Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Event Management for Large-Scale Distributed Systems

John Warne

Abstract

This document defines an architectural framework for event management in large-scale distributed systems. The framework adopts a client-server view of computation interactions in which every interaction point between a client and a server is a potential event. Facilities are provided which enable different parties to register an interest in explicitly “named” events, to observe their occurrences, to evaluate the parameters of their interaction points, and to take action upon them if specific conditions are met. Both the framework and its facilities conform to the ODP Basic Reference Model [ODP 93].

APM.1284.00.02 **Draft** 29th September 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Event Management for Large-Scale Distributed Systems

Event Management for Large-Scale Distributed Systems

John Warne

APM.1284.00.02

29th September 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document defines an architectural framework for event management in large-scale distributed systems. The framework adopts a client-server view of computation interactions in which every interaction point between a client and a server is a potential event. Facilities are provided which enable different parties to register an interest in explicitly "named" events, to observe their occurrences, to evaluate the parameters of their interaction points, and to take action upon them if specific conditions are met. Both the framework and its facilities conform to the ODP Basic Reference Model [ODP 94].

1.1 Motivation

The effective management of a distributed system requires the ability to observe the occurrence of both basic and composite events of interest, and, when appropriate, to trigger designated management actions.

The distinction between basic and composite events is simply one of number:

1. a *basic event* has a single source of generation, namely, the computation interaction point at which it occurs and at which it can be signalled to interested parties;
2. a *composite event* signifies several basic event occurrences which can be signalled as a collective whole.

Such basic and composite event tracking plays a crucial role in the solution to many fundamental distributed system problems, including deadlock detection and resolution, liveness and termination detection, token loss detection and regeneration, checkpointing and recovery, monitoring and debugging, and general system failure detection.

The solutions to these problems all share the common requirement of observing the occurrence of significant system events in order to evaluate some *System State Predicate* (SSP). The goal is to determine whether the state of the system satisfies a given set of conditions and thus a given SSP. Generic examples of such predicates are *fixed points* and *consistent cuts* [MISRA 92].

Consider the specific example of a distributed system checkpointing protocol. The protocol reacts to computation interactions by periodically taking partial snapshots of system state such that these snapshots can be used by a recovery process to reconstruct a previous global (consistent) system state. The SSP for this protocol specifies the conditions for which a set of partial snapshots will yield a global system state - a consistent cut.

An event-driven scheme for SSP evaluation can thus be seen as the core of a generic solution to many distributed system problems. This general need has motivated the event management framework presented in this document. With this framework in place, all that remains to be done is the formulation of appropriate SSPs and the construction of the actions which are to be taken when the predicates are satisfied. The framework itself provides the

foundations for animating such SSP evaluations and triggering the associated system actions.

An important class of SSP evaluations is related to *Quality of Service* (QoS) *Management*. In this class, each SSP specifies a set of conditions which, if realised, signifies a failure in the provision of QoS for some given set of service resources. By monitoring such resource usage and detecting deviations in expected QoS behaviour, the event management framework can be used to report QoS failures to QoS Management for remedial action.

It is a primary goal that the event management framework shall provide a QoS monitoring capability for the ANSA real-time QoS extensions proposed in [LI 94].

1.2 Scope

The document as whole scopes the applicability of event management for large scale distributed systems in open environments and the requirements for the supporting infrastructure. It provides the foundations for detailed designs of specific event management service implementations.

1.3 Audience

The intended audience are technical designers who are familiar with the principles of open distributed systems in general and object-based client-server models in particular.

The audience would be greatly assisted by some familiarity with ANSA principles [ANSA 91] and the ODP Basic Reference Model [ODP 94].

1.4 Document structure

The remainder of the document is divided into four chapters as follows.

Chapter 2 enumerates the functional requirements of the event management framework. These requirements are the fundamental guidelines for the design choices presented in Chapter 3.

Chapter 3 present the principles, concepts and functional components of the event management framework. It begins by describing the attributes of the model underlying the framework: an extension of the *Event-Condition-Action* (E-C-A) model for event based “triggers” enunciated in [McCARTHY 89]. The chapter continues by developing the computational view of triggers in terms of three components classes: event agents, condition evaluators and action monitors. It is shown how these components can be placed in a distributed system to achieve scalable, efficient event management.

Chapter 3 concludes with a statement of where the event management framework fits into the general picture of the ANSA management engine. This statement also indicates how the framework can serve as the basis for monitoring QoS related events and for reporting deviations from expected QoS performance to Qos management services.

Chapter 4 illustrates a practical application of event management.

Finally, chapter 5 provides a summary and some directions for future work.

2 Requirements

This chapter enumerates the functional requirements of the event management framework. These requirements are the fundamental guidelines for the design choices presented in Chapter 3.

2.1 Events as named computation interaction points

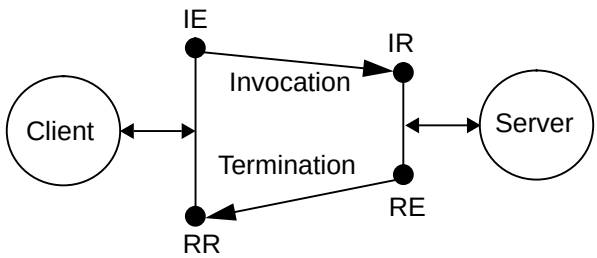
An event is some system state transition caused by a computation activity within an object. Each event thus has a scope which is local to a particular object and to a particular activity.

It is a requirement that any such event can be named and reported to interested parties. Accordingly - and in contrast to systems that support a fixed set of predefined events - the event management framework treats the interaction points of every object operation as a potential event. Moreover, parties can register an interest in an event by reference and binding to an associated named “event interface”. The details of this interface and its method of generation are developed in chapter 3

2.2 Support for client and server event observations

It is a requirement that the event management framework conforms to the ODP Basic Reference Model [ODP 94] and in particular to the ODP client-server interaction model illustrated in Figure 2.1

Figure 2.1: ODP client-server interaction model



The following brief explanation of this model is similar to that given in [REES 93a] and [ODP 94].

