Modeling and Managing Interactions among Business Processes

Fabio Casati, Angela Discenza[1]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-159
December 4th , 2000*

E-mail: casati@hpl.hp.com, discenza@elet.polimi.it

Events,
workflows,
interoperability

Most workflow management systems (WfMSs) only support the separate and independent execution of business processes. However, processes often need to interact with each other, in order to synchronize the execution of their activities, to exchange process data, to request execution of services, or to notify progresses in process execution. Recent market trends also raise the need for cooperation and interaction between processes executed in different organizations, posing additional challenges. In fact, in order to reduce costs and provide better services, companies are pushed to increase cooperation and to form virtual enterprises, where business processes span across organizational boundaries and are composed of cooperating workflows executed in different organizations. Workflow interaction in a cross-organizational environment is complicated by the heterogeneity of workflow management platforms on top of which workflows are defined and executed and by the different and possibly competing business policies and business goals that drive process execution in each organization. In this paper we propose a model and system that enables interaction between workflows executed in the same or in different organizations. We extend traditional workflow models by allowing workflows to publish and subscribe to events, and by enabling the definition of points in the process execution where events should be sent or received. Event notifications are managed by a suitable event service that is capable of filtering and correlating events, and of dispatching them to the appropriate target workflow instances. The extended model can be easily mapped onto any workflow model, since event specific constructs can be specified by means of ordinary workflow activities, for which we provide the implementation. In addition, the event service is easily portable to different platforms, and does not require integration with WfMS that supports the cooperating workflows. Therefore, the proposed approach is applicable in virtually any environment and is independent on the specific platform adopted.

* Internal Accession Date Only

[1] Politecnico di Milano, Milan Italy, 20133

Modeling and Managing Interactions among Business Processes

Fabio Casati
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA, USA 94304
`casati@hpl.hp.com`

Angela Discenza
Politecnico di Milano
Via Ponzio 34/5
Milan, Italy 20133
`discenza@elet.polimi.it`

November 30, 2000

Abstract

Most workflow management systems (WfMSs) only support the separate and independent execution of business processes. However, processes often need to interact with each other, in order to synchronize the execution of their activities, to exchange process data, to request execution of services, or to notify progresses in process execution. Recent market trends also raise the need for cooperation and interaction between processes executed in different organizations, posing additional challenges. In fact, in order to reduce costs and provide better services, companies are pushed to increase cooperation and to form *virtual enterprises*, where business processes span across organizational boundaries and are composed of cooperating workflows executed in different organizations. Workflow interaction in a cross-organizational environment is complicated by the heterogeneity of workflow management platforms on top of which workflows are defined and executed and by the different and possibly competing business policies and business goals that drive process execution in each organization.

In this paper we propose a model and system that enable interaction between workflows executed in the same or in different organizations. We extend traditional workflow models by allowing workflows to publish and subscribe to events, and by enabling the definition of points in the process execution where events should be sent or received. Event notifications are managed by a suitable event service that is capable of filtering and correlating events, and of dispatching them to the appropriate target workflow instances. The extended model can be easily mapped onto any workflow model, since event specific constructs can be specified by means of ordinary workflow activities, for which we provide the implementation. In addition, the event service is easily portable to different platforms, and does not require integration with the WfMS that supports the cooperating workflows. Therefore, the proposed approach is applicable in virtually any environment and is independent on the specific platform adopted.

1 Introduction and Motivations

Most workflow management systems (WfMSs) only support the separate and independent execution of business processes. However, processes often need to interact with each other, in order to synchronize the execution of their activities, to exchange process data, to request execution of services, or to notify progresses in process execution.

Interaction among workflows is needed both within and between organizations. While in some cases interacting processes executed in the same organization may be unified within a single workflow, sometimes this approach is not feasible or not satisfactory, the semantics of the processes may require that workflows remain separate, not to introduce constraints which are absent in the business processes they model. As an example, in the following we introduce two processes involving the management of pacemakers and patients: while the two workflows do need to interact, they must remain separate (even when they are managed by the same organization and by the same WfMS), since a patient may change several pacemakers and a pacemaker may be implanted in several patients, thereby changing their "partner" patient process.

Recent market trends also raise the need for cooperation and interaction between processes executed in different organizations, posing additional challenges. In fact, in order to reduce costs and provide better services, companies are pushed to increase cooperation and to form *virtual enterprises*, where business processes span across organizational boundaries and are composed of cooperating workflows executed in different organizations.

Workflow interaction across organization has additional complexity with respect to interaction among workflows of the same organization, due to the heterogeneity of the environments in which processes are executed. In fact, workflows in different organizations are typically specified in different workflow languages, executed on top of WfMSs of different vendors, and managed according to different business policies and business goals. Furthermore, while organizations do need to interact, they want to preserve the *autonomy* of their processes, i.e., they do not want to expose the details of these processes and they want to be able to modify and improve them without affecting cooperating ones.

In this paper we present a *model* for specifying interaction among workflows and a *system* that implements the model and provides the communication infrastructure.

The workflow model extends traditional approaches by allowing the definition of *event nodes*, that denote points in the workflow execution where events should be sent to (or received from) cooperating workflows. Event nodes may be used to synchronize execution of activities, to exchange data between workflows, to notify state changes (i.e., to notify that certain milestones have been reached in process execution), or to request the activation of a remote process. Events can also be used to handle exceptions. In fact, exceptions are often asynchronous with respect to the flow of task executions. For this reason, as observed by many authors (see, e.g., [10, 16, 32]), it is often difficult to capture and represent exceptions within the graph of activities that describes the behavior of the process. Event nodes are instead well-suited, since they can model the occurrence of events that are asynchronous with respect to the execution flow and can activate a separate (exception-handling) part of the flow, as explained in this paper.

Event dispatching is based on a publish-subscribe model, where workflows may notify events to all interested partners and receive events of interest from them. Events may include parameters, whose value is taken from workflow variables of the sender and can be assigned to workflow variables of the receiver. We also provide a simple but powerful language by which event nodes specify filtering rules over events of interest, based on the values of event parameters and on properties related to correlation among event instances.

The core of the system is the *event service*, which is capable of filtering and correlating events and of dispatching them to the appropriate workflow instances. The interaction between the workflow environment and the event service is achieved by means of special-purpose workflow activities that, once invoked by the WfMS, send or request events to the event service and manage data exchange. This approach allows the implementation of event nodes and of the proposed model on top of any existing WfMSs, which require no extension in order for workflows to interact. In addition, the event service is easily portable onto different platforms, and does not need to interface with the WfMSs on top of which interacting workflows are executed. Therefore, the proposed approach is applicable in virtually any environment and is independent on the specific WfMS adopted.

2 A model for the specification of workflow interactions

In order to enable workflow interoperability, we extend traditional, WfMC-like models with the capability of exchanging events with other workflows (or even to generic, non-workflow agents) executed in the same or in different organizations.

