Specification and Implementation of Exceptions in Workflow Management Systems

FABIO CASATI
and
STEFANO CERI, STEFANO PARABOSCHI, and GIUSEPPE POZZI
Politecnico di Milano

Although workflow management systems are most applicable when an organization follows standard business processes and routines, any of these processes faces the need for handling exceptions, i.e., asynchronous and anomalous situations that fall outside the normal control flow.

In this paper we concentrate upon anomalous situations that, although unusual, are part of the semantics of workflow applications, and should be specified and monitored coherently; in most real-life applications, such exceptions affect a significant fraction of workflow cases. However, very few workflow management systems are integrated with a highly expressive language for specifying this kind of exception and with a system component capable of handling it.

We present Chimera-Exc, a language for the specification of exceptions for workflows based on detached active rules, and then describe the architecture of a system, called FAR, that implements Chimera-Exc and integrates it with a commercial workflow management system and database server. We discuss the main issues that were solved by our implementation, and report on the performance of FAR. We also discuss design criteria for exceptions in light of the formal properties of their execution. Finally, we focus on the portability of FAR and on its unbundling to a generic architecture with detached active rules.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems; *Rule-based databases*

General Terms: Design, Languages, Management, Performance

Additional Key Words and Phrases: Active rules, asynchronous events, exceptions, workflow management systems

The research in this paper was sponsored by the WIDE Esprit project. 20280, by CNR-CESTIA, and by the Hewlett-Packard Internet Philanthropic Initiative.
Authors' addresses: F. Casati; S. Ceri, S. Paraboschi, and G. Pozzi, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza L. Da Vinci, 32, Milano, I-20133, Italy; email: ceri@elet.polimi.it; parabosc@elet.polimi.it; pozzi@elet.polimi.it.

1. INTRODUCTION

Workflow management systems (WfMSs) are software systems for supporting coordination and cooperation among members of an organization, helping them to perform complex business processes [Georgakopoulos et al. 1995]. Workflow systems represent business processes by means of elementary activities and connections among them; activities represent either fully automated tasks executed by computers, or tasks assigned to human actors executed with the support of a computer. WfMSs control the evolution of processes by initiating and assisting their different activities and by checking that they are correctly performed by all the actors.

An important feature of workflow systems is the ability to represent exceptions that alter the normal flow of processes [Dayal et al. 1990]. Among exceptions, a class which is gaining recognition and importance is that of *expected exceptions* [Eder and Liebhart 1995], i.e., of those anomalous situations that are known in advance to the workflow designer. When an exception is unexpected, the exception handler typically resorts to halting the process and invoking a human intervention. Instead, when exceptions are expected, the exception handler can rely on the semantics of the workflow application in order to handle the exception, typically by means of some form of reactive processing. For instance, in a car rental workflow, an accident to a rented car causes an exception to the regular rental process. The accident, although expected, is an unlikely event; once it has occurred, however, a variety of activities become needed, including, e.g., giving assistance to the renters, scheduling the car's repair, and rescheduling the future rentals for the affected car. All such activities constitute the (planned) *reactions* to raising the exception.

Workflows and expected exceptions capture different aspects of the application semantics, as summarized in Table I. Workflows describe the "normal behavior" of a process, while expected exceptions model the "occasional behavior." Expected exceptions are unpredictable, and therefore cannot be conveniently represented in the process in the form of special tasks and connections among tasks. They are not frequent, but once they occur they may require special treatment, which may lead to the execution of a completely different process. They are asynchronous (hence, initiated at an arbitrary stage of the process) and highly influenced by external factors. Their execution may cause the backtracking of previous steps in the process or even sudden termination.

There is growing interest and need for languages and systems to integrate workflow specifications with expected exceptions; commercial workflow systems typically support only a selected number of them, without enough generality. In this paper we present a comprehensive approach to the management of expected exceptions; we define a new language for expressing expected exceptions, and then describe the features of a system for integrating the exception handler with the workflow manager. For brevity, we use the term "exception" to denote asynchronous expected

exceptions (a classification of the various kinds of exceptions can be found in Section 7).

The exception-handling mechanism must be able to *capture* exceptional events and to *react* to them. Each reaction must first assess the state of the process and then, in a few cases, adopt the corrective action; in many cases events correspond to false alarms and do not need to be followed by a corrective action. This model has a strong similarity to the trigger management strategy used in active databases [Ceri and Widom 1990; Widom and Ceri 1996]. In fact, since most workflow systems execute on top of commercial databases, it is quite natural to use active database functionality to manage exceptions.

Active rules are characterized by the following components, each with an immediate correspondence to exceptions:

—The *event part* defines the symptoms of an exception, e.g., database modifications or signals coming from other components of the workflow, that *trigger* the rule, i.e., put it in the set of rules to be considered by the rule management system.

—The *condition* is a boolean predicate that checks that the symptoms really identify an exception to be managed; it can also be used to select, among several exception management alternatives, the most adequate to deal with the current workflow state.

—The *action* describes the updates and procedures that must be invoked to respond to the exception occurrence.

Each rule is executed in a new transactional context, different from the one in which the exceptional event was generated; in terms of active databases, rules have a *detached* execution mode [Dayal et al. 1990]. In most cases, rules do not need immediate service and should interfere as little as possible with regular workflow processing. Thus, we have engineered an exception handler in which triggered rules are batched and considered at given periods of time. Very few events are classified as “real time” and cause an immediate invocation of the rule management system.

In this paper we present the design and implementation of a rule-based exception handler for workflow management. Exceptions are specified in Chimera-Exc, a new language specifically designed for expressing exceptions in a WfMS; Chimera-Exc is an extension of Chimera [Ceri et al. 1996] for the conceptual specification of database applications incorporating object orientation, deductive rules, and active rules. We describe the innovative features of Chimera-Exc and the corresponding expressive power in modeling exceptions; then we describe the implementation of the language and its interfaces for a commercial WfMS and DBMS. Next, we discuss the formal properties of workflow applications augmented with exceptions, and indicate criteria for a sound use of exception handlers in workflow management. Finally, we focus on the portability of FAR and on its unbundling to a generic architecture with detached active rules.

Table I. Comparing Flows and Exceptions

	Workflow Process	Exception
Essence	Captures the essential behavior of the process (occurring more than 90 percent of the time)	Captures the occasional behavior of the process (occurring less than 10 percent of the time)
Time of specification	Specified first	Specified last
Spread	It is learnt by all agents	It is learnt by a few specialized agents
Initiation	Starts at a given stage of the process	Starts at an arbitrary stage of the process
Termination	Once concluded, causes a progression towards the end of the process	Once concluded, may cause the iteration of previous steps or the sudden termination of the process
Scope	Behavior depends essentially on local variables	Behavior depends essentially on external factors
Expressive power	Action does not use WF control primitives	Action uses WF control primitives

(This research is part of the WIDE project, funded by the Esprit IV program of the European Union, with the goal of developing an advanced commercial workflow management system, called FORO. The team at Politecnico di Milano designed and implemented FAR, the FORO Active Rule component for exception management in the FORO environment.)

2. PRELIMINARIES

This section introduces basic workflow concepts and terminology, based on the model and definitions of the Workflow Management Coalition (WfMC) [Workflow Management Coalition 1996; 1998]. FORO provides several extension to that model, but such extensions are not relevant in the context of this paper; for a complete description, see Grefen et al. [1999]. This section then presents a case study used throughout the paper to exemplify the new concepts. Finally, we show the need for exceptions in the case study, drawing requirements on the exception-specification language.

2.1 Basic Workflow Concepts

A *workflow process definition* (or simply *workflow schema*) is the formal representation of a business process. A workflow schema is composed of subprocesses and elementary activities (*tasks*) that collectively achieve the business goal. Activities are organized into a directed graph (the *flow structure*) that defines the order of execution among the activities in the process. Arcs (transitions) in the graph may be labeled with transition predicates defined over process data, meaning that as an activity is completed, tasks connected to outgoing arcs are executed only if the

corresponding transition predicate evaluates to true. A *process instance* (or simply *case*) is an enactment of a workflow schema. A schema may be instantiated several times, and several instances may be running concurrently. WfMSs support case execution by scheduling tasks (as defined by the flow structure) and by assigning them for execution to human or automated agents.

A process may create and access several types of data. The WfMC identifies three types of workflow data: *workflow relevant data* includes typed data created and used by a process instance; this data can be made available to subprocesses and activities, and accessed by the WfMS in order to evaluate transition predicates. *Application data* is domain-specific, must be processed with external tools and generally cannot be accessed by the WfMS, although the system may control and restrict accesses to them. *System and environmental data* is maintained by the local system environment or by the WfMS itself, and can also be used to evaluate transition predicates. Activities are typically executed atomically, and data modifications are made visible as the task is completed.

A critical issue in workflow management is the assignment of tasks and cases to the appropriate *agent* (also called *workflow participant*), in order to execute activities or supervise their execution. The approach adopted by most WfMSs consists in allowing the definition of an *organization schema* that describes the structure of the organization relevant to workflow management. In the organization schema, agents are grouped in several ways, for instance, according to their skills or to the organizational unit they belong to. In the definition of the workflow schema, processes and activities are (statically) bound to elements of the organization schema (e.g, to *roles* or *organizational units*) rather than to individual agents. This approach decouples the definition of the process from the definition of the organization, and provides more flexibility, since changes in the organization schema may not affect process definitions.

At runtime, as a task is scheduled for execution, the WfMS determines all the agents allowed to execute it and inserts the task into their *worklists*. As an agent *pulls* the task from his/her worklist in order to start working on it, the task is removed from the worklists of the other agents. Once a task execution is completed, the WfMS schedules the next task(s) to be activated.

2.2 The Car Rental Case Study

The process model (or *workflow schema*) of the car rental example is shown in Figure 1. A new process instance (or *case*) is started as the car rental company receives a car reservation request. The first activity (or *task*) collects the customer data along with the desired car size and rental day (*getRentalData*), updating the corresponding workflow relevant data. The next activity *chooseCar* is then started, to find a car of the requested size available on the requested day; the execution of the task may involve dealing with the customer in order to adapt the request as a function of car

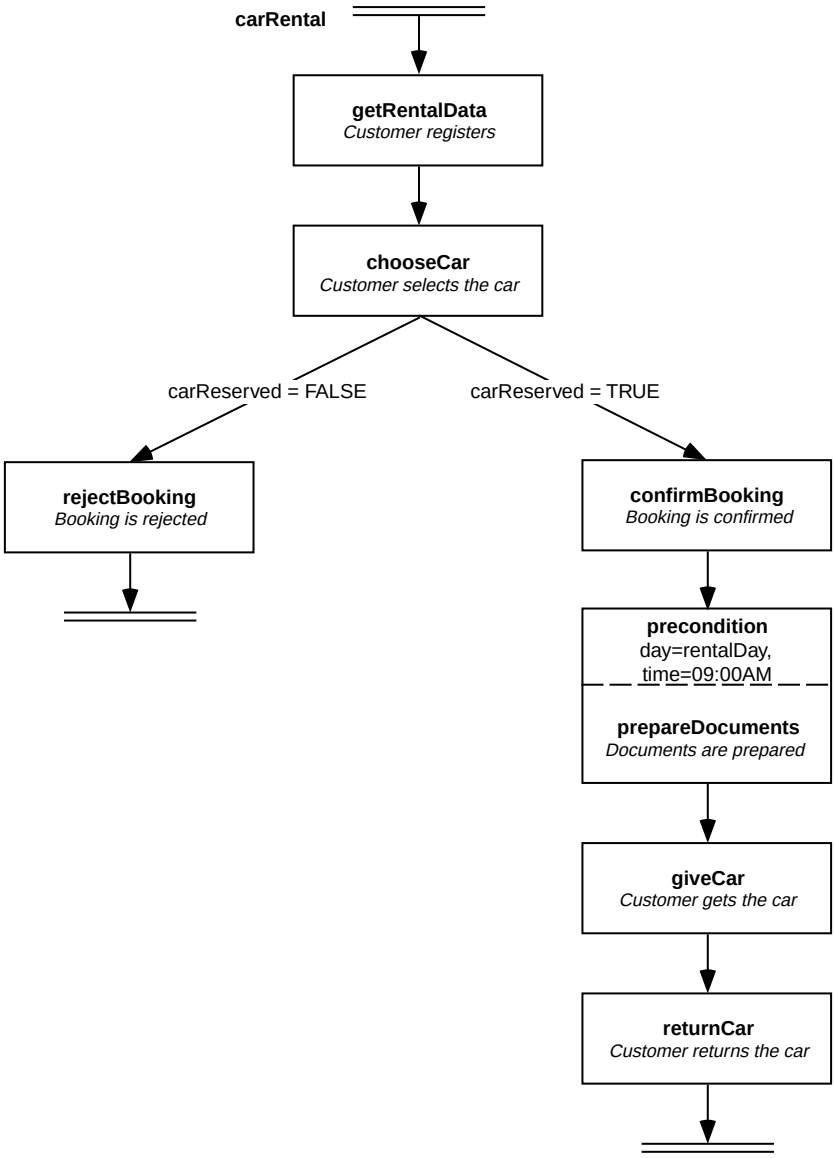

Fig. 1. The *carRental* process model. The dashed line for the *prepareDocuments* activity delimits the preconditions from the body of the activity.

availability. If, after termination of the task, no suitable car is found (variable *carReserved* is *false*), then the task *rejectBooking* is activated and causes the end of the case: otherwise, task *confirmBooking* is started, causing the confirmation to the customer that the reservation is accepted.

After the confirmation, the precondition of the task *prepareDocuments*—delimited by a dashed line inside the task rectangle in Figure 1—waits until the morning of the rental day, when the documents of rentals for the day are prepared by the agents at the local rental site: The precondition is

an expression that must be satisfied before the activity can start. The subsequent task (*giveCar*) is pulled by an agent when the customer shows up. The final task (*returnCar*) is pulled by an agent when the customer returns the car.

2.3 Need for Exceptions

The car rental schema models the typical car rental process; the flow structure corresponds to the sequence of operations that must be performed in order to achieve the business goal, i.e., renting the car and satisfying the customer. This process is very simple, but in reality business processes can be quite complex, and the corresponding workflow schemas can be very intricate. However, it frequently happens that a particular execution of a process needs to deviate from the normal behavior defined by the flow structure. The typical exceptions to the car rental process can be summarized as:

—Changes to shared variables stored in the database; for example, a delay in returning a car as perceived by another case where that same car is booked to a different customer.

