Event-based Distributed Workflow Execution with EVE

Andreas Geppert, Dimitrios Tombros

Department of Computer Science, University of Zurich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
{geppert | tombros}@ifi.unizh.ch

Technical Report **96.05**
May 1996 (revised March 1998)

Abstract

In event-driven workflow execution, events and event-condition-action rules are the fundamental metaphors for defining and enforcing workflow logic. Processing entities enact workflows by reacting to and generating new events. The foundation on events also eases the integration of processing entities into coherent systems.
In this paper, we present an event engine, called EVE, implementing event-driven execution of distributed workflows. Its functionality includes event registration, detection and management, as well as event notification to distributed, autonomous, reactive software components which represent workflow processing entities. EVE also maintains a history of all event occurrences in the system used for the monitoring and analysis of executing workflows. We describe the distributed, multi-server, multi-client architecture of EVE and illustrate its usage for workflow execution.

Keywords: event-based systems, workflow management, distributed workflow execution

1 Overview

Currently, many enterprises optimize and streamline their business processes. Examples include insurance claim handling, credit card applications, etc. These business processes can be—completely or partially—automated when represented as *workflow specifications. Workflow management systems* (WFMS) are software systems providing workflow definition and implementation functionality (scheduling, execution, and control) [13]. Workflow specifications—or workflow types—consist of subworkflows and atomic work steps, data flows between them, execution order constraints, as well as assignment of tasks to PE. Productive use of WFMS requires that they effectively support the integration of heterogeneous information resources, people and applications (called processing entities—PE). The resulting integrated system is called a *workflow system* (WS).

Current commercial WS architectures do not effectively address problems related to the representation, control, and coordination of PE that operate in environments distributed across multiple organizational entities, that are heterogeneous with respect to their automation degree and interfaces, that evolve over time, and that operate both independently and as part of a WS (e.g., [3]). Several research efforts address various aspects of these problems (e.g., [4], [16], [27], [32]), especially those pertaining to distribution and heterogeneity. The proposed solutions, however, do not effectively support flexible representation/integration of PE and runtime evolution of the WS architecture; they also propose a rather strongly coupled integration framework and rigid task structure.

Although event-based systems are recognized as the architectural style of choice for loosely coupled systems [6], to the best of our knowledge, no other approach so far uses events as the *only* integration and coordination mechanism for distributed, heterogeneous, and dynamically configurable WS. Several researchers—including ourselves—[7, 9, 12, 14] have proposed the use of *event-condition-action rules (ECA-rules)* as provided by active database management systems (ADBMS, [2]) for workflow execution. These approaches, however, use a centralized ADBMS, which renders distribution, openness, and scalability hard to achieve. Event services (as, e.g., specified in CORBA Services [22]) also support the notion of event, but these services are restricted to primitive events and, typically, are hybrid in the sense that they rely on both, messages and events as coordination paradigms.

In [28, 29], we propose a *layered event-based architecture* for WS. The WS is described in a domain-specific architecture model which represents PE by autonomous reactive components communicating through parameterized events. The behavior of these components is expressed by ECA-rules. In this paper, we propose an event-based middleware layer and execution platform—called EVE—able to integrate these reactive components and make the WS-architecture description executable. Event-based WS-architectures and workflow execution have various advantages:

- The event-based coordination model allows complete process specification without imposing any limiting assumptions about the concrete process architecture. Complex process situations are expressed by composite events, and coordination is accomplished by defining the appropriate reactive behavior for PE.
- The architecture of the WS can be changed during workflow execution (subject to process restrictions). Interaction patterns are dynamically administered by EVE.
- A powerful and uniform mechanism to express component behavior is provided; the use of this mechanism for failure and exception handling is straightforward.
- The correct execution (with respect to the specification) and the monitoring of the workflows is guaranteed based on formal semantics [30].
- The logging of workflow situations (events) is done at practically no extra cost compared to a message-based architecture.

This paper presents an architecture for WS focusing on the distributed event engine, EVE. In section 2 we position EVE and survey related work. The WS-architecture model is presented in section 3 for the sake of comprehensiveness. We describe the architecture and functionality of EVE in section 4 and its approach to distributed event detection and workflow execution in section 5.

2 Related Work

ECA-rules have been proposed by several authors for workflow execution, e.g., [5, 7, 9, 12]. Some of these systems [5, 12] use composite events to detect complex workflow situations. EVE, however, is the first system using ECA-rules for workflow management addressing the problem of distributed event-based workflow execution.

Current commercial WFMS (e.g., ActionWorks Metro [1], or XSoft's InConcert [19]) are typically built around a centralized process engine and relational database server. Two event-based coordination systems, Yeast [17] and CoopWARE [20], have a purpose similar to EVE;

both have, however, a centralized event-detection and rule execution architecture. Additionally, Yeast supports only event-action rules.

A number of research systems consider distributed workflow execution. ObjectFlow [16] uses a graph-based workflow definition model. Steps are executed by agents coordinated by a (potentially) distributed workflow engine which however accesses a centralized DBMS to store workflow states. WIDE [10] proposes a distributed hierarchical workflow engine which through a basic access layer stores process state in a centralized relational DBMS. In $METEOR_2$ [27], process scheduling is distributed among various task managers. The distribution of the task managers is implemented through CORBA [21]. In Mentor [32], workflows are modeled using state-charts which are partitioned to each involved PE. Each PE-specific state-chart is executed locally on the PE workstation.