A client object interacts with a server object by invoking operations on interfaces provided by the server.

Four interaction points are distinguished in Figure 2.1: IE (*Invocation Emission*), IR (*Invocation Received*), RE (*Response Emission*) and RR (*Response Received*).

Interaction point IE is where the client engages in the initiation of the invocation. The event for this point has a value comprising the operation name and zero or more parameters. The client's expectation will change when it engages in this event; it will be expecting a response to the requested invocation.

Interaction point IR is where the server observes the client's invocation request. This is the "request event" [REES 93b] which informs the server that it has been invoked. The value of this event is expected to be the same as the value of the event at interaction point IE.

Interaction point RE generates the "termination event" [REES 93b] that occurs when the server has completed the evaluation of the operation. Its value comprises a termination name followed by zero or more parameters. The particular name and parameters are expected to be the ones defined by the behaviour of the server given the value of the event occurring at interaction point IR and any other events that may have been observed by the server.

Interaction point RR is the client's observation of the termination. It is expected to have the same value as the event generated at point RE.

By registering an interest in selected event pairs (IE, RR) and (IR, RE), different parties can observe event occurrences generated at selected client and server interaction points respectively, thus meeting the general requirement laid down in Section 2.1.

2.3 Use of rule based triggers as a unifying paradigm

Of late, there has been a surge of interest in the development of a unifying reactive capability for monitoring, evaluating and acting upon event occurrences in database management systems [HSU 88, McCARTHY 89, DAYAL 90, GEHANI 92, ANSWAR 93]. All models derived from this research are elegant and well-formed, and all are based on the unifying computation paradigm of "triggers". In its simplest form, a trigger is a rule which comprises a *triggering event*, a *condition* to be evaluated when the event occurs, and an *action* to be taken if the condition evaluates to be true.

There is a similar requirement to provide such a reactive capability for distributed computing systems in general, of which conventional distributed database systems are but a subset. In view of this, the event management framework adopts the trigger paradigm as the unifying approach for structured event handling in distributed systems.

Moreover, the definitions of triggers need to be made independent of the objects they monitor, thereby eliminating the need to change the definitions of objects every time triggers are added or deleted.

2.4 Events, conditions and actions with first class interfaces

Just as events are required to have well defined event interfaces (Section 2.1), so there is a similarly important requirement to associate well defined interfaces with conditions and actions. In this way, instances of events, conditions and actions can each be generated with an interface of its specific type.

The details of the interfaces for events, conditions and actions are developed in chapter 3.

2.5 Support for composite triggers

Many services require support for triggers of two types, the first type of which is an essential element of the second type:

1. *basic triggers* for monitoring the specific conditional outcome of an individual event occurrence
2. *composite triggers* whose actions can be fired only when a combination of events occur and their associated conditions all become true.

The event management framework accordingly supports both types.

2.6 Support for causality tracking

Causality is fundamental to many problems occurring in distributed systems. Indeed, many of the problems requiring the evaluation of SSPs (*System State Predicates*) (see Section 1.1) share a common need to track causal relationships between events. It is therefore appropriate that the event management framework should provide effective support for potential causality tracking.

Although many distributed event tracking techniques are application specific, they are all essentially based on the fundamental property of Lamport Clocks [LAMPORT 79]. To this end, the event management framework presented herein labels each event with a timestamp derived from a Lamport Clock and provides sufficient additional information about the event occurrence to permit event monitoring application services to apply specific causal event tracking techniques, such as those based on Vector Clocks [SCHWARZ 94] or even more elaborate Matrix Clocks [WUU 84].

The reader is referred to [HOFFNER 93] for a survey of ordering algorithms for monitoring in which the principles of various logical clock systems are presented.

3 Event Management Framework

This chapter present the principles, concepts and functional components of the event management framework. It begins by describing the attributes of the model underlying the framework: an extension of the *Event-Condition-Action* (E-C-A) model for event based "triggers" enunciated in [McCARTHY 89]. The chapter continues by developing the computational view of triggers in terms of three components classes: event agents, condition evaluators and action monitors. It is shown how these components can be placed in a distributed system to achieve scalable, efficient event management.

The chapter concludes with a statement of where the event management framework fits into the general picture of the ANSA management engine. This statement also indicates how the framework can serve as the basis for monitoring QoS related events and for reporting deviations from expected QoS performance to Qos management services.

3.1 Principles of event based triggers

The concept of a **trigger** is central to the event management framework. A trigger is a rule that essentially comprises a *triggering event*, a *condition*, and an *action*. When the event occurs, the condition is evaluated and, if found to be true, the action is "fired".

The following imperative form illustrates the attributes of a basic trigger.

trigger-name: **{{**{*short-lived* | *long-lived*, *coupling mode*}
on {*event-expression*}
{if {*condition-statement*}
then {*trigger-action*}
[else {*alternative-action*}**]}**
[unless {*limiting-constraint*}**]}**

Triggers do not fire unless they are active. A *short-lived trigger*, once activated, is deactivated the moment it fires. This is in contrast to a *long-lived trigger* which, once activated, remains active unless it is explicitly deactivated.

A trigger is explicitly activated (deactivated) by invoking the operation:

trigger-name.*activate* (.*deactivate*)

Thus the name of a trigger is the name given to its interface (discussed in detail in Section 3.4).

The *coupling mode* expresses the relationship between the computation activity that causes the event to occur and the computation activity which evaluates the *condition-statement* or fires the *trigger-action* (or both). The options for this are defined in Section 3.5.

An *event-expression* may specify a basic event (i.e., one which occurs at a singular interaction point (identified below)) or a combination of basic events

that expresses some logical outcome of event occurrences. Such a combination of basic events is termed a composite event and is discussed in Section 3.6.