Interactions among workflows are defined by means of *event nodes.* Event nodes denote points in the workflow where an event is sent or received, thereby allowing synchronization and data exchange with other processes, as well as notification of state changes or requests of service execution. Event nodes can be part of the flow structure of a workflow, just like ordinary tasks, and can be of two types: *send* (produce) nodes or *request* (consume) nodes. Send nodes notify events to other processes; they are non-blocking, and thus they do not affect the execution of the workflow in which they are defined: as the event is sent, process execution proceeds immediately by activating the subsequent task. Request nodes correspond instead to requests of events to be received from other processes. They are blocking: as a request node is reached in the control flow of a given workflow instance, the execution (along the path where the request node is defined) stops until an event with the specified characteristics is received. Having event nodes explicitly defined as part of the process model allow the process designer to specify what to notify, when, and what data to send along with the event notification. We do not impose any restriction on the event semantics, which are application specific.

In the following, we first present the characteristics of events, and then we show how event notifications and requests are modeled within workflow schemas.

2.1 Event classes

In the event-based model, every event belongs to an *event class*. An event class groups events that share common semantics, and is characterized by a class name, unique within an event service, and by an arbitrary number of parameters with their respective name and type. Parameters can be of type integer, string, real, boolean, datetime, record, or payload (the payload is a generic stream of bytes).

Some parameters are user-defined and application specific, while others are application-independent. Some application-independent parameters must be defined for each event, while others are optional but, when defined, they have specific semantics, known to the event service. Mandatory parameters are:

- *className*: a string, set by the sender process, that identifies the class to which the event belongs.
- *identifier*: an integer, automatically assigned by the event service, that identifies each single event instance.
- *timestamp*: the date and time of the event notification, also automatically assigned by the event service.

Optional parameters are:

- *sourceOrganization*: a string that identifies the organization that produced the event.
- *sourceCase*: a string that identifies the process instance that produced the event.
- *destOrganization*: a string that identifies the organization to which the event is intended.
- *destCase*: a string that identifies the process instance to which the event is intended.

As an example, consider a pacemaker management system, handling the process of building, charging, delivering, maintaining, and replacing pacemakers, and a patient management system, concerning patients whose heart disease is cared by means of pacemakers. We will describe these processes in detail in the following section.

The two processes require a tight interoperability, but are managed by (and executed within) different organizations (patients are managed by hospitals, while pacemakers are managed by companies that produce medical equipment). In addition, a hospital may get pacemakers from several companies, and a pacemaker producer may provide medical devices to several hospitals.

These processes will need to exchange several events in order to interact and synchronize the execution of their activities. For instance, one of these events may inform the pacemaker management process that a given pacemaker has been explanted. For this purpose, an event class `pmExplant` has been defined, with the following parameters (mandatory parameters *identifier* and *timestamp* are implicit and their values are not defined by the process designer but rather assigned by the event service):

```
define event class pmExplant {
sourceOrganization: string
destOrganization: string
serialNumber: integer
explantDate: datetime
physician: string
}
```

2.2 Send nodes

Send nodes denote points in the flow where events are produced. They are characterized by a name, unique within a workflow schema, and by a set of parameters that define the event to be produced. Event parameters are specified by pairs $\langle \mathit{parameter\_name}, \mathit{value} \rangle$. Values may be constants or may refer to the name of a workflow variable, meaning that the actual parameter is set at the value of the workflow variable at the time the event is sent.

As an example, consider the send node `pmExplanted` in the patient workflow. The send node has the purpose of notifying the explant to the organization managing the explanted pacemaker. The send node generates an event of class `pmExplant`, and specifies three parameters (besides the class name): the target organization to which the event is intended, the pacemaker serial number, and the explant date. The syntax used for the definition of the events is as follows:

```
⟨ ⟨ className, "pmExplant"⟩
⟨ destOrganization, pmSenderOrg ⟩
⟨ serialNumber, pmSerialNumber ⟩
⟨ explantDate, pmExplantDate⟩ ⟩
```

The class name is defined by a constant, written between quotes, while the other parameters are specified (in this example) by references to local variables of the workflow instance in which the send node is executed. As the send node `pmExplanted` is reached, the sender workflow generates one event instance with the above parameters, and references to workflow variables are replaced by their respective value, computed at the time the event is raised.

Event producers do not have to specify all the parameters when they notify an event instance of a given class. The only parameter they have to specify is the class name. However, if parameters are specified, they must be a subset of the parameters defined for that class.

2.3 Request nodes

Request nodes denote points in a process where the execution is suspended, waiting for an event to be delivered. Each event request receives at most one event instance, and only events that occur after the request has been issued can be delivered to the node. As the instance is delivered, process execution proceeds with the activation of the task connected in output to the request node.

For each request node, the workflow designer specifies a name unique in the workflow schema and a *request expression*, that defines the kind of events the request nodes is interested in (specified by a *filtering rule*) and the event parameters whose value should be captured within workflow variables (specified by a *capturing rule*). The syntax of the request expression is the following:

```
<request expression> := <filtering rule> [":" <capturing rule>];
```

The filtering rule is mandatory, while the capturing rule must be specified only if the requesting workflow needs to capture the value of event parameters into local workflow variables. In the following we describe these components in detail.

2.3.1 Filtering rule

The filtering rule defines the characteristics of the (composite) event in which the request node is interested. Filtering rules define a constraint over event instances in terms of names and values of event

parameters. In addition, it is possible to define constraints over multiple event instances by requesting *composite events*, defined as sequences, disjunction, conjunction, or negation of event instances. The event service will deliver to the requesting node the first (possibly composite) event instance matching the filtering rule and produced after the request has been issued. If an event instance matches the filtering rule of several requests, it is delivered to all of them.

As an example, consider a pacemaker process managed by the *MediPace* company: the process needs to be notified when the pacemaker it manages is explanted from a patient, in order to trigger the appropriate actions (such as arranging for the shipment of the pacemaker back to the factory, its recharge, etc.) In order to specify this behavior, the pacemaker schema includes a request node `explanted`, that captures all events of class `pmExplant` related to the pacemaker at hand, qualified by the organization name and by the pacemaker serial number. This semantics can be specified by associating the following filtering rule to the request node `explanted`:

```
⟨ ⟨ className, "pmExplant"⟩
⟨ destOrganization, "MediPace"⟩
⟨ serialNumber, pmId⟩ ⟩
```

In the expression above, the event class (`pmExplant`) and the name of the pacemaker company (`MediPace`) are constants, and are independent from the specific process instance. The rule then specifies that the event parameter `serialNumber` must be equal to the value of the workflow variable `pmId` (this will be true for only one workflow instance). No requirement is set on the explant date, meaning that the node is interested in the event regardless of the explant date. At the time the request node is reached by the control flow, the pacemaker process issues a request to the event service, where references to names of workflow variables (e.g., `pmId`) are replaced by the values of such variables. Issued requests are not affected by subsequent changes in the values of these variables.