	CORBA event service	**EVE**
Communication model	push/pull	push
Typed events	yes	yes
Composite events	no	yes
Event parameters	one of type any	multiple, typed
Event filtering	yes	yes
Disconnected consumers and suppliers	yes	yes
Multicasting	yes	yes
ECA-rules	not yet	yes

Table 1. Comparison of CORBA event-related functionality and EVE

Various aspects of the implementation of EVE (e.g., naming services) can be facilitated by the functionality of CORBA [21] or other distributed object technologies. The emphasis of our research in EVE however, is to complement these efforts by concentrating on a reactive event-based coordination and integration architecture—with its accompanying advantages for heterogeneous system integration—instead of a message-based approach. Event-specific functionality in EVE is richer than in CORBA Event Services [22] (see Table 1). OMG is currently in the process of specifying a workflow management facility [23]. This facility uses the event service for communicating state changes of workflow execution elements (activities). Since the event service is restricted to primitive events, complex workflow situations cannot be detected and communicated. Also, EVE fully supports ECA rule services, the standardization of which, is still only under consideration by OMG[1].

3 A Model for Workflow System Architectures

In this section, we briefly introduce the model we propose for the description of workflow system architectures. In general, workflow types can be specified in a more abstract way (and e.g., using a graph-based formalism) and mapped onto the architecture model level. Here we do not presume any particular workflow modeling approach, but simply assume that the basic elements of a workflow can be specified in some way (i.e., workflow structure in terms of atomic

1. [18] for example discusses the integration of ECA-rules in a CORBA environment.

activities and subworkflows, dependencies between the steps of a workflow, etc.). This specification is subsequently mapped to the behavior of processing entities which compose a WS architecture. Note however, that the event-based coordination layer we propose is independent from the modeling formalism used. This section serves to illustrate the purpose of an abstract-level specification layer in the overall event-based architecture.

3.1 A Workflow Example

In order to better illustrate the concepts we introduce, we use as an example workflow the processing of a health insurance claim (HIC) (Fig. 1). The workflow is initiated once a HIC arrives at an insurance agency. An insurance agent creates a file containing the diagnosis, treatments, cost (from the HIC), and the insurance number (step `A1:HandleHIC`). If the claimed amount does not exceed 300 CHF the claim is directly accepted, a corresponding entry is made in the customer database (`A2:LogClaim`), and a check is prepared (`A3:PrepAccept`) and printed (`A4:Print`). Otherwise, various controls have to be made: a control in the local database whether the prescribed treatment is covered by the patient's insurance (`A5:CntrlCoverage`). A request to a central clearing house to control whether equivalent but less expensive medication is available. This is done in subworkflow `SW1`, consisting of the step `A6:CntrlMedication`. On the other hand, in subworkflow `SW2` executed at the insurance company headquarter, a medical expert controls whether the treatment actually suits the diagnosis (`A7:CntrlTreatment).` When all controls are completed, a corresponding entry is made in the customer database (`A3:LogClaim`), and either a rejection letter (`A8:PrepReject`) or a payment check (`A4:PrepAccept`) is prepared by the insurance agent and printed (`A4:Print`).

3.2 Brokers and Services.

The BROKER/SERVICES MODEL (B/SM [28, 29]) is used to describe the software and process architecture of the resulting WS. Workflow type specifications are mapped to elements of the B/SM in a way described in detail elsewhere [29]. In general, each atomic step corresponds to the execution of a service. ECA-rules define when and how services are executed in the context of workflows. From the perspective of B/SM, a WS consists of interacting, reactive components called *brokers* representing PE. Broker behavior is defined by ECA-rules describing their reaction to simple events (e.g., service requests) or composite events (e.g., a request within a specific time interval) in the action-part of the rules. Different types of brokers represent human user interfaces, organizational groups, external applications, and WFMS-components.

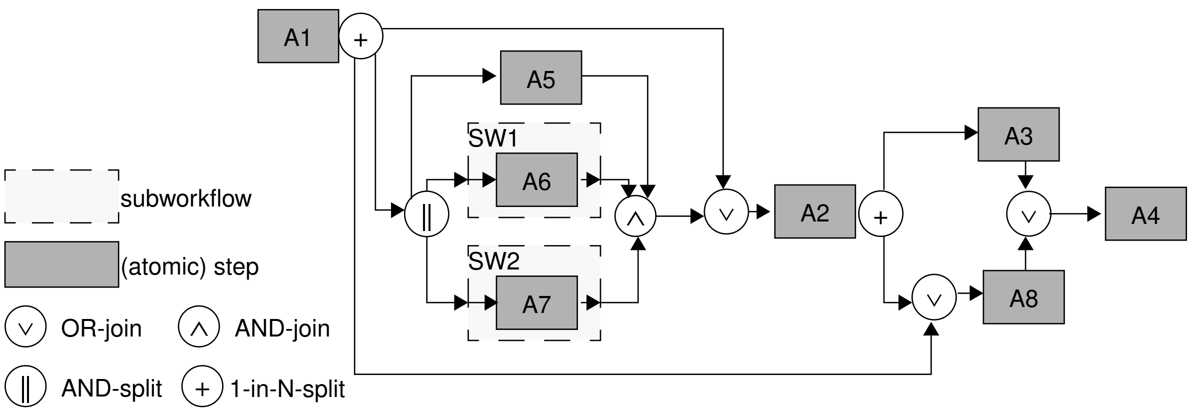

Figure 1: Structure of the example workflow type

Primitive event types represent atomic happenings. Two kinds of primitive event types are distinguished: broker interaction events (requests (`REQ`), replies (`RPL`), and exceptions (`EXC`) for services and workflows) and time event types. Time events can be absolute, relative, or periodical. Composite event types represent complex, process-specific situations by combining primitive or further composite events. They are constructed using one of the following type constructors. A *sequence* (`SEQ`) occurs when the two component events occur in the respective order. An *exclusive disjunction* (`DEX`) occurs when either one of the component events occur. A *conjunction* (`CON`) occurs when both component events occur, regardless of their order. A *repetition* (`REP`) occurs when the component event occurs a predefined number of times. A *negation* (`NEG`) of an event occurs when the component event has not occurred within an interval. Finally, two events occur *concurrently* (`CCR`) when both occur at the same point in time.

