the utility to narrow the rule set and events to look for cycles when the application does not finish normally. A user can reduce the scope of debugging rules and events by pruning irrelevant trees from the tool.

Besides, this thesis presents a general way to extend active capability to the web-based distributed environment. The work extended the previous 3-tier architecture and redesigned the proxy to make web-based GUI possible. The redesigning of proxy has considered making remote process invocation possible to set up the communication among processes, also.

Currently the visualization tool supports interaction with LED module. The same can be extended to interaction with GED module easily. Running multiple visualization tools, for example, one for each application and one for GED, will be helpful to analyze the rule execution at a global level. One challenging work involves detecting cyclic rule sets across applications. Two independent applications work as producer and consumer in

GED and they may contain a cyclic rule in a global view, but may not have any cyclic local rule set. If a tool can help in detecting these cycles, it would be useful for application designer to redesign rules when it is necessary.

The other feature we can think of in the tool is rule modification functionality at run time. Currently, we are able to only enable/disable rules at run time. It would be helpful for rule designer to be able to change the condition, action part, and priority of a certain rule at run time.

APPENDIX A
A PROTOTYPE IMPLEMENTATION OF JAVA LED

As explained in the previous chapters, the platform independent features were a strong reason for choosing Java as the primary language for developing user interfaces. While we explored the Java language features, we noticed many new language features that would overcome some of the limitations in C++. In this chapter, we analyze the limitation of LED implementation in C++ and discuss features in Java that can be applied to improve the event detection process. Finally, we discuss the detailed prototype implementation of Java LED, in terms of event propagation.

1. Limitation of LED in C++

Parameter contexts were introduced to categorize the event detection as well as parameter computation. Each composite node maintains separate lists for each context and each child. For example, An ‘And’ node keeps 4 separate lists for the left child (One for each context computation) and another 4 lists for the right child. In order to propagate the same event for distinct contexts and independent parameter computation, the same parameter information was duplicated in several places. This approach increases storage requirement as the event graph grows without providing any additional advantage. Sharing of parameter values across context scan reduce storage overhead significantly. In C++, when we allocate memory for a data structure or a class, we have to free the

memory explicitly. Because we use the event propagation tree to detect the composite event, not freeing the memory appropriately could easily lead to memory leaks.

Another disadvantage of the LED implementation in C++ is that it has difficulty in passing complex data types as parameters. It only supports passing only simple data types (e.g., integer, float, character, and string by value) as parameter. This limitation comes from run-time type information in C++. C++ does not have any base class to point or reference for later manipulation. Current implementation in C++ keeps 4 member types (INT, CHAR, STRING, FLOAT, TEMPORAL) for passing the parameter even if only one of them is actually used. No other complex class or data structure, which may be defined in user application, can be passed as a parameter.

In addition, the language feature also limits Rule representation in C++. Currently, condition and action in C++ are implemented as global function and passed to LED. LED cannot access the code for condition and action, which is not in user program space. Suppose that we have action part of code stored in implementation repository. LED may want to execute those codes at run time. In C++, LED cannot use those codes if user did not link and load those codes in application when starting the application.

2. Java language features for LED

The most frequently reported problem in C++ is memory leak. Users always should pair memory allocation with memory free operation explicitly in C or C++. There is no automatic mechanism to free the memory even if the machine is short of memory. On the contrary, Java uses a technique called garbage collection to automatically detect objects that are no longer being used (an object is no longer in use when there are no more references to it) and to free them [14]. It is a technique that has been around for

years in languages such as Lisp. The Java interpreter knows what objects it has allocated. It knows which variables refer to which objects, and which objects refer to which other objects. Thus, Java can figure out when any other object or variable no longer refers to an allocated object.

LED implementation in C++ has difficulty in passing complex data types as parameters. But, passing complex data type or user defined class as a parameter is the easiest one in Java because Java has a superclass Object and type-upcasting is done automatically. Complex data type or user defined class is upcasted to Object and passed around easily.

