An Open ECA Server for Active Applications

Florian Daniel
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento
Via Sommarive 14
I-38100 Povo (TN), Italy
daniel@disi.unitn.it
http://www.floriandaniel.it

Giuseppe Pozzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. Da Vinci, 32
I-20133 Milano (MI), Italy
giuseppe.pozzi@polimi.it
http://home.dei.polimi.it/pozzi/

April 14th, 2008

PAPER CATEGORY: RESEARCH PAPER

An Open ECA Server for Active Applications

Abstract

Event monitoring and active behaviors are important aspects in many software systems and application domains, not only in database management systems. In this paper, we propose an Event-Condition-Action (ECA) approach that spans from application data to application components and behaviors. Starting from an exception manager we previously developed in the context of a workflow management system, we derived an autonomous active component capable of handling a variety of events and of enacting actions in response to detected events. The ECA server runs as an autonomous engine and can be seamlessly integrated with existing systems, thus enhancing the systems' functionalities and maintainability by separating active and non-active design concerns.

Keywords

ECA rules, Active rules, Event monitoring, Open ECA server, OES, Autonomous ECA server, Active applications

INTRODUCTION

Until the emergence of the first operating systems and high-level programming languages allowed developers to disregard hardware peculiarities, computers had to be programmed directly in machine code Then, only in the eighties, Database Management Systems (DBMSs) provided efficient, external data management solutions, and in the nineties Workflow Management Systems (WfMSs) extended this idea and extracted entire processes from still rather monolithic software systems. We believe that in similar way also active (also known as reactive) behaviors, which are present in many modern applications (see for instance Section 2), can be more efficiently managed by proper active software supports, such as active rules and rule engines (Section 3).

The basic observation underlying this idea is that, when abstracting from the particular application and domain, most of the active behaviors in software systems adhere to the rather regular and stable ECA (Event-Condition-Action) paradigm. ECA rules have first been introduced in the context of active DBMSs, where operations on data may raise events, conditions check the status of the database, and actions perform operations on data. Our previous experience in the field of WfMSs (Casati, Ceri, Paraboschi, and Pozzi, 1999; Combi and Pozzi, 2004) allowed us to successfully apply high-level ECA rules to WfMSs for the specification and handling of expected exceptions that may occur during process execution. By leveraging this experience, in this paper, we propose an ECA paradigm accompanied by a suitable rule language, where events represent data, temporal, application or external events, conditions check the state of data or of the application, and actions may act on data, applications, or external resources. Active rules may thus not only refer to the data layer, but as well to the whole application, comprising data and application-specific characteristics. Elevating active rules from

the data layer to the application layer allows designers to express a broader range of active behaviors and, more importantly, to address them at a suitable level of abstraction (Section 4). This could turn out beneficial for example in requirements engineering approaches, such as the ones described by Loucopoulos and Kadir (2008) or by Amghar, Meziane, and Flory (2002), as well as in re-engineering approaches like the one described in Huang, Hung, Yen, Li, and Wu (2006).

For the execution and management of ECA rules, we further propose an open ECA server (OES), which runs in a mode that is completely detached from the execution of the actual application, so as to alleviate the application from the burden of event management. OES is highly customizable, which allows developers to easily add application- or domain-specific features to the rule engine (Section 5 describes the customization process, Section 6 illustrates a use case of the system). Instead of implementing the OES system from the scratch, we shall show how we unbundled and reconfigured the necessary components from a previously developed exception manager for a WfMS (Casati et al., 1999) (Section 7) – unbundling is the activity of breaking up monolithic software systems into smaller units (Gatziu and Koschel, 1998). We thus move from the ECA server we developed within the EC project WIDE to manage exceptions in the context of Sema's FORO commercial WfMS, where the exception manager (FAR) was tightly bundled into FORO.

RATIONALE AND BACKGROUND

Active mechanisms or behaviors have been extensively studied in the field of active DBMSs as a flexible and efficient solution for complex data management problems. Many of the results achieved for relational or object-oriented active databases have recently been extended to tightly related research areas such as XML repositories and ontology storage systems. To the best of our knowledge, only few works (Dittrich, Fritschi, Gatziu, Geppert, and Vaduva, 2003; Chakravarthy and Liao, 2001; Cugola, Di Nitto, and Fuggetta, 2001) try to elevate the applicability of active rules from the data level to the application level and to eliminate the tedious mapping from active behavior requirements to data-centric active rules (Section 8 discusses related works in more detail). Besides DBMSs, there are several application areas, which could significantly benefit from an active rule support that also takes into account their application- or domain-specific peculiarities. Among these application areas, we mention here:

- WfMSs or in general business process management systems allow one to define the system-assisted execution of office/business processes that may involve several actors, documents, and work items. Active mechanisms could be exploited for an efficient enactment of the single tasks or work items, and the management of time constraints during process execution (Combi and Pozzi, 2003; Combi and Pozzi, 2004).
- Web services and (Web) applications, which use Web services as data sources or incorporate their business logic (Li, Huang, Yen, and Chang, 2007), may rely on an asynchronous communication paradigm where an autonomous management of incoming and outgoing events (i.e., messages) is crucial. Suitable active rules could ease the integration of Web services with already existing (Web) applications. Active rules could further serve for the coordination of service compositions, similar to the coordination of actors and work items in a WfMS (Charfi and Mezini, 2004; Daniel, Matera, and Pozzi, 2006).

- Exception handling is gaining more and more attention as a cross-cutting aspect in both WfMSs and service compositions. The adoption of active rules for the specification of exception handlers to react to application events has already proved its viability in the context of WfMSs (Casati et al., 1999; Combi, Daniel, and Pozzi, 2006). Their adoption for handling exception also in service compositions would thus represent a natural evolution.
- Time-critical systems or production and control systems, as well as the emerging approaches to self-healing software systems (Mok, Konana, Liu, Lee, and Woo, 2004; Minsky, 2003), intrinsically contain features or functionalities that are asynchronous with respect to the normal execution of the system (e.g., alerting the user of the occurrence of a production error). Their execution may indeed be required at any arbitrary time during system execution, and may thus not be predictable. Active rules are able to capture this peculiarity at an appropriate level of abstraction.
- Adaptive applications or context-aware, ubiquitous, mobile, and multi-channel applications incorporate active or reactive behaviors as functional system requirements (Wyse, 2006). The event-condition-action paradigm of active rules thus perfectly integrates with the logic of adaptivity, proper of such classes of software systems. The use of a dedicated rule engine for the execution of rules representing adaptivity requirements fosters the separation of concerns and the possibility of evolution of the overall system (Daniel et al., 2006; Daniel, Matera, and Pozzi, 2008).