An event type definition can be restricted by a time interval, defined by two events; events are considered as relevant only if they occur within the interval. Finally, further restrictions can be specified for composite events. Particularly, it can be required that the component events have to occur within the same workflow. The formal semantics of distributed composite events are described in [30].

Broker functionality is described by *services* specified by a signature (i.e., the service name, its request parameters, and resulting replies or exceptions). Broker interaction is based on broadcasting parameterized events. Service execution is started by a request event from the client broker and is terminated when a reply or one of the exceptions defined for the service is generated by its server broker. The server is dynamically determined based on service parameters, organizational relationships, or the workflow execution history in ways not further specified here. A workflow instance starts executing when its initiation request event occurs. Workflows are thus executed by the provision of the services requested. Services and their providers are associated by `m:n` relationships (*capabilities)* implying a predefined behavior for a broker whenever the service is requested, i.e. ECA-rules of the form:

```
ON   service-request-event (service parameters)
[IF  condition on parameters or broker state is true]
DO   execute service provision actions;
     generate reply event;
```

Workflow production data (manipulated by PE in workflow steps) is generally stored in external systems which might not be accessible to the WFMS. Brokers may, however, provide access to some of the production data to the WFMS-engine by providing interfaces as part of their state. The broker state can be manipulated in the action parts of their ECA-rules. The brokers and services of the example are shown in Table 2.

Broker	**Type**	**Services**	**Location**
MEDICHECK	application	`CntrlMedication`	clearing house
Medical Expert	user interface	`CntrlTreatment`	headquarter
DBCL	Oracle client	`LogClaim`	agency
Insurance Agent	user interface	`HandleHIC, CntrlCoverage, PrepAccept, PrepReject`	agency
LPF	printer frontend	`Print`	agency

Table 2. Brokers and services of the example

The main advantages of using the B/S model to describe workflow system architectures consist in the simplification of the system integration task and in the provision of powerful general purpose definition mechanisms for functional, informational, and behavioral aspects of workflow systems. Various aspects of workflow specifications as well as heterogeneous processing entities can be mapped to a relatively simple and homogeneous representation layer, which can then be directly transformed into an executable form.

4 EVE: A Middleware Component for Workflow Execution

From a point of view of the software architecture model, a runtime system and workflow engine is needed to make a B/SM system executable. This runtime system forms the middleware allowing brokers to execute workflows. A straightforward mapping between the event-based B/SM and the execution engine is achieved by providing an *event-based middleware layer*. In other words, workflows are executed by brokers reacting to (and generating new) events. This approach implies that the following groups of services are provided by EVE:

1. **Distribution**. Since processing entities typically are distributed over a network and workflows should be executable in a distributed manner, EVE must support distribution. This requires distributed event detection and communication facilities between EVE(-servers) and its clients (i.e., brokers).
2. **Runtime repository**. In order to maintain the necessary meta information to operate a workflow system, naming, persistence, and retrieval services are needed. This is accomplished by the runtime repository which stores and provides information about brokers, event types, ECA-rules, etc. This information can be altered at runtime and evolution of a WS is thus possible without shutting down and recompiling the whole system.
3. **Workflow execution**. In order to execute workflows (in an event-driven style), event detection, event notification, and task assignment services are needed. These services should also allow to execute workflows on remote sites. These tasks are accomplished by leveraging event-based systems and services (such as active database systems or notification) to workflow execution.
4. **Event history management**. In order to facilitate maintenance of workflow systems, logging, monitoring, and analysis services have to be provided. These services rely on event histories maintained by EVE. The event history is made persistent and recoverable by using the corresponding services of the underlying object manager (Shore [8]).
5. **Failure handling.** In general, two types of failures can be distinguished: system failures and failure at the workflow execution level. EVE provides exception event notification mechanisms which combined with ECA-rules are used in the definition of reactions to workflow execution errors (as proposed e.g., in [12]). Furthermore, by communicating with brokers via special adapters and persistent event queues, EVE is able to handle temporary unavailability and failures in the connectivity of brokers.

The execution of a workflow starts (or proceeds) as soon as some event is generated by a broker. The local EVE-server then performs event detection and rule execution. Within the execution of each rule, task assignment determines responsible brokers, which are then notified and subsequently react as defined by their ECA-rules. Particularly, brokers can generate new events, which again are handled by EVE-servers, and so on transitively (see Fig. 2). The advantage of this approach is that—due to the event-driven style—brokers do not have to know the providers of the requested services. Furthermore, once events are forwarded to responsible PE, they can be handled asynchronously by the PE.