A trigger can specify an optional else-clause which defines an action to be taken when the event occurs and when the condition is evaluated and found to be false. Such an action is termed an *alternative-action* - discussed further in Section 3.3.

Although a trigger (more precisely, the *trigger-action*) can be explicitly deactivated (disabled), it can also be implicitly deactivated during periods in which its *limiting-constraint* is "true". This applies even when the evaluated condition is satisfied.

The usefulness of combining a condition evaluation with a limiting constraint is discussed in Section 3.3.

3.2 Events and event agents

Chapter 2 introduced the operational nature of events in the context of the ODP interaction model. Specifically, an event is a selected computation occurrence that gets signalled from one of the four interaction points involved in a client-server operation invocation sequence.

The following expands upon these points and explains how events of interest are selected and how they are signalled.

3.2.1 Interaction points and basic event selection

Each interaction point (IP) is shown below, together with an Event Type label that is introduced to assist the reader in interpreting the kind of basic event (simply, "event") which is generated at that interaction point. The reader is also referred to Figure 2.1.

IP	Event type	Comment
IE	ClientRequest	*Client requests an invocation of a server operation*
IR	ServerReceipt	*Server receives a client request to invoke an operation*
RE	ServerResponse	*Server sends a response to an invoking client*
RR	ClientResult	*Client receives a result from an invoked server operation*

A specific event is selected for signalling by qualifying an operation with the required interaction point using a SelectEvent specification as an attribute of a client or a server.[1]

The following examples illustrate the principles.

In a client's case, a specification of the form (as an attribute of the client)

SelectEvent (ClientRequest: ServerX.op1, ClientResult: ServerY.op3, ...)

refers to operations on named server interfaces (e.g. op1 on ServerX and op3 on Server Y) whose respective initiation and outcome are to be signalled as client events of interest.

In a server's case, a specification of the form (as an attribute of the server)

1. The details of how SelectEvent specifications are included in the bodies of client and servers is mostly implementation language dependent and therefore a subject for further study. The ideas presented in this section are simply intended to illustrate the principles.

SelectEvent (ServerReceipt: OwnX.op2, ServerResponse: OwnY.op1, ...)

identifies those events to be signalled at different interaction points within the server, involving different operations and different interfaces (e.g. interfaces OwnX and OwnY with operations op2 and op1 respectively).

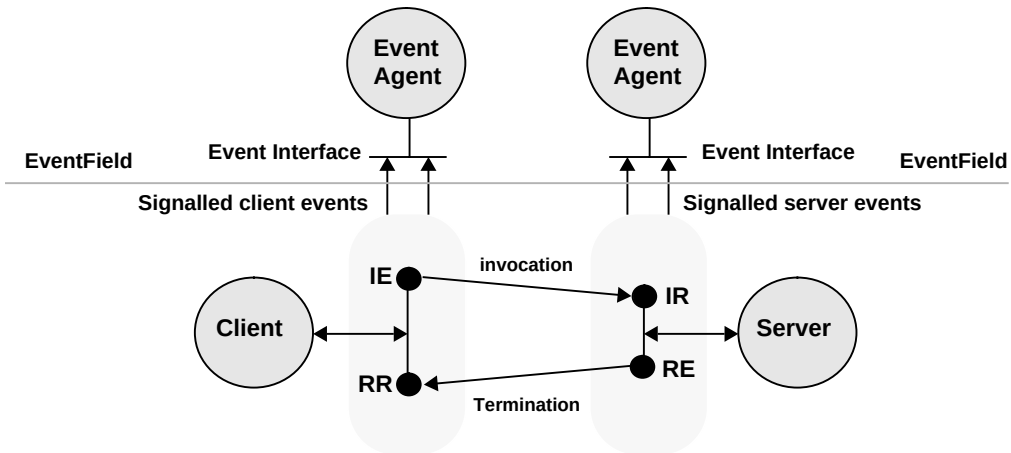
Hence, only those clients and servers that are required to be monitored need be made instances of a reactive class capable of signalling selected events.

The same object may participate in interactions with many different services of the same or different interface type (concurrently and/or consecutively and as client or server or both). In such cases, the SelectEvent specification for the object will enumerate those interaction points which are to be signalled for the client and the server roles.

3.2.2 Signalling events to event agents

A client or server signals the occurrence of a selected event by invoking a signal operation on the event interface of an event agent. This arrangement is illustrated in Figure 3.1, again based on the ODP client-server interaction model sketched in Chapter 2.

Figure 3.1: Signalling event occurrences



There is one event agent for the client and one for the server[1,2,3]. Each event agent acts as a consumer of event occurrences, recording them and eventually

1. For the implementation of an object that interacts as both client and server, the two event agents could be combined to yield a single agent. The details are not discussed in this document.

2. It is envisaged that event interfaces will be generated by the transformer toolkit of the event management framework from the SelectEvent specifications given as components of client/server specifications. This is a subject for further study.

3. In ANSA engineering terms, the efficient processing of signalled events will typically require event agents to be placed as near as possible to their event generating services, ideally, collocated with them in the same capsule or, even stronger, collocated within the same cluster. The latter is necessary if event agents are to be automatically moved with their services whenever the services relocate to another capsule. This follows since the cluster and its contained objects are moved as a collective whole.

distributing knowledge of them to other interested consumers in the system (detailed in Section 3.3).

The state of an event comprises the parameters supplied to the event manager when the event occurrence is signalled. Specifically, these parameters include:

(i) activity name - the identity of the computation activity that raises the event

(ii) timestamp - a Lamport Clock value that denotes the logical time of the event occurrence

(iii) event name - one of the four interaction point names ClientRequest (IE), ClientResult (RR), ServerReciept (RE) or ServerResponse (RE)

(iv) operation name - the name of operation for which the event is selected

(v) operation parameters - the actual parameters supplied at the point of event occurrence, which can either be operation request/receipt parameters or operation response/result parameters.