Formally, a *filtering rule* is defined according to the following syntax:

```
<filtering rule> := "<"  <event filter>  ">"|
                    "Seq" "(<" <event filter> "> , <" <event filter> ">)" |
                    "Or" "(<" <event filter> "> , <" <event filter> ">)"  |
                    "And" "(<" <event filter> "> , <" <event filter> ">)" |
                    "Not" "(<" <event filter> "> , <" <event filter> " > ,
                            <" <event filter> ">)"
```

while an *event filter* is defined as follows:

```
<event filter> := "<" <attribute name> "," <attribute term> ">" [<
event filter>]
```

Attribute terms are constants, workflow variables, or event variables. Workflow variables (denoted by strings starting with a lowercase letter, such as `pmId`) allows the specification of filters that depend on the state of the requesting workflow instance, and are replaced by the value of the variable at the time the request is issued. Event variables (denoted by strings starting with an uppercase letter) allows the designer to specify filtering rules that include correlation constraints, as clarified below.

An event instance matches an event filter (i.e., satisfies the filtering rule and can be delivered to the corresponding request node) when (1) all the attribute names in the event filter occur also in the event, and (2) for each attribute name occurring both in the event instance and in the event filter, the corresponding value in the event instance is equal to the attribute term in the event filter.

If the attribute term is an event variable, then any value matches it, and unification of that variable with the respective value is performed, as in logic programming languages. For instance, the expression `<explantDate, X>` does not actually filter events on the basis of the explant date (it only requires the *explantDate* parameter to be part of the event instance). The filter `<explantDate, X>, <shipmentDate,X>` also does not impose requirements on the explant date in itself, but it does requires that it is equal to the date the pacemaker is shipped back.

The semantics of composition operators `Seq`, `Or`, `And`, and `Not` is defined below. Consider event filters $\mathtt{ef}_1$, $\mathtt{ef}_2$ and $\mathtt{ef}_3$:

- Sequence: $\mathtt{Seq(ef_1,ef_2)}$ is satisfied by two events $\mathtt{e}_1$ and $\mathtt{e}_2$ such that $\mathtt{e}_1$ matches $\mathtt{ef}_1$, $\mathtt{e}_2$ matches $\mathtt{ef}_2$, and $\mathtt{e}_1$ occurs before $\mathtt{e}_2$.
- Disjunction: $\mathtt{Or(ef_1,ef_2)}$ is satisfied by any event that matches either $\mathtt{ef}_1$ or $\mathtt{ef}_2$.
- Conjunction: $\mathtt{And(ef_1,ef_2)}$ is satisfied by two events $\mathtt{e}_1$ and $\mathtt{e}_2$ such that $\mathtt{e}_1$ matches $\mathtt{ef}_1$ and $\mathtt{e}_2$ matches $\mathtt{ef}_2$.
- Negation: $\mathtt{Not(ef_1,ef_2,ef_3)}$ is satisfied by two events $\mathtt{e}_1$ and $\mathtt{e}_2$ such that $\mathtt{e}_1$ and $\mathtt{e}_2$ satisfy $\mathtt{Seq(ef_1,ef_2)}$ and no event that matches $\mathtt{ef}_3$ has occurred between $\mathtt{e}_1$ and $\mathtt{e}_2$.

For a formal specification of the semantics of the filtering language and of the detail of the event composition semantics we refer the reader to the technical report [12].

We now present an example of filtering rule involving an event correlation: in a workflow managing a stock portfolio, the designer might need to define a request node sensible to two successive raise of the same stock title, and perform suitable actions as the (composite) event occurs, such as buying some stocks of that title. The filtering rule can be specified as follows:

```
Not (<<Stock, X> <Raise, R1>>
       <<Stock, X> <Raise, R2>>
       <<Stock, X> <Fall, F>>)
```

The filtering rule is matched by the composite event formed by two raises in the stock price of a title not interleaved by a fall in the price of the same title. The evaluation of the filtering rule binds event variable X to the stock title and event variables R_1 and R_2 to the amount of the two raises. R_1 and R_2 do not actually contribute to the computation of the composite event: the only constraints are imposed by the `not` operator and by the presence of event variable X in all the three event filters, meaning that all events must refer to the same stock title. Event variables R_1 and R_2 will be used instead in the capturing rule to pass data to the requesting workflow instance, as detailed in the following subsection.

2.3.2 Capturing rule

Besides defining the event of interest by means of a filter over event parameters, a request node may also capture the value of event parameters into local workflow variables. For this purpose, the filtering rule is coupled with a *capturing rule*. For instance, assume that the pacemaker process is also interested, upon notifications of explants, in capturing the explant date, needed for tracking purposes. Therefore, the request node `explanted` also needs to capture the value of parameter `explantDate` of event `pmExplant` into the local variable `wfExplantDate`. The capturing rule that defines this behavior is defined as follows:

$\langle\ \langle$ `wfExplantDate, explantDate` $\rangle\ \rangle$

If event variables have been introduced in the filtering rule, the value to be assigned to a given workflow variable can be taken from the value of an event variable, assigned during the evaluation of the filtering rule.

For instance, in the stock raise example, if the workflow designer wants to capture the stock title that had two successive raises as well as the amount of the raises, she can define the following capturing rule (stockTitle, firstRaise, and secondRaise are names of variables in the workflow that requested the event):

$\langle\ \langle$ `stockTitle,` $X\ \rangle$
$\langle$ `firstRaise,` $R_1\ \rangle$
$\langle$ `secondRaise,` $R_2\ \rangle\ \rangle$

The syntax for the definition of the capturing rule is the following:

```
<capturing rule>         := "<" <single capturing rule>  ">"
<single capturing rule>  := "<" <workflow variable name> ","
                             <event attribute term> ">"
                            [<single capturing rule>]
<event attribute>        := <event parameter name> | <event variable name>
```

The event service also takes care of data conversion between the different data type representations adopted by the event service and by the WfMS where the requesting workflow is executed. We will come back on this issue in Section 4.

3 Patients and pacemakers

We next present a case study that illustrates how event nodes are used to model complex processes. The case study involves a patient and a pacemaker management process. The workflow schemas of the two processes are depicted in Figures 1, 2, and 3. They are defined by means of activity graphs (where boxes represent activities and diamonds model decision points), with the addition of send and receive nodes (graphically represented by a circle with an outgoing and

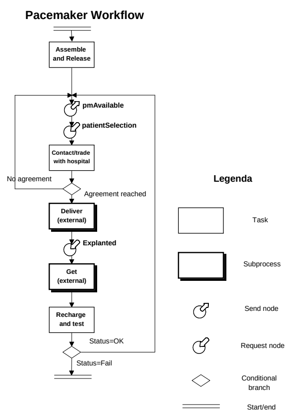

Figure 1: Pacemaker Workflow.

an incoming arrow respectively). We first describe the patient and pacemaker processes, and then detail the events exchanged between the two.