—Violation of time-related constraints or alarms, such as deadlines for the execution of tasks and cases; for example, the no-show of a client two hours after the agreed reservation time.

—Asynchronous external events, notified to the WfMS by a human or automated agent; for example, the cancellation of a car reservation or the notification of a car accident.

—Violation of constraints over workflow data; for example, booking the same car to two different customers during overlapping time intervals.

—Changes in the process organization; for example, the sudden unavailability of agents at the local rental company.

Such situations cannot be efficiently modeled and handled within the flow structure, since they are asynchronous and their occurrence is not related to the completion of other tasks in the case. For instance, consider an exception related to a late car return, causing the rescheduling of the rentals for that car: the rescheduling is an intrinsically asynchronous event, but in order to take it into account it has to be checked at all stages of the flow following car selection and prior to giving the car to the customer. Figure 2 shows the impact of such a modification upon the workflow schema in Figure 1; clearly, the resulting schema is more complex and less effective in conveying the semantics of the application.

However, such exceptions need to be considered by the WfMS, because their handling is part of the semantics of the process. Therefore, ad-hoc constructs, extending those introduced in the previous sections, must be added to the workflow model. These constructs should enable the specifica-

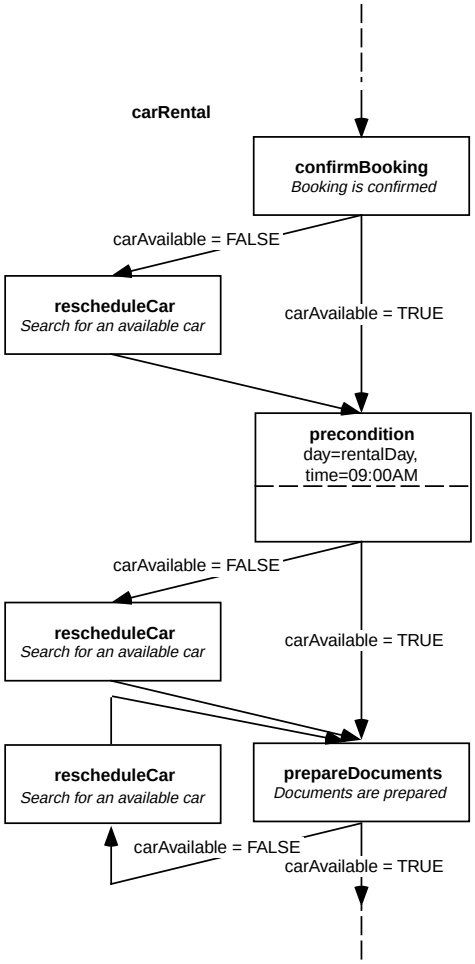

Fig. 2. Changes to a part of the *carRental* process model in absence of an exception management system.

tion of both the conditions to be monitored by the WfMS and of the sequence of operations to be performed in order to handle them.

3. THE EXCEPTION LANGUAGE: CHIMERA-EXC

The exceptions need to query the state of the workflow cases, which is stored within a database shared by the workflow interpreter and the exception handler itself. Therefore, prior to introducing the exception language, we describe the schema definition language for managing state information about workflows.

3.1 Class Definitions for Chimera-Exc

Chimera-Exc rules execute upon a simple, object-oriented schema, consisting of object classes. Chimera-Exc does not need structural and behavioral

complexity, which is typical of object-oriented databases (and can be found in Chimera). So an object class is just a record of attributes; each attribute has an atomic type, chosen among the classical types supported by database systems (e.g., character, string, integer, date). The names of all classes within the database and of all attributes within classes must be different. Each class is simply a collection of typed objects; all objects have a distinct object identifier.[1]

Classes in Chimera-Exc are of three different kinds:

—Workflow management classes store metainformation about workflows and their enactment; they are defined within the FORO WfMS.

—Exception management classes store metainformation about exceptions and their management; they are defined in the FAR exception handler.

—Workflow-specific classes store the values of variables defined within workflow schemas. All variables relative to a given workflow schema are collected within one object class, whose name derives from the workflow name. For example, a particular car reservation may be represented by an object *C* of *carRental* class, and the expression *C.reservationNumber* denotes the string variable containing the reservation number.

Note that workflow and exception management classes are workflow-independent, hence their structure is defined once and for all in the system;[2] their content changes while flows are executed or exceptions are managed. Workflow-specific classes are instead defined when a given workflow is presented to the WfMS; each case is associated to an object generated when the case is created with a unique case identifier (*caseId*).

Workflow-specific classes use a restricted form of inheritance: they all inherit from the generic, workflow-independent class *case*. Thus, a workflow-specific object, e.g., belonging to the class *carRental*, inherits the attributes of the class *case*, e.g., the case *responsible*; having a *responsible* is a generic property that holds for all workflows.

3.2 Events, Conditions, Actions, and Priorities in Chimera-Exc

This section describes informally the characteristics of events, conditions, actions, and priorities in Chimera-Exc. The full syntax of the language is given in the Appendix, which appears in electronic format only; see http://www.acm.org/pubs/contents/journals/tods/1999-24/.

3.2.1 *Events*. Each rule in Chimera-Exc can monitor multiple events. Events in Chimera-Exc belong to four categories: data manipulation events, external events, temporal events, and workflow events. Data manipulation events are present in Chimera, but external, temporal, and workflow

[1]Given this simple schema, classes of Chimera-Exc can be stored within relational tables; each tuple is provided with a system-generated object identifier.

[2]In the context of the WIDE project, workflow classes are defined and managed within FORO; the exception handler has only retrieval privileges regarding workflow classes.

events are defined specifically for Chimera-Exc. These categories are detailed in the remainder of the section.

Data manipulation events enable the monitoring of operations that change the content of the database; these events include instantiating an object via the *create* statement, removing an object via the *delete* statement, or changing the value of one of the object variables via the *modify* statement.[3] For instance, the event *modify(carRental.returnTime)* is raised as the return time of a car is modified, where *carRental* is the name of a workflow and *returnTime* is the name of one of the variables for that workflow.

External events are raised by external applications interacting with the exception handler. External events must first be *registered* by applications; all event names must be different. Then events can be *raised* by external applications. For instance, the event *carAccident* can be raised by the application that responds to calls from the renters and provides them with assistance. With the *raise* primitive, external applications can provide several application-specific parameters (e.g., the license number of the car and the place and time of the accident). In Chimera-Exc, *raise* is also the event language construct that defines the external events to which a rule is made sensible; thus, event *raise(carAccident)* is raised as an external application notifies an occurrence of the *carAccident* event.

Workflow events enable the monitoring of starts and completions of tasks and cases. The progression of a given case is marked by the times of initiation and termination of the case itself and of its tasks. These times are expressed by means of the predefined events *caseStart*, *caseEnd*, *taskStart*, and *taskEnd*, denoting the *DateTime* in which the case or a given task is started or completed. The workflow executor notifies the exception handler of these events.

Temporal events are classified as instant, periodic, and interval events. Informally, an instant event occurs when a certain temporal instant is reached (e.g., at midnight of the 31st of December 1999); an interval event occurs when a given temporal interval elapsed from a given time (e.g., two hours since the start of a case); and a periodic event occurs when periodic time conditions are fulfilled (e.g., every first Monday of the month). We next characterize each class of events more precisely.

—*Instant events* are expressed as *DateTime* constants of SQL-92 [Cannan and Otten 1992]; partial specifications, e.g., of the date or the time, have a default completion. An instant event can occur only once; if it has already occurred at the time when the rule is compiled, it is disregarded.

—*Periodic events* are expressed using a notation taken from Leban et al. [1986]. A period is defined by means of two components, separated by the *during* keyword; the latter component indicates the time period and the former component indicates how many units of time should be spent

[3]The obvious relational counterparts of these operations are insertions, deletions, and updates to specific attributes.

waiting since the start of the period in order for the event to occur. For instance, *1/days during weeks* denotes the period of time defined by the first day of each week; *18/hours during days* is a periodic event ocurring at 6:00 pm every day; *any/hours during day* denotes a periodic event at every hour. We also enable comparison expressions for limiting periodic events within a time window, e.g., *date '12/25/1997' < 1/days during weeks < date '12/25/1998'*; this expression denotes every Sunday between the days of Christmas in 1997 and 1998.[4]

—*Interval events* are defined by means of two components, a duration and a *DateTime*, called the anchor time; the event occurs when the specified duration has elapsed since the anchor time. For ease of reference, interval events are named in the condition. Syntactically, the duration is preceded by the keyword *elapsed* and followed by the keyword *since*, which precedes the anchor time. For instance, *elapsed (interval 60 days) since date '1/1/1998'*, or *elapsed (interval 60 days) since caseStart* are simple examples of interval events.

Legal anchor times are *DateTime* constants, the raising of external events, or the start or completion of tasks and cases; when an anchor time is not defined (because it refers to an external event that is not raised or to a workflow event that has not occurred), the interval event is also not defined. Durations are expressed in the notation used in SQL-92 [Cannan and Otten 1992]; partial specifications have a default completion. Predefined constants, such as the maximum or expected duration of a case or task, can be used. Arithmetical operations on time values are permitted, such as the addition or multiplication of durations with constants.

3.2.2 *Conditions*. The condition part enables us to verify that rule triggering really corresponds to an exception that needs to be processed: false conditions denote false alarms, while true conditions denote an exceptional situation that must be handled. In Chimera-Exc, conditions and actions normally share some variables; when the evaluation of the condition produces bindings for these variables, the condition is satisfied, thus identifying the objects of the database that are affected by an exception. When no bindings are produced, the condition is not satisfied and the action is not executed. Conditions in Chimera-Exc derive their syntax and semantics from Chimera, which in turn derives them from the logical language Datalog, adapted to an object-oriented style. Thus, conditions in Chimera-Exc have a declarative nature, are set-oriented, and use a logic-programming style.

[4]The notation presented in Leban et al. [1986] allows us to specify complex periodic events, such as the election day in the US (the first Tuesday after the first Monday of November), which can be expressed as *1/(3/days during weeks) > (1/(2/days during weeks) during 11/months)*; in Chimera-Exc, comparison expressions are only used between periodic events and given *DateTime* constants representing the extremes of a time interval.

Syntactically, a condition is a conjunction of atomic formulas [Ceri et al. 1990]. Class formulas introduce variables ranging on the current extent of an object class (e.g., *carRental(C)*); type formulas introduce variables of a given type (e.g., *integer(I)*); comparison formulas are built by means of comparisons between expressions, which in turn are built by means of constants, attribute terms, or aggregate terms, composed in the usual way. Attribute terms use the classical object-oriented notation in which the attribute name acts as a selector of an object variable (e.g., *C.bookedCarPlate*) while aggregate terms are introduced by aggregate functions applied to attribute terms (e.g., *avg(C.carMaxSpeed)*), again with a classical object-oriented notation. The following condition binds objects of class *carRental* whose customer is Italian and whose rental day is on the first of January 1998:

carRental(C), C.customer.nationality = "Italy", C.rentalDay = 1/1/1998

The condition is evaluated at a time subsequent to the time of the triggering, and there is no guarantee that the database state has not been changed meanwhile. The metapredicate *old* enables the evaluation of attribute terms in the state prior to the operation that causes the triggering. Use of the *old* predicate requires logging such database states into special object classes. The following condition binds objects of class *carRental* such that the boolean variable *carReserved* was true before the triggering and is false at the time of rule evaluation:

carRental(C), old(C.carReserved) = true, C.carReserved = false

The predicate *occurred*, followed by an event specification, binds a variable defined on a given class to the objects of that class that are affected by the event. This predicate can be used in many different contexts, shown below:

(1) *agent(A), occurred(create(agent), A)*

(2) *agent(A), occurred(modify(agent.name), A)*

(3) *agent(A), occurred(delete(agent), A)*

(4) *externalEvent(E), occurred(raise("carAccident"),E)*

(5) *temporalEvent(T), occurred(lateCarReturn,T)*

(6) *task(T), occurred(taskStart("assignCar"),T)*

(7) *case(C), occurred(caseEnd,C)*

In Examples 1, 2, and 3, *A* is a workflow-specific object *agent* that has been created, modified, or deleted. In Examples 4 and 5, *E* and *T* are objects of exception-management classes *externalEvent* or *temporalEvent*, respectively, whose events, the raising of the exception *carAccident* or the temporal event *lateCarReturn*, have occurred. Finally, in Examples 6 and 7, *T* and *C* are objects of workflow-independent classes *task* and *case*, which are bound by workflow events.

3.2.3 *Actions*. The action part of a rule consists of one or more primitives executed in sequence. Unlike conditions, Chimera-Exc actions are instance-oriented: a different primitive call is issued for every binding produced by the condition. Primitives are divided into the following main categories:

—data modification primitives for creating objects (*create* primitive), modifying the value of an object's attributes (*modify* primitive), or deleting objects (*delete* primitive). These primitives are applied to the set of objects that are bound by conditions; their semantics is the same as in Chimera [Ceri et al. 1996];

—workflow management primitives for the notification of alarms to agents, starting new tasks, cases, or subprocesses, reassigning tasks to a different agent, rejecting or canceling tasks, or the global rollback of cases. Alarms may refer to the values of the variables that are bound by the condition, typically manipulated as character strings and inserted within notification messages. Several examples of these actions are given in the next sections, and their concrete syntax is given in the Appendix.

Active rules in Chimera-Exc must be *safe*, i.e., the input parameters of the action part must appear in some positive literal (i.e., a term outside the scope of the *not* operator) of the condition part.

Note that the expressive power of Chimera-Exc also enables the implementation of the basic workflow functionality (i.e., task scheduling and assignment) by means of Chimera-Exc rules. Indeed, we have designed and implemented a prototype workflow engine by exploiting active rules [Casati et al. 1996; Pernici and Sanchez 1996]. However, after assessing the performance of the prototype implementations, we decided to implement the workflow engine in WIDE by means of an interpreter of a workflow definition language, generated by the workflow compiler from the graphical description of the schema.