Since an event (occurrence) has state, it must also have structure. The structure of an event (instance) is thus an “object”. Consequently, event objects may exhibit the properties of other objects in that they may be created (manufactured), deleted (collected as garbage), accessed (referenced) and made persistent (continued to be referenced). The maintenance of event objects (instances of an event class) is the job of their related event agent.

As a recording medium, each event agent, together with its event object repository, can be viewed as an event history: an ordered set of event occurrences. The collection of all event histories (a partial order) represents the EventField. The EventField can thus be treated as a repository of existing event objects that can potentially be interrogated as a database (e.g. using an interactive query language such as SQL[1]).

3.3 Conditions and condition evaluators

An event agent serves not only as consumer of events but also as a producer of event notifications for other interested parties. In the proposed framework such parties are termed condition evaluators[2].

3.3.1 Conditional events

Figure 3.2 illustrates the relationships between a condition evaluator and an event agent, arranged to form a conditional event configuration.

The following described the functions of this component and its interactions with an event agent.

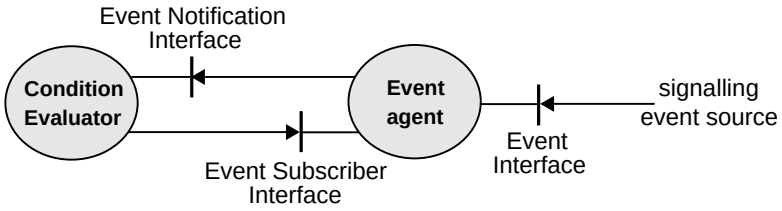
As shown, the condition evaluator binds to an event subscriber interface of the event agent. Thereafter, it invokes a subscribe operation on this interface to inform the event agent of an event notification interface through which it is to notify the

1. The idea of the EventField as a persistent database was suggested by Dave Otway at APM. The detailed design, implementation and application of this database concept are subjects of further study.

2. The recipient of event notifications need not be a condition evaluator. For example, it might simply be a basic event logging function. However, the framework presented in this document is based on distributed triggers, of which a condition evaluator is a fundamental component.

condition evaluator of event occurrences[1]. The parameters of this notify operation are identical to those supplied at the time the event was signalled at its service interaction point (Section 3.2.2).

Figure 3.2: A conditional event configuration



This conditional event configuration represents the bulk of the if-then-else-unless components of the trigger specification outlined in Section 3.1. The reader is advised to keep the details of this specification close to hand throughout the following description of condition evaluation.

While the *limiting-constraint* is "true", all event notifications are received by the condition evaluator but otherwise ignored[2].

If, however, the *limiting-constraint* is found to be "false" (i.e., disabled), the *condition-statement* is evaluated. If this result is also "false", the event notification is similarly ignored; unless, that is, there is an else-clause which specifies an *alternative-action*[3]. If so, this action is taken.

If the *condition-statement* proves "true", the condition evaluator notifies the subscribing action monitor to take the *trigger-action* (Section 3.4). At this point the trigger is said to "fire".

Note that although the above discussion has focused on the interactions between an event agent and one condition evaluator, there is nothing to prevent a "many-to-one mapping" in which the condition evaluators of several different triggers all subscribe to the same event agent service and thereby all receive notifications of the same event occurrences.

1. Event interfaces also provide an unsubscribe operation to allow condition evaluators to stop receiving event notifications.

2. Sometimes it is a requirement that event occurrences need only be reacted to within some specific window of interest. The *limiting-constraint* (a condition) of a trigger specification provides such a capability. Outside this window (while the *limiting-constraint* is imposed ("true")), each *condition-statement* evaluation is effectively disabled. For example, the *limiting-constraint* might only enable the evaluation of the *condition-statement* on every fifth occurrence of an event. A more complex example might enable *condition-statement* evaluation only when certain parameter values of the event are observed.

3. An *alternative-action* is a useful device for those applications which require some action to be taken even when a condition evaluator does not invoke the trigger-action. For example, it is effectively deployed in flexible transaction applications where monitored event occurrences are used to evaluate the existence of specific dependencies between transactions. The outcome of each evaluation frequently involves a choice of two actions: one action (the *trigger-action*) if the evaluated result is "true" (meaning that a specified dependency between given transactions exists); and another action (an *alternative-action*) if the dependency does not exist. The interested reader is referred to [WARNE 94] for a detailed account of trigger-based flexible transactions and their application to distributed system workflows.

3.4 Actions and action monitors

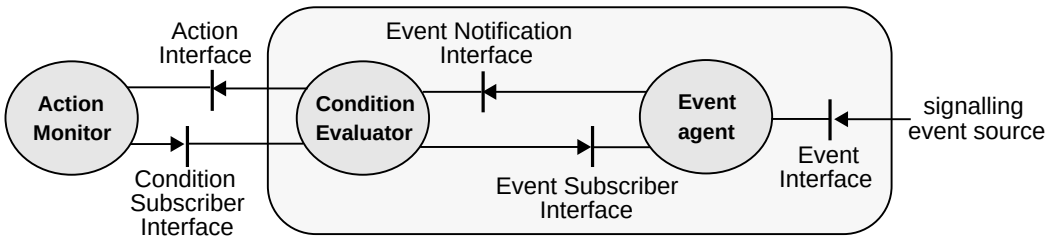
The third and final component of a basic trigger is the action monitor, whose sole purpose is to execute a sequence of operations in response to notifications from its associated condition evaluator.

The details of this component are examined below.

3.4.1 Basic trigger configuration

Figure 3.3 illustrates how a conditional event configuration is extended to represent a basic trigger configuration.

Figure 3.3: Basic trigger configuration



An action monitor and a condition evaluator are related via two interfaces:

- condition subscriber interface though which the action monitor (via a subscribe operation[1]) informs the condition evaluator of its interest in receiving notifications of the (satisfied) conditional event
- action interface (passed to the condition evaluator via a parameter of the subscribe operation) through which the action monitor receives conditional event notifications (via a notify operation).