The pacemaker process, managed by a medical equipment company, starts with the construction of a new pacemaker (Figure 1). As the pacemaker is built, a notification is sent to a third-party, non-profit health care organization that supports patients and hospitals in finding pacemakers with the required characteristics. We are not concerned in this example with the business logic of this component, and we do not a priori assume that it is driven by a workflow engine. We only assume that it collects information about available pacemakers and makes it available to hospitals. Next, the pacemaker company waits for a notification from the health care organization that informs it on the identity of the patient and hospital that has selected the pacemaker. After receiving the notification, the pacemaker company contacts the hospital's physician to agree on technical, economical, and logistic issues. If no agreement is reached or if the physician decides, after a discussion with the pacemaker company, that the pacemaker is not suitable, then the pacemaker re-issues the notification of the pacemaker availability. Otherwise, the pacemaker is delivered to the selected hospital. The shipment is performed by another organization, thereby requiring interaction with the business processes of the shipping company, which can also be specified and implemented with the model and architecture presented in this paper, although for brevity it is not discussed here.

After the delivery, the pacemaker is managed by the hospital, and the pacemaker process is required to take no action, until the hospital notifies that the pacemaker has been explanted and that is going to be returned (this can happen if the pacemaker needs to be recharged, or if it is not working properly, or if the patient does not need it any more). After the hospital notifies the explant, the pacemaker company picks-up the pacemaker at the hospital site (again through a shipping company), repairs and tests the pacemaker, and notifies its availability. If instead the pacemaker cannot be repaired, it is trashed and the process terminates.

The patient workflow (Figure 2) starts with the registration of the patient that needs a pacemaker. Next, a suitable pacemaker is selected through the health care organization, and the pacemaker company is contacted in order to check the details about the selected pacemaker and agree on times, terms, and conditions with the pacemaker organization (subprocess `Get pacemaker and schedule surgery` of Figure 3). Then, a surgery is scheduled and a task that manages pacemaker reception and registration is activated. When the pacemaker has been delivered and the surgery has been scheduled, the pacemaker is implanted. Next, the patient undergoes follow up visits, until either the pacemaker is not working properly or it is not needed any more. In the first case, the patient process selects and gets a new pacemaker and schedules the surgery to explant the old pacemaker and implant the new one. If no pacemaker is needed, after the old one has been explanted, the process terminates.

Note that with traditional workflow models it is very difficult to describe such interactions, even between processes executed with the same WfMS. In fact, in commercial WfMSs, workflows are typically executed in isolation with respect to other workflows. Every required synchronization and data exchange between workflows must be designed and implemented, for instance by defining ad hoc tasks that interact with the database or with the WfMS API. Furthermore, interactions would be hidden in the task implementations, instead of being explicitly specified in the workflow, with obvious disadvantages.

In the following, we show how interactions between workflows can be modeled within workflow specifications, we detail the interactions occurring in the patient/pacemaker case study, and we show the architecture and implementation of the system that supports workflow interaction on top of a generic WfMS platform.

3.1 Event nodes in the patient/pacemaker case study

This section summarizes the inter-process interaction occurring in the patient/pacemaker case study. Figure 4 shows the scenario in which the patient and pacemaker workflows operate and interact. We assume the existence of several pacemaker producers and several hospitals. For simplicity, we assume that there is just one health care organization, called *HeartCare*, that helps physicians and hospitals in finding a suitable pacemakers, and we also assume that there is a single event service (possibly managed by the same *HeartCare* organization) through which hospitals and pacemaker producer companies cooperate. We will discuss architectures with several distributed event service components in a following section.

In the pacemaker workflow, as a new pacemaker is built (or after it has been repaired), an event is sent (send node `pmAvailable` of Figure 1) to the *HeartCare* organization in order to notify the availability of the pacemaker. We assume that the *HeartCare* organization accepts and recognizes events of class `newPm` that inform about available pacemakers. Class `newPm` includes parameters `sourceOrganization` and `destOrganization` of type string, `serialNumber` of

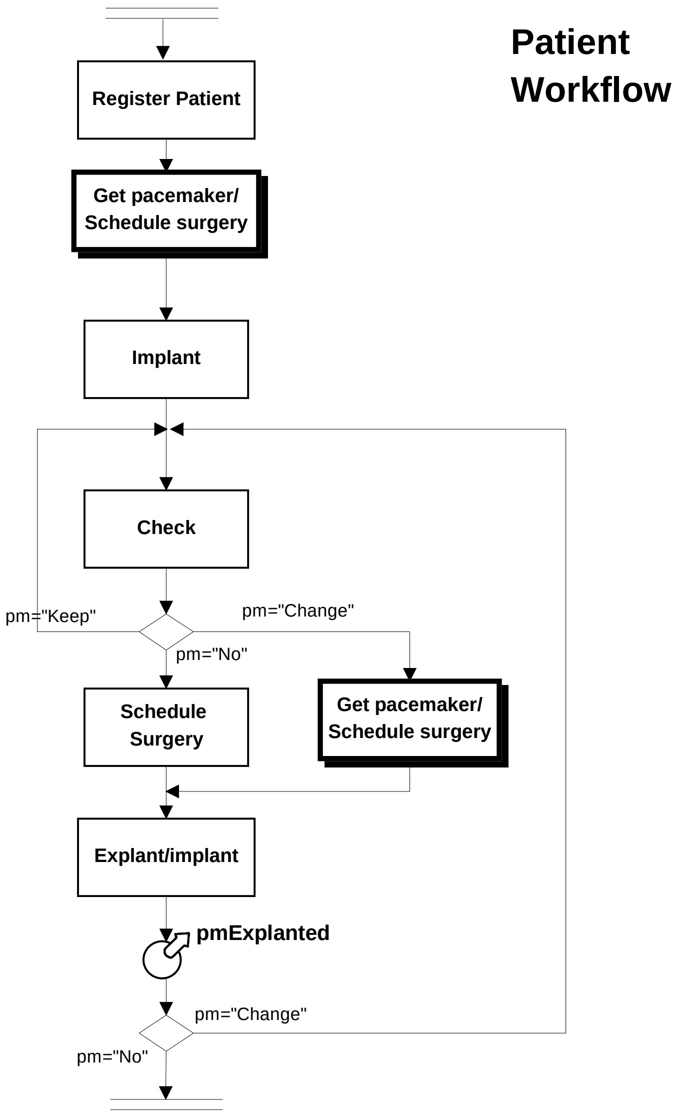

Figure 2: Patient Workflow

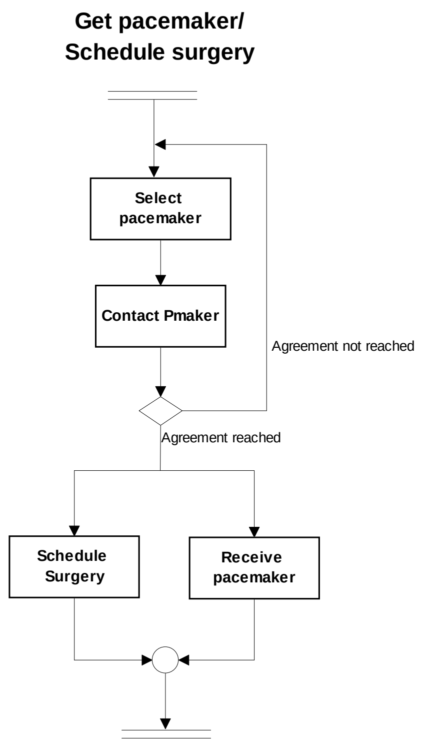

Figure 3: Expansion of the `Select and get pacemaker` subprocess of the `Patient` workflow