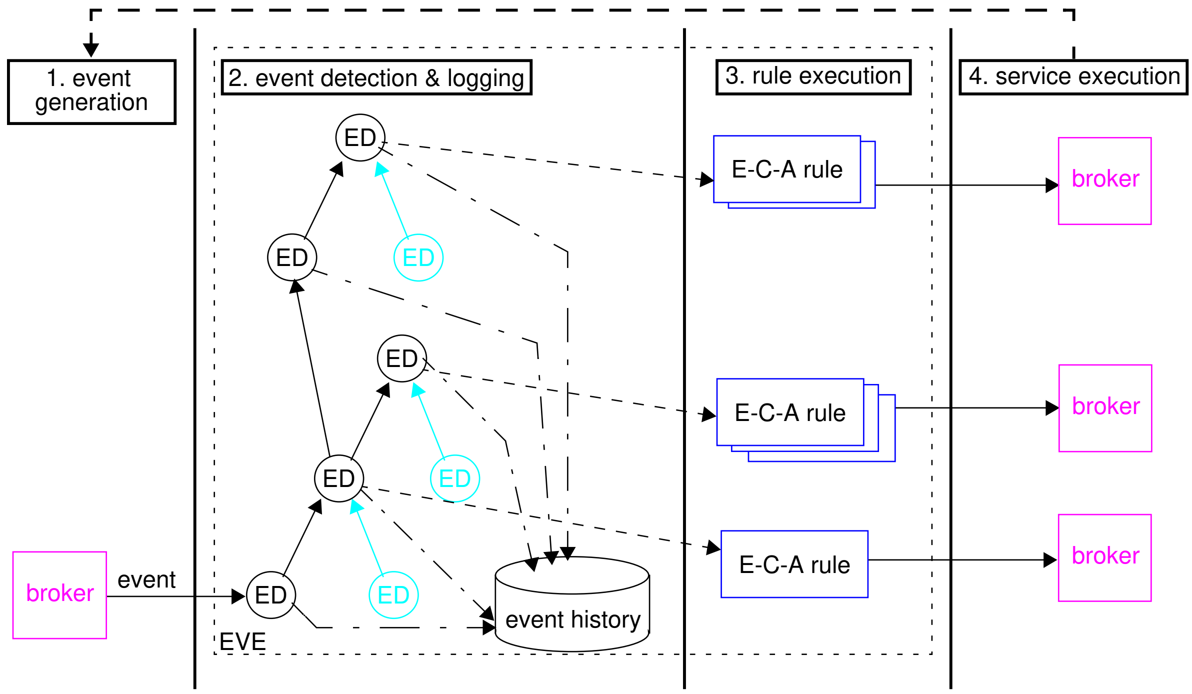

Figure 2: The workflow execution process in EVE

Brokers can also request the execution of (sub-)workflows by generating workflow request events in one of their ECA-rules. Such subworkflows can be executed at the same or a remote server. We however require that all the atomic steps directly contained in the same workflow are executed on the same site, i.e., on any machine of the LAN managed by a particular EVE-server. This site is called the *home site* of the workflow. Subworkflows can be executed at sites other than the home site of their parent workflows, and different instances of the same workflow type can also have different home sites. This is not a limitation as the required distribution can be achieved by appropriately specifying subworkflows, whereas it significantly eases synchronization of servers and leads to better synchronization performance (see section 5.2).

4.1 Multi-Server Architecture and Communication

Brokers involved in workflows typically reside on different machines, which results in workflow systems that are physically distributed to a varying degree (e.g., on different machines within the same domain/LAN, or on different organizational subnets). EVE supports the distributed execution of workflows through a *multi-server/multi-client architecture*. By means of the provided middleware services, this distribution is kept transparent for brokers (and the PE they represent), which in general should not be required to know the physical location of their service providers.

Workflow systems distributed over large areas, consist of multiple *EVE-servers*. They provide all the aforementioned required services. The most important ones of these are described in more detail in the following sections, while we immediately address EVE's client/server architecture and communication. EVE-servers communicate directly with their local brokers and with remote servers. Thus, each broker communicates solely with its local EVE-server. Brokers interact transparently with each other by forwarding the events they generate through platform-specific *EVE-adapters* to their local server.

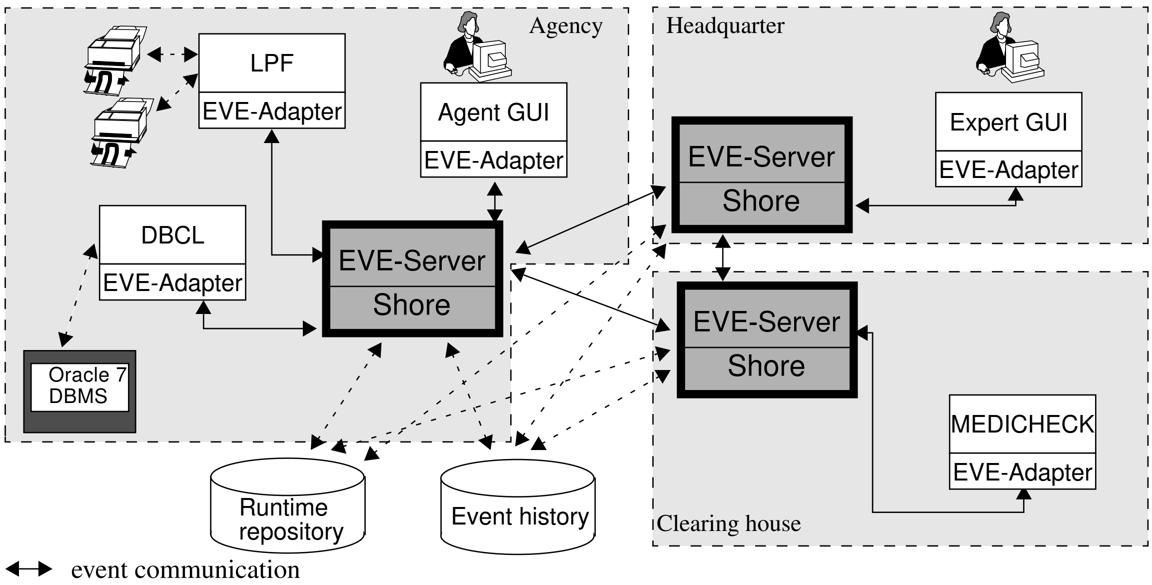

Figure 3: Architecture of EVE and the example workflow system

EVE-servers control the progress of workflows. Control is defined in terms of ECA-rules, which determine what has to happen when a particular (composite) event occurs. They accept and handle client connection requests over a "well-known" network address. Brokers may log in into and logout from the system, whereby each logged-in broker establishes its own communication channel with its server. EVE-servers exchange administration information about the current location and connectivity of brokers. They also provide persistent queues of events to be forwarded to disconnected brokers. Workflow monitoring is achieved by using special-purpose brokers which provide event monitoring services.