This configuration reflects the generality of the event management framework. It not only permits several condition evaluators to subscribe to the same or different event agents, but also permits several action monitors to subscribe to the same or different condition evaluators. This capability is conducive to the construction of flexible configurations of triggers using libraries of standard (reusable) action monitors, condition evaluators and event agents as the basic building blocks.

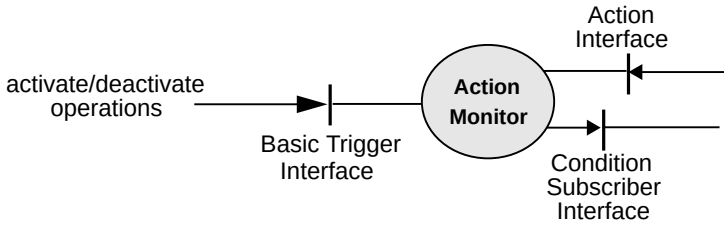
3.4.2 Basic trigger controls

An action monitor also supports a basic trigger interface through which the reactive property of the trigger as a whole can be controlled. This interface is illustrated in Figure 3.4

A basic trigger is initially activated by issuing an explicit activate operation on the basic trigger interlace. If the trigger is *long-lived* its 'firing" will be perpetuated until it is explicitly deactivated via a deactivate operation on the basic trigger interface. This is contrast to a short-lived trigger which is automatically deactivated the moment it "fires". Any trigger, once deactivated, can only "fire" again if it is explicitly activated.

1. Condition evaluators also provide an unsubscribe operation to allow action monitors to stop receiving notifications of satisfied conditional events.

Figure 3.4: Basic trigger interface and control operations



3.5 Coupling modes

Coupling modes specify the relationships between the computation activities that signal events and the computation activities that evaluate condition-statements or fire trigger-actions (or both).

An event occurrence is typically propagated to its consumers asynchronously, thus allowing the event signalling computation activity to continue processing independently of the activit(y)(ies) created to carry the event notification(s) to the monitoring condition evaluator(s). In such cases, the *coupling mode* between these computations is said to be *immediate-independent*.

This, however, is only one example of four potential coupling modes. The other three modes are

1. *immediate-dependent*: the computation activity that signals the event pre-empts its current processing to carry the event notification, to evaluate the condition(s), and, wherever satisfied, to “fire” the required actions by invoking the respective action monitor(s);
2. *deferred-dependent*: the computation activity that signals the event defers event evaluation(s) and any subsequent action firing(s) until it is ready to terminate its own execution;
3. *deferred independent*: the computation activity that signals the event defers event evaluation(s) and any subsequent action firing(s) until it is ready to terminate its own execution, whereupon it creates the activit(y)(ies) necessary to process these deferments.

The *immediate-dependent* coupling mode is appropriate in situations where a *trigger-action* is to be fired prior to further normal processing of the event signalling computation activity. One example is where an event occurrence is to trigger the execution of an “interceptor” that performs some transparency function on behalf of the event signalling computation activity [APM/TR.043.00].

The *deferred-dependent/independent* coupling modes were originally intended to meet the needs of flexible atomic computation activities (transactions), although they might also be gainfully deployed in non atomic transactional environments (but this is not specifically examined here).

The *deferred-dependent* mode permits a computation activity to complete some part of its processing before attempting to evaluate a condition which is dependent, say, on some outcome of that processing. With an atomic transaction such a condition evaluation might relate to an integrity constraint which must be verified after that computation has performed some set of processing actions and before it commits. If this integrity check fails then the atomic computation activity can simply be aborted.

The *deferred-independent* mode differs from the dependent mode in that it always performs pending condition evaluation(s) in the independent computation activit(y)(ies), but only when the event signalling computation has completed its processing and terminates. With atomic transactions this means that even if a computation commits or aborts, each pending condition evaluation is still performed by an independent atomic activity.[1]

The reader is referred to [WARNE 94] for a detailed account of flexible atomic transactions models and the manner in which they make effective use of inter-transaction dependency rules and coupling modes.

3.6 Composite triggers

Different basic conditional events and actions can be combined to create composite triggers using logical operators and special event specification operators. An example is illustrated in the figure below.

Figure 3.5: Example of a composite trigger

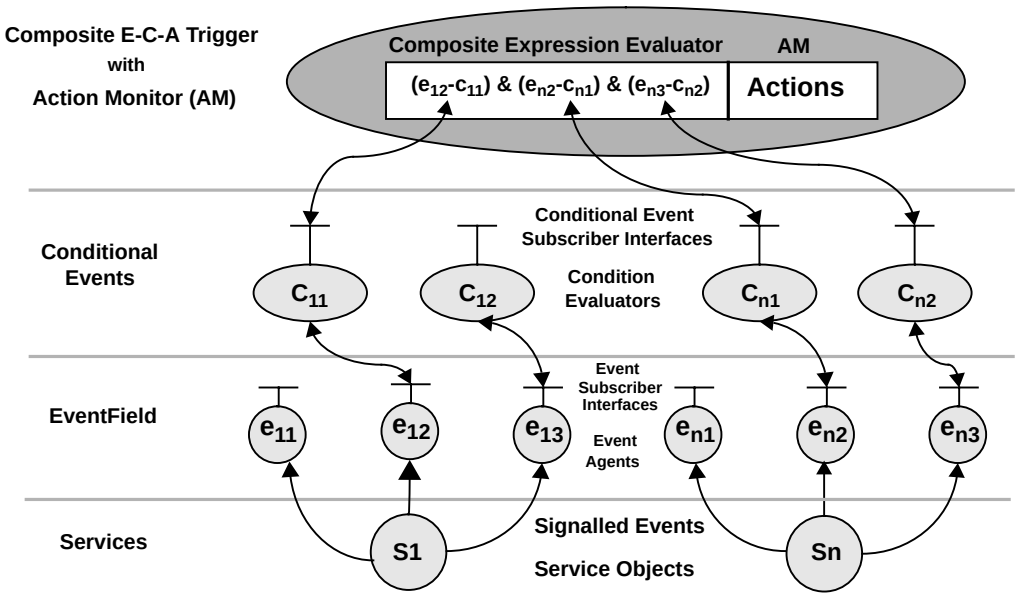


Figure 3.5 depicts an object configuration comprising a composite trigger, several condition evaluators and event agents, and two service objects.