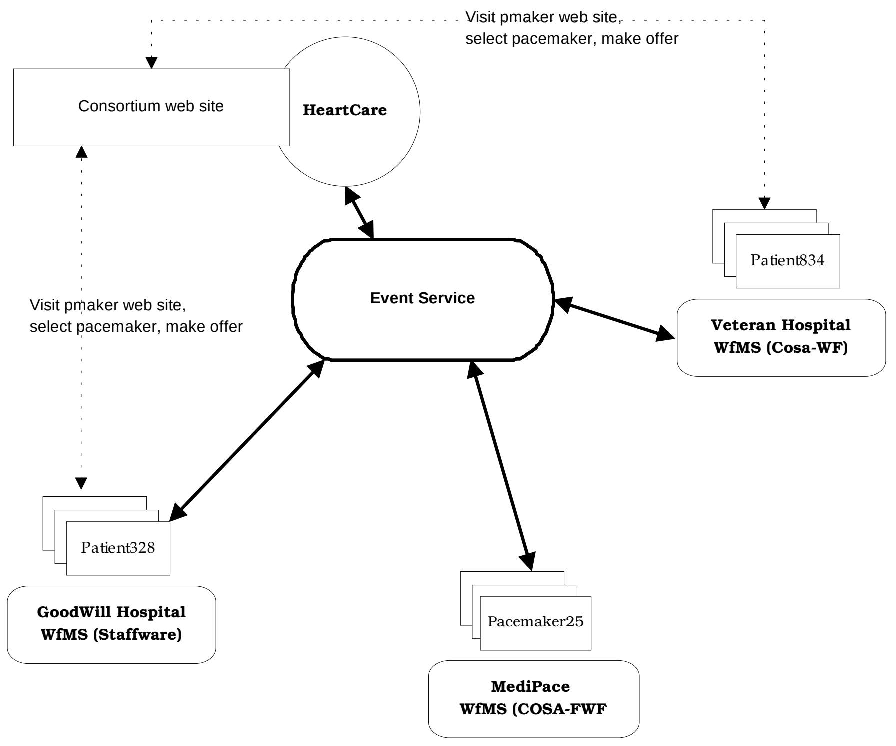

Figure 4: Interactions among patients, pacemaker, and the health-care organization *HeartCare*

type integer, and `dataSheet` of type payload.

Within send node `pmAvailable` of a pacemaker workflow run at the *MediPace* company, the event is formed in the following way:

```
⟨ ⟨ className, "newPm"⟩
⟨ destOrganization, "HeartCare" ⟩
⟨ sourceOrganization, "MediPace" ⟩
⟨ serialNumber, pmSerialNumber ⟩
⟨ dataSheet, pmDataSheet⟩ ⟩
```

Information about the class name and the target organization will be used by *HeartCare* in order to filter the events it is willing to receive (i.e., those of class `newPm` intended to *HeartCare*). Information about the sender organization, the pacemaker serial number, and the data sheet is needed in order to properly modify a web site that includes information available to physicians.

Note that we do not require business processes of the health care organization to be supported by workflow managers. Indeed, workflows can use events to interact with non-workflow applications. In this case, we only require the *HeartCare* company to “listen” to notifications of `newPm` events by issuing requests to the event service filtered as follows:

```
⟨ ⟨ className, "newPm"⟩
⟨ destOrganization, "HeartCare" ⟩ ⟩
```

After notifying the pacemaker availability, the pacemaker process waits for a message from *HeartCare* informing that the pacemaker has been selected by a patient. We assume that *HeartCare* produces events of class `pmSelected` in order to inform that a given pacemaker has been selected for a patient by a physician, also including event parameter `physicianNameAndAddress`, of type string, that the pacemaker process captures into the local workflow variable `contactInfo`.

This behavior is specified by request node `patientSelection`, that has the following request expression:

```
⟨ ⟨ className, "pmSelected"⟩
⟨ destOrganization, "MediPace" ⟩
⟨ serialNumber, pmId ⟩ ⟩ :
⟨ ⟨ contactInfo, physicianNameAndAddress⟩ ⟩
```

Finally, the pacemaker workflow includes request node `explanted`, that captures all events of type `pmExplant` related to the pacemaker at hand, qualified by the pacemaker serial number. The following request expression is associated to the request node `explanted`:

```
⟨ ⟨ className, "pmExplant"⟩
⟨ destOrganization, "MediPace"⟩
⟨ serialNumber, pmId⟩ ⟩ : ⟨ ⟨ wfExplantDate, explantDate ⟩ ⟩
```

`pmExplant` events are produced by the patient workflow (Figure 2) to notify to the pacemaker organization that the pacemaker has been explanted. The event expression of the send node is as follows:

```
⟨ ⟨ className, "pmExplant"⟩
⟨ destOrganization, pmSenderOrg ⟩
⟨ serialNumber, pmSerialNumber ⟩
⟨ explantDate, pmExplantDate⟩ ⟩
```

4 The DERPA event service

This section presents the architecture and implementation of the DERPA event service and shows how the interaction between the workflow environment and the event service is managed. The interested reader is referred to [12] for details on DERPA.

The DERPA event service is based on a publish/subscribe model, and allows generic software agents to interact by producing and consuming events. Agents are qualified as event sources when they produce events and as event receivers when they receive them. An agent can play both roles. The goal of the event service is to capture all the events generated by event sources and to deliver them to all interested receivers. DERPA controls and directs the event traffic according to publication and subscription requests issued by agents. In addition, the event service provides correlation and filtering capabilities, achieved by programming the event service to recognize a small number of relevant events out of sequences of elementary events, according to filtering rules. Events produced through filtering will be called *filtered* events, to distinguish them from the input ones, called *raw* events.

When an event E is published, DERPA checks if any filtered (composite) event $FE_1, .., FE_n$ has also occurred due to the occurrence of E. Next, it delivers each of these occurred raw or filtered events to all the agents who have subscribed to those events. The order in which events are delivered is non-deterministic.

4.1 Architecture and implementation of DERPA

The architecture of the DERPA event service is depicted in Figure 5. DERPA implements two orthogonal features: it filters events and it manages the subscriptions and unsubscriptions. Subscriptions and unsubscriptions are handled by the *Filter Machines Manager*, that generates a Filter Machine for each subscription that it receives. Each subscription includes the request expression, defined according to the syntax defined in Section 2.