3.2.4 *Priorities*. Rules have a statically defined priority, specified as an integer (positive or negative), which is used to determine the order of execution within a set of triggered rules. Default priority is set to zero; the rule with the highest number has the highest priority. When two rules have the same static priority, their relative priority is determined by their creation time, with oldest rules having highest priority.

3.3 Examples

We show Chimera-Exc at work by means of some of the exceptions informally introduced in Section 2.3. Trigger *lateCarReturn* notifies the clerk responsible for a car rental that the car's return is delayed.

```
define trigger lateCarReturn
  events     modify(carRental.returnTime)
  condition  carRental(C), occurred (modify(carRental.returnTime),C),
             C.returnTime > old(C.returnTime)
  actions    notify (C.responsible, "Late return for car "
             + C.bookedCarPlate)
  order -2
end;
```

The following version of the same trigger checks whether the car that is returned late is booked by some other customers at a time preceding the new expected return time of the car; and, if so, the notification goes to the agents who are responsible for these car rentals.

```
define trigger lateCarReturn
   events      modify(carRental.returnTime)
   condition   carRental(C1), occurred (modify(carRental.returnTime),C1),
               carRental(C2), C1.bookedCarPlate = C2.bookedCarPlate,
               C1.returnTime > C2.rentalTime
   actions     notify (C2.responsible, "Need of rescheduling car "
               + C2.bookedCarPlate)
   order 3
end;
```

The trigger *customerCancel* reacts after cancelling a reservation by notifying the responsible agent and by starting the task *rejectBooking*, which belongs to the workflow schema (see Figure 1). The external event notification includes as a first parameter the reservation number of the affected car rental.

```
define trigger customerCancel
   events      raise("customerCancel")
   condition   carRental(C), externalEvent(E), occurred(raise
               ("customerCancel"), E),
               C.reservationNumber = E.parameter1
   actions     notify (C.responsible, "Customer cancelled reservation: "
               + E.parameter1),
               startTask(C, rejectBooking)
   order -3
end;
```

The realtime trigger *carAccident* notifies the agent responsible for the rental of the damaged car and all the agents for future reservations of that car of such an external event. The event also starts the execution of a new case of another workflow, called *Accident*, which receives as a parameter the plate number of the damaged car and manages its repair.

```
define trigger carAccident
   events      realtime raise("carAccident")
   condition   externalEvent(E), carRental(C1), occurred(raise
               ("carAccident"), E),
               E.parameter1 = C1.reservationNumber,
               carRental(C2), C1.bookedCarPlate = C2.bookedCarPlate
   actions     notify (C1.responsible, "Accident for car "
               + C1.bookedCarPlate),
               notify (C2.responsible, "Need of rescheduling car "
               + C2.bookedCarPlate),
               startCase(Accident, C1.bookedCarPlate)
   order 5
end
```

The trigger *noShow* is activated every hour, and selects those car rental reservations where the customer has not shown up two hours after the reservation time; the customer's name and driver's license number are

recorded. The condition is formulated in such a way that the recording is done only once.

```
define trigger noShow
  events      any/hours during days
  condition   carRental(C), C.rentalTime > sysTime - 3:00:00,
              C.rentalTime <= sysTime - 2:00:00, C.carRented = false
  actions     notify (C.responsible, "No show for " + C.reservationNumber),
              create(noShowRecord,
              [C.customer.name, C.customer.drivingLicence,sysDate])
  order 2
end;
```

The trigger *stopWork* is activated every day at 6:00 pm, and notifies every task executor of the *carRental* workflow that work-time is over.[5]

```
define trigger stopWork
  events      18/hours during days
  condition   carRental(C), task(T), T.caseId=C, T.status="running"
              agent(A), T.executor=A
              actions    notify (A, "Time to go home!")
  order 100
end;
```

Note that each agent receives at most one notification message due to the execution of the *stopWork* trigger. In fact, one notification action is executed for each agent bound to object variable A after the condition evaluation. Replacing the action with the primitive `notify (T.executor, "Time to go home!")` yields slightly different semantics, where a notification message is sent for every active task. So agents executing several tasks will receive several notifications.

The trigger *slowCaseEnd* is activated two hours after the start of the *returnCar* task, and checks whether the case is still running. If so, it notifies the case responsible as to who must use a form in order to enter the final data about the rental and creates a record reporting the delay.[6]

```
define trigger slowCaseEnd
  events      slowCaseEnd: elapsed (interval 2 hour)
              since taskStart(returnCar)
  condition   temporalEvent(TE), occurred(slowCaseEnd,TE),
              carRental(C), TE.dependsOnCase=C, C.status="running"
  actions     notify(C.responsible,"Enter final data about "
              + C.reservationNumber),
              create(slowCaseRecord,
              [C.responsible,C.reservationNumber,sysTime])
  order 1
end;
```

3.4 Rule Execution Semantics

Rule execution semantics is defined operationally by explaining the mechanisms of rule scheduling. After their creation, rules are stored within a

[5]Attribute *caseId* of *Task* links each task activation to its case.
[6]Attribute *dependsOnCase* of *temporalEvent* links a temporal event to the case that caused the event occurrence.

rule repository, defined next. Each rule is initially *untriggered*, and becomes *triggered* when any of its events occur.

The exception handler has a component called *Scheduler* that is activated either periodically or when special events, tagged as *realtime*, are raised. When the Scheduler is activated, it first determines which rules are triggered; these rules are entered into the set of *ready rules* and the state of these rules is set to *untriggered*. Building the ready rules set and changing the state of ready rules to untriggered occurs atomically. Then, the Scheduler orders ready rules according to a statically defined priority, and starts their execution according to the ordering. Rules can be executed in parallel, so rules for different priorities can execute concurrently. When a given rule is started, it is deleted from the set of ready rules; its condition is evaluated, and if the condition is satisfied then its action is executed.

When the period expires or is interrupted by a realtime rule, there may be ready rules that have not been executed; these rules carry on to the next period, where they are executed according to their relative priority within the set of all ready rules. Thus, a rule with low priority could, in principle, carry on from one period to the next one, without ever being executed. However, we make it a requirement that exceptions should normally be considered within the period that immediately follows its triggering; in Section 5.3 we discuss how such a result can be accomplished by suitable parameter settings.

After the triggering of a realtime rule, the Scheduler immediately recomputes the set of ready rules (which now includes the realtime one) and orders them according to their priority, then executes the rules of the new ready set in priority order. In this way, if a realtime rule is given high priority, it has good chances of being scheduled for execution quite rapidly after its triggering. However, rules scheduled prior to the triggering of the realtime rule, and which could still be executing at the end of the ordering operation, are not preempted.

During the evaluation of the condition *occurred*, predicates bind suitably typed variables to the events that have caused rule triggering. At each rule execution, the binding of events to variables takes place for those events that have occurred since the start of the period preceding the last execution of the same rule and up to the start of the current period. If a triggered rule is not considered during some scheduling periods (e.g., because the rule has low priority), bindings to its *occurred* predicates include all bindings accumulated during those periods: we informally say that bindings carry on to the next period in the same way as rules. In this way, each event is considered by each rule exactly at one execution; we say that the event is *consumed* by the rule. The *old* predicate allows rules triggered by data events to refer to the database state prior to the modification. If the same object is modified several times between two successive rule executions, then the *old* predicate binds variables to the most recent old value.

Note that the triggering and execution of an exception does not necessarily suspend or abort the execution of the case in whose context the exception was generated. However, a Chimera-Exc rule may identify the

exceptional case in the condition part, and then suspend or rollback the execution of selected tasks or of the entire case.

Chimera-Exc rules are *detached*, i.e., they are executed in a different transaction than the triggering one. The choice of a detached execution mode is mandatory because Chimera-Exc actions must often be performed outside the database context, and therefore cannot be rolled back. Thus, it is essential that the triggering transaction commits before executing actions; in fact, only as the transaction commits are we guaranteed that the exception actually occurred. The use of a detached mode allows us to manage data events in a uniform fashion with respect to temporal, external, and workflow events; rule detachment, coupled with set-oriented execution semantics and periodic scheduling, is a good solution in terms of performance, as demonstrated in Section 5.3.

3.5 Rule Execution Contexts

Several workflow schemas can be managed by the same WfMS that can concurrently activate cases of the various workflows. Rules in Chimera-Exc are either defined in the context of a specific workflow or are global. In the former case, we say that rules are *targeted* to a workflow, and their side effects are propagated only to the cases and tasks of that workflow. Syntactically, we indicate the rule target in the definition, by adding the keyword *for* and the workflow name, as follows:

```
define trigger slowCaseEnd for carRental
```

All the rules in Section 3.3 are targeted and should be completed with the *for* clause. Once targeted, the trigger may be simplified by assuming as context the target workflow. Thus, trigger *stopWork* can be simplified as shown below, where the variable T implicitly ranges over the tasks of the *carRental* workflow:

```
define trigger stopWork for carRental
  events     18/hours during days
  condition  task(T), T.status="running", agent(A), T.executor=A
  actions    notify (A, "Time to go home!")
  order 100
end;
```

Untargeted rules, also called *global* rules, have side effects that affect all the cases of all the workflows managed by the WfMS. They can be used for writing generic WfMS exceptions, which monitor all the workflow schemas, or exceptions that connect two or more workflow schemas, enabling a form of interoperability among workflows. The following example shows a generic WfMS exception calling for the reassignment of all the agent's tasks, which become suddenly unavailable.

```
define trigger replaceAgent
  events     modify(agent.status)
  condition  agent(A), occurred(modify(agent.status),A),
             old(A.status) = "available", A.status = "unavailable",
             task(T), T.executor = A
  actions    reassign(T)
end;
```

The next exception relates the workflows for submitting and evaluating proposals, and detects the situation where the proposer's company is the same as the evaluator's. In such cases the evaluator must be changed.

```
define trigger replaceResponsible global
  events     caseStart(ProposalEvaluation)
  condition
  case(C1), case(C2),
             C1.wfName = "ProposalSubmission",
             C2.wfName = "ProposalEvaluation",
             occurred(caseStart,C2),
             C1.proposer.company = C2.evaluator.company
  actions    notify(C2.responsible, "Conflict of interests: Proposal "
             + C1.proposalNumber + " should have a different evaluator"
end;
```

4. ARCHITECTURE

Exceptions written in Chimera-Exc are processed by the FAR (FORO Active Rule) system, implemented at Politecnico di Milano. The architecture of FAR consists of four modules:

—The *compiler* accepts rules written in Chimera-Exc as input and produces their translation to an internal language specifically developed within this project. In addition, the compiler produces relational triggers for capturing data events directly within the database.

—The *time manager* is responsible for the optimal management of time-dependent events (including workflow and external events).

—The *scheduler* is activated periodically or in response to realtime rule triggering; it orders the rules and then submits them to the interpreter for execution.

—The *interpreter* is responsible for executing rules with a given degree of parallelism.

The above modules interact with a database interface, called the **basic access layer** (BAL), capable of executing standard SQL queries on top of a relational server, thus providing interoperability with any relational database; BAL was developed at SEMA and is used by FORO. High-level communication with the FORO WfMS occurs through the **FORO interface**, which informs the time manager of workflow events and receives from the interpreter the exception management primitives. This overall architecture is shown in Figure 3.

In the next sections we describe the more interesting features of each module.

4.1 Compiler

The **FAR compiler** compiles Chimera-Exc rules by filling in the rule repository and by translating the event, condition, and action parts of a rule.

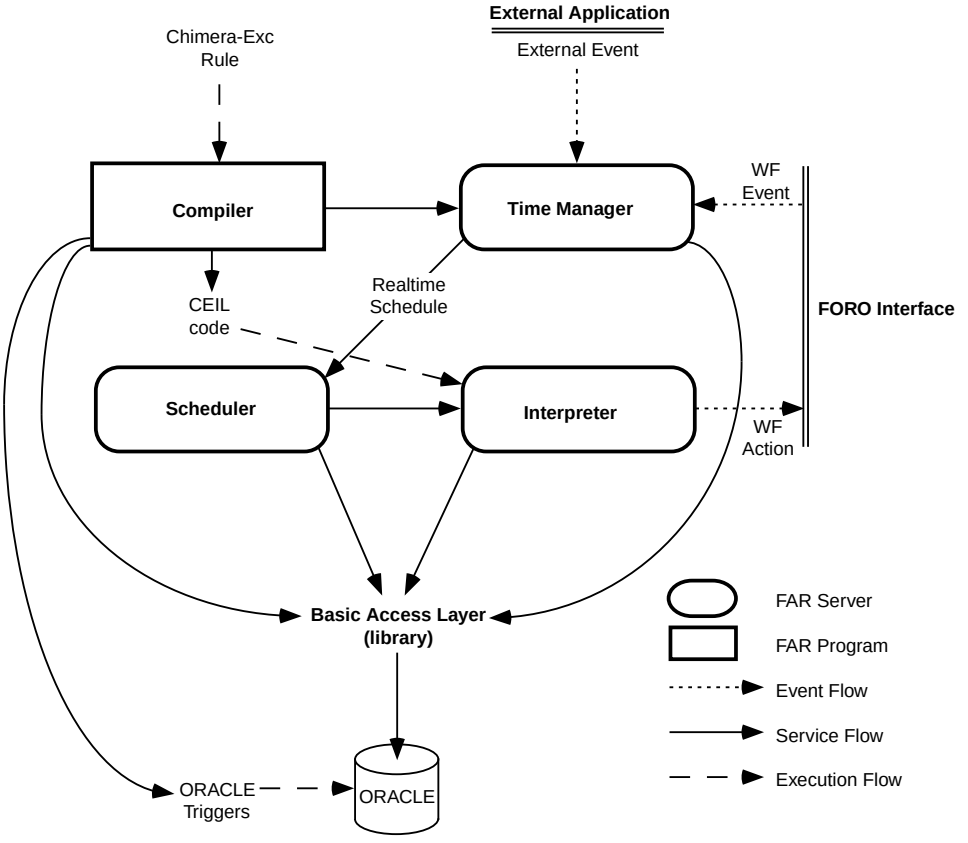

Fig. 3. FAR architecture. The FAR compiler is activated on demand; the FAR interpreter, the FAR scheduler, and the FAR time manager are daemon programs, i.e., they are always active waiting for requests coming from other modules.