The composite trigger has an associated composite expression evaluator that fires the action monitor only after it has been notified of the occurrence all conditional events given in its composite event expression

$$(e_{12}\text{-}c_{11}) \,\&\, (e_{n2}\text{-}c_{n1}) \,\&\, (e_{n3}\text{-}c_{n2})$$

1. It might seem somewhat inappropriate to permit the “effect” part of a computation to proceed if its related “cause” part is aborted. However, there are certain specific exceptions. For example, it is still a requirement that the accounting and charging functions for resource usage by atomic computations be continued, even for those computations that are aborted. Thus it is sometimes necessary to break away from a strict causality model and create a variant in which certain practical requirements can override the otherwise stricter causality principle of not allowing an “effect” to exist without its “cause”.

where each e-c component expresses a particular event (e) and a condition (c) over that event.

The trigger is subscribing to the respective condition evaluators C_{11}, C_{n1} and C_{n2}. The flow of conditional event notifications from these evaluators to the trigger is indicated by the upward arrows (the notification interface is not shown).

Each condition evaluator is itself subscribing to event notifications from the appropriate event agent, for example, C_{11} is subscribing to the agent that emits notifications of event e_{12}.[1]

(Note that the convention adopted in Figure 3.5 is that each condition evaluator (each event agent) has the same name as its associated condition (event).)

Finally each event agent is receiving event signals from a specific service object and is in turn emitting event notifications to its subscribing event evaluator[2] (the direction of event notifications is indicated by the upward arrows, but again the notification interfaces are not shown).

3.7 Composite event expressions and event operators

A flexible event management framework should not be unnecessarily restrictive on the combinations of events that can be monitored. Accordingly, it is proposed that the work on “Composite Event Specifications in Active Databases: Model and Implementation” [GEHANI 92] be used as fundamental guidelines. This work is of sufficient generality to permit any logical combination of conditional events to be expressed and to admit efficient implementation.

While it is not possible to detail that work here (this would certainly require the focus of a separate paper), some preliminary design and implementation pointers are given in the next two sections.

3.7.1 Design note

The example presented in Figure 3.5, illustrates the use of the logical conjunction operator for expressing a combination of conditional events, all of which must thus be true before the trigger action can be fired. Other useful operators, although not exhaustive, are disjunction and sequence. Moreover special operators which permit combinations of different operators must also be accommodated. The design presented in [GEHANI 92] meets these needs and others. However, the work is not placed in the context of object-based

1. It was mentioned earlier that in the interest of efficiency it is usual to collocate each event agent with its event generating source (Page 9, footnote 3). Similarly, it may be efficient to collocate condition evaluators with their monitored event agents. The event management framework affords this opportunity, since each condition evaluator is itself a separate object and thus potentially capable of migration to directed locations. The same capability might equally be given to the triggered object itself (action monitor), but only in those situations where it is efficient to do so and the trigger is basic and thus not monitoring several distributed condition evaluators. When, however, the trigger is a composite concerned with monitoring several distributed conditional events, its placement in the system with respect to its monitored objects (interfaces) must be carefully considered. Such configuration issues are subjects for further study.

2. The reader is reminded that several different condition evaluators can subscribe to the same event agent, but this is not the case here.

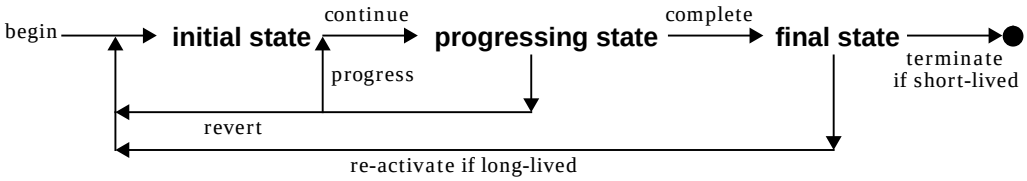
systems, such as the event management framework presented in this document. Thus further work in this direction is still required.

An interesting variant of the work by Gehani et al has been developed by John Bates in his Ph.D thesis on the Authoring Model [BATES 94]. Bates presents an elegant script language for expressing any possible combination of event occurrences in specific system contexts of interest. It is proposed that this script notation be used as the basis for developing an object-based language for the event management framework.

3.7.2 Implementation note

Both [GEHANI 92] and [BATES 94] compose composite event expressions as regular expressions. Such expressions admit efficient implementation in the form of extended finite state automata. The principle of such state machine operation is illustrated in Figure 3.6.

Figure 3.6: Generic extended state machine operation



This extended state machine is representative of a composite expression evaluator, a practical example of which was identified in Section 3.6.

Inputs to the state machine are conditional event notifications (simply, events or event notifications in the immediate following).

The machine begins in an initial state where it awaits notification of some initial event (of which there may be a choice). When this occurs, the machine moves to a progressing state where it awaits further events of interest (either a specific event or any one of some number of events).

As notifications of interesting events occur, the machine continues to progress and thus remains in the progressing state. If an event occurs which “invalidates” the progressing state (i.e., causes the composite expression evaluation to yield a “false” result) the machine reverts back to the initial state to begin over again.

When all required event notifications have been received and have also satisfied all required progressing state transitions, the final state is reached. At that point the trigger is “fired” by its associated action monitor (not shown in Figure 3.6).