DERPA filters events as follows: as a raw event is received, it is inserted into the *Incoming Event Queue.* Raw events are processed one at a time by an *Event Matcher* that filters and correlates events. The Event Matcher is composed of an *Event Distributor* and several *Filter Machines.* The Event Distributor receives

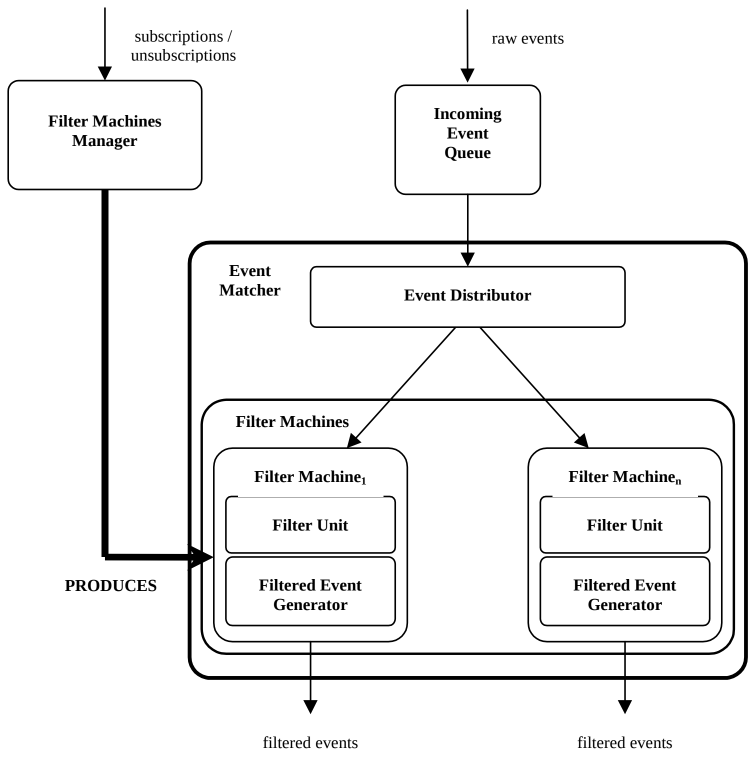

Figure 5: Architecture of the event service

an event from the Incoming Event Queue and produces several copies of the event, that are distributed to the Filter Machines. The number of copies of an event depends on the number of Filter Machines that are waiting for it, which equals the number of subscriptions to events of the same class of the raw events (or that include events of that class in the filtering rule). Each Filter Machine performs two tasks: it has a *Filter Unit*, which verifies whether the event matches the filtering rule, and a *Filtered Event Generator*, which produces one or more events as output of the filtering, including the parameters specified in the capturing rule.

The behavior of the Event Filter can be modeled by means of *statecharts*, derived from the filtering rule. We refer the interested reader to [12] for a detailed and formal description of the behavior of the event service.

A prototype of DERPA has been developed at CEFRIEL and Politecnico di Milano as the evolution and generalization of software components developed in previous projects. The event service has been implemented in Java, and each subscription corresponds to the generation of a new thread Java, that processes raw events, generates filtered events, and delivers them to the appropriate request application. Events are delivered to agents either by invoking callback functions that accepts a *DERPAEvent* object (basically, an hash map) or by sending XML messages over HTTP (this modality is typically used for inter-organizational interactions).

4.2 using DERPA to enable interactions among workflows

We next describe how the DERPA event service is exploited in order to enable interactions among workflows. We recall that one of our main goals is to enable workflow interoperability across organizations by reusing existing, "legacy" WfMSs. The WfMS is unaware of the interaction, which is achieved without the need of modifying the WfMS in place.

The interfacing between the event service and the workflow environment is managed by means of *send* and *request* applications, provided with DERPA, that implement the logic of the send and request nodes. When a workflow designer needs to specify a send node, she actually defines an ordinary workflow task, performed by the send application provided with the event service. The send application (i.e., the task) receives from the calling environment the name, type, and value of the workflow variables as well as the send expression that will be used to form the event to be sent. The send application terminates immediately after the event is sent, thereby allowing case execution to proceed. The application is independent from the specific event being sent, although a different version of the application must be provided for each WfMS type, in order to take into account data conversion issues between the representation adopted by the event service and that of the WfMS.

Similarly, request nodes are modeled by means of ordinary tasks, implemented through a *request* application provided with the event service. Besides event parameters, the request application also receives a well-formed string that describes the request expression and that will be passed along to the event service in order to define the kind of events the request application is interested into. Request nodes, in order to request an event to the DERPA event service, actually *subscribe* to receive events, and informs the DERPA filter about the structure of the events to be received (defined by the capturing rule) and about how raw events should be filtered in order to generate the event (defined by the filtering rule).

4.3 Centralized and distributed configurations

The DERPA event service supports local, centralized, hierarchical, and distributed event service configurations. The choice of the optimal configuration is application-specific, and depends on many factors such as the number and the location of the cooperating processes or the business agreements established among partners. In the patient-pacemaker example we have presented the *centralized* configuration (see Figure 6(b)), where a single event service act as an event broker for several different organizations.

In the context of a single WfMS, a *local* event service can be exploited to allow interoperation among the workflows in the WfMS (see Figure 6(a)). A local event service ignores parameters *sourceOrganization* and *destOrganization*, since events are always sent and received locally.

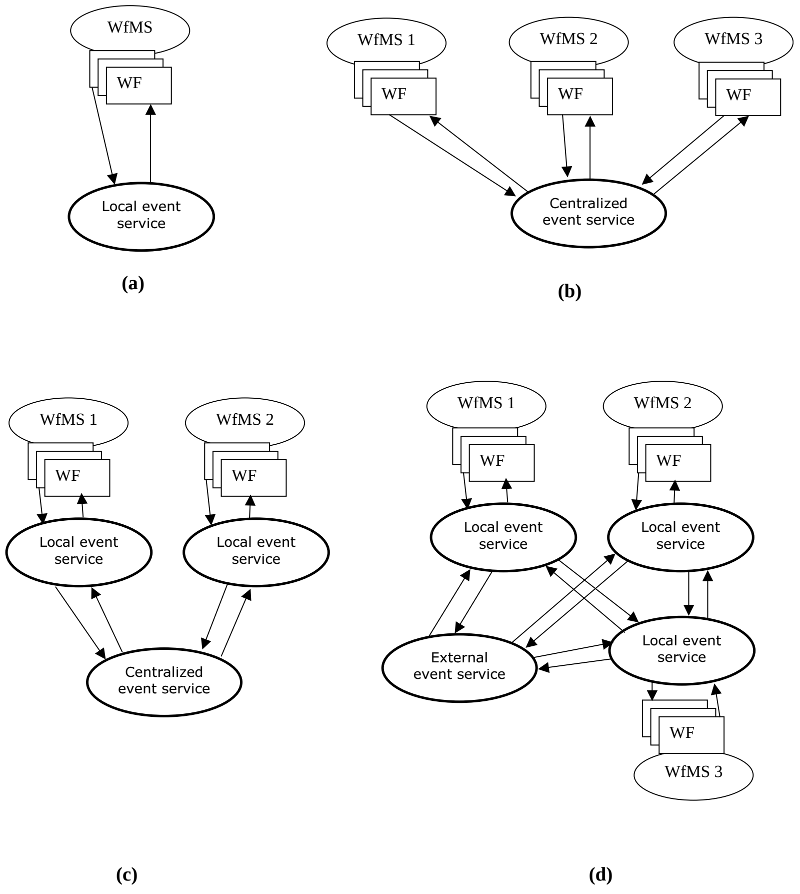

Figure 6: Event service architectures

The *hierarchical* configuration (Figure 6(c)) combines the local and the centralized approach: a local event service is used to exchange events among workflows in an organization, while a centralized event service distributes events among the local event services. When the value of the *destOrganization* (*source-Organization*) parameter has not been defined or has been set to the special keyword `local`, the event notification (request) is managed locally, and the event can be only notified to (received from) a local workflow. When the organization name is specified (or the special keyword `external` is used), the event is forwarded to (requested from) the centralized event service. The keyword `any` denotes instead that the node is willing to receive (send) events from (to) both local and remote event services. Forwarding a request for an event (i.e., a filtering rule) to the centralized event service actually causes the local service to subscribe to events matching the filtering rule at the centralized service. When an event matching that rule is notified to the centralized service, it is in turn forwarded to the local service that subscribed to it, that will in turn forward it to the local requesting workflow.