SUPPORTING ACTIVE BEHAVIORS IN APPLICATIONS

The above mentioned application areas show a wide range of potential applications of active mechanisms and rule engines. Current approaches, however, mainly operate on the data level and do not provide an adequate abstraction to also address application logic when specifying events, conditions, and actions. As a consequence, developing applications with active behaviors requires developers to address – each time anew – some typical problems:

- the definition of a set of events that trigger active behaviors and the development of suitable event management logic (the *event manager*);
- the implementation of generic and application-specific action *executors*, which enable the enactment of the actual active behaviors;
- possibly, the design of appropriate *rule metadata*, required to control rule execution and prioritization;
- the specification of a suitable *rule specification formalism*; and
- the development of an according rule interpretation and execution logic (the *rule engine*).

Figure 1 arranges the previous design concerns into a possible architecture for active applications. Of course, in most cases, the described modules and features might not be as easily identifiable, because the respective functions are buried in the application code or because they are just not thought of as independent application features. Nevertheless, conceptually we can imagine the internal architecture be structured like in Figure 1.

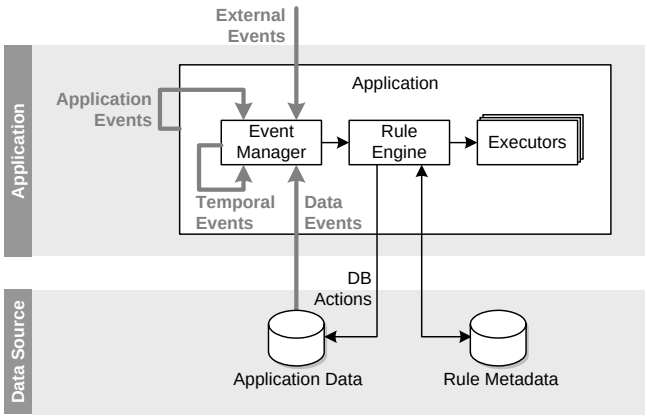

Figure 1 With no decoupled support for the management of active rules, each application internally needs to cater for suitable rule management functions and rule metadata.

Typically, we classify events as *application events*, *data events*, *temporal events*, or *external events*. Application events originate from the inside of the application; data events originate from the application's data source; temporal events originate from the system clock; and external events originate from the outside of the application. All possible events in active applications can be re-conducted to these four classes of events (Eder and Liebhart, 1995).

Given the previous considerations, developing active application may represent a cumbersome undertaking. We however believe that developers can largely be assisted in the development of such applications by introducing a dedicated, detached rule execution environment that extracts the previously described active components from applications and acts as intermediate layer between the application's data and its application logic. This further fosters the separation of concerns between application logic and (independent) active behaviors and the reuse and maintainability of active rules.

The idea is graphically shown in Figure 2. Applications provide for the necessary *application events* (now *external events* with respect to the rule engine) and the set of action *executors* that enact the respective active behaviors; each application may have its own set of executors. The *customizable rule engine* allows the applications to delegate the capturing of data events, temporal events, and external events as well as the management of the set of rules that characterize the single applications. The rule engine includes the necessary logic for maintaining suitable *rule metadata* for multiple applications. The described architecture requires thus to address the following research topics:

- the specification of a customizable *rule specification language*;
- the development of a proper *runtime framework* for rule evaluation;
- the provisioning of easy *extension/customization mechanisms* for the tailoring of the generic rule engine to application-specific requirements.

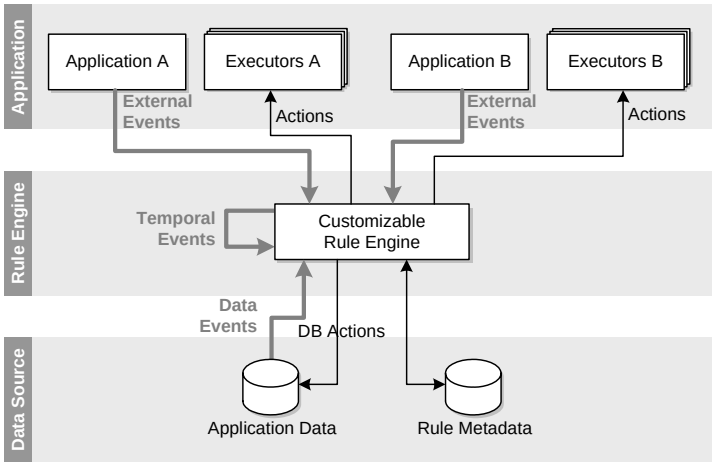

Figure 2 The introduction of a decoupled rule engine may largely assist the development of active applications.

In the following, we propose the OES system, a rule execution environment that provides an implementation of the idea expressed in Figure 2. OES is based on the so-called OpenChimera language for rule specification and provides for advanced customization support.

THE OES SYSTEM

The OES system consists of two main logical components that complement each other: the OpenChimera rule language for the definition of active behaviors and the OES rule engine for the execution of OpenChimera rules. Both rule language and rule engine are extensible and easily customizable, in order to be able to manage application-specific events, conditions, and actions.

The OpenChimera Language

The OpenChimera language is derived from the Chimera-Exception language (Casati et al., 1999), a language for the specification of expected exceptions in WfMSs. Chimera-Exception is based, in turn, on the Chimera language (Ceri and Fraternali, 1997) for active DBMSs. OpenChimera builds on an object-oriented formalism, where classes are typed and represent records of typed attributes that can be accessed by means of a simple dot-notation. Rules adhere to the following structure:

```
define trigger <TriggerName>
  events    <Event> [(,<Event>)+]
  condition [<Cond> [(,<Cond>)+]|none]
  actions   <Action> [(,<Action>+)]
  [order <PriorityValue>]
end
```

A trigger `<TriggerName>` has one or more disjunctive triggering events (`<Event>`), a condition with one or more conjunctive conditional statements (`<Cond>`), and one or more actions (`<Action>`) to be performed in case the condition of the triggered rule holds. Rules may have an associated priority (`<PriorityValue>`) in the range from 0 (lowest) to 1000 (highest). Priorities enable the designer to define a rule execution order.

Events

Events in OES can be specified according to the following taxonomy:

- *Data events* enable the monitoring of operations that change the content of data stored in the underlying (active) DBMS. Similarly to rules in active databases, monitored events are `insert`, `delete`, and `update`. Data events are detected at the database level by defining suitable rules (or triggers) for the adopted active DBMS.
- *External events* must be first registered by applications in order to be handled properly. External events are recognized by means of the `raise` primitive, which – when an external event occurs – provides the name of the triggering event and suitable parameters (if needed).
- *Temporal events* are related to the occurrence of a given timestamp and are based on the internal clock of the system. In order to cope with a worldwide environment, all the temporal references of these events are converted to the GMT time zone. Temporal events are categorized as instant, periodic and interval events:
  - *Instant events* are expressed as constants preceded by an @-sign (e.g. `@timestamp ''December 15th, 2005, 18:00:00''`);
  - *Periodic events* are defined using the `during` keyword, separating the start of the event from the respective time interval (e.g. `1/days during weeks` denotes the periodic time defined by the first day of each week). The full notation and additional details can be found in (Casati et al., 1999);
  - *Interval events* are expressed as `elapsed duration since instant`, where instant is any type of event used as anchor event (e.g. `elapsed (interval 1 day) since modify (amount)`).