As a rule is submitted to the FAR for compilation, the compiler first generates one tuple in the table *RuleDictionary*, storing the rule's name and identifier, the priority, and the timestamp of its latest execution (initially set to the compilation time), needed to restrict rule execution to process only those events that occurred after the previous execution of the rule. The compiler then populates table *RuleEvents* by adding one tuple for every event of every rule, each defining the characteristics of a triggering event, such as its type (data, workflow, temporal, or external), the involved object class and attributes (e.g., *carRental* and *carStatus*, respectively, for event `modify(carRental.carStatus)`), whether the event is to be managed in *realtime* or not, and finally the identifier of the newly compiled rule triggered by the described event. This table enables the identification of which rule is triggered by which event.

The event, condition, and action part of the rule are then managed by the FAR compiler as follows:

—*Data manipulation events* are captured by simple row-level granularity triggers as currently defined in the SQL3 standard; each Chimera-Exc data manipulation event is translated into a corresponding database trigger [Cochrane et al. 1996]. Triggers operate in the context of transactions running in the FORO environment; for each tuple inserted, updated, or deleted from the underlying tables, triggers write a tuple to a suitable table, called *DataEvent*, to signal that a given object has been created, deleted, or updated. For instance, the compilation of the exceptions presented in Section 3.3 produce only one database trigger, needed to detect modifications of the car return time for the *lateCarReturn* exception.

—*External, temporal, and workflow events* are analyzed by the time manager, and are described next.

—The *condition* is translated into a query written in standard SQL2; the query computes a *binding table* that contains all the bindings of the variables shared between the condition and the action. Thus, if the binding table is empty, the condition is not satisfied. If an *old* predicate is used in the condition, then the compiler writes a tuple in a suitable *RuleLog* table that defines the characteristics of the object attribute to be logged (the attribute's name and type and the object's class). Furthermore, the compiler also generates one database trigger for each *old* predicate found in the condition that, before an attribute is modified or deleted, captures the value of the attribute and logs it in suitable *Log tables*. One Log table exists for each data type; the trigger adds one tuple in the appropriate table for every rule interested in logging the value. In the car rental example, compilation of the exceptions presented in Section 3.3 produces only one database trigger that logs old values needed by exception *lateCarReturn* in order to refer to the old value of the *returnTime* attribute. Thus, compiling the *carRental* exceptions produces two triggers, both resulting from the compilation of exception *lateCarReturn*: one to detect updates to attribute *returnTime* and one to log the value of this attribute prior to the modification.

  Note that native database triggers are defined only to detect data manipulation events and to log old values. No other trigger or database-specific construct is needed by the FAR: this is an important feature that eases the portability of our system onto different database platforms.

—*Actions* are translated into calls to the BAL (for changing the database content) or the FORO interface (for interacting with the WfMS). Explicit iterators generate as many calls as the tuples of the binding table; each tuple provides the actual input parameters of a call.

The translation of condition and actions produces a piece of code written in an intermediate language, responsible of submitting the query to the BAL and then retrieving its tuples and generating the action calls.

4.2 Time Manager

The time manager is sensible to external events, workflow events, and wake up requests; it manages temporal events detected by means of the local clock.

External and workflow events are considered as special cases of interval events, in which the interval is set to zero and the anchor time is set equal to the time at which the external event is raised or the workflow event occurs. Similarly, instant events are reduced to special cases of interval events, with the interval set to zero. Therefore, all events are reduced to either interval or periodic events. Interval events are characterized by their anchor time and duration, while periodic events are characterized by their periodic duration and, possibly, two anchor times limiting the time interval in which periodic events should be considered.

Anchor times of a given event *e* are either time constants or other events that may occur in the future; in the former case we say that *e* is defined, and in the latter case we say that *e* is undefined. Thus, a given event may initially be undefined, then it may become defined, and finally it may occur.

The time manager keeps a list of defined events, sorted by their time of occurrence. Insertions into the list of defined events are due to the occurrence of workflow events and external raises or to the compilation of a defined event. Whenever a periodic event occurs, a suitable algorithm must inspect the periodic event definition in order to possibly enter another event, relative to the next period, into the defined events list. Suitable data structures hold undefined events (together with the specification of their anchor times) and periodic events; algorithms generate defined events from these data structures.

The time manager constantly compares the first event in the defined events list with the current value of the clock. Whenever an event occurs, the time manager deletes the event from the list and inserts a tuple into the *TemporalEvent* table, storing the event's identifier and timestamp. By this act, each temporal event that has occurred is matched with an insert to the database, and this insert causes an action of the Scheduler that will eventually consider all the rules triggered by that event.

External applications must register events to the time manager prior to raising them. We enable an arbitrary ordering of either the registration of the event or the compilation of the event within a rule, but both of them must occur before considering the external event as defined.

4.3 Scheduler

The scheduler is activated periodically or because a real-time event has occurred. It initially copies and then empties the tables *DataEvent* and *TemporalEvent*, which contain the events that have occurred since the last execution of the scheduler. A copy of these tables is made to enable access to them by rules, in particular to evaluate the *occurred* predicates within conditions. Then, the scheduler determines which rules are triggered and

orders them according to their priority, writing them into the *ReadyRules* table.

The motivation behind periodic scheduling is the need for batching multiple instances of the same exception. Exceptions may be raised rather frequently (many times as false alarms). Batching their handling and using a set-oriented evaluation for testing all the event occurrences at once is highly beneficial. This has been proven both experimentally and through simulation, as described in Section 5.3. We considered both the batching of rule executions and immediate scheduling as alternatives to periodic scheduling. Batching rules means activating the scheduler after a fixed number of events. Periodic activity dominates batching rules because it yields similar performances, but with much smaller variance in exception service time. On the other hand, immediate scheduling would generate a much lower throughput because every event would be followed by execution of the corresponding rule, without any scaling advantage. In our prototype we observed that immediate execution was infeasible even for simple workflows.

The scheduler is multithreaded and supports the parallel execution of rules; each rule is independently executed. Given the maximum degree of parallelism n, the scheduler selects the first n rules and invokes a thread of the interpreter for each of them; rule $n + 1$ is considered as soon as one thread becomes available, and this process continues until either all triggered rules have been considered or the period has expired. In this way, the number of rules executed in parallel is constant, at least during the time that follows the start of a scheduling period.

The period of activation and degree of parallelism are critical parameters for characterizing FAR behavior; we expect them to be set so that all rules are normally processed within one scheduler period. We consider this issue further in Section 5.3.

4.4 Interpreter

The interpreter executes each exception as an atomic thread. Instructions are written in CEIL (Chimera-Exc Internal Language), an intermediate language that enables table generations, set-oriented query submission, and tuple-oriented retrieval and update actions. These are executed by invoking the corresponding BAL services.

5. IMPLEMENTATION OF FAR

The FORO and FAR systems operate on top of Oracle Server, a commercial relational system. In particular, triggers generated by the FAR compiler are written in the Oracle SQL and installed into the Oracle database; Oracle Server must be version 7.2 or subsequent [Oracle Corporation 1996] to enable multiple triggers on the same event.

5.1 Transactions in FAR

FAR uses transactions as provided by Oracle Server. Mapping computations to ACID transactions was carefully studied to provide the required operational semantics and minimize the interference of FAR with the workflow applications managed by FORO. Interference occurs in the table *DataEvent*, whose content is written by Oracle triggers running in the workflow applications context and must be copied and deleted by the scheduler. The scheduler adds the tuples in the *DataEvent* table into its own private copy, then deletes all its content and commits as an ACID transaction; during this time, workflow applications cannot produce data events because they conflict in the *DataEvent* table, which is exclusively locked by the scheduler. However, such time is quite negligible, and the interference is acceptable.

For similar reasons, the scheduler adds the tuples of the *TemporalEvent* table into its own private copy, then deletes all its contents, and commits as an ACID transaction; during this time, the time manager cannot produce new temporal events. Finally, the scheduler accesses its own *DataEvent* and *TemporalEvent* copies, determines which rules are scheduled, and writes their rule identifiers into the *ReadyRules* table. Again, these operations are performed as an ACID transaction, so that the determination of triggered rules is reliable. Each rule is then executed by the Interpreter as an ACID transaction. Before committing the transaction, the scheduler deletes from the *ReadyRules* table the tuple corresponding to the executed rule. Exceptions may interfere with other exceptions or with workflow applications, and such interference is properly regulated by means of standard transactions. In particular, we assume that deadlocks are detected by the underlying database server that chooses suitable victims; the corresponding transactions are rolled back and then restarted.

The private copies of *DataEvent* and *TemporalEvent* tables are used to evaluate the *occurred* predicate; given that rules and their events may carry on from one period of scheduling to the next one, the content of these two tables can be deleted only when the FAR becomes idle after the execution of all the rules of a given period, since at such time all events are certainly consumed by their relevant rules.

5.2 Interaction of FAR with FORO

The FAR environment interacts with that of FORO in the following way:

—At *system generation time*, CORBA-IDL classes storing the WfMS data dictionary are generated. Translators IDL2SQL and SQL2Chimera generate the appropriate data dictionary descriptions for use in the Oracle database and in the FAR environment. Chimera-Exc exceptions may refer to FORO metadata, so their translation is invalidated when the system is regenerated.

—At *workflow definition time*, one class definition is associated to each workflow, storing the variables and constants associated to each case.

The workflow-specific class inherits from class *case*. Inheritance is implemented in the underlying relational database by means of the *vertical approach* [Atzeni et al. 1999]. Each case corresponds to one tuple in the generic table *case* (storing the cases' common attributes such as the *responsible* or the *initiator*) and one tuple of the workflow-specific table (storing the value of workflow-specific attributes such as the *plate* of the rented car). Inherited attributes of a given case can be retrieved by joining the two tuples that share the same case identifier. Again, IDL2SQL and SQL2Chimera translators provide suitable mappings.

—Due to *case execution*, the FORO interface informs the time manager of the beginning and end of cases and tasks. The time manager also provides a *WakeUpRequest* service to the FORO interpreter, which accepts requests for wakeup with a parameter *DateTime*. At the specified time, the time manager raises a *WakeUpCall* to the FORO interpreter.

—Due to *rule execution*, the FAR Interpreter notifies the FORO interface of the workflow management primitives to be executed in order to handle exceptions.

5.3 Performance Analysis

In order to evaluate the performances of the FAR server, we ran a few experiments on top of our prototype.[7] Our first observations were concerned with confirming the adequacy of our rule execution mode. To further confirm the advantages of the detached execution mode over a deferred execution mode (in which rules are executed at the task's commit time), we built a detailed queuing network model of our system. The model is described in Ceri et al. [1998]. Synthetically, the model is an open queuing network model with three service centers (CPU, disk, and exception manager) and four client classes (user transactions, also called tasks; exceptions with true conditions; exceptions with false conditions; and an exception scheduler). The distribution of task' interarrival times is considered exponential, assuming a random arrival. The interarrival times for the exception scheduler are assumed constant.

We then developed a model for a deferred execution mode for exceptions, and compared it with the detached mode. The curves show the tasks' execution time as a function of the tasks' frequency of arrival; execution times tend to the infinite when the system saturates. In Figure 4, we show the behavior of a system with no rules, or with detached rules (as in Chimera-Exc), or with deferred rules. Deferred rules saturate the system much earlier than detached rules, due to the batching factor. With detached execution all the event instances occurring in the same period (e.g., of all task starts) are processed together by a single rule execution.

[7]The database server was a Sun Sparc 10 with 32 MB of RAM, running the Oracle server v. 7.3.2.1 on Solaris 2.5.1. The machine running the FAR environment was a Sun Sparc Ultra 2/140 with Solaris 2.5.1.

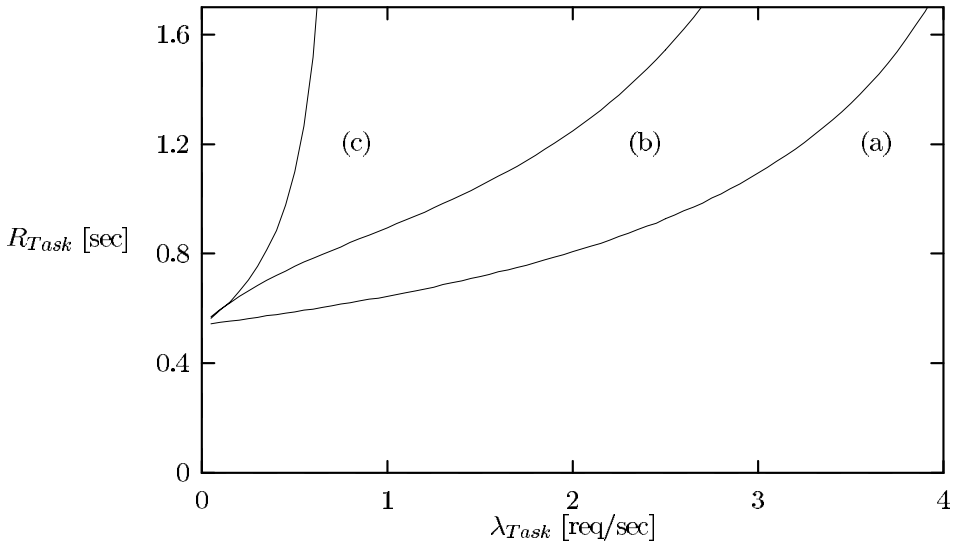

Fig. 4. System throughput for: (a) no rules; (b) detached rules; (c) deferred rules.

Therefore, we take advantage of set-oriented processing. Several ad-hoc experiments have confirmed the results of the queueing model.

In order to obtain an adequate level of performance, we dedicated a considerable effort to obtain a multithreaded implementation of the FAR scheduler. In our implementation, the threads are a fixed number, each one responsible of a connection with the database. The FAR scheduler dispatches exceptions to threads, monitoring the status and assigning a new exception as soon as the thread has terminated the execution of the previous one, until there are no more exceptions to run. The diagram in Figure 5 illustrates the throughput of the system, measured in rules per second, and how it correlates with the number of threads dedicated to the execution of exceptions. The figure shows a quasilinear increase of throughput with the increase in the number of threads;[8] the flattening of the curve of the throughput as the number of threads increases is due to the fact that the system tends to reach saturation.