Publish/subscribe event-based architectures naturally allow for a fully *distributed* architecture (Figure 6(d)), where event services can subscribe in order to receive selected events from other event services. This is also possible with DERPA, that is indeed capable of managing publications and subscriptions. In a fully distributed architecture there is no a priori, fixed topology. However, each event service must be configured in order to specify the set of event service(s) to which the event requests or notifications should be forwarded. The specification can be made at the event service level, but it is also possible to specify a different set of event services to notify or request for each event class.

5 Related work

Several hundreds business process models have been proposed by the research literature and by commercial WfMSs in the last decade. Most of them (and the totality of the one supported by successful commercial products, such as Changengine [17], MQ Workflow [18], Staffware [29], Forte [21], Cosa[3], or InConcert [23]) are based on activity graphs, since this seems to be the referred way for users to model business processes. However, the large majority of these activity-graph models do not allow for the specification of any kind of workflow interactions. A few of them, such as Staffware and Cosa, allow the definition of simple events (qualified by their name) in the flow. *COSA*, by Cosa Solutions [3], includes in its workflow model the notion of *trigger*, defined as an event-action rule that can be triggered by external events or upon deadline expiration, and can react to the triggering event by activating a task or a new (sub)process instance. *InConcert*, by InConcert Inc. [23], 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*, by Staffware Corporation [28], a special kind of task called *event step* can be defined. The event step suspend 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.

These approaches have several limitations with respect to supporting workflow interactions:

- the designer can only model request for events. No facility for sending events is provided;
- when an application (or the system administrator) need to send an event, they explicitly specidy the process instance to which the event is directed.

It is not possible to raise an event and let all interested process instances capture the event, and it is necessary to have a priori knowledge of which instance is interested in which event. In large-scale systems, this is unfeasible;

- there is no provision for event filtering and for capturing event data into process variables.

Recently, the issue of interactions among processes has received more attention in the research community, especially in the context of cross-organizational workflows. However, many projects are still in their infancy and many issues are still to be identified and resolved. One of the first contribution in the field of interorganizational workflows comes from Van Der Aalst [31]. In that paper, the author presents a taxonomy of possible interaction types: *chained execution* occurs when the completion of a process on one system should trigger the activation of a process on another system. *Subcontracting* occurs when a subprocess is assigned for execution to a remote WfMS, possibly operated by a different organizational entity. In *case transfer*, each WfMS has a copy of the workflow specifications, and cases can be transferred from a WfMS to another one. Finally, *loosely coupled* interaction occurs when two or more parts of a process managed by different WfMSs may be concurrently active. The coupling is loose since each process definition only needs to make interaction points publicly available, while it is free to change the remainder of the workflow at will. The paper then focuses on loosely coupled workflows: in particular, it defines an interorganizational workflow as a set of loosely coupled workflows and an interaction structure. The interaction structure basically allows the definition of two types of dependencies among activities in component workflows: *asynchronous* dependencies allow data exchange and the specification of causal dependencies among activities. (e.g., activity B in process instance P_2 can only be executed after activity A in process instance P_1 has been completed.) *Synchronous* dependencies allow the designer to specify that two specific tasks should be activated at the same time. Our event-based approach also allows the definition of synchronous and asynchronous dependencies, but it is more flexible in that it allows for *dynamic* virtual enterprises, where a process is not required to know in advance the processes it will interact with. This is an important requirement in a dynamic environment such as the Internet.

The problem of interorganizational processes is also addressed by the WISE project [1, 2]. WISE aims at developing an infrastructure for business-to-business electronic commerce. The WISE architecture includes a component for the specification of virtual business processes, a component for their enactment, a component for process monitoring and analysis, and finally a component that manages context-aware communication among process participants. These papers do not however present a concrete process model for the specification of cooperating processes and do not show how the interaction is achieved.

Virtual enterprises are also addressed by *CrossFlow* [11], a newly started Esprit project aiming at the definition of an infrastructure for crossorganizational workflows. CrossFlow assumes a centralized description of the process that is to be executed in cooperation, which is then translated into the workflow languages of the participating WfMSs. Suitable gateway components, configured according to the business agreements that have been reached among participating organizations, manage the interactions among the WfMSs. The approach of [19], also developed within CrossFlow, focuses on a particular form of workflow interaction, where an organization refers to another organization for the execution of part of its business process (subcontracting, in the terminology of [31]). An organization acts as a service provider, receives service requests and input data, carries out the requested process (possibly by periodically notifying progresses in process execution), and eventually completes the service, returning output data to the calling organization. The proposed approaches are interesting, although the papers only provide high-level descriptions but do not present any concrete workflow interaction model.

Eder and Panagos [13] propose an event based infrastructure to support cooperative workflows. In their approach, workflow participants, workflow engines, and workflow administrators can subscribe to several types of events, possibly published by different workflow engines. Allowed events are process state changes, activation and completion of activities, and changes in the organization. By subscribing to this kind of events, a workflow engine becomes aware of advancements in the execution of processes enacted by other engines, and can use this information to trigger the execution of activities in its own processes. Our approach differs in that we allow the process designer to define application-specific events and to specify when they should be notified (as well as which data they should carry along), thereby providing for autonomy of cooperating processes. In addition, we allow event filtering and the definition of composite events by means of a powerful filtering and correlation language.

A very interesting and detailed approach to process management in virtual enterprises has been developed in the context of the CMI project at MCC [14]. The project focuses on methods and tools for defining processes that compose *services* provided by different companies. The authors present an advanced workflow model with several new primitives for managing coordination among services. One of the main features of the model is that it allows, for each service, the definition of application-specific states (e.g., *loan requested* or *loan approved* for a loan management service) and operations (e.g., *cancel loan request*). When the designer specifies a process by composing services, she can define control flow conditions based on the (application-specific) states of component services, and can specify when (application-specific) operations should be invoked on the component service. CMI does not aim at designing a new workflow management system from scratch: instead, they assume that processes are enacted by a commercial WfMS, and that the advanced model will be mapped on top of the selected WfMSs. Our approach is similar, in that we also allow application-specific notifications of advancement in process execution, we allow a high degree of autonomy of cooperating processes (services), and we do not aim at designing a new workflow management system. However, we do not define a new complete workflow model but rather provide some extensions to existing ones, that can be applied to any existing workflow model. The mapping of such extension is very simple, and therefore the extended model can be easily implemented on top of any WfMS. In addition, the event-based model and the filtering language allows for greater expressive power, since processes can also exchange application-specific data and since the event service can monitor composite events, i.e., can monitor conjunctions, disjunctions, or sequences of process events.