Conditions

Conditions bind elements and perform tests on data. Since the adopted mechanism for rule execution is detached, i.e. the triggering event and the rule execution take place in two separate transactions, at rule execution time the context of the triggering event is reconstructed for condition evaluation. For instance, if we consider a data event triggered by the modification of a tuple, the `occurred` predicate of the OpenChimera language is used to select only the tuples that have really been modified and on which the trigger can, possibly, execute the specified action.

Actions

Standard actions that can be performed include changes to the database and notifications via e-mail messages. Other application-specific actions can be defined by means of external executors. Several executors may be available, each one typically dedicated to one specific action. As we shall show in Section 5, the customization of the actions that are available for rule definition represent the real value of the OES system.

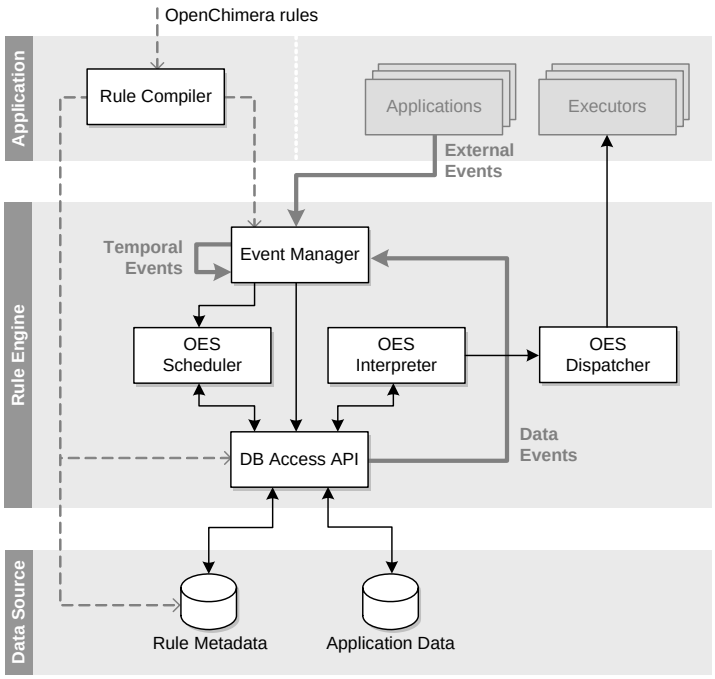

Figure 3 The architecture of the autonomous ECA server OES.

The OES Rule Engine

The internal architecture of the OES system, detailed in Figure 3, is composed of: Rule Compiler, Event Manager, Scheduler, Interpreter, DB access API, and Dispatcher. The main features of the constituent modules are described in the following.

- *OES Rule Compiler*: The Compiler accepts rules at rule creation time and translates them into an intermediate execution language, proper configurations of the Event Manager, and suitable rule metadata that are accessed at rule evaluation time. The Compiler is invoked by specifying (i) the name of the file containing the source code of the rule and (ii) the name of a file containing a data dictionary for the specific application domain, which is basically a standard text file describing the data types used for type checking at compile time.

  Besides rule compilation, the Compiler is also in charge of rule management: commands inside a source file provided in input to the compiler allow the developer to add new rules (`define trigger`), to remove existing rules (`remove trigger`), or to modify existing rules (`modify trigger`), thus enabling an incremental rule definition and a flexible rule management.

- *OES Event Manager*: The Event Manager is sensitive to external and temporal events. For the correct interpretation of interval events, the module registers those events that are used as anchor events and raises the actual event only once the respective interval has elapsed. Instant and periodical events are managed by means of a proper `WakeUpRequest` service. Finally, the Event Manager may invoke the OES Scheduler directly if a real-time event is raised.

- *OES Scheduler*: The Scheduler periodically determines the rule instances which have been triggered by monitoring the rule metadata and schedules triggered rules for execution according to the rules' priorities. The Scheduler is automatically invoked in a periodical fashion, but it can also be invoked directly by the Event Manager: this forces an immediate scheduling of the respective rule, still respecting possible priority constraints.
- *OES Interpreter*: The Interpreter is called by the OES Scheduler to execute a specific rule in the intermediate language. The Interpreter evaluates the rule's condition and computes respective parameters. If a condition holds, actions are performed via the DB access API or via the OES Dispatcher.
- *OES Dispatcher*: The Dispatcher provides a uniform interface for the execution of actions by external executors and hides their implementation details to OES. External executors play a key role in the customization of the system.
- *OES DB Access API*: The DB Access API provides a uniform access to different DBMSs. At installation time, OES is configured with the driver for the specific DBMS adopted. Specific drivers are needed, since OES also exploits some DBMS-specific functionalities for the efficient execution of database triggers.

CUSTOMIZING THE OES SYSTEM

As described in the previous section, OES comes with a default set of generic events and actions; domain-specific events and actions can be specified in form of external events and suitable external executors. Hence, if the default set of events and actions suffices the needs of the developer, he/she can immediately define rules without performing any additional customization. If, instead, domain-or application-specific events and actions are required, he/she needs to customize the OES system.

Customizing Events

New events are specified as *external events*, which are supported by the OES system through a proper raising mechanism. External events must be registered in the OES system, in order to enable their use in the definition of OpenChimera triggers. If notified of the occurrence of an external event, OES inserts a respective tuple into the rule metadata. The metadata is periodically checked by the OES Scheduler and enables condition evaluation and action execution.

When customizing events, the customizer has to implement the external program(s) that might raise the event(s). Communications between external program(s) and OES are enabled through a CORBA message passing mechanism. We observe that if the adopted DBMS has no active behavior, no data event can be defined; temporal and external events, instead, can be normally defined, detected, and managed as they do not require any specific active behavior from the DBMS.

Customizing Conditions

The syntax of OpenChimera conditions can be extended with new data types, abstracting tables in the underlying database. The definition of new types occurs by means of a so-called *data dictionary*, which is a standard text file containing a name and a set of attributes for each new

data type. At rule compilation time, the OES Compiler, besides rule definitions themselves, requires the data dictionary to evaluate the proper use of data types for the variables included in the trigger definition. The definition of the data dictionary is the only situation where the Compiler has to read data that are specific to the application domain.