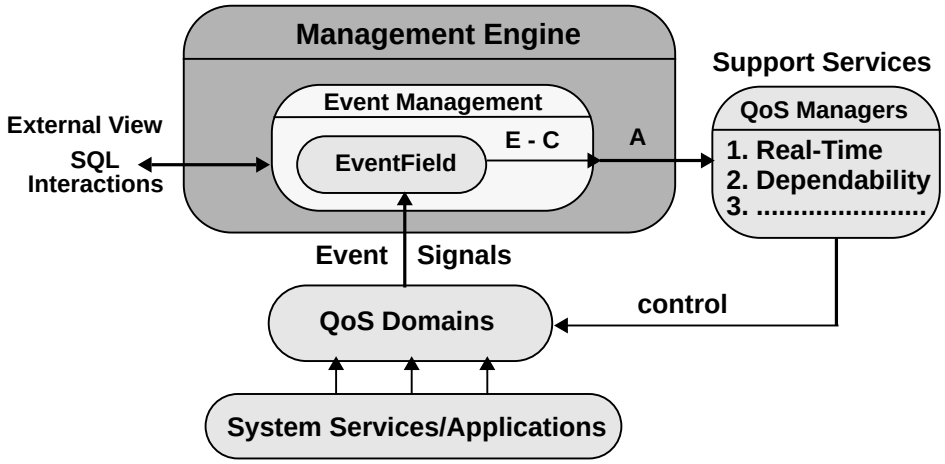
Thereafter, and finally, one of two state transitions occurs: the terminated state, if the trigger is *short-lived*; or the initial state, if the trigger is *long-lived*.

3.8 Architectural implications

The event management framework is an integral component of the of the ANSA Phase III Generic Management Engine [APM 94].

Figure 3.7 illustrates the relationships between event management and the other components with which it interacts.

Figure 3.7: Event management engine



System services and applications, operating within specific QoS domains, signal events to the event management engine, thereby entering them into the EventField for condition evaluation (denoted by E -C). In this process, all satisfied triggers are “fired”, causing their action components (A) to interact with all interested QoS Managers. Such QoS mangers are typically notified of any monitored deviations in required QoS performance upon which they can consequently take remedial action by interacting with and controlling the signalling QoS domains.

An external view is provided that allows interactions with Event Management and its EventField via, for example, an SQL front-end.[1]

1. External views and their underlying interaction languages are subjects for further study. SQL is proposed here to suggest the persistent database property of the EventField.

4 Example

This chapter illustrates how event management can be applied to a simple application.

(The reader should note that the application chosen in not related to the QoS discussion outlined in Chapter 3[1]. Instead, it illustrates the utility of the event management framework in a simple business application.)

4.1 Application

The proposed application (borrowed) from [GEHANI 92]) considers the stock market and its involvement with three companies, namely, IBM, DowJones and Parker. Parker is interested in tracking changes in price and percentage in IBM stock and the DowJones index respectively. When a change occurs in both of these interests, Parker performs an analysis to determine whether it is advisable to purchase IBM stock. The conditions that need to be satisfied for Parker are that the price of IBM stock is below $55 and the corresponding percentage change set by DowJones is less than 3.4%.

4.2 Event Management Strategy

A distributed computer system is constructed to automate the purchase of IBM stock on behalf of Parker whenever the aforementioned conditions are satisfied.

It is assumed that the system can monitor and access IBM stock changes via an object IBM which also supports an interface of the same name. Through this interface it is possible to subscribe to ServerReceipt events (Section 3.2.1) for the operation IBM.SetPrice. This operation specifies the new price as a parameter named IBMprice.

Similarly, percentage changes in the DowJones index can monitored and accessed via the DowJones object and the operation DowJones.SetValue. This operation specifies the new index value as a parameter from which it is possible to calculate the percentage change, DowJonesChange, over the monitored period.

Finally, Parker has an associated object called ParkerPortfolio via which it can purchase IBM stock by invoking the operation ParkerPortfolio.GetIBMStock.

Figure 4.1 specifies the composite trigger (CleverParker) for effecting the purchase of IBM stock in the system. The trigger is long-lived and has “immediate-independent” as its coupling mode. There is no need for an

1. It is difficult to provide suitable event-based QoS examples, since at the time of writing, the work on QoS is still at an early stage of development. Furthermore, the integration of QoS work with the proposed event management framework requires much further study.

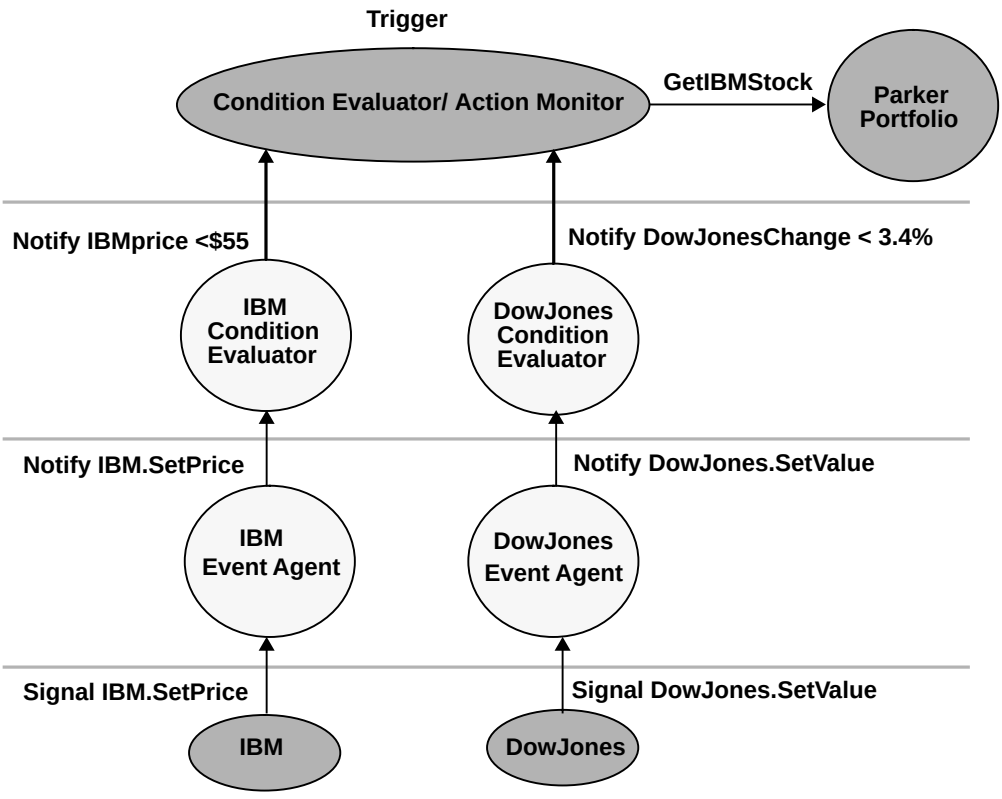
Figure 4.1: Composite trigger for purchasing IBM stock

```
{Trigger = CleverParker(long-lived, immediate-independent)
        On
            {IBM.SetPrice() & Dowjones.SetValue()}
            if
               {IBMPrice < $55 & DowJonesChange <3.4%}
              then
                    {ParkerPortfolio.GetIBMstock()}}
```