In summary, the original features of our model with respect to existing approaches are the following:

1. we introduce events as part of the workflow model, in order to make explicit in the workflow schema when and how workflows interact.

2. we allow the use of user-defined events which include event parameters, that are set according to the state of the sending workflows and affect the state of the receiving workflows;

3. we allow workflows to specify complex filtering and correlation conditions in order to identify the events of interest.

4. we define an approach that is immediately applicable with current technology and with "legacy" systems;

5. through the publish-subscribe approach, the model allow for *dynamic* interaction, so that the workflow designer does not need to know (and to specify) at compile time which are the interacting processes.

Event-based workflow interoperability requires an event service that manages event notifications and subscriptions and dispatches event to the appropriate receivers. Event services are becoming very popular, and a number of models, systems, and architectures have been proposed as a natural mechanisms for achieving interoperability between software applications.

Many event services, based on publish/subscribe models or on its variations have been developed in recent years, (e.g., RAPIDE [20], Polylith [25], Field [26], ToolTalk [30], Softbench's BMS [15], GEM [22]), also by standardization bodies such as OMG [24], and by the active database and Telecommunication Management Network communities (see. e.g., the active databases Snoop [9] and Chimera [8], and the network event correlator platforms ECS by Hewlett-Packard [27] or SMARTS by InCharge [33]).

For supporting our model, we needed an event dispatching service capable of managing publications and subscriptions, of allowing the definition of filtering rules over events and their parameters, and of performing event correlation, in order to enable the specification of interest in composite events. In addition, we needed a system that is easily portable over different platform, in order to provide a component that could be used in any environment where the interaction take place. For these reasons, we exploited the capabilities of the DERPA event service [7], which already provides many of these features, such as advanced event filtering and correlation facility, and we extended it in order to make it applicable for our purposes.

6 Concluding remarks

In this paper we have presented a model and system that enable interaction between workflows executed in the same or in different organizations. We have extended traditional workflow models by allowing workflows to publish and subscribe to events, and by enabling the definition of points in the process execution where events should be sent or received. Event notifications are managed by a suitable event service that filters, correlates, and dispatches them to the appropriate target workflow instances. The model can be used to define a variety of behaviors, by exploiting (possibly pre-defined) workflow fragments: for instance, in [4] we present a set of workflow fragments that allow the definition of many typical interaction structures. Events are also the base for managing asynchronous behaviors and for handling *expected exceptions*, as discussed in [5, 6].

A distinguished feature of the proposed model is that it can be easily implemented on top of any WfMS, since event specific constructs can be specified by means of ordinary workflow activities, for which we provide the implementation. In addition, the event service has been developed in Java, in order to ease its portability, and does not require integration with the WfMS that supports the cooperating workflows.

In our future work, we will address the development of methods and tools that enable the conversion of DERPA events into several different XML formats, as defined by B2B standards such as RosettaNet, CBL, or cXML. These conversions will be performed by suitable DERPA agents. Another orthogonal issue in our research agenda is related to the DERPA event service, and involves the definition and implementation of security features, managing authentication of the involved organizations and secure notification of events over the Internet.

References

[1] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. Processes in electronic commerce. In *ICDCS Workshop on Electronic Commerce and Web-Based Applications (ICDCS 99)*, Austin, Texas, USA, May 1999.

[2] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. WISE: Business to business e-commerce. In *Proceedings of RIDE-VE'99*, Sidney, Australia, Mar. 1999.

[3] *COSA Reference Manual*, 1998.

[4] F. Casati. *Models, Semantics, and Formal Methods for the Design of Workflows and Their Exceptions.* PhD thesis, Dipartimento di Elettronica e Informazione, Milano, Italy, Dec. 1998.

[5] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Transactions on Database Systems*, 1999. to appear.

[6] F. Casati and G. Pozzi. Modeling and managing exceptions in commercial workflow management systems. In *Proceedings of CoopIS'99*, Edimburgh, UK, Sept. 1999.

[7] S. Ceri, E. Di Nitto, A. Discenza, A. Fuggetta, and G. Valetto. DERPA: a generic distributed event-based reactive processing architecture. Technical report, CEFRIEL, Mar. 1998.

[8] S. Ceri, R. Meo, and G. Psaila. Composite events in Chimera. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, Mar. 1996. Springer-Verlag, Berlin.

[9] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts, and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 606–617, Santiago, Chile, 1994.

[10] D. Chiu, K. Karlapalem, and Q. Li. Exception handling with workflow evolution in "adome-wfms": a taxonomy and resolution techniques. In *Proceedings of the First Workshop on Adapive Workflow Systems*, Seattle, Washington, USA, Nov. 1998. Available at http://ccs.mit.edu/klein/cscw98/paper06.

[11] CrossFlow. Esprit project n. 28635. Information available from www.crossflow.org, 2000.

[12] A. Discenza. Filtering events in a distributed architecture. Technical Report 98.079, Dipartimento di Elettronica e Informazione, 1998.

[13] J. Eder and E. Panagos. Towards distributed workflow process management. In *Proceedings of the WACC workshop on Cross-organizational workflows*, San Francisco, CA, USA, Feb. 1999.

[14] D. Georgakopoulos, H. shuster, A. Cichocki, and D. Baker. Managing process and service fusion in virtual enterprises. *Information Systems*, 1999. to appear.

[15] C. Gerety. HP SoftBench: A New Generation of Software Devlopment Tools. *Hewlett-Packard Journal*, 41(3):48–59, 1990.

[16] C. Hagen and G. Alonso. Flexible exception handling in the OPERA process support system. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998.

[17] *Changengine Process Designe Guide*, 2000.

[18] *MQ Series Workflow - Concepts and Architectures*, 1998.

[19] J. Klingemann, J. W. sch, and K. Aberer. Adaptive outsourcing in cross-organizational workflows. Technical Report REP-IPSI-1998-30, Dipartimento di Elettronica e Informazione, GMD-German National Research Center for Information Technology, Aug. 1998.

[20] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995.

[21] J. Mann. Forte' fusion. *Patricia Seybold Group report*, 1999.

[22] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), June 1997.

[23] R. T. Marshak. Inconcert workflow. *Workgroup Computing report, Patricia Seybold Group*, 20(3), 1999.

[24] Object Management Group. Event Service Specification. OMG Document. Available at ftp://ftp.omg.org/pub/docs/formal/97-02-09.ps, Mar. 1995.

[25] J. Purtilo. The POLYLITH software bus. *"ACM" Transactions on Programming Language and Systems*, 16(1):151–174, 1994.

[26] S. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, 7(4), July 1990.

[27] K. Sheers. HP OpenView event correlation service. *Hewlett-Packard Journal*, Oct. 1996.

[28] Staffware Corporation. *Staffware Global - Staffware for