If the goal of the system was simply to maximize the exception throughput, then a high number of threads should be selected, so as to guarantee system saturation. Such a solution, however, would cause the system to suddenly dedicate most of its resources to exception handling at the beginning of a scheduling period, thereby slowing the processing of task-originated transactions. Instead, our goal is to choose the number of threads that offers a good throughput for exception execution, while causing marginal impact on the response time of task-originated transactions.

[8]This figure indicates as well that each rule requires about one second of execution time, which is better than acceptable from an application standpoint (each "rule" is indeed a rather complex transaction that accumulates and processes all the events occurred during a period of time, translated to a high number of SQL calls through the BAL interface).

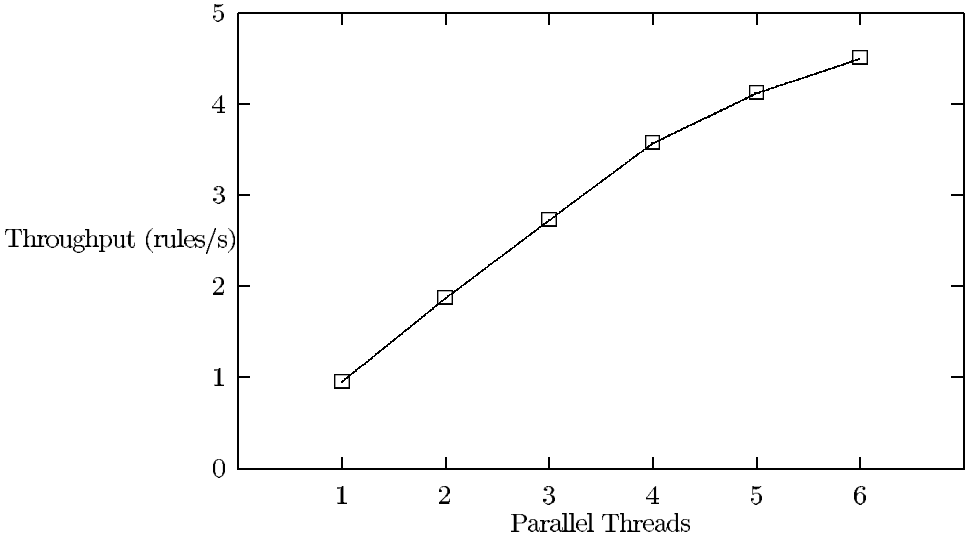

Fig. 5. Throughput of exceptions, varying the number of threads.

We measured the average response times of tasks and exceptions on our prototype, increasing the number of threads. The results of these observations are shown in Figure 6. The top curve in the figure represents the sum of both response times, whose minimum indicates a good empirical tradeoff between exceptions and task-related user transactions. We observe that in that particular experiment the number of threads that minimizes the top curve is four.

The scheduling period is another parameter with a strong impact on performance. With short scheduling periods, it is possible to shorten exception-handling delay, thereby increasing its quality, but also increasing the cumulative response time due to the lack of batching effects for events (as discussed above, see Figure 4). Thus, the behavior of the FAR system depends essentially on the tuning of these two parameters: the number of threads and the length of the scheduling period. In addition, in a real system the load changes during execution (e.g., due to peak times during the day). To this end, we implemented an adaptive algorithm, which monitors the load on the system. This adaptive algorithm dynamically modifies the scheduling period, but not the number of threads. In fact, in the Oracle implementation we experienced that dynamically changing the number of threads does not improve throughput, due to the overhead required in opening and closing the database channels associated with them.We perform the fine tuning on the scheduling period instead, which can be adapted after a previous period according to an iterative formula. A complete description of the adaptive algorithm for setting the scheduling period can be found in Ceri et al. [1998].

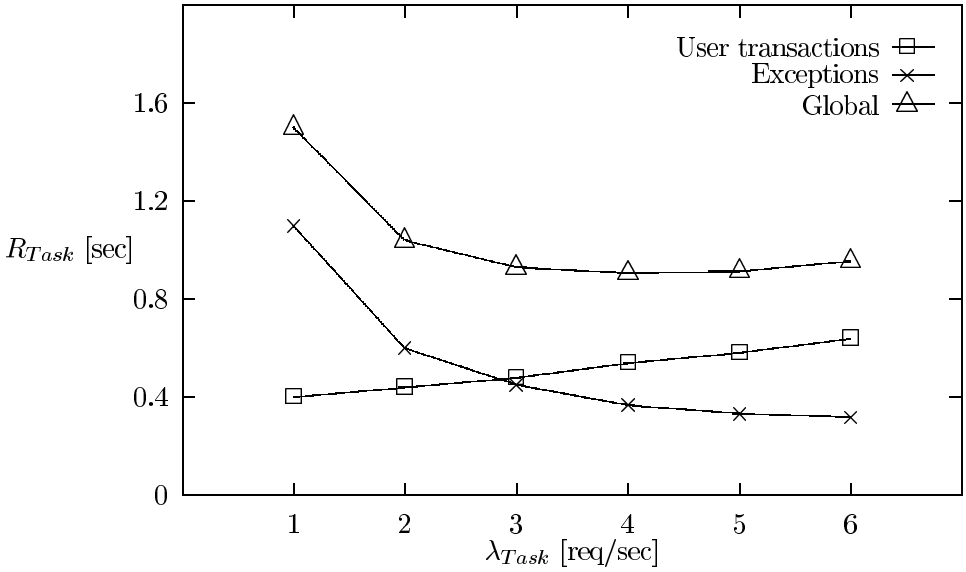

Fig. 6. Response times for tasks and exceptions.

6. DETECTING TERMINATION IN WORKFLOW EXCEPTIONS

Although exceptions are typically designed individually, by considering the exceptional event and by defining the appropriate reaction to it, the designer must also consider the characteristics of the entire set of the defined exceptions in order to verify whether undesired interactions can occur.

Two significant properties of a set of exceptions are *termination* and *confluence*. Informally, termination occurs when exceptions do not trigger each other indefinitely, so that exception processing eventually terminates, while confluence requires that the effect of exception processing be independent of the order in which triggered rules are executed.

In general, it is impossible to devise sufficient conditions for confluence in practical workflow applications, mainly due to the difficulty of performing a semantic analysis of the effect of exception execution. In addition, the Chimera-Exc language is intrinsically nondeterministic because it associates a set-oriented, declarative condition with a tuple-oriented imperative action, without any language construct for imposing an order on the bindings selected by the condition. Thus, it is sufficient that action executions on collections of objects be nonconfluent for the entire rule evaluation to also be nonconfluent. For instance, if a collection of objects representing selected tasks to be assigned needs to be matched with another collection of objects representing available agents and these are not semantically related, then it is not possible to repeat the same assignment at different executions of the same rule. This situation is not surprising, as it occurs whenever the language supports the declarative set selection, followed by

the individual management of each element in the set. So it occurs in most SQL triggers and procedures of conventional database applications.

Termination has been widely studied in the context of active databases, but the results of such analysis are not immediately applicable to workflow exceptions. Active databases consider rules that are triggered by data events and act within the scope of given transactions, while workflow exceptions consider arbitrary events and are executed within separate transactions in a detached mode. The very essence of termination has to be understood well, since the exception handler is a nonterminating machine that processes external and temporal events. In order to concentrate our analysis on exceptions, we assume that all processes terminate in finite time, and disregard the issue of process termination; the interested reader is referred to Jensen [1992] and ter Hofstede et al. [1998].[9]

We introduce a *Termination Analysis Machine (TAM)* as an abstraction of WfMS and exception handler that sequentially processes the exceptions and workflows that are explicitly initiated by triggered exceptions. The assumption of sequential processing simplifies the use of the TAM in proving termination, but it is not restrictive because rules and tasks execute as ACID transactions upon the same database, and their serializability is guaranteed by the underlying database server. Therefore, any concurrent execution of rules and tasks is equivalent to some serial execution of the same rules and tasks, and therefore to some execution of the TAM machine. The TAM does not accept new workflow cases besides those generated by exceptions, so it is the appropriate device for studying pathological cases of infinite executions, which are caused by the exception-handling behavior.

We define the *State of the TAM* as the triple $S = \langle DB, R, A \rangle$, where DB represents the WfMS database state; R represents the set of triggered rules taken from the set of workflow exceptions E; and A represents the set of active tasks operating in the set of workflows W.

A TAM execution is a sequence of TAM states; at each step the TAM executes either a rule or a task and removes it from either R or A; due to execution, new rules can be triggered or tasks may become active. We denote the sequence of executed rules and tasks as a *trace* of the TAM.

Definition 1. A given set of exceptions E, defined for a set of workflow schemas W, terminates iff, for an arbitrary initial state S_0, the TAM executes and produces a final state $S_f = \langle DB_f, \emptyset , \emptyset \rangle$ such that no rule is triggered and no task is active.

In the following we give sufficient conditions for termination applicable to classes of rules; we first concentrate on data-dependent rules and then consider workflow-dependent and time-dependent rules.

[9]Note that process termination is trivial if the process scheme does not include loops, and otherwise is based on a semantic analysis of loop conditions. But semantic analysis is possible only if the variables used in the analysis are isolated, so they cannot be subject to changes induced by exceptions.

6.1 Data-Dependent Rules

We consider exceptions E_D which are only triggered by data events and generate only data manipulation actions. The following definitions can be computed by a straightforward syntactic analysis over the rules in E_D.

—*TriggeredBy* denotes a function taking a rule r in input and producing the set of data events that trigger r.

—*Performs* denotes a function taking a rule r and producing the set of database modifications that may be performed by r's action.

—$Triggers_D$ denotes a function taking a rule r and producing the set of rules r' that become triggered as a result of the data manipulation actions of r (possibly including r itself): $Triggers_D(r) = \{r' \in E_D \mid Performs(r) \cap TriggeredBy(r') \neq \emptyset\}$.

—The *Triggering Graph* TG_D is a directed graph whose nodes are rules in E_D; an arc from r_i to r_j exists iff $r_j \in Triggers_D(r_i)$.

THEOREM 1. *If there are no cycles in TG_D, then the rules in E_D are guaranteed to terminate.*

PROOF. The rule execution algorithm presented in this paper is similar to that of Aiken et al. [1995], and the proof has the same structure. Suppose that TG_D is acyclic, and the TAM execution is still infinite. Tasks cannot be started by rules in E_D, so the number of tasks in the TAM trace is finite. This implies that at least one rule r appears an infinite number of times in the trace. And, in turn, that there exists an action $a \in TriggeredBy(r)$ performed infinitely many times. There is only a finite set of rules and tasks that produce action a and so trigger r, and since tasks cannot execute an infinite number of times, there must exist a rule r_1 producing action a ($a \in Performs(r_1)$), thus triggering r ($r \in Triggers_D(r_1)$), which is performed infinitely many times. Iterating the above reasoning, we can prove that there is also a rule r_2 that triggers r_1 and appears infinitely many times in the trace, and so on. Since we assumed that there are no cycles in the TG_D, this reasoning generates infinitely many rules, which is a contradiction. □

6.2 Workflow-Dependent Rules

We consider exceptions E_W, triggered by data or workflow events and generating data manipulation or workflow management primitives; clearly, this class of exceptions includes E_D. When a workflow action executed by a rule activates a new task, subprocess, or case, these may subsequently cause the activation of several tasks, according to the workflow schema. These tasks may generate data or workflow events. This behavior is modeled by a function w, from workflow actions that can be generated by rules to all the workflow and data events that actions can generate due to

workflow enactment. Determining such function in a conservative way requires a simple syntactical analysis of the workflow schema. More accurate analysis is possible by introducing the workflow state in the model, but such analysis is outside the scope of this paper. Thus, the following definitions can be computed by an extensive syntactic analysis over the rules in E_W and workflows in W:

—*StartedBy* denotes a function taking a rule r in input and producing the set of workflow events that trigger r.

—*Activates* denotes a function taking a rule r and producing the set of workflow actions that may be produced by r.

—$Triggers_W$ denotes a function taking a rule r in input and producing the set of rules r' that become triggered as a direct or indirect result (via the workflow engine) of the actions of r (possibly including r itself);
$Triggers_W(r) = \{r' \in E_W \mid (Performs(r) \cap TriggeredBy(r') \neq \emptyset) \vee (\exists w \in W \mid w)Activates(r)) \cap TriggeredBy(r') \cup StartedBy(r')) \neq \emptyset)\}$.
The second term evaluates the rules that can be triggered by the data or workflow events generated by the workflows activated by r's workflow actions.

—The *Triggering Graph* TG_W is a directed graph whose nodes are rules in E_W; an arc from r_i to r_j exists iff $r_j \in Triggers_W(r_i)$.

THEOREM 2. *If there are no cycles in TG_W, then the rules in E_W are guaranteed to terminate.*

PROOF. Suppose that TG_W is acyclic, and TAM execution still does not terminate. Given our assumptions on workflows, it is not possible that the trace includes a finite number of rules and an infinite number of tasks. This implies that at least one rule r appears an infinite number of times in the trace. In turn, this means that there exists an action $a \in (TriggeredBy(r) \cup StartedBy(r))$, which is performed infinitely many times. There is only a finite set of rules and tasks that produce action a, and thus, either:

(1) there exists a rule r_1 executed infinitely many times and such that $a \in Performs(r_1)$ (which also means that $r \in Triggers_W(r_1)$); or

(2) there exists at least a task t that produces action a; this implies that t is executed infinitely many times. Since we assume that workflow processes do not include endless loops, task t is started as an effect of a rule r_1 executed infinitely many times and such that $a \in w(Activates(r_1))$ (which also means that $r \in Triggers_W(r_1)$).

Iterating the above reasoning, we can prove that there also exists a rule r_2 that triggers r_1 and appears infinitely many times in the trace, and so on. Since we assume that there are no cycles in the TG_W, this reasoning generates infinitely many rules, which is a contradiction. □

6.3 Simple Time-Dependent Rules

Next, we consider exceptions E_T triggered by data, workflow, or simple time events, including instant events or interval events that are anchored to constant times or to workflow events. As before, E_T exceptions can generate workflow or data manipulation actions, and clearly this class of exceptions includes E_W.