Brokers communicate with their local EVE-server through an EVE-adapter. EVE-adapters —normally a library linked to the broker code—thus provide the interface of the EVE-system to brokers. They provide communication transparency for server login as well as for event generation and notification. Adapters also handle temporary server unavailability by keeping persistent queues of client-generated events. Brokers are informed of the occurrence of interesting events by callbacks from their adapter which provides a listening port to the EVE-server.

An EVE-server forwards over the network event occurrences to the adapters of brokers that are registered in the runtime repository as interested in these events. During the execution of broker actions, new events may be generated by the broker. The broker then informs its local EVE-server about the occurrence of these events again through its EVE-adapter. The EVE-server in turn performs event detection, logs these events as well as eventual resulting composite events in the event history, and forwards the occurrences to other interested local brokers and to the servers of interested remote brokers. An overview of the architecture of EVE and the example brokers is presented in Fig. 3. EVE-servers reside at the insurance agency (site s1), at the company headquarter (site s2), and the clearing house (site s3).

The communication infrastructure is implemented using the *Adaptive Communication Environment* framework [24]. EVE-servers are implemented on Solaris 2.5 through extensions to the acceptor pattern [25] that dispatches a handler for each connecting broker. This handler is responsible for incoming events and initiates the appropriate actions in the workflow engine (i.e., triggers the event detection process). The adapter implementation is client-platform specific. Currently, Java and C++ versions have been implemented.

4.2 The EVE Runtime Repository

The runtime repository implements naming, persistence, and querying services and thus contains information needed to execute workflow instances. The explicit representation of this meta-information in the runtime repository leads to a higher degree of flexibility (compared to approaches that use hard-wired, interpretive techniques for workflow execution) and better maintainability of workflow systems (compared to compilation-based approaches). The repository is managed by EVE-servers on top of the object-oriented database system Shore [8]. The runtime repository contains the following information:

- workflow types (name, initiation and termination events, active and terminated instances),
- specification of the participating brokers including their context-specific behavior and responsibilities,
- organizational relationships among brokers and between brokers and groups,
- event types corresponding to those defined in the event part of broker-ECA-rules, and
- ECA-rules defining the structure of workflows.

ECA-rules. ECA-rules in EVE—not to be confused with broker-ECA-rules—implement processes and broker interactions. An EVE-rule is defined as a triple `(event type, condition, action)`. Whenever an instance of the event type occurs and the condition holds, the action is executed. Conditions return a set of object references, they are thus considered to hold if they return a non-empty result. The resulting set of references is passed to the action part. Conditions can, e.g., be used to implement *task assignment*: the condition computes the set of eligible brokers for a request and selects one based on the task assignment strategy specified for the request (event type). Possible task assignment strategies are:

- random (any eligible broker is chosen),
- load-based (the broker with the minimal number of pending requests is chosen),
- responsibility-based (the effectively chosen broker must fulfill further requirements as specified in the current request),
- a combination of these strategies.

Actions are arbitrary code fragments, typically including notifying a broker of a service request. EVE-rules reside on EVE-servers; they are compiled and stored in dynamically linkable libraries and loaded upon execution time whenever the corresponding rule fires.

The execution of the service `LogClaim` for example, can start when the execution of the subworkflows `SW1`, `SW2`, and the service `CntrlCoverage` is completed. This is expressed by rule `R1` residing on server `s1` [2]:

```
ON   CON(RPL(CntrlCoverage.OK, HIC),
     CON(RPL(SW1.Done, Result),RPL(SW2.Done, Result)))
DO   raise(REQ, LogClaim, HIC, SW1.Done.Result, SW2.Done.Result)
```

and a further rule for *task assignment* residing on `s1`:

```
ON   REQ(LogClaim,HIC))
IF   br = assign(LogClaim)            // filter: returns assigned broker
DO   br->notify(REQ, LogClaim, HIC, SW1.Done.Result, SW2.Done.Result)
```

2. The checks for equality of workflow ids and broker ids are not shown here.

5 Workflow Execution

In order to allow event-driven workflow execution, EVE has to implement detection and signaling of primitive and composite events to interested brokers. Composite event detection is an issue extensively considered in centralized ADBMS [e.g., 11]. However, the semantics of composite events are different in a distributed system. This fact renders the concepts and techniques used in centralized ADBMS unfeasible or inadequate for distributed environments and thus need to be extended.

5.1 Event Occurrences and Event Detection

Event occurrences are the *actual happenings* of interest at some point in time and are considered as instances of event types. Each occurrence has several attributes:

- the `event type` of the occurrence
- the occurrence `site`
- a unique site-specific `occurrence identifier`
- the `timestamp` when the event occurred (see section 5.2 and [30])
- the name of the requested `service`
- a `request identifier` (in case of request, confirmation and reply events)
- a `workflow identifier` within which the event has occurred
- the identifier of the `broker` which raised the event
- a list of component occurrences in case of composite events
- a list of typed parameters (i.e., production data).