In addition, Java has new feature to enable a dynamic rule creation and loading. When a user defines action and condition part using a rule editor, for example, the rule editor will create files making a new class and put those files into some directory. After that, Java compiler can be invoked to compile and move the new rule definitions into rule repository. Then, using reflection or dynamic extension, new rules can be made available to user application. In other words, Java allows us to dynamically extend the program at runtime. Java programs can dynamically extend themselves by choosing, at runtime, classes and interfaces to load and use. It means we don’t have to know about all the classes and interfaces of user application programs at compile time. Using dynamic extension feature in Java, we can pass the string type as a name of new action type or condition type in rule constructor in user application. Then, the rule class loads this new type in JVM so that it is available to the application. Because the type name is handled as a String at runtime, the program can be written such that actual contents of the Strings do not need to be known at compile time.

To make dynamic extension possible, Java has two kinds of class loader - the primordial class loader and class loader objects. Whenever the JVM loads a class or interface, it uses either the primordial class loader or a class loader object. Every type that a JVM loads, it creates an instance of class java.lang.Class to represent the type to the rest of the application. When you start the application, JVM finds out what class you need, load those classes, and create an instance for each class.

Java also supports dynamic method call binding. Connecting a method call to a method body is called binding. When binding is performed before the program is run (by compiler and linker), it's called early binding. Structural language such as C, Pascal, supports these kind of binding. The core idea of dynamic binding is "send a message to an object and let the object figure out the right thing to do". This is called "late binding". It is also called run-time binding because the binding occurs at run-time based on the type of object.

Of course, C++, one of object oriented languages, supports some kind of late binding, using virtual function, which was exploited in our C++ version of LED. But, this is a very limited way when we compare the Java late binding mechanism. Java extends late binding in such a way that some sort of type information can be stored in every object. And it also introduces an abstract class and interface, a new feature to support the core object model, which was not in C++.

An Abstract class is used to manipulate a set of class through a common interface. All-derived method calls that match the signature of the base-class declaration will be called using dynamic binding mechanism. An Abstract class is a good way to express only

the interface and not a particular implementation. This language feature was exploited and explained in detail when we discuss the class design for JAVA LED prototype.

The interface keyword takes the abstract concept one step further. An interface is a collection of abstract methods and constants. One thing we should note is that interface is considered as a new type in Java. In other words, When we create an instance of an interface, it is considered as an object. This follows the core object model. The interface is used to establish “protocol” between classes.

3. Design and Implementation of prototype Java LED

In the previous section, we discuss new features in Java that is useful for LED implementation. As a prototype, we focus on event propagation in binary operators for various contexts and implement those operators for Java LED to overcome a memory problem in C++ and improve event detection process. First, we show the design issues and then discuss the implementation details.

3.1 Design

Each composite node in C++ LED maintains separate lists for each context and each child. For example, An ‘And’ node keeps 4 separate lists for the left child (One for each context computation) and another 4 lists for the right child. So, the same parameters can be duplicated at most 4 different list nodes.

In Java LED, this duplication has been eliminated. Each composite node has one parameter context table for each child node. An event with parameter information propagated from the child is associated with 4 context bits, one for each context. Whenever the event is propagated from child, it is stored in the table and sets one of the

appropriate bits according to the event context. After an event is propagated and consumed, it is treated as a garbage entry if the associated context bit of the event becomes 0000. The bits indicate that there is no event left to be used for detection of event node. Garbage entry is collected and freed from data structure according to merge and propagation algorithm. For example, the context bits for e_1^1 of AND node in (c) of Figure 6-1 becomes 0000 (which means it is no longer needed) after event propagation is performed. As a part of algorithm, the event e_1^1 is discarded from the parameter context table.

The overall class hierarchy was designed to incorporate the new Java features such as dynamic method call binding, and reflection. C++ supports multiple inheritance of method implementations from more than one superclass at a time. Using this feature, LED in C++ version has a RULE class, which inherits from Reactive class and Notifiable class. This feature has been omitted in Java because it may introduce many complexities. Instead, Java encourage us to use interface so that we have to reconsider the class hierarchy according Java language features, without losing any features of C++ version. We make use of these features when we redesigned the class hierarchy. In short, a class in Java can only inherit method implementations from a single superclass, but it can inherit method declarations from any number of interfaces.