As concerns termination, instant events or interval events anchored to constant times can be disregarded because they can trigger rules at most once, and therefore cannot be the cause of infinite execution. On the other hand, interval events anchored to workflow events trigger the rules after a fixed delay since the workflow events themselves. Delays are not considered in termination analysis (e.g., in the proof of Theorem 2), so we can disregard intervals provided that we include all rules of E_T whose interval event is defined into the set $R \subseteq E_T$ of triggered rules. Under this interpretation, results demonstrated for rules in E_W are immediately applicable to E_T.

6.4 Rules Dependent on Periodic and External Events

We finally consider rules that depend on periodic events, external events, or interval events, which in turn are anchored to external events. When any of these events is present, termination is not guaranteed, and indeed not desired. To see this, consider exceptions *noShow* and *stopWork* in Section3.3; we know that they will be indefinitely triggered every hour or every day. The same semantics can be achieved by a permanently active external application that periodically calls the exception handler, or by adding an arbitrary time interval to these events.

6.5 Termination Analysis Summary

On the basis of results presented in the previous sections, the analysis of a generic set of exceptions should be done by subtracting the rules that depend on periodic or external events by computing the triggering graph of residual rules and by testing its acyclicity. Rule analysis is based on pessimistic assumptions, and therefore cycles represent only potential causes for nontermination. Cyclic triggering graphs are thus often acceptable, provided that each cycle is analyzed to test whether termination is in danger. Semantic rule analysis methods described in Baralis et al. [1998]; Baralis and Widom [1994]; and Ceri et al. [1995] can be extended to our context, although with some practical difficulties.

In many practical cases, cycles are quite simple and rule analysis is very easy. As an example, consider the following exception:

```
define trigger carReassignment for carRental
  events      modify(carRental.selectedCar)
  condition   carRental (C), occurred(modify(carRental.selectedCar), C),
              C.carStatus="unavailable"
  actions     modify(carRental.selectedCar, C, C.secondChoiceCar)
end;
```

Trigger *carReselection* is raised by a modification of the *selectedCar* attribute of an object in the *carRental* class. The condition checks if the assigned car is not actually available for rental, and in this case assigns a different car to the customer, corresponding to his second choice. If this car is also unavailable, the exception continuously triggers itself, repeatedly assigning the same second-choice car. In this case, rule *carReassignment* triggers itself, thus the triggering graph TG_D has a ring; nontermination can be avoided by modifying the condition, so that the second-choice car is assigned only if it is available.

Next, we consider nontermination caused by workflow-dependent rules. We change the action of the exception *carReassignment* so that it activates the task *assignCar*, which selects a new car to be assigned to the customer, thereby retriggering the exception. Again, if the task is badly designed, it may automatically assign unavailable cars, thereby causing the trigger's condition to be true and endless processing: trigger *carReassignment* and task *assignCar* are alternatively executed forever.[10]

```
define trigger carReassignment2 for carRental
  events     modify(carRental.selectedCar)
  condition  carRental (C), occurred (modify(carRental.selectedCar), C),
             C.carStatus="unavailable"
  actions    startTask(C,assignCar)
end;
```

Note that the exception is triggered by a data event and generates a workflow action; in turn, a task is started by a workflow action and generates a data event. This situation also corresponds to a triggering graph TG_W with a ring.

7. RELATED WORK

In this section we first describe the use of active rules for workflow enactment, and then describe exception management in research projects and commercial systems, and finally overview exception handlers for programming languages and software process languages.

7.1 Active Rules and Workflow Enactment

Active rules introduce a significant increase in the expressive power of database languages, resulting in the enhancement of processing capabilities within database servers [Ceri and Ramakrishnan 1996; Widom and Ceri 1996]. They are supported by most relational database systems. Active rules in these systems support basic functionality and are inspired by the forthcoming SQL3 standard [Cochrane et al. 1996]. Triggers are also supported in several object-oriented database prototypes (e.g., HiPAC, Chimera, Reach, Samos, Sentinel, and many others [Widom and Ceri 1995; Paton 1999]). Some of these research projects also aim at developing

[10]Note that if the task is executed by a human agent, the process will soon terminate; endless execution occurs when the WfMS environment reacts to exceptions automatically.

detached ECA rule servers. Of these projects, HiPAC is the forerunner. HiPAC is an active object-oriented database system that extends traditional object-oriented systems with ECA rules. In HiPAC, ECA rules may be triggered by data, temporal, and external events. HiPAC supports several *coupling modes* that allow the definition of when the condition should be evaluated with respect to the event occurrence (event/condition coupling), and when the action should be executed with respect to the condition evaluation (condition/action coupling). It also supports the immediate, deferred, decoupled, and causally-dependent decoupled coupling modes. With respect to the event/condition coupling, *immediate* means that the condition is evaluated immediately after the event is detected (within the same transaction); *deferred* means that the condition is evaluated after the last operation in the triggering transaction (but within transaction boundaries); while *decoupled* means that the condition is to be evaluated in a separate transaction. Condition/action coupling has the same options, with the addition of *causally-dependent decoupled* mode, which further constrains the action transaction for serialization after the one evaluating the condition. The Chimera-Exc detached mode corresponds to a decoupled event-condition and an immediate condition-action coupling mode, with the difference that in Chimera-Exc the detached transaction always starts after the triggering one has committed.

TriGS [Kappel and Retschitzegger 1998]; *Reach* [Zimmermann and Buchmann 1999]; *SAMOS* [Gatziu et al. 1996]; and *Sentinel* [Chakravarthy 1997] have similar characteristics and objectives, in that they allow the definition of ECA rules triggered by object manipulation, temporal, and external events (in Chimera-Exc terminology), and aim to provide detached rule execution. However, all these prototypes have several limitations, such as reduced portability, lack of documentation and maintenance support, which made them unsuitable for our goals. Finally, none of the above-mentioned systems has been fully implemented. For instance, the Sentinel, SAMOS, and Reach implementations do not support the detached rule execution mode, while several prototypes have been developed for HiPAC, each supporting only a subset of the planned features.

In the context of workflow management, active rules have been proposed as a mechanism for *enacting* workflows in a few research prototypes such as Casati et al. [1996]; Geppert and Tombros [1995]; and Kappel et al. [1995]. In principle, the ECA paradigm is suitable for describing enactments; for instance, task completions activate rules whose condition detects the next task to be executed and whose action generates the appropriate task start. In practice, active rules are not used for workflow enactment, due to a rather low performance of the resulting workflow engine implementation. Active rules have the drawback that they must build the current context of the workflow by inspecting the database state, while a workflow interpreter typically maintains information about the current context within easy-to-access state variables. However, active rules can be used for defining a precise, albeit operational, semantics of workflow enactment.

7.2 Exception Handling in Workflow Research Projects

Although research in workflow models and systems has been very active in recent years, and the need for modeling exceptions in information systems has been widely recognized (see Auramaki and Leppanen [1989]; Borgida [1985]; Borgida et al. [1990]; Borgida et al. [1984]; and Saastamoinen [1995]), the workflow community has only recently tackled the problem of integrating exception-handling mechanisms in workflow models.

Eder and Liebhart [1995] provide a classification of the different types of failures and exceptions that can occur in WfMS. Exceptional situations are classified as *basic failures*, corresponding to failures at the system level (e.g., DBMS, operating system, or network failure); *application failures*, corresponding to failures of the applications implementing a given task, and therefore to be handled at the application level; *expected exceptions*, to be handled at the workflow level and corresponding to deviations from the normal behavior of the process; and *unexpected exceptions*, corresponding to the semantics of the business process, which are not properly modeled by the corresponding workflow representation, and are to be handled at the process-definition level. A similar classification, proposed in Heinl [1998], divides exceptions according to the consistency constraints they violate regarding execution semantics, application programs, the business process, and those between the workflow and the corresponding business process. Another classification is presented in Gray and Reuter [1994], centered on causes for failure (*environment*, *operations*, *maintenance*, *hardware faults*, and *software faults*). Failures are generally assumed to be handled at the system and application level, typically by relying on the transactional properties of the underlying database platform that enable forward recovery (unless failures also result in a *semantic* failure, such as when a system failure causes a deadline to be exceeded).

The workflow research community is mostly concerned with *expected* and *unexpected* exceptions, since these are related to the workflow and process modeling domain. Unexpected exceptions are caused by inconsistencies between a business process in the real world and its corresponding workflow representation [Heinl 1998], due to design error, incomplete workflow specification, or to change and improvement in the business process not yet implemented in the workflow description. Such exceptions occur frequently in processes that are very complex or with a high variability; they are normally captured and managed by human agents, typically by halting process execution and invoking human intervention.

The frequent and repeated occurrence of unexpected exceptions is an indication that "traditional" workflow technology may be unsuited for supporting the execution of the process under consideration and that *groupware* applications or *adaptive* workflow systems are probably more suitable. Adaptive systems enable the dynamic modification of workflow definitions, and are therefore particularly suited for supporting the execution of processes whose unfolding is unknown at workflow definition time. Such processes include *ad hoc* processes, i.e., processes that are to be

executed once or only a few times and whose exact form is determined as process execution proceeds.

Unexpected exceptions are typically handled by dynamically modifying the workflow, in order to resolve the inconsistency between the workflow model and business process [Heinl 1998]. Dynamic modification may be restricted to the workflow instance for which the exception occurred, or may be extended to the workflow schema in order to prevent further occurrences of the same exception in other workflow instances. A flexible process support system called *PROSYT* is presented in Cugola [1998]. PROSYT was explicitly developed to tolerate deviations from the defined process. The user is not forced to execute activities in the order specified by the process definition, but may instead execute operations whose preconditions for execution are not satisfied. However, depending on the process data, some operations may be "strictly" forbidden. It is possible to define *invariants* which, if violated, cause abortion of the invoked operation or a notification message to be sent to the process manager, who will have to intervene in order to restore a "legal" situation. These constraints help in limiting the degree of freedom allowed to the user, since allowing a completely unregulated process execution is useless.

In this paper we address the handling of asynchronous *expected* exceptions (Eder and Liebhart [1995] classification) or exceptions violating the consistency constraints of the process (Heinl [1998] classification). We deal not only with semantic failures, but also address generic exceptional situations that are part of the semantics of the process, but do not correspond to "normal" process execution.

Most approaches for handling *expected exceptions* are based on the integration of advanced transaction constructs into workflow models [Worah and Sheth 1997]. We regard them as *exception handlers based on extended transactional models*. In the following we briefly discuss these approaches.

ConTracts provide an execution and failure model for long-lived transactions and for workflow applications [Reuter et al. 1997]. A ConTract is a long-lived transaction composed of *steps*, where the order of execution of the steps is specified by a *script*. Isolation between steps is relaxed, so that the results of completed steps are visible to other steps; in order to guarantee semantic atomicity, each step is associated with a *compensating step* (or subscript, if the compensation is a complex process) that semantically undoes the effect of the step. ConTract provides support for managing situations in which a step is unable to fulfill its goal. At the step level, transactional steps are rolled back while nontransactional steps must be recovered by human intervention. At the script level, both forward and backward recovery are provided. Backward recovery is achieved by compensating completed steps, typically in reverse order of their execution. Compensation may be *partial*, meaning that it is performed up to the point where forward execution in the contract can be resumed, possibly along a path that is different from the faulty one.

In Leymann and Roller [1997] a transactional model based on the notion of *compensation spheres* is introduced. A compensation sphere is a group of tasks such that all tasks have to be executed successfully or all executed tasks must be compensated. As in the previous approach, tasks (or entire spheres) are associated with compensating activities that are executed in reverse order in the case of a task-semantic failure. Different behaviors can be associated with a compensation: besides partial rollback and forward recovery, control flow may be resumed at the start of the compensation sphere without performing the compensation, or administrative actions may be requested.

WAMO [Eder and Liebhart 1995] follows a similar approach for handling failure, although it is implemented within a more flexible and expressive model that is more suitable for workflow applications. The conceptual workflow model is enriched by the specification of the transactional properties of each task (e.g., a task may be *compensatable*, it may be *critical*, meaning that it cannot be undone or compensated, it might not require compensation in case of semantic failure, or it may be forced to succeed semantically, possibly with the help of human intervention). If a task fails semantically, compensation is started according to the transactional properties of tasks, until a decision point is reached in the flow structure where forward execution may be resumed by following an alternative path. Preliminary work on WAMO was extended in Eder and Liebhart [1998], where the model was modified so that any activity can be the turning point from backward compensation to forward execution (based on the observation that decisions are sometimes also taken within a task, and not only in branches in the control flow) by allowing complex activities rather than compensating tasks only, and by providing support for dynamic workflow instance modification when the backward compensation/forward recovery approach does not handle the exception satisfactorily.

Within the *Exotica* project, methods and tools to implement advanced transaction models on top of FlowMark (IBM's workflow product) have been developed [Alonso et al. 1996; 1994]. The basic idea is to provide the user with an extended workflow model that integrates advanced transaction concepts, allowing the definition of a compensating task for each ordinary task. It also enables the user to translate these specifications into plain FDL (FlowMark Definition Language) by properly inserting additional "compensating" paths after each task or group of tasks, to be conditionally executed upon a task failure (captured by means of the task return code). In particular, it is shown how sagas and flexible transactions can be implemented in FlowMark (similar approaches can be followed for other transaction and workflow models). A preprocessor was also developed in order to automate the mapping.

CREW [Kamath and Ramamritham 1998] extends the approaches above by providing more flexibility in the backward compensation/forward execution process. For each step, it allows the definition of the point to which execution should be rolled back in case of failure and the specification of whether execution should be restarted or aborted from there; furthermore,

based on a predicate defined over workflow variables, a task involved in a partial rollback and forward recovery may or may not be compensated or re-executed.

The transactional approaches offer limited exception-handling capabilities. In fact, there are a number of exceptional situations that cannot be managed with backward compensation and forward recovery on a different path. These are extreme and expensive solutions (in terms of lost work), unnecessary in many cases. Furthermore, the WfMS is incapable of discriminating among the different causes of semantic failure, which could require different handling strategies. Chimera-Exc and the transactional approach are orthogonal and can be combined. This is the solution adopted in WIDE, where the workflow model includes concepts taken from *sagas* and *nested transactions*. The rollback of a workflow instance can be triggered explicitly by Chimera-Exc actions. Next, we review two systems called TREX and OPERA, which extend the transactional approach, adding more features to deal with expected exceptions.