Events are detected by persistent event detector (ED) objects residing in EVE-servers. In order to generate a new primitive event, a broker notifies its server about the occurrence via its adapter. The server then forwards the event occurrence to the appropriate event detector. Time events are detected based on the system clock. After primitive event detection, composite event detection takes place. For that matter, we adapted the approach originally proposed for the centralized ADBMS Sentinel [11]. A composite ED is a graph (see Fig. 4), where nodes are event types and edges represent event composition. Nodes are marked with references to not-yet-consumed component occurrences. Whenever an event is detected, the parent nodes are informed and check whether the new event together with already obtained component events can form a new event composition. This check is performed in an event type-dependent way. If multiple instances of a component event type exist, the oldest adequate one is chosen (this is the `chronicle` consumption mode [11], which is the appropriate one for workflow management). In case the new event cannot be consumed, the ED stores a reference to it until a com-

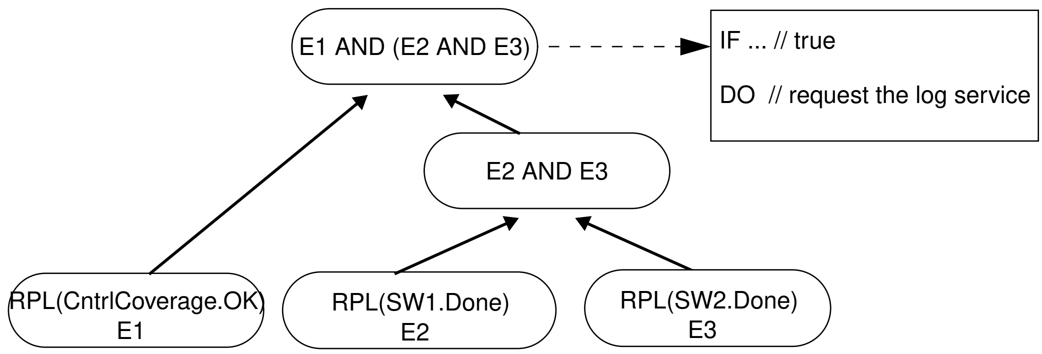

Figure 4: Event detector graph with its attached rule(s) for the event in rule R1

position is possible (i.e., until new sibling component occurrences have been received). Event detection is recursive; whenever a parent node can compose a new occurrence, it in turn informs its parents, a.s.o. in a bottom-up manner.

In addition to event composition edges, event nodes reference rule objects. If a (primitive or composite) event occurrence is detected which has a rule attached to it, then the rule is added to the list of rules to be fired. This firing actually takes place as soon as the event detection cycle has been terminated.

5.2 Distributed Composite Event Detection and Server Synchronization

Timestamps play a crucial role in defining semantics of composite events and define the ordering of event occurrences. In centralized environments a global time can be assumed for this ordering, whereas it does not exist in distributed environments. In EVE we thus adopt the approach of [26] to deal with distributed event ordering. We assume that each site has a local clock. Local clocks are synchronized with a precision p. Event occurrences can be globally ordered, provided that the granularity g of the global time-base is larger than p. In the 2g-precedence model [31], two events occurring at different sites are ordered whenever the timestamp of one event is at least 2g larger than that of the other one. If the difference of the timestamps is less than 2g, then the events are considered to occur concurrently. If two events occur at the same site, then they can be ordered whenever they are at least one clock tick apart. For details on timestamp computation with the 2g-precedence model, see [26].

Correct event composition in the chronicle consumption mode [11] relies on the ordering of component occurrences using timestamps. Upon event composition, the oldest eligible instance is chosen whenever there are multiple candidates. Thus, new events can only be composed if no older adequate component event has occurred at some site. This poses a special problem in distributed environments, since the signaling of events from remote sites typically takes different amounts of time or may even be temporarily impossible due to a detector site crash, and the order of arrival of events from different remote sites may not be the same as their order of occurrence. Thus, an ED can correctly compose new events out of components originating from multiple sites only if it is synchronized with the detectors of its components and so on recursively. Furthermore, efficiency considerations dictate that the effort to synchronize servers and their event detectors should not overly burden the workflow execution process (e.g., with respect to the number of affected detectors and the number of messages needed to exchange synchronization information).

Only servers cooperating within some specific workflow instance need to be synchronized, because events generated within instances of unrelated workflow types are neither temporally nor causally related. Thus, synchronization is needed for event detectors that detect workflow termination events or directly or indirectly contain such events as components.

The detector synchronization works as follows (see Fig. 5). Whenever a broker attached to a server S_1 requests a (sub) workflow and S_1 forwards this request to another server S_2, then the affected detectors are informed that they need to synchronize with S_2. Thus, each detector maintains the information on which servers it needs to synchronize (the *synchronization set*) with. For each server S_i in such a set, a reference counter records the number of workflow instances requested by S_1 at S_i. S_1 also records for each server in the set when the last event was received. From this set the global time can be derived until which the local server is synchronized with the involved remote servers (the *synchronization point*, which is the minimum of the events' timestamps least recently received from the remote servers). The server which has signalled this event is called the *most recently synchronized site,* MRS. Since it is guaranteed