The prototype Java LED concentrated on event propagation algorithm for binary operator. Figure 6-1 shows overall class hierarchy and the relationship between classes and interfaces. Oval shapes, shaded rectangles, and plain rectangles represent interfaces, abstract classes, and implementation classes respectively. The prototype has two interfaces (Notifiable and Executable), and two abstract classes (Table and Event). Table

data structure was introduced to store the context information merged and propagated from the child node. Abstract Event class has a subscribed event list as a member, which contains all the references to the subscribed events. When the event is detected and propagated, LED traverses this list and gets reference to abstract table of subscribed events. We create table as an abstract class because parameter context table and event node has a mutual reference relationship. In other words, An event node needs reference of the parameter context tables of the subscribed events when the event is detected and propagated up to event tree. And a parameter context table needs to know what kind of event the table is associated with to decide the semantics of merge & propagation.

Up-direction arrow means a relationship between interface and implementation. As mentioned earlier, each event node has a subscribed event list as a member, which contains all the references to the subscribed events. Because the list may contain different kinds of event classes such as ‘Primitive’, ‘AND’, ‘SEQ’, and ‘OR’, the subscribed events are upcasted to Notifiable when they are accessed for the purpose of event propagation. Note that Java treats interface as a type. All subscribed events implement Notifiable interface so that the event node does not need to check run-time type and cast in order to call proper method of each subscribed event.

The arrow from Rule to Event class means a subscription relationship. When a rule instance subscribes to an event object and event occurs, the corresponding rule object will be triggered. Down-direction arrow means an inheritance relationship between classes. Because an abstract class in Java usually declares abstract methods with no implementation details and expect the subclasses will implement the corresponding method, abstract Event class just declares the Notifiable interface. Actual implementation

for Notifiable interface is done in the lower level classes like Primitive, AND, SEQ, and OR. In other words, these event operator classes inherit from Event class and implement their own operation of merge and propagation according to their semantics. Abstract Table class is impelemented by PCTable class, which is the data structure for merge computation. Also, this data structure checks garages and delete from the table after propagation to overcome a memory leak problem.

Finally, Rule has Executable interface and Notifiable interface, because rule class itself can be an event generator and can be notified from the associated event.

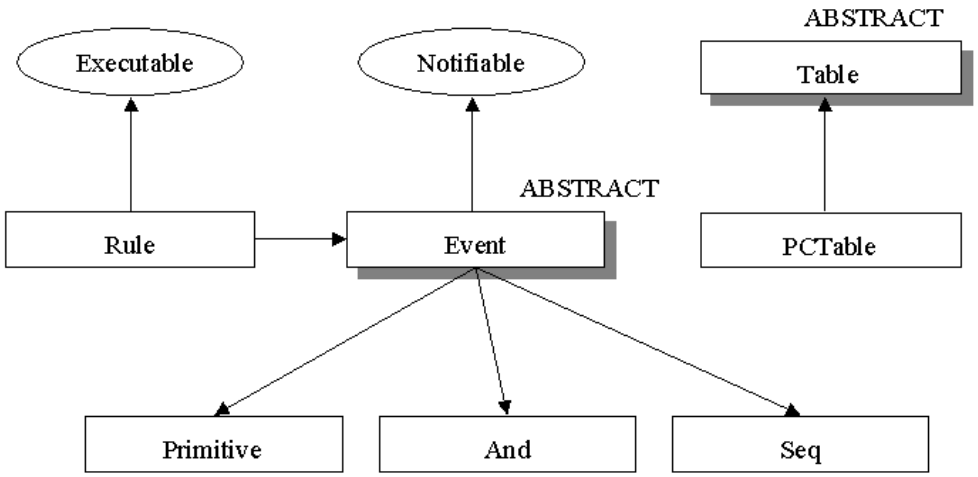

Figure 6-1 Overview of Class Hierarchy for Java LED

3.2 Merge and Propagation Algorithm

First we give simple example of event propagation and then generalize the algorithm. The example and algorithm follows the definition of each context given in Chapter 2.