In *TREX*, [van Stiphout et al. 1998] several types of task failures are identified, including deadline expiration, unavailability of resources, inability to access a piece of data, explicit "fail" messages sent from a task to the parent subprocess to indicate that it is unable to continue execution, and "abort" messages sent from a subprocess to its component tasks to indicate that they must abort execution. Exception handling is specified by means of a mapping rule of the type

$$(task \times exception) \rightarrow exception\ handler$$

where a specific exception-handling strategy is associated with every type of failure for each task. Several types of exception-handling strategies can be defined and broadly classified as *continuation handlers*, which modify the flow and then resume normal execution, allowing the insertion of new activity, execution of an alternate activity, or the retrial of the failed activity, and *abortion handlers*, which provide for partial compensation and forward recovery mechanisms analogous to those described above. Although TREX extends transactional approaches to exception handling by allowing discrimination between several kinds of failures and transfers the control to suitably defined exception-handling activities as the exception occurs, it still lacks the required flexibility for handling generic exceptional situations not related to task failures and is restricted in the class of allowed reactions.

The *OPERA* process support system [Hagen and Alonso 1998] offers a flexible exception-handling mechanism that integrates concepts developed for programming languages with advanced transaction models. Workflows in OPERA have a modular structure. In case a task fails, execution is stopped and the control is transferred to the exception handler of the parent activity (or to a default handler if none was defined in the parent activity); in turn the handler may also propagate the exception to higher levels in the hierarchy of activities. An exception may be signaled by

external applications (e.g., by means of the workflow API or application return codes), it may be defined within the flow structure by a conditional split leading to the exception handler rather than to an ordinary activity, or it may be signaled when a given predicate on process data is satisfied. Exception handling is specified using the same mechanism as for normal flow by defining tasks and control/data flow among tasks. After managing the exception, the handler may resume execution of the signaling task or subprocess or abort (compensate) it.

Although exceptions in OPERA can be expressed in a variety of ways, they are globally less powerful than those of FAR, for instance, in dealing with temporal events, or in composing different kinds of events within homogeneous expressions. Furthermore, each event must be separately managed, and a different exception handler must be activated for every case affected by the exception.

7.3 Exception Handling in Commercial Workflow Management Systems

Workflow management has become very popular in recent years, and several hundred commercial products exist on the market (see Stark and Lachal [1995] for a review of some and Mohan [1997] for a discussion of recent trends in workflow management).[11] However, very few products provide support for managing exceptional situations; typically, only exceptions that are synchronous with the progression of the workflow can be captured by the model. So alternative execution paths in the control flow are made in reaction to exceptions, resulting in workflow specifications that are complex and difficult to understand.

For instance, widespread products such as IBM's *FlowMark* [IBM 1996] or Hewlett Packard's *AdminFlow* [Hewlett Packard 1998] are not able to capture external, temporal, or data events, and the only support for "exceptional" situations is the ability to define *deadlines* for tasks and processes and to capture task semantic failures (e.g., by means of the task return code). Reactions to these exceptions must be modeled by suitably modifying the control flow. For instance, in AdminFlow the designer may set a task deadline and specify that as the deadline expires the process should either terminate or proceed to set the task state to `TIME OUT`; in the latter case, the task state can be checked by a conditional split in order to execute an "exceptional" path as a result of a deadline expiration. Instead, FlowMark manages deadlines by sending warning messages to agents, regardless of the process state or other conditions.

A few products, such as COSA, StaffWare, InConcert, and Action Request System, offer some modeling features for capturing and reacting to events, although with much less functionality than provided by FAR.

---

[11]Some web sites maintain references to the home pages of the most popular commercial WfMSs. The interested reader may visit the following URLs:
http://www.do.isst.fhg.de/workflow/pages/Produkte_Englisch.html and
http://www.ifi.unizh.ch/groups/dbtg/Workflow/workflow_sites.html

COSA, from Cosa Solutions [Software-Ley GmbH 1996], includes in its workflow model the notion of a *trigger*, defined as an event-action rule that can be triggered by external events, workflow events, or deadline expiration, and can react to the triggering event by activating a task or a new (sub)process instance. *InConcert*, from InConcert Inc. [McCarthy 1993], also includes event-action triggers in its workflow model. Triggering events can be process state changes (e.g., a task becomes ready for execution), external (user-defined) events, or temporal events. Allowed actions include notification of messages to agents, activation of a new process, or invocation of a user-supplied procedure. In *Staffware*, from Staffware Corporation [Staffware Corporation 1997], a special kind of task, called an *event step*, can be defined. The event step suspends case execution until a defined event occurs. Event notifications must include the specification of the workflow, case, and step identifier in order to identify the event step.

Although COSA, InConcert, and Staffware provide some exception-handling support, they have several limitations with respect to WIDE. They support a subset of Chimera-Exc events and actions (being particularly weak in detecting data events), and in addition they share the same limitations as OPERA; in particular, each event must be managed separately by the exception-handling mechanism. The reaction can only affect a single case, with the above-mentioned disadvantages in performance and expressive power. Furthermore, notifications of external events must explicitly indicate the affected case, while a Chimera-Exc rule can automatically determine the set of cases (often more than one) interested in a given external event and manage it for all affected cases, as in the *carAccident* event example.

Finally, *Action Request System*, from Remedy Corporation [1996], takes a different approach to workflow modeling, where the emphasis is on data rather than on activities. The designer models a process by defining the data items and the operations that can be performed on the data. The workflow and task assignments to agents are then defined by a set of ECA rules capable of reacting to operations executed over workflow data by notifying messages to selected agents and presenting the work to them (i.e., the data and the set of operations that can be performed next). Rules can also monitor predicates over data and time and can react to them (possibly depending on conditions over time and over the process state) by modifying process data or activating external applications. Thus, the rules in the Action Request System allow several types of events to be monitored and several actions to be performed as the event occurs and as a defined condition is satisfied. The drawback is that it offers low-level constructs for workflow modeling, so that it is difficult to have a global view of the process that is entirely modeled by means of rules. Thus, a considerable effort is required in order to define and maintain the workflow, and although exceptions can be defined, the specification of normal and exceptional behaviors are undistinguishable within the set of workflow rules.

7.4 Exception Handling in Programming Languages

The notions of exception and exception handling are part of many programming languages, e.g., Smalltalk, Ada, C++, Java, and PL/SQL. However, although we recognize similar concepts, the domain and purpose of exceptions in programming languages are quite different from exceptions in workflows. In the following, we describe the exception-handling support offered by Ada and then analyze the differences with respect to our approach. Similar considerations hold for the other programming languages mentioned above.

In Ada, an exception denotes an event that suspends the ordinary program execution [Booch 1983]. Events can be predefined or declared by the user. The first category includes events that can be raised by the runtime system such as *NUMERIC ERROR*, raised when an operation produces a result outside the implemented domain (e.g., a division by zero), or *STORAGE ERROR*, raised when storage limits are exceeded. User-defined exceptions, qualified by their names, must instead be raised explicitly within the application code by means of the *raise* statement.

A handler for every exception can be defined in any block or body of a subprogram, package, or task. As an exception is raised, normal execution is suspended and the code associated with the corresponding handler within the same block/unit is executed. If no handler is found, the exception is propagated to the containing block. Finally, if no handler has been defined, the exception is passed to the operating system, which halts program execution. After the execution of the exception-handler code, program execution proceeds from the end of the block in which the exception was handled. Raising an exception provides no parameters to the handling part, which can only operate on the variables that are in the scope of the unit in which the handler is defined. Furthermore, no assumption can in general be made on the values of these variables nor on the instruction that raised the exception.

Exceptions in Ada (and in other programming languages) are mainly intended to detect erroneous conditions and avoid abnormal program termination in order to recover the error or at least allow a graceful degradation. Typical reactions involve abandoning the execution of the unit in which the exception was raised, retrying an operation, using an alternative approach to perform an operation, or repairing the cause of the error when possible [Booch 1983]. This is different from exceptions in workflow systems, which operate at a higher level of abstraction, since they are not intended to protect from system or programming errors, but rather to enable the specification of a complex behavior that is anomalous with respect to the "normal" semantics of the business process, but yet driven by the application semantics.

7.5 Exception Handling in Software Process Modeling Languages

Languages developed for software processes could be applicable to workflow management because they are intended to drive the software process

whose flow requires a complex sequence of steps and uses sophisticated automatic tools. These environments make use of rich, process modeling languages, which are often based on rule-based paradigms, and typically integrate reactive and asynchronous elements as part of their normal behavior. Thus, process languages offer high-level paradigms that have comparable semantics with (and even include the semantics of) Chimera-Exc. However, such languages are not currently integrated with WfMS, and therefore do not provide a solution to the problem of adding the management of expected exceptions to existing WfMS systems, which is the primary motivation of our work. A major shortcoming of process languages in our context is that the above languages are often not fully and robustly implemented, especially concerning integration with transactional systems. It is however interesting to review their features.

APPL/A [Sutton, Jr. et al. 1995] is a software process language that extends Ada with relations, triggers, constraints, and transactions. In APPL/A, the reactive component is integrated with the imperative paradigm of the programming language; emphasis is given to the management of consistency, giving the designer the flexibility to adapt the consistency-checking mechanism to the context (disabling it, enforcing it, or only partially enabling it). Other process languages are built directly on top of the rule paradigm. Alf [Canals et al. 1994] uses active rules as the main paradigm to represent the software construction process; Merlin [Junkermann et al. 1994] uses Prolog-based rules as the foundation of its environment. Epos [Conradi et al. 1994] integrates the procedural and reactive paradigms in its description language. The seamless integration of rules for the specification of exceptions within the rule paradigm of the above languages is very attractive. However, it may generate solutions that are hardly managed, due to the complexity of the interactions between a large number of rules.

8. UNBUNDLING ACTIVE FUNCTIONALITIES FROM FORO

A new direction for development of database systems is to "break the database box" [Silberschatz and Zdonik 1997]. Databases offer services that are typically integrated within a complex DBMS, and many applications require only a fraction of such functionality. Thus, there is a trend to "unbundle" the DBMS components and let applications combine the modules they need in arbitrary ways. This trend is applicable to the FAR system, which is separable from the WIDE architecture, and can provide a module for specifying and executing detached rules without being connected to the FORO WfMS.

We have followed a strategy presented in Gatziu et al. [1998] for unbundling the active component of a DBMS. Clearly, unbundling active functionalities requires the separation of the active component from the underlying data source, which may or may not be a DBMS. Such separation is present in the WIDE architecture, which is based on the Oracle DBMS. Further, unbundling requires the separation of an *event service*, responsible

for managing complex and heterogeneous events, and a *rule service*, responsible for scheduling and executing rules. Such separation is also present in the FAR architecture, since the former is provided by the time manager, and the latter by the chain of the compiler, scheduler, and interpreter.

We experimented with the use of the WIDE architecture in combination with an arbitrary Oracle application. Such unbundling was relatively simple, since FAR has a bidirectional interface with FORO, which was introduced in order to support the separate development of FORO and FAR. We therefore achieved unbundling by disconnecting FAR from FORO by connecting it directly to Oracle and adapting the FORO interface. We used the Sql2Chimera translator to make an Oracle schema definition available to FAR (this corresponds to building a wrapper; see Garcia Molina et al. [1997]), and extended the FAR action language so it can launch arbitrarily named stored procedures with parameters. One invocation is performed for each binding produced by the condition. These calls are channeled by the adapted FORO interface directly to the Oracle server. In this way the FAR environment provides detached active rules that run in the context of a generic Oracle application; in particular, we experienced its use with an application that provides access to the Car Rental information, though not managed by a WfMS.

The most critical point of this architecture is the detection of data events. In Gatziu et al. [1998], event detection is delegated to a subsystem that interacts with a passive DBMS. Instead, in the FAR solution, data events are captured by means of native Oracle triggers, which are automatically generated by the rule compiler and then installed in the Oracle DBMS. This solution is not portable because there is no published standard for triggers [Widom and Ceri 1996], and so Oracle triggers differ from those of other vendors. We, however, limited the trigger generation code to a portion of the rule compiler that can be easily separated and recoded.

9. CONCLUSIONS

This paper first introduced exception handling in workflow management as a new and interesting problem, and then showed the solution of the problem by means of a specifically designed language (Chimera-Exc) and architecture (FAR). In the following, we summarize the main contributions of our work:

(1) With the support of our partners in the WIDE project, we have identified the *user requirements* on expected exceptions, identifying the different types of exceptional events and of exception-handling strategies.

(2) We have proposed a rich *exception-specification language,* supporting an expressive event language, detached evaluation of conditions and actions, declarative set-oriented conditions, and imperative tuple-oriented actions for interacting with the workflow manager.

(3) We have developed formal methods for *proving the termination* of the exception handler in interaction with workflow management, showing that sufficient conditions for termination can be reduced to testing the acyclicity of suitable triggering graphs.

(4) We have performed a *full implementation* of an exception handler, facing a number of architectural choices concerning, e.g., the efficient management of temporal events, selective logging of past states, use of native relational triggers to detect data changes, periodic rule scheduling with an adaptive setting of the period of execution, and so on.

(5) We have designed and engineered the system's components in order to emphasize its *portability*. Our dependence on a given DBMS is limited to the use of triggers for capturing data events. All other database accesses are performed through the Basic Access Layer that guarantees platform independence. Interfaces are specified in IDL, and the system executes on top of CORBA.

(6) We have guaranteed *interoperability* between the FAR and the workflow engine by preserving the autonomy of each system and guaranteeing minimal interference (and delay) with the processing of normal workflow execution. Portability and interoperability are essential for unbundling the FAR system from FORO.

The first version of FAR, integrated with FORO, has been available since July 1997 to partners of WIDE: the ING Bank Group in Holland, HGM, and the Hospitàl General de Manresa, Catalunya, Spain. Exceptions, written in Chimera-Exc, were specified and then implemented in the context of workflows for claim management (by ING) and patient admission (by HGM). The specification process led to the careful redesign of several features of the language, so as to increase its expressive power and simplify the trigger code, reported in this paper. Experiments have shown several performance bottlenecks, but also provided us with a positive assessment of our architectural choices. Based on these feedbacks, the final version of FAR was delivered in June 1998. Some residual efforts in WIDE are being dedicated to developing a design interface for specifying exceptions based on patterns, and also to developing a methodology for building workflow applications. Such a methodology includes exceptions as an important ingredient, and will be integrated with Sema's "Iteor" methodology for business process reengineering.