OES adopts a detached trigger execution model, where the triggering part of a rule is detected in one transaction, and the condition and action parts of the trigger are executed in another transaction. The definition of suitable data types in the data dictionary allows OES to reconstruct at condition evaluation time the status of the transaction in which the rule was triggered.

Customizing Actions

Adding a new action to the syntax of the OpenChimera language requires adding suitable descriptions and action executors to a so-called *Action Dictionary*. At rule compilation time, if the OES Compiler encounters an action that is not included in the set of predefined actions, it checks whether the specified action is included in a specific view in the database (the view `Action-Dictionary` can be seen in Figure 6) by searching the specified action in the `ActionName` attribute of the table `Action`. If the action is described in the view and its signature (as specified by the `Action_Tag` table) complies with the parameters of the rule to be compiled, the action is valid. If the OES Compiler fails in finding a matching tuple in the Action Dictionary, a suitable error message is generated. At rule execution time, the OES Interpreter processes the rule and the OES Dispatcher invokes the specified executor, as defined by the Action Dictionary, launching it as a child process.

Executors in OES can be characterized according to three orthogonal aspects: the location of the executor, dynamic vs. static parameters, and XML support:

- *Location*. Executors can be either *local* applications, running on the same system where OES is running, or *remote* services accessible via the Internet. We observe that services, even if running on the same system as OES, are always considered remote services.
- *Parameters*. Executors typically require input data. Parameters can be *dynamically* computed by the OES Interpreter at run time, or they can be *statically* defined. If dynamic parameters are required, the Interpreter performs a query over the application data, computes the actual parameters, and writes them into an XML file. Static parameters can be directly taken from the definition of the action and added to the XML file.
- *XML support*. Some executors are able to parse XML files, others do not. If an executor parses XML, it is up to the executor to extract the parameters correctly. If an executor does not parse XML, an intermediate parser is used to extract the parameters from the XML file and to invoke the executor, suitably passing the required parameters.

According to the above criteria, executors are divided into the following categories:

a) *Commands*. Local applications with static parameters that are not capable of parsing XML. The Dispatcher of OES constructs the command line and invokes the local system service according to the parameters stored in the `Executor` table of Figure 6. Such an executor is identified by the attribute `CommandType="CMD"`, e.g. this may happen for a periodical backup service performed via the `tar` command of a Unix system.

b) *Executors capable of reading XML files*. Dynamic parameters are computed by the OES Interpreter and stored in an XML file. The executor, in turn, can be a local application or a client connecting to a remote service. Executors reading XML files are classified as follows:

   b1) *Local applications*. The Dispatcher of OES invokes the local application and passes it the name of the XML file with the parameters.

   b2) *Client connecting to an XML-enabled remote service*. The Dispatcher of OES starts a client application that connects to the remote service and sends the XML file via the HTTP POST method. The executor, in turn, may reply with another XML file, e.g. containing the results or the return code of the service.

c) *Executors not capable of reading XML files*. Dynamic parameters are computed by the OES Interpreter and stored in an XML file. The invocation of the executors is performed via specific, intermediate parsers, which extract the necessary parameters from the XML file and invoke the executors by suitably passing the required dynamic parameters. Analogously to XML-enabled executors, not XML-enabled executors are classified as follows:

   c1) *Local applications*. The parser invokes the local application passing it the dynamic parameters in the appropriate format.

   c2) *Client connecting to a remote service which is not XML-enabled*. The parser sets up a client-server connection with the remote service and passes it the dynamic parameters in the appropriate format, possibly receiving results back.

It can be observed that executors not capable of reading XML files are internally treated like executors capable of reading XML files by leveraging an intermediate layer of suitable parsers, one parser for each specific executor. Figure 3 summarizes the taxonomy of executors.

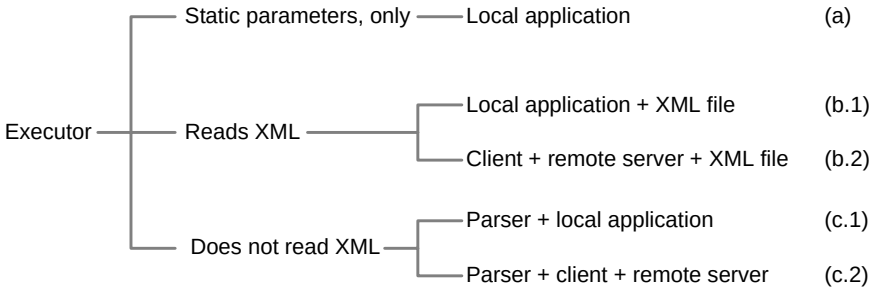

Figure 4 Taxonomy of executors.

CASE STUDY – THE NEW YORK STOCK EXCHANGE

In order to show how to customize OES in practice, we consider the stock exchange market. The customers of a personal stock management software would like to be notified via SMS if the price of one of their stocks (e.g. "MCP") exceeds predefined limits; stock prices are to be updated every 30 minutes during working days. Figure 5 shows an excerpt of the data structure underlying the stock management software, to be used for the integration with OES.

As we shall show in the following, supporting the required SMS feature requires the OES system to be extended with two new actions: one (`updateStock`) for the periodic update of the stock value, and one (`sendSMS`) to send the SMS notification message.

stockValue

name	value	timeStamp
MCP	4:35	10:13 GMT 31-Oct-2003
IUO	18:52	11:33 GMT 30-Oct-2003

notification

customerId	stockName	min	max	active
2043	MCP	4.00	4.50	yes
2045	MCP	4.10	4.40	yes
2043	IUO	17.50	20.15	no

customer

Id	cellNumber
2043	+347-0123456
2045	+348-7654321

Figure 5 The stockValue, customer, and notification tables as defined by the management software.

Customizing OpenChimera and the Rule Engine

The event for the periodic update of the stock price in the underlying database is a *periodic temporal event*, while the event triggering the sending of the SMS notification is a *data event*. As both events are default OpenChimera events, no customization of OpenChimera events needs to be performed.

The definition of suitable conditions over the database tables described in Figure 5, requires the definition of according data types in the data dictionary. More precisely, the three data types `stockValue`, `notification`, and `customer`, referring to the respective tables in the database, must be included into the data dictionary, in order to be able to bind variables to them and formulate proper data queries.

The two new actions (`updateStock` and `sendSMS`) can be made available to the OpenChimera environment by means of two new tuples in the `Action` table of the OES system. In table `Action_Tag` of Figure 6, the three tuples with attribute `ActionName` set to `sendSMS` or `updateStock`, respectively, serve this purpose and conclude the customization of the OpenChimera syntax. For the customization of the rule engine, we need to implement and to register the two actions `sendSMS` and `updateStock` as external executors.