alternative action (else-clause) or a limiting constraint (unless-clause) in this particular case.

This trigger specification would be transformed (by transformer tools[1]) to yield the object configuration shown in Figure 4.2.

Figure 4.2: Object configuration for CleverParker



The details of subscribing to event agents and condition evaluators are not discussed or shown. The principles are fully covered in Chapter 3.

1. A toolkit for transforming trigger specifications into event management object configurations is a subject for further study.

5 Summary and Future Directions

This paper has provided a detailed overview of the proposed ANSA Event Management Framework. The principle requirements for the framework were identified and reviewed, specifically

(i) events as named interaction points based on the ODP client-server model

(ii) support for observing selected client and server events

(iii) use of basic and composite rule-based triggers as the unifying paradigm underpinning the framework

(iv) events, conditional events, and actions as first class objects with first class interfaces

(v) elementary support for causality tracking at the event signalling level.

Using these requirements as the guiding design principles, the concept of a trigger was developed to produce a flexible object structure for general event management in large scale distributed systems. The interaction details of the event management components were described in outline and an example was provided to further illustrate the principles.

5.1 Directions for future work

It is proposed that the following work items be produced in response to the architectural framework presented in this document:

- Detailed design specification of the framework, including:
  - — detailed design of the event management specification language;
  - — complete specification of the interfaces for the event agent, condition evaluator and action monitor;
  - — Detailed design of appropriate external views and interaction languages (e.g. SQL) for allowing query and control access to the event management engine
  - — detailed design of interworking between QoS domains, QoS management and the event management components.
- Methods and tools for constructing event management configurations, including:
  - — detailed design of the tools for transforming trigger specifications to run-time object configurations
  - — specifications of run-time support libraries.

To validate the event management framework and the above projected detailed design, it is further proposed that this work be followed by the implementation of a demonstrator which can be used to prototype the ideas.

References

[ANSA 91]

"A System Designer's Introduction to the Architecture", RC.253, April 1991, APM Ltd., Poseidon House, Castle Park, Cambridge.

[ANSWAR 93]

E. Answar, L. Maugis, S. Chakravarthy, "A New Perspective on Rule Support for Object-Oriented Databases", Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, 1993.

[APM 94]

"1994 - 1996 ANSA Workplan", APM.1275.00.05, August 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[BATES 94]

John Bates, "Presentation Support for Distributed Multimedia Applications", Technical Report No. 341, June 1994, University of Cambridge Computer Laboratory, England.

[DAYAL 90]

U. Dayal, M. Hsu, R. Ladin, "Organizing Long Running Activities with Triggers and Transactions", Proceedings of the ACM-SIGMOD International Conference on Management of Data, 1990.

[GEHANI 92]

N.H. Gehani, H.V. Jagadish, O. Shmueli, "Composite Event Specification of Active Databases: Model and Implementation", Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada, 1992.

[HOFFNER 93]

Yigal Hoffner, "A Survey of Ordering Algorithms for Monitoring", TR.040.00, May 1993, APM Ltd., Poseidon House, Castle Park, Cambridge.

[LAMPORT 79]

L. Lamport, "Time, clocks and the ordering of events in distributed systems", Communications of the ACM, 21(7), 1978

[LI 94]

Guangxing Li, "A model for Real-Time QoS", APM.1151, March 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[McCARTHY 89]

D. R., McCarthy, U. Dayal, "The Architecture of an Active Database Management System", Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, May-June 1989, 215-224.

[ODP 94]

Basic Reference Model for Open Distributed Processing - Part 3: Prescriptive Model, Secretariat ISO/IEC JTC1/SC21/WG7, American National Standards Institute, February 1994.

[REES 93a]

R.T.O.R. Rees, “ANSA Computational Model”, AR.001.01, April 1993, APM Ltd., Poseidon House, Castle Park, Cambridge.

[REES 93b]

R.T.O.R. Rees, “Using Path Expressions as Concurrency Guards”, TR.22.00, April 1993, APM Ltd., Poseidon House, Castle Park, Cambridge.

[SCHWARZ 94]

R. Schwarz, F. Mattern, “Detecting causal relationships in distributed computations: in search of the holy grail”, Distributed Computing, 7, Springer Verlag, 1994.

[WARNE 94]

J. P. Warne, “Flexible Transaction Framework for Dependable Workflows”, Architecture Report: APM.1263.02, June 1994, APM Ltd., Poseidon House, Castle Park, Cambridge.

[WUU 84]

G. T. J. Wuu, A. J. Bernstein, “Efficient solutions to replicated log and dictionary problems”, Proceedings of the 3rd ACM Symposium on PODC, 1984.