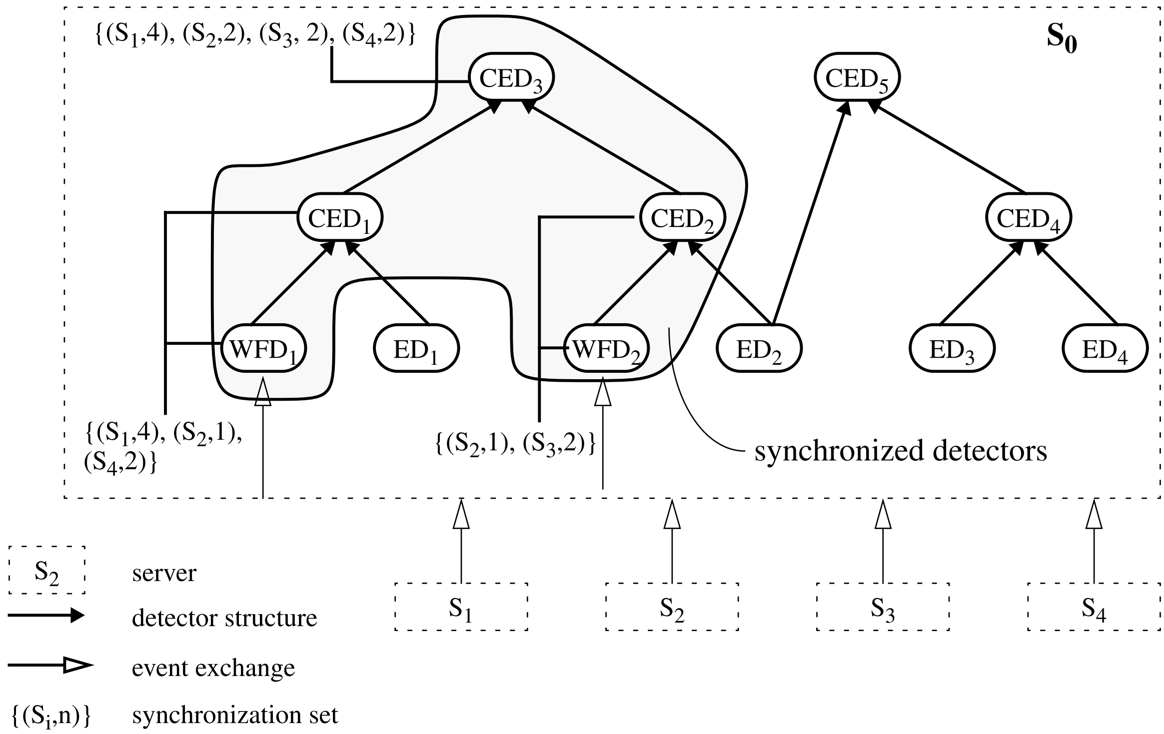

Figure 5: Synchronization of Servers and Event Detectors

that all events with timestamps earlier than the synchronization point have been received, it is safe for an ED to consume these events for composite event detection.

Whenever one of the affected detectors at S_1 receives an event or a component event, the timestamp of this event will be larger (i.e., later) than the synchronization point. In case the event was *not* sent by the MRS, the received event is queued and not further processed. Otherwise, the new MRS is determined (which then also determines the new synchronization point), and all queued events that occurred prior to the new synchronization point are flushed (i.e., processed as in centralized event detection).

Two kinds of events exchanged between servers are relevant for synchronization: synchronization events and workflow termination events. Synchronization events do not bear any specific meaning, but only indicate that the sending server is still alive. They are sent out at predefined synchronization intervals to all servers from which request events for still executing (sub) workflow instances have been received. In case a server receives such an event, it knows that it has received all previous workflow termination events from the remote server, and can exploit this information for synchronization of its detectors.

Workflow termination events indicate the completion of a workflow instance; they are processed by ED. In case such an event is received (i.e., a workflow reply or exception), the reference counter of the respective server in the corresponding synchronization set is decremented. If the new value of the reference counter is 0, then no more instances of the corresponding workflow type are active at the remote server. This server can thus be deleted from the synchronization set of the respective detectors (and their ancestors).

This procedure allows to minimize for each server and workflow type the number of remote servers it has to be synchronized with. Consequently, detectors at S_1 are only then synchronized with S_i if S_i in fact currently executes on behalf of S_1 one or more instances of the workflow type.

The event detection locations for the execution of the example workflow are shown in Table 3. In our example, note that the detector of the event `CON(RPL(CntrlCoverage.OK, HIC), CON(RPL(SW1.Done, Result), RPL(SW2.Done, Result)))` has to be synchronized with the ED for its component events.

Agency site (s1)	Headquarter site (s2)	Clearing house site (s3)
REQ(HandleHIC);		
RPL(HandleHIC.Done);		
REQ(CntrlCoverage)	**REQ**(SW1);	**REQ**(SW2)
	REQ(CntrlTreatment);	**REQ**(CntrlMedication)
	RPL(CntrlTreatment.Done);	**RPL**(CntrlMedication.Done)
RPL(SW1.Done); **RPL**(SW2.Done); **RPL**(CntrlCoverage.OK)		
CON(**RPL**(CntrlCoverage.OK), **CON**(**RPL**(SW1.Done), **RPL**(SW2.Done)))		
REQ(LogClaim);		
RPL(LogClaim.Done);		
REQ(PrepAccept);		
RPL(PrepAccept.Done);		
REQ(Print);		
RPL(Print.JobCompleted);		

Table 3. Event detection sites for a sample workflow execution (time precedence is depicted by the horizontal table lines, event parameters are omitted for simplification)

5.3 Event History

At a specific point in time, the sequence of event occurrences describes what has happened in the past. These occurrences form the *event history.* First, the elements of the history together with other events occurring subsequently constrain what will happen in the future. This is the case for an occurrence if its type is a component of a composite event type and has not yet been consumed for an occurrence of that type. Second, (parts of) the entire event history may provide important information about broker actions and the course the single workflows have taken. The event history thus is a database for monitoring and analysis (and successive optimization) of workflows [15]. It is maintained by EVE-servers and is physically distributed among them. Each server maintains a consistent view of the global event history because composite events are inserted in it only after detector synchronization has taken place meaning that no earlier candidate component events have occurred. A logically integrated view of the global event history is possible based on the partial ordering of the timestamps of events. The history also forms the basis for the formal definition of workflow execution correctness [30].

5.4 Summary

Based on the services described above, EVE is a middleware layer that allows distributed reactive components to cooperate in a way that renders distribution transparent. The event-based style of EVE eases the integration and coordination of these reactive components and the maintenance and evolution of workflow systems. EVE's multi-server architecture along with distributed event detection and detector synchronization avoids a centralized architecture; it is therefore potentially more efficient and less vulnerable to functionality degradation in case of site crashes (depending of course on the workflow specifications currently executing).

Special-purpose services in EVE (such as task assignment) enable reactive components to execute workflows in an event-driven style; workflow systems described in terms of an event-based model can thus be made executable in a seamless way.