In Figure 6-1 event tree for AND node consists of primitive e1, e2 and the time sequence of event occurrence were indicated with the arrows and time value t_1, t_2 and t_3 ($t_1 < t_2 < t_3$). At time t1, the event e_1 was propagated but no additional propagation would

not be performed because the left table of AND node is empty. At time t2, the event e_1 occurred, again. This changes the recent context bit of the previous event parameter like Figure 6-1 (b). At time t3 when e_2 occurs, the complex merge and propagation operation is performed in AND node. For explanation purpose, Figure 6-1(c) shows the intermediate computation and the result together. The merge operation would be different according to the semantics of context and operator. Notice the bit change in left and right table of AND node after merge. e_1^1 in Figure 6-1 (c) is garbage-collected and freed from left table of AND.

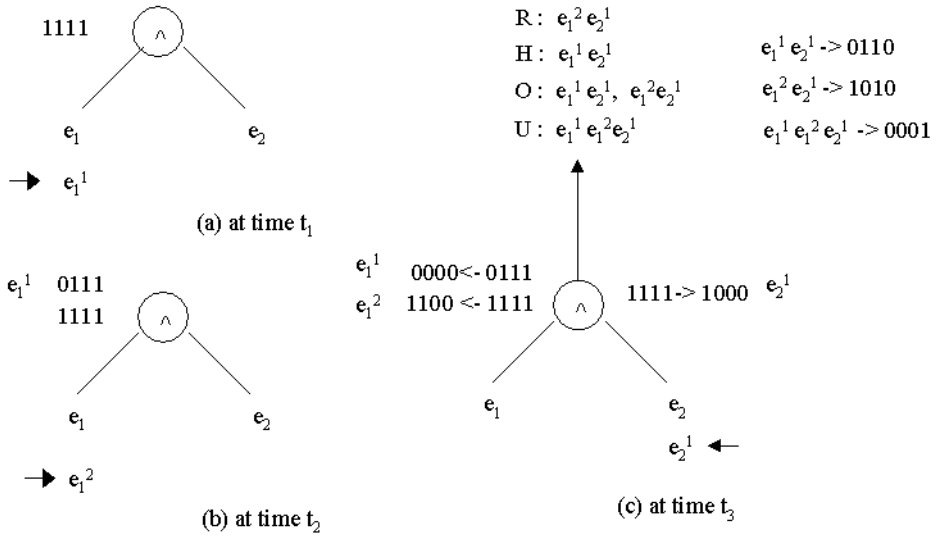

Figure 6-2 Example of event propagation in AND composite event

Now, we are ready to give generalized merge and propagation algorithm. The algorithm works in 2 steps, one for propagation, and one for merge in each composite node.

Algorithm 1 Merge & Propagation Algorithm

1. Propagate from child to parent & adjust existing bits in the parents
2. When merge(Check for combine and propagate TS and bits)
   Propagate merged events and their bits
   Readjust or Modify existing bits from where it is propagated

Garbage collection when it becomes 0000

Note that there are common behaviors for binary operators. First, when a newly detected event is propagated within RECENT context, the previous recent event in the parent event node will be no longer used for merge within Recent context. Instead, the newly detected event and parameter will be used for merge and propagation in the parent node. For instance, in Figure 6-1 (b), we can observe that the recent context bit of e_1^1 is clear and the recent context bit of e_1^2set after propagation. Secondly, when a composite event is propagated within RECENT context, the constituent events are not consumed in the recent context, while in the other contexts (CHRONICLE, CONTINUOUS, and CUMULATIVE) the events are consumed and the corresponding context bits are clear. For example, in Figure 6-1 (c), recent context bit of e_2^1 is still 1 but other context bits are reset after propagation. This is because an initiator of an event (primitive or composite) will continue to initiate new event occurrence until a new initiator occurs in recent context. Thirdly, because all occurrence of an event type are accumulated in cumulative context as instances of that event until a terminator is detected, all the occurrences that are used for detecting the event are packaged and propagated up to event tree. This is shown as e_1^1 e_1^2 e_2^1 in Figure 6-1 (c) with cumulative context bit set. When terminator event e_2^1 occurs, the cumulative context computation in AND accumulates e_1^1, e_1^2, and e_2^1, and propagate the result up to tree. This cumulative behavior is commonly shown in all operators.