As for the `sendSMS` action, the transmission of short text messages to cell phones can be performed free of charge from the Internet sites of major mobile telephone companies and of major portals. Our executor for the new defined `sendSMS` action thus connects to a suitable Web server and requests the transmission of messages. We assume that the executor `myBrowser` serves this purpose. The definition of the new action requires thus the insertion of a new tuple into the `Action` table and the definition of proper attributes (see Figure 6):

- `ActionName` defines the name of the action;
- `Priority` defines the default priority for the action (i.e. `10`), which can be overwritten by means of the `order` statement in the rule definition;

- `CommandType` defines whether the action corresponds to an executor not capable of reading XML files and with static parameters ("`CMD`"), or an XML-enabled executor ("`XML`");
- `CommandRequest` defines the actual invocation command to be launched by the Dispatcher;
- `ExecutorId` is the unique identifier of the executor.

We consider now the action named `sendSMS` with executor id `22`: `CommandType` is "`XML`", indicating that the executor is XML-enabled. `CommandRequest` is the name of the executor that receives the XML file via the command line, connects to the remote server, and forwards the XML file. The first tuple of the `Action` table thus binds the `sendSMS` action to a proper executor.

To complete the definition of the action, we have to specify how static parameters can be passed to the executor. Static parameters are defined by tuples in the `Executor` table (see Figure 6):

- `ExecutorId` is the unique identifier of the executor;
- `Location` defines the location where the executor can find the remote service, if needed. In fact, if the executor requires a remote service, the executor runs as a client, connects to a valid URL defined by `Location`, and sends out the XML file created by the Interpreter. If `Location` is set to `localhost`, no remote service is needed;
- `Par1`, `Par2`, `Par3` define the static parameters that may be used by local commands which are not capable of reading XML files. We recall that this kind of executors is labeled "`CMD`" in the attribute `CommandType` of the `Action` table.

Action

ActionName	Priority	CommandType	CommandRequest	ExecutorId
sendSMS	10	XML	/usr/local/bin/myBrowser	22
sendEMail	5	XML	/usr/bin/myMailer	25
Backup	1	CMD	/usr/local/bin/tar	30
updateStock	20	XML	/usr/local/bin/myUpdateStock	6

Executor

ExecutorId	Location	Par1	Par2	Par3
22	http://freesms.jumpy.it			
25	localhost			
30	localhost	-xvf	/usr/home/agents	/dev/rmt8
6	http://quotazioni.borsitalia.it			

Action_Tag

ActionName	Tag	Pos
sendSMS	CellNumber	1
sendSMS	CellMessage	2
sendEMail	ESubject	1
sendEMail	EAddressee	2
sendEMail	EText	3
updateStock	StockName	1

Figure 6 Action-Dictionary view: Action, Executor, and Action_Tag tables. By joining them on the ExecutorId and on the ActionName attributes, we obtain the Action-Dictionary view. The Action_Tag table is used to check the signature of executors at rule compilation time. The names of system tables and of related attributes are capitalized.

As can be seen in Figure 6, the `sendSMS` action requires dynamic parameters that will be computed at runtime and stored in an XML file. Specified parameters are translated into suitable tags in the XML file and sorted according to the order in which they appear in the source code of the rule. Dynamic parameters are specified in the `Action_Tag` table:

- `ActionName` defines the name of the action;
- `Tag` is the name of the tag inside the XML file (tag names must match the data dictionary);
- `Pos` defines the order of the parameters to be used in the OpenChimera language.

Thus, if the action is `sendSMS`, the two topmost tuples of `Action_Tag` define that the XML file to be sent to the executor must be constructed as follows: the first dynamic parameter is the number of the cell phone of the customer, and the second dynamic parameter is the message to be sent to the customer.

The specification of the executor for the `updateStock` action is analogous to the one of the `sendSMS` executor. The information we need to store represents the price of a stock at a given time instant. To access this information, we again use an executor that uses the Web to accomplish its task by searching the Web for the stock price and storing it into the application's data source.

To make the action `updateStock` available, we deploy a suitable executor, namely `myUpdateStock`, available in the directory `/usr/local/bin`. Again, its inclusion into OES requires inserting a suitable tuple in the `ActionDictionary` view of Figure 6. The name of the action is `updateStock`, its priority is `20`, its type is `"XML"`, the executor is `myUpdateStock`, and the id is `6`. Dynamic parameters for the executor are defined by the `Action_Tag` table: for the current action, the only dynamic parameter needed is the name of the stock. The executor `myUpdateStock` thus receives in input an XML file containing the name of the stock and connects to the remote server. The invoked remote service replies with another XML file, from which `myUpdateStock` reads the stock name, its value and its timestamp as defined by the remote server, and stores these data in the database.

Specifying the Active Rules

Now we can specify the actual rules to define the required active behavior. For presentation purposes, we assume that all customers are interested in the "`MCP`" stock, only.

The `myUpdateStock` executor accesses the DBMS and stores the stock name, the stock price and its timestamp in the `stockValue` table. According to the customized syntax of the OpenChimera language, we can now define the `periodicalStockUpdate` rule as follows.

```
define trigger periodicalStockUpdate
  events     30/minutes during days
  condition  stockValue(S), S.name="MCP"
  actions    updateStock(S.name)
end
```

The event part of the rule states that the rule must be invoked every 30 minutes. The condition part considers all the instances `S` of the `stockValue` type (i.e., all the tuples inside the table named `stockValue`) and selects only the tuples where `S.name` equals "`MCP`". The action part

invokes the executor `myUpdateStock`, corresponding to the `updateStock` action. The OES Interpreter computes the required dynamic parameter by assigning the value "`MCP`" to the tag `StockName` inside the XML file passed to the `myUpdateStock` executor. The `periodicalStockUpdate` rule thus periodically stores the price of the chosen stock in the database.

A second rule is needed to compare the stored price with the allowed range of variability. The respective data are stored in the database and can be accessed by the following rule `stockOutOfRange`, in order to trigger possible SMS notifications:

```
define trigger stockOutOfRange
  events     modify(stockValue.value)
  condition  stockValue(S), notification(N), customer(C),
             S.name=N.stockName, N.customerId=C.Id,
             occurred(modify(stockValue.value),S),
             not(N.min<S.value<N.max), N.active="yes"
  action     sendSMS(C.cellNumber,"Stock "+S.name+
             " out of range. Its current price is "+S.value),
             N.active="no"
end
```

The event part of the rule states that the rule must be invoked each time the attribute value of a tuple inside the `stockValue` table is changed (data event). The condition part has a twofold goal. First, it aims at binding the instances of `stockValue` (`S`), of `notification` (`N`) and of `customer` (`C`). The binding states that the stock must be related to a request of notification by an interested customer: this is performed by a join operation. Second, the conditions part verifies that tuples selected from the `stockValue` table are only those for which there has been a change of the value attribute since the last execution of the rule (`occurred(modify(stockValue.value),S)`), that the new price falls outside the allowed variability range (`not(N.min<S.value<N.Max)`), and that the notification service is active (`N.active="yes"`). The action part is executed after all the conditions are true. The action invokes the executor `sendSMS` whose parameters are the cell phone number of the customer (`C.cellNumber`) and a string message including the name of the stock and its current price. In order to prevent a continuous sending of the same message, a second action disables the notification service (`N.active="no"`) for the sent message. Users can easily enable the service again through their stock management software.

IMPLEMENTATION

The OES system described in this paper is derived from the exception manager FAR (FORO Active Rules), developed within the EC project WIDE and aimed at managing expected exceptions in the workflow management system FORO (Casati et al., 1999). In the following, we shortly outline the architecture of the FAR system and show how OES has been unbundled from FAR. Then, we discuss termination, confluence, and security in OES.

The FAR System

Exception handling in WfMSs typically involves a wide scenario of events and actions. In the case of the FAR system, the rule engine is able to manage the following four categories of events

(Casati et al., 1999): data events, temporal events, workflow events (e.g. the start or the end of a task or of a case), and external events. Concerning the actions that can be enacted through FAR, the rule engine supports the following actions: data manipulation actions, workflow actions (e.g. the start or completion of a task or a process instance, the assignment of a task or case to a specific agent), and notification actions.

Figure 7 graphically summarizes the FORO/FAR architecture. Exceptions are specified by means of the active rule language Chimera-Exception (Casati et al., 1999), from which we derived the OpenChimera language adopted in OES. Besides data events (originating from an active Oracle database shared with the FORO system), temporal events and external events, FAR is directly notified of workflow events coming from the FORO workflow engine. On the action side, database actions are directly supported by the FAR system, while notifications and workflow actions are performed via the FORO workflow engine.

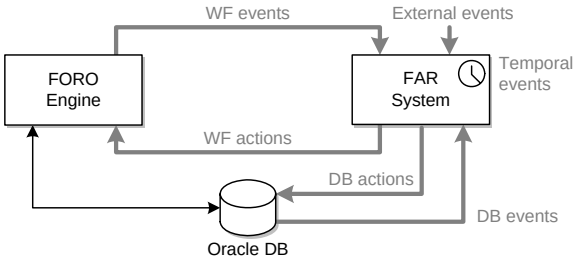

Figure 7 FAR architecture and dependencies with FORO. FAR is bundled into FORO.

Unbundling the Rule Engine

The implementation of the OES system leveraged as much as possible the already existing implementation of the FAR system. Instead of developing a new rule engine from scratch, we decided to unbundle (Gatziu, Koschel, von Bultzingsloewen , and Fritschi, 1998; Silberschatz and Zdonik, 1997) the necessary functionalities and modules from the FORO/FAR system. When unbundling the rule engine from FORO/FAR, we had to re-consider all the interactions of the tightly-coupled, bundled modules. In particular, we had to consider how events are notified to the rule engine and how the rule engine enacts actions.

An extension of FAR's built-in support for both external events and external executors provided efficient means to enable users of OES (i.e., developers of active applications) to define application-specific events and actions. The unbundled OES system thus inherits the support for data events, temporal events, and external events from the FAR system, while workflow events are not supported any longer, due to the unbundling of the rule engine from the WfMS. Analogously, we were able to reuse FAR solutions to support the execution of database actions and the flexible definition of external executors for customizable actions; again, workflow-specific actions were discarded. The introduction of intermediate parsers allows OES to select appropriate executors according to the specifications received from the rule engine.

In order to be capable of detecting events and of performing actions, the unbundled OES system must implement suitable communication channels among the modules composing the system. For example, OES must be able to start transactions over a given DBMS and to invoke external applications, possibly passing some parameters. For the communication between internal

modules, OES leverages CORBA and shared tables in the underlying database. While a shared database works fine for internal modules, the adoption of a specific DBMS (i.e., Oracle) may cause interoperability difficulties with external modules, such as external executors for customized actions. Therefore, the communication with external executors added to the OES system is based on XML as common format for accessing and sharing information. Data is passed in form of XML documents, containing possible static and/or dynamic parameter values or responses from the external executors.

Remarks

Termination

An active system guarantees *termination* if its rules are not allowed to trigger each other indefinitely. If we define a rule *r1* that reacts to the event *e1* by executing the action *a1*, which in turn triggers the event *e1*, the active system enters an endless loop if the condition of *r1* always holds (*self-triggering*). We may also define a rule *r1* that reacts to the event *e1* by executing the action *a1*, which in turn triggers the event *e2* of a rule *r2* whose action *a2* triggers again *e1*. Should the conditions of *r1* and *r2* always hold, the active system enters an endless loop (*cross-triggering*). Similarly, an active system may encounter a situation of *cascaded triggering*, if the endless cycle involves more than two rules.

Potential situations of non-termination can be avoided by *static* and *dynamic* checks. Compile time (static) detection is performed at rule compilation time by the OES Compiler: for each potential loop, it issues a proper warning message. The static check is performed by a suitable termination analysis machine, properly adapted to OES from (Casati et al., 1999). The resolution of possible loops is up to the developer.

Run time (dynamic) detection of loops is more complex in OES than in FAR, as involved actions can be external to OES itself. A self-triggering situation may occur when an action *a1* invokes the server *s1*, which in turn invokes a server *s2* that is external and unknown to OES, and *s2* invokes another server *s3*, whose actions trigger the event *e1* of *r1*. This self-triggering situation is very hard to detect, as it comes from subsequent server invocations outside OES. A simple yet effective *avoidance mechanism* is limiting the maximum cascading level for rules: rules are simply not allowed to trigger other rules indefinitely. OES (like most active DBMSs) adopts this solution and uses an upper limit for cascaded activations that can be easily configured. With respect to generic DBMSs, OES however does not limit this technique to data events only.

Confluence

In a system featuring active behaviors, *confluence* means that the final effect of the processing of multiple concurrently triggered rules is independent of the ordering by which rules are triggered and executed. The problem of confluence arises in many situations, like SQL triggers and stored procedures in most conventional database applications. Typically, those situations generate non-confluent behaviors, because actions are performed over sets of tuples, which by definition come with no ordering criteria.

The same consideration applies to OES: each rule is intrinsically non-confluent, because it associates a set-oriented, declarative condition with a tuple-oriented imperative action, and there

is no language construct to impose a rule-internal order on the bindings that are selected by the condition evaluation part. If in OES we assume to trigger a rule t_1, its condition part may for instance return a set of *n* unordered data tuples to which the rule's actions are to be applied; at this point, we cannot say for sure in which order the actions are enacted, as this typically depends on the underlying active DBMS.

If, instead, we assume to trigger two (or more) rules t_1, t_2, the usage of priorities (i.e., the `order` token of OpenChimera) enables the designer to define an ordering among the rules t_1, t_2, where the highest priority rule is processed first. This option enables the designer to state a partial order among the triggered rules t_1, t_2, but not an order that is internal to each rule.

Security

Security in OES relates to three different aspects: rule definition, event generation, and action execution. At *rule definition* time, the customizer logs into OES and uses the OES Compiler. As triggers and rule metadata are stored inside the DBMS, the security level provided by OES is the one provided by the DBMS.

At *event generation* time, security issues concern data events, temporal events, and external events. Data events require to access the DBMS and to insert, delete, or update data: again, the security level provided by OES is the one provided by the underlying DBMS. Temporal events are triggered by the internal clock of OES: their security level is the one provided by the operating system on which OES is running. External events are triggered by external applications: the security level of the entire system is the one implemented by the external application, which has however to be registered into OES by the customizer prior to being able to trigger any event.

At *action execution* time, security issues concern database actions and external actions. Database actions are preformed locally by OES itself, which connects to the local DBMS and performs all the actions defined by the involved rule over locally stored data: the security level provided by OES is the same as the one provided by the DBMS. External actions, instead, require OES to reach executors external to OES itself. The same criteria as those for external applications apply.

RELATED WORK

Active Database Management Systems

The scenario of event management in active DBMSs is the most relevant one.

Samos (Dittrich et al., 2003) is a very complex active OODBMS, which provides several active functionalities, including event management similar to the one of OES. Samos runs coupled to the Object-Store passive OODBMS, only. OES, which is not an active DBMS but a pure event manager, can be mapped onto any active DBMS accepting the SQL language, and it provides suitable interfaces for most common DBMSs. Samos provides a very powerful event definition language, including relationships in event capturing (before..., after...), event composition (sequence..., conjunction...), and an execution model which accepts both attached and detached exception management. On the contrary, OES provides a very simple model featuring a numeric prioritization of rules and the only detached mode of execution.

Sentinel (Chakravarthy, 1997) was started as an OODBMS with event based rules capable of defining composite events by an extended set of operators. Later on, the authors (Chakravarthy and Liao, 2001) extended the system to include asynchronous events for a distributed cooperative environment, obtaining a server which is not connected to any particular DBMS, but runs as a message broker. With respect to Sentinel, OES adopts a more simplified event definition mechanism and language. OES can detect database modification events at the very database level, without requiring services from external event detectors, as required by Chakravarthy and Liao, 2001. According to OES, the event detection takes place only locally, even if in a distributed database environment, and the consequent action – if needed – may require communication with other sites of the distributed environment. Thus, in OES distributed events cannot be defined directly but need to be mapped as sets of local events and of local actions. Local actions may also include communications among the sites of the distributed environment.

EvE (Geppert,Tombros, and Dittrich, 1998) is an event engine implementing event-driven execution of distributed workflows. Similarly to OES, EvE adopts a registration, detection, and management mechanism, and it runs on a distributed, multi-server architecture. The main differences of OES, with respect to EvE, are that: a) OES does not use rules to schedule tasks according to a process model for the managed business process defined inside the WfMS; b) OES does not select executors (brokers in EvE's terminology) at runtime, choosing from a pool of resources since only one executor is defined for every action; c) OES does not require a WfMS environment as a core unit. In fact, OES can be run as a completely autonomous ECA server and the definition of events is not related to any WfMS. OES is extremely free, autonomous, can reference heterogeneous executors and allows one to define almost any type of event.

Framboise (Fritschi, Gatziu, and Dittrich, 1998) is a framework for the construction of active DBMSs inheriting the rule language of Samos. Framboise represents a database middleware, extending (Dittrich et al., 2003) to provide individual and customizable active services for any arbitrary passive DBMS. With respect to Framboise, OES aims at providing active services exploiting ECA rules over an existing active DBMS, capable of accepting standard SQL statements and the definition of triggers. While the language of OES is much simpler than Framboise's, OES does not necessarily require a DBMS, thus limiting itself to manage temporal and external events. On the other hand, if the application domain requires a DBMS, data events can be managed by OES provided that the DBMS supports active behaviors. OES can be more conveniently mapped on most commercial active DBMS, without requiring to recompile the kernel of the active DBMS itself neither requiring to modify existing applications.

Workflow Management Systems

Some WfMSs - e.g., Mentor (Wodtke, Weißenfels, Weikum, Kotz Dittrich, and Muth, 1997), Meteor (Krishnakumar and Sheth, 1995) - allow one to define a task to be executed whenever a specified exception is detected and the related event is raised. Pitfalls for this solution are that there is a wide separation between the normal evolution flow and the exception management flow, and that an exception can only start as a new activity. Additionally, the detection of the event must be formally performed whenever a task is terminated and before the next one is started: the detection cannot be performed while a task is running. In other systems - e.g., ObjectFlow (Hsu and Kleissner, 1996) - a human agent is formally dedicated to the detection of

asynchronous exceptions: after the event occurs, task execution is aborted and suitably defined execution paths are executed.

The use of OES coupled to a WfMS to manage asynchronous events overcomes some of these limitations. In fact, the detection of an event can take place even during the execution of a task, and not only after the completion of the task and before the successor is activated. Furthermore, the management of the exception can be completely automated, and may not require any human intervention to identify compensation paths.

Active Middleware Systems

Middleware technology aims at providing low- to medium-level services, which can be exploited by higher-level applications. In this area, Siena (Carzaniga, Rosenblum, and Wolf, 2001) is a wide area notification service, and it is mainly focused on scalability issues. With respect to OES, Siena can capture a reduced number of events, e.g. temporal events are not considered.

Amit (Adi and Etzion, 2004) is a "situation manager" which extends the concept of composite events. An event is a significant instantaneous atomic occurrence identified by the system; a situation requires the system to react to an event. The middleware aims at reducing the gap between events and situations. Amit comes with a situation definition language enabling one to capture events (immediate, delayed, deferred) and to detect situations. Applications are then notified when required situations occur.

CONCLUSIONS AND FUTURE WORK

In this paper, we described the autonomous, open ECA server OES and its active rule language, OpenChimera. OpenChimera supports the definition and the management of generic active rules following the Event-Condition-Action (ECA) paradigm, while the OES rule engine, derived from the FAR exception handler (Casati et al., 1999) of the FORO WfMS, supports the execution of OpenChimera rules.

OES comes with a standard set of events and actions. Events cover data manipulation events, temporal events, and events raised by external applications; the standard set of actions includes data manipulation actions. It is possible to customize the OES system to application- or domain-specific needs by adding new events and actions. OES can be coupled and customized with relatively little effort with any existing system that requires event and rule management solutions. The extended system allows designers to easily define application-specific active rules and to insulate active application requirements from the core application logic.