ACKNOWLEDGMENTS

The overall architecture of FAR and its integration with FORO and with advanced transactional services was jointly designed by the teams at Sema, Twente University, and the Politecnico di Milano. We are particularly grateful to Paul Grefen and Gabriel Sànchez Gutierrez for their contribution in the design of the architecture, and to Carlos Lopez Alonso for his role as chairman of the software integration board. We also acknowledge

the contributions of Barbara Pernici and Giuseppe Psaila to the language design, and thank Alex Borgida and Alfonso Fuggetta for the fruitful discussions that helped us in the revision of the first version of this paper.

Fifteen masters' students, from the Politecnico di Milano and the Universit a Statale di Milano, participated in the implementation of FAR: Luca Barozzi, Davide Benvenuti, Dario Canepari, Patrizio Ferlito, Roberto Ferrari, Maurizio Manni, Cristian Mauri, Luca Moltrasio, Simone Rodigari, Marcello Ronco, Riccardo Sabadini, Lazzaro Santamaria, Franco Varano, Alberto Villa, and Daniele Zampariolo.

REFERENCES

AIKEN, A., HELLERSTEIN, J. M., AND WIDOM, J. 1995. Static analysis techniques for predicting the behavior of active database rules. *ACM Trans. Database Syst. 20*, 1 (Mar. 1995), 3–41.

ALONSO, G., AGRAWAL, D., ABBADI, A. E., KAMATH, M., KAMATH, M., GUNTHOR, R., AND MOHAN, C. 1996. Advanced transaction model in workflow context. In *Proceedings of the 12th IEEE International Conference on Data Engineering* (New Orleans, LA) IEEE Press, Piscataway, NJ.

ALONSO, G., KAMATH, M., AGRAWAL, D., ABBAD, A. E., GUNTHOR, R., AND MOHAN, C. 1994. Failure handling in large scale workflow management systems. Tech. Rep. RJ9913. IBM Almaden Research Center.

ATZENI, P., CERI, S., PARABOSCHI, S., AND TORLONE, R. 1999. *Database Systems: Concepts, Languages and Architectures*. McGraw-Hill, Inc., Hightstown, NJ.

AURAMAKI, E. AND LEPPANEN, M. 1989. Exceptions and office information systems. In *Office Information System: the Design Process* Elsevier Sci. Pub. B. V., Amsterdam, The Netherlands.

BARALIS, E., CERI, S., AND PARABOSCHI, S. 1998. Compile-time and run-time analysis of active behaviors. *IEEE Trans. Knowl. Data Eng. 10*, 3 (May-June), 353–370.

BARALIS, E. AND WIDOM, J. 1994. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB'94, Santiago, Chile, Sept.) VLDB Endowment, Berkeley, CA, 475–486.

BOOCH, G. 1983. *Software Engineering with Ada*. Benjamin/Cummings series in computing and information sciences. Benjamin-Cummings Publ. Co., Inc., Redwood City, CA.

BORGIDA, A. 1985. Language features for flexible handling of exceptions in information systems. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 565–603.

BORGIDA, A., MYLOPOULOS, J., SCHMIDT, J., AND WETZEL, I. 1990. Support for data-intensive applications: Conceptual design and software development. In *Proceedings of the Second International Workshop on Database Programming Languages* (San Mateo, CA) Morgan Kaufmann Publishers Inc., San Francisco, CA, 258–280.

BORGIDA, A., MYLOPOULOS, J., AND WONG, H. 1984. Generalization as a basis for software specification. In *On Conceptual Modeling* Springer-Verlag, New York, NY, 87–114.

CANALS, G., BOUDJLIDA, N., DERNIAME, J.-C., GODART, C., AND LONCHAMP, J. 1994. ALF: A framework for building process-centred software engineering environments. In *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Research Studies Press Advanced Software Development Series. Research Studies Press Ltd., Taunton, UK, 153–185.

CANNAN, S. J. AND OTTEN, G. A. M. 1992. *SQL-The Standard Handbook*. McGraw-Hill, Inc., Hightstown, NJ.

CASATI, F., CERI, S., PERNICI, B., AND POZZI, G. 1996. Deriving active rules for workflow enactment. In *Proceedings of the Seventh International Conference on Database and Expert Systems Applications* (DEXA '96, Zurich, Switzerland, Sept.), R. Wagner and H. Thoma, Eds. Springer-Verlag, New York, 94–115.

CASATI, F., CERI, S., PERNICI, B., AND POZZI, G. 1998. Workflow evolution. *Data Knowl. Eng. 24*, 3, 211–238.

CERI, S., BARALIS, E., FRATERNALI, P., AND PARABOSCHI, S. 1995. Design of active rule applications: Issues and approaches. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases* (Singapore, Dec.) Springer-Verlag, Berlin, Germany, 1–18.

CERI, S., FRATERNALI, P., PARABOSCHI, S., AND TANCA, L. 1996. Active rule management in Chimera. In *Active Database Systems*, J. Widom and S. Ceri, Eds. Morgan Kaufmann Publishers Inc., San Francisco, CA, 151–176.

CERI, S., GENNARO, C., PARABOSCHI, S., AND SERAZZI, G. 1998. Scheduling exceptions in a workflow management system. Tech. Rep. TR-98.27. Dip. di Elettronica e Informazione, Politecnico di Milano, Milan, Italy.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic Programming and Databases*. Springer-Verlag, New York, NY.

CERI, S. AND RAMAKRISHNAN, R. 1996. Rules in database systems. *ACM Comput. Surv. 28*, 1, 109–111.

CERI, S. AND WIDOM, J. 1990. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Data Bases* (VLDB, Brisbane, Australia, Aug.) VLDB Endowment, Berkeley, CA, 566–577.

CHAKRAVARTHY, S. 1997. Sentinel: an object-oriented DBMS with event-based rules. *SIGMOD Rec. 26*, 2, 572–575.

COCHRANE, R., PIRAHESH, H., AND MENDONÇA MATTOS, N. 1996. Integrating triggers and declarative constraints in SQL database sytems. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (VLDB '96, Mumbai, India, Sept.) 567–578.

CONRADI, R., HAGASETH, M., LARSEN, J.-O., NGUYÊN, M. N., MUNCH, B. P., WESTBY, P. H., ZHU, W., JACCHERI, M. L., AND LIU, C. 1994. EPOS: Object-oriented cooperative process modelling. In *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Research Studies Press Advanced Software Development Series. Research Studies Press Ltd., Taunton, UK, 33–70.

CUGOLA, G. 1998. Inconsistencies and deviations in process support systems. Ph.D. Dissertation. Dip. di Elettronica e Informazione, Politecnico di Milano, Milan, Italy.

DAYAL, U., HSU, M., AND LADIN, R. 1990. Organizing long-running activities with triggers and transactions. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (SIGMOD '90, Atlantic City, NJ, May 23–25, 1990), H. Garcia-Molina, Ed. ACM Press, New York, NY, 204–214.

DENNIS, R. AND MCCARTHY, S. K. S. 1993. Workflow and transactions in InConcert. *IEEE Data Eng. 16*, 2 (June), 53–56.

EDER, J. AND LIEBHART, W. 1995. The workflow activity model WAMO. In *Proceedings of the International Conference on Cooperative Information Systems* (Vienna, Austria, May)

EDER, J. AND LIEBHART, W. 1998. Contributions to exception handling in workflow management. In *Proceedings of the Sixth International Conference on Extending Database Technology* (Valencia, Spain, Mar.), H. -J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds.

GARCIA-MOLINA, H., PAPAKONSTANTINOU, Y., QUASS, D., RAJARAMAN, A., SAGIV, Y., ULLMAN, J., VASSALOS, V., AND WIDOM, J. 1997. The TSIMMIS approach to mediation: Data models and languages. *J. Intell. Inf. Syst. 8*, 2, 117–132.

GATZIU, S., FRITSCHI, H., AND VADUVA, A. 1996. SAMOS. An active object-oriented database system: Manual. Tech. Rep. 96.02. University of Zurich, Zurich, Switzerland.

GATZIU, S., KOSCHEL, A., VON BÜLTZINGSLOEWEN, G., AND FRITSCHI, H. 1998. Unbundling active functionality. *SIGMOD Rec. 27*, 1, 35–40.

GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. 1995. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases 3*, 2 (Apr. 1995), 119–153.

GEPPERT, A. AND TOMBROS, D. 1995. Event-based distributed workflow execution with EVE. Tech. Rep. 96.05. University of Zurich, Zurich, Switzerland.

GRAY, J. AND REUTER, A. 1994. *Transaction Processing Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

GREFEN, P., PERNICI, B., AND SANCHEZ, G. 1999. *Database Support for Workflow Management: the WIDE Project*. Kluwer Academic Publishers, Hingham, MA.

HAGEN, C. AND ALONSO, G. 1998. Flexible exception handling in the Opera process support system. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems* (ICDCS98, AAmsterdam, The Netherlands, May)

HEINL, P. 1998. Exceptions during workflow execution. In *Proceedings of the Sixth International Conference on Extending Database Technology* (Valencia, Spain, Mar.), H. -J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds.

HEWLETT-PACKARD. 1998. Changengine Admin Edition (AdminFlow) Process Design Guide. Hewlett-Packard, Fort Collins, CO.

IBM. 1996. IBM FlowMark - Modeling Workflows. IBM Corp., Riverton, NJ.

JENSEN, K. 1992. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, New York, NY.

JUNKERMANN, G., PEUSCHEL, B., SCHÄFER, W., AND WOLF, S. 1994. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Research Studies Press Advanced Software Development Series. Research Studies Press Ltd., Taunton, UK, 103–129.

KAMATH, M. AND RAMAMRITHAM, K. 1998. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of the 14th International Conference on Data Engineering* (Orlando, FL, Feb.) IEEE Computer Society Press, Los Alamitos, CA.

KAPPEL, G., LANG, P., RAUSCH-SCHOTT, S., AND RETSCHITZEGGER, W. 1995. Workflow management based on objects, rules, and roles. *IEEE Data Eng. 18*, 1 (Mar.), 11–18.

KAPPEL, G. AND RETSCHITZEGGER, W., Eds. 1998. The TriGS active object-oriented database system— an overview. *SIGMOD Rec. 27*, 3, 36–41.

LEBAN, B., MCDONALD, D. D., AND FORSTER, D. R. 1986. A representation for collections of temporal intervals. In *Proceedings of the Conference on AAA-I* (AAAI'86, Philadelphia, PA) 367–371.

LEYMANN, F. AND ROLLER, D. 1997. Workflow-based applications. *IBM Syst. J. 36*, 1, 102–123.

MOHAN, C. 1997. Recent trends in workflow management products, standards, and research. In *Proceedings of the NATO Advanced Study Institute on Workflow Management Systems and Interoperability* (Aug.)

ORACLE CORP. 1996. Oracle 7 Server Concepts Manual. Oracle Corp., Redwood City, CA.

PATON, N. 1999. *Active Rules in Database Systems*. Springer-Verlag, New York, NY.

PERNICI, B., Ed. 1989. *Office Information System: the Design Process*. Elsevier Sci. Pub. B. V., Amsterdam, The Netherlands.

PERNICI, B. AND SANCHEZ, G. 1996. The WIDE workflow model. Tech. Rep. 3002-3, WIDE Consortium.

REMEDY CORP. 1996. Action Request System 3.0 Administrator's Guide. Remedy Corp..

REUTER, A., SCHNEIDER, K., AND SCHWENKREIS, F. 1997. Contracts revisited. In *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds. Kluwer Academic Publishers, Hingham, MA.

SAASTAMOINEN, H. 1995. On the handling of exceptions in information systems. Ph.D. Dissertation. University of Jyvaskyla.

SILBERSCHATZ, A. AND ZDONIK, S. 1997. Database systems—breaking out of the box. *SIGMOD Rec. 26*, 3, 36–50.

SOFTWARE-LEY, GMBH. 1996. Cosa Reference Manual. Software-Ley GmbH.

STAFFWARE, CORP. 1997. Staffware Global-Staffware for Intranet based Workflow Automation.

STARK, H. AND LACHAL, L. 1995. Ovum Evaluates: Workflow. Ovum, London, UK.

SUTTON, S. M., JR., OSTERWEIL, L. J., AND HEIMBIGNER, D. 1995. APPL/A: A language for software process programming. *ACM Trans. Softw. Eng. Methodol. 4*, 3 (July), 221–286.

TER HOFSTEDE, A. H. M., ORLOWSKA, M. E., AND RAJAPAKSE, J. 1998. Verification problems in conceptual workflow specifications. *Data Knowl. Eng. 24*, 3, 239–256.

VAN STIPHOUT, R., MEIJLER, T. D., AERTS, A., HAMMER, D., AND LE COMTE, R. 1998. Trex: Workflow transaction by means of exceptions. In *Proceedings of the Sixth International Conference on Extending Database Technology* (Valencia, Spain, Mar.), H. -J. Schek, F. Saltor, I. Ramos, and G. Alonso, Eds.

WIDOM, J. AND CERI, S., Eds 1996. *Active Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

WODTKE, D., WEISSENFELS, J., WEIKUM, G., AND KOTZ DITTRICH, A. 1996. The Mentor project: Steps toward enterprise-wide workflow management. In *Proceedings of the 12th IEEE International Conference on Data Engineering* (New Orleans, LA) IEEE Press, Piscataway, NJ, 556–565.

WORAH, D. AND SHETH, A. 1997. Transactions in transactional workflows. In *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds. Kluwer Academic Publishers, Hingham, MA.

WORKFLOW MANAGEMENT COALITION. 1996. Terminology and glossary. Tech. Rep. WFMC-TC-1011. Workflow Management Coalition.

WORKFLOW MANAGEMENT COALITION. 1998. Process definition interchange v 1.0. Tech. Rep. WfMC-TC-1016-P. Workflow Management Coalition.

ZIMMERMANN, J. AND BUCHMANN, A. 1999. REACH. In *Active Rules in Database Systems* Springer-Verlag, New York, NY.

Received: September 1998; revised: February 1999; accepted: May 1999