The algorithm significantly reduced the duplicate pointers and does not require 4 separate list for each child of composite node to do parameter computation. The garbage collection is performed as a part of merge operation to reduce the overhead for the next

operation, and overcome memory leak in C++. The parameter type can be any user-defined type or class. For the primitive type such as float, int, string, and char, Sentinel package provides the wrapper to convert those into object internally and convert back, so user does not need to care about conversion details.

APPENDIX B
SAMPLE TRACE FILES

Toplevel 1
Event STOCK_e1 6363832
SubTransaction 1000
Rule R2 6367792 STOCK_e1
Event STOCK_e_3 6365376
SubTransaction 1000000
Rule R0 6367688 STOCK_e_3 **(a)**
SubCommit 1000000
SubCommit 1000 **(b)**
Event STOCK_e2 6365096
Event STOCK_e_AND 6365608
Event STOCK_e_SEQ 6367432
SubTransaction 1001
Rule R3 6367944 STOCK_e_SEQ
Event STOCK_e1 6363832
Event STOCK_e_AND 6365608
Event STOCK_e_SEQ 6367432
SubTransaction 1001000
Rule R3 6367944 STOCK_e_SEQ
Event STOCK_e1 6363832
Event STOCK_e_AND 6365608
Event STOCK_e_SEQ 6367432
SubTransaction 1001001
Rule R2 6367792 STOCK_e1
Event STOCK_e_3 6365376
SubTransaction 1001000000
Rule R3 6367944 STOCK_e_SEQ
Event STOCK_e1 6363832
Event STOCK_e_AND 6365608
Event STOCK_e_SEQ 6367432
SubTransaction 1001000001
Rule R2 6367792 STOCK_e1
Event STOCK_e_3 6365376
SubTransaction 1001000000000
Rule R3 6367944 STOCK_e_SEQ
Event STOCK_e1 6363832
Event STOCK_e_AND 6365608
Event STOCK_e_SEQ 6367432
SubTransaction 1001000000001
Rule R2 6367792 STOCK_e1
Event STOCK_e_3 6365376 **(c)**

REFERENCES

[1] S. Han, Three-Tire Architecture for Sentinel Applications and Tools: Separating Presentation from Functionality. Master's thesis, University of Florida, Gainesville, 1997

[2] V. Krishnaprasad, Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Mater's thesis, University of Florida, Gainesville, 1994

[3] S. Neelakantan, Scheduling Rules in an Active DBMS using Nested Transactions. University of Florida, Gainesville, 1998

[4] A. J. O. Diaz, and N.W. Paton, “DEAR: A Debugger for Active rules in an Object-Oriented Context,” presented at 1st International Conference on Rules in Database Systems, September 1993.

[5] S. D. U. Alexander, and Suzanee W. Dietrich, “PEARD: A Prototype Environment for Active Rule Debugging,” Intelligent Information Systems : Integrating Artificial Intelligence and Database Technologies, vol. 7, Number 2, October 1996.

[6] s. G. Anca Vaduva, and Klaus R. Dittrich, “Investgating Rule Termination in Active Database Systems with Expressive Rule Languages,” presented at 3rd International Workshop on rules in Database Systems (RIDS 97), Skoevde, Sweden, June 1997.

[7] J. G. a. H. McGilton, “The Java Language Environment: A White Paper” , 1996.

[8] IONA, The OrbixWeb 3.0 Programmer's Guide, http://www.iona.com/products/internet/orbixweb/fordevelopers.html, July 1997

[9] L. Hyesun, Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996

[10] S. C. Elena Baralis, and Setfano Paraboschi, “Compile-Time and Runtime Analysis of Active Behaviors,” IEEE Transactions on Knowledge and Data Engineering, vol. Vol 10, May/June 1998.

BIOGRAPHICAL SKETCH

Seokwon Yang was born in Chuncheon, Kangwon Province, Korea. He received the Bachelor of Computer Engineering, Honors degree in Computer Science and Engineering department from Hanyang University, Korea, in February 1997. He will receive his Master of Science degree in Computer Science from the University of Florida, Gainesville, in August 1999. His research interests include active, object-oriented databases.