OES therefore fosters separation of concerns in the application development process (i.e., active and non-active requirements) and provides a robust solution to a cross-cutting implementation issue: active rule management. The nature of the OES rule engine minimizes the efforts required to integrate OES into other applications and further supports a flexible management of rules even after application deployment, i.e., during runtime. At design time, the built-in support for the detection of infinite loops represents a valuable tool to developers who typically have to deal with a multitude of rules and interdependencies.

Acknowledgments

We are grateful to Catia Garatti and Marco Riva for the implementation of the OES system, starting from FAR, and we thank prof. Stefano Ceri of Politecnico di Milano, Italy, prof. Stefano Paraboschi of the University of Bergamo, Italy, and prof. Fabio Casati of the University of Trento, Italy, for fruitful discussions and suggestions.

REFERENCES

Adi, A., & Etzion, O. (2004). Amit - the situation manager. *VLDB Journal*, 13(2), 177-203.

Amghar, Y., Meziane, M., & Flory, A. (2002). Using business rules within a design process of active databases. In S. Becker (Ed.), *Data Warehousing and Web Engineering* (pp. 161-184), Hershey, PA: IRM Press.

Carzaniga, A., Rosenblum, D. S., & Wolf, A. L. (2001). Design and evaluation of a wide-Area event notification service. *ACM Transactions on Computer Systems*, 19(3), 332-383.

Casati, F., Ceri, S., Paraboschi, S., & Pozzi, G. (1999). Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3), 405-451.

Ceri, S., & Fraternali, P. (1997). *Designing database applications with objects and rules: the IDEA methodology*. Reading, MA: Addison-Wesley.

Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., & Matera, M. (2002). *Designing Data-Intensive Web Applications*. San Francisco, CA: Morgan Kauffmann.

Chakravarthy, S. (1997). Sentinel: An object-oriented DBMS with event-based rules. In J. Peckham (Ed.), *SIGMOD Conference* (pp. 572-575). New York: ACM Press.

Chakravarthy, S., & Liao, H. (2001). Asynchronous monitoring of events for distributed cooperative environments. In H. Lu, & S. Spaccapietra (Eds.), *Proceedings of CODAS'01* (pp. 25-32). Beijing: IEEE Computer Society.

Charfi, A., & Mezini, M. (2004). Hybrid Web service composition: business processes meet business rules. In M. Aiello, M. Aoyama, F. Curbera, & M. P. Papazoglou (Eds.), *Proceedings of ICSOC'04* (pp. 30-38). New York: ACM Press.

Combi, C., Daniel, F., & Pozzi, G. (2006). A portable approach to exception handling in workflow management systems. In R. Meersman & Z. Tari (Eds.), *OTM Conferences (1)*, LNCS 4275 (pp. 201-218). Montpellier, France: Springer Verlag.

Combi, C., & Pozzi, G. (2003). Temporal conceptual modelling of workflows. In I. Song, S. W. Liddle, T. Wang Ling, & P. Scheuermann (Eds.), *Proceedings of ER'03*, LNCS 2813 (pp. 59-76). Chicago: Springer Verlag.

Combi, C., & Pozzi, G. (2004). Architectures for a temporal workflow management system. In H. Haddad, A. Omicini, R. L. Wainwright, & L. M. Liebrock (Eds.), *Proceedings of SAC'04* (pp. 659-666). New York: ACM Press.

Cugola, G., Di Nitto, E., & Fuggetta, A. (2001). The JEDI event-based infrastructure and its application to the development of the OPSS wfMS. *IEEE Transactions on Software Engineering*, 27(9), 827-850.

Daniel, F., Matera, M., & Pozzi, G. (2006). Combining conceptual modeling and active rules for the design of adaptive web applications. In N. Koch & L. Olsina (Eds.), *ICWE'06 Workshop proceedings* (article no.10). New York: ACM Press.

Daniel, F., Matera, & Pozzi, G. (2008). Managing runtime adaptivity through active rules: the Bellerofonte framework, *Journal of Web Engineering*, in press.

Dittrich, K. R., Fritschi, H., Gatziu, S., Geppert, A., & Vaduva, A. (2003). Samos in hindsight: experiences in building an active object-oriented DBMS. *Information Systems Journal*, 28(5), 369-392.

Eder, J., & Liebhart, W. (1995). The workflow activity model WAMO. In S. Laufmann, S. Spaccapietra, & T. Yokoi (Eds.), *Proceedings of CoopIS'95* (pp. 87-98). Vienna, Austria.

Fritschi, H., Gatziu, S., & Dittrich, K. R. (1998). Framboise - an Approach to framework-based active database management system construction. In G. Gardarin, J. C. French, N. Pissinou, K. Makki, & L. Bouganim (Eds.), *Proceedings of CIKM '98* (pp. 364-370). New York: ACM Press.

Gatziu, S., Koschel, A., von Bultzingsloewen, G., & Fritschi, H. (1998). Unbundling active functionality. *SIGMOD Record*, 27(1), 35-40.

Geppert, A., Tombros, D., & Dittrich, K. R. (1998). Defining the semantics of reactive components in event-driven workflow execution with event histories. *Information Systems Journal*, 23(3-4), 235-252.

Hsu, M., & Kleissner, C. (1996). Objectflow: towards a process management infrastructure. *Distributed and Parallel Databases*, 4(2), 169-194.

Huang, S., Hung, S., Yen, D., Li, S., & Wu, C. (2006). Enterprise application system reengineering: a business component approach, *Journal of Database Management*, 17(3), 66-91.

Krishnakumar, N., & Sheth, A. P. (1995). Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2), 155-186.

Li, S.H., Huang, S.M., Yen D.C., & Chang, C.C. (2007). Migrating legacy information systems to web services architecture. *Journal of Database Management*, 18(4), 1-25.

Loucopoulos, P., & Kadir, W.M.N.W. (2008). BROOD: Business rules-driven object oriented design. *Journal of Database Management Systems*, 19(1), 41-73.

Minsky, N. H. (2003). On conditions for self-healing in distributed software systems. *Proceedings of AMS'03* (pp. 86-92). Los Alamitos, CA: IEEE Computer Society.

Mok, A. K., Konana, P., Liu, G., Lee, C., & Woo, H. (2004). Specifying timing constraints and composite events: an application in the design of electronic brokerages. *IEEE Transactions on Software Engineering*, 30(12), 841-858.

Wyse, J. E. (2006). Location-aware query resolution for location-based mobile commerce: performance evaluation and optimization. *Journal of Database Management*, 17(3), 41-65

Wodtke, D., Weißenfels, J., Weikum, G., Kotz Dittrich, A., & Muth, P. (1997). The Mentor workbench for enterprise-wide workflow management. In J. Peckham (Ed.), *Proceedings of SIGMOD'97* (pp. 576-579). New York: ACM Press.