6 Conclusions

A proper software architecture of workflow systems is crucial in order to adequately structure the entire environment (including PE, not only the coordination system) and to fulfill further requirements such as flexible integration of PE, reusability of PE-definitions, etc. We use a layered event-based architecture for workflow systems. Specifically, we have introduced the distributed event engine EVE serving as the underlying execution platform for workflows. An event-based approach to workflow execution combines runtime efficiency with flexibility. The contributions of this paper can be summarized as follows:

- a novel architecture for a distributed WS is presented providing the functionality for event-driven workflow execution,
- an integrated framework addresses problems relevant to the distributed execution of workflows (e.g., event-detection, global event history).

Further work remains to be done on how to specify and implement semantic recovery mechanisms in EVE based on ECA-rules. Ultimately, we will address workflow type evolution, particularly with respect to long running workflows and organizational change. The capabilities in this area obviously depend on the semantics of workflows, and we will rely on our previous work on rule-base evolution in object-oriented ADBMS.

References

1. ActionWorks. The ActionWorks Metro Solution (http://www.actiontech.com/Metro/overview/index.html).
2. The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS FEatures. *ACM SIGMOD Record* 25:3, September 1996.
3. G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan. Functionality and Limitations of Current Workflow Management Systems. *IEEE-Expert* (to appear in a special issue on Cooperative Information Systems), 1997.
4. G. Alonso, C. Hagen, H.-J. Schek, M. Tresch. Distributed Processing over Stand-alone Systems and Applications. In *Proc. 23rd VLDB*, Athens, Greece, August 1997.
5. D. Barbará, S. Mehrota, M. Rusinkiewicz. INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical Report, Department of Computer Science, University of Houston, May 1994.
6. D.J. Barrett, L.A. Clarke, P.L. Tarr, A.E. Wise. A Framework for Event-Based Software Integration. *ACM Trans. on Software Engineering and Methodology* 5:4, October 1996.
7. C. Bussler, S. Jablonski. Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems. *Proc. 4th RIDE-ADS*, Houston, February 1994.

8. M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, M.J. Zwilling. Shoring Up Persistent Applications. *Proc. ACM SIGMOD*, Minneapolis, May 1994.
9. F. Casati, S. Ceri, B. Pernici, G. Pozzi. Deriving Active Rules for Workflow Management. *Proc. 7th DEXA*, Zurich, Switzerland, September 1996.
10. S. Ceri, P. Grefen, G. Sanchez. WIDE - A Distributed Architecture for Workflow Management. *Proc. RIDE*, 1997.
11. S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. *Proc. 20th VLDB*, Santiago, Chile, September 1994.
12. U. Dayal, M. Hsu, R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *Proc. SIGMOD*, Atlantic City, NJ, May 1990.
13. D. Georgakopoulos, M. Hornick, A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, Kluwer Academic Publishers, September 1994.
14. A. Geppert, M. Kradolfer, D. Tombros. Realization of Cooperative Agents Using an Active Object-Oriented Database Management System. In *Proc. 2nd Intl. Workshop on Rules in Database Systems*, Athens, Greece, September 1995.
15. A. Geppert, D. Tombros. Logging and Post-Mortem Analysis of Workflow Executions based on Event Histories. *Proc. 3rd Intl. Workshop on Rules in Database Systems*, Skoevde, Sweden, June 1997.
16. M. Hsu, C. Kleissner. ObjectFlow: Towards a Distributed Process Management Infrastructure. *Distributed and Parallel Databases*, 4:2, April 1996.
17. B. Krishnamurthy, D.S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21:10, October 1995.
18. H. Lam, S.Y.W. Su. ECAA Rules and Rule Services in CORBA.
19. R. Marshak. InConcert Workflow. *Workgroup Computing Report* 20:3, 1997.
20. J. Mylopoulos, A. Gal, K. Kontogiannis. A Generic Integration Architecture for Cooperative Information Systems. In *Proc. 1st Int'l Conf. on Cooperative Information Systems,* Brussels, Belgium, June 1996.
21. The Common Object Request Broker: Architecture and Specification. Revision 2.0. The Object Management Group, July 1995.
22. CORBA Services: Common Object Services Specification. OMG, July 1997 (http://www.omg.org/corba/sectran1.htm).
23. Workflow Management Facility. Joint Submission to the Workflow RFP, OMG, August 1997.
24. D.C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. *Proc. 6th USENIX C++ Technical Conf.*, Cambridge, MA, April 1994.
25. D.C. Schmidt. Acceptor A Design Pattern for Passively Initializing Network Services. *C++ Report*, November 1995.
26. S. Schwiderski, A. Herbert, K. Moody. Monitoring Composite Events in Distributed Systems. Technical Report 387, Computer Laboratory, Cambridge University, UK, February 1996.
27. A. Sheth, K. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, I. Shevchenko. Supporting State-wide Immunization Tracking using Multi-Paradigm Workflow Technology. *Proc. 22nd VLDB*, Bombay, India, September 1996.

28. D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek (eds.), *Object-Orientation with Parallelism and Persistence*, Kluwer Academic Publishers, 1996.
29. D. Tombros, A. Geppert, K.R. Dittrich. Design of Coopearive Process Oriented Environments Using Reactive Components. Technical Report 97.06, Department of Computer Science, University of Zurich, June 1997.
30. D. Tombros, A. Geppert, K.R. Dittrich. Semantics of Reactive Components in Event-Driven Workflow Execution. *Proc. 9th Intl. Conf. on Advanced Information Systems Engineering*, Barcelona, Spain, June 1997.
31. P. Verissimo. Real-Time Communication. In S. Mullender (ed): *Distributed Systems*. 2nd ed., Addison-Wesley 1993.
32. D. Wodtke, J. Weissenfels, G. Weikum, A. Kotz-Dittrich. The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. *Proc. 12th ICDE*, New Orleans, February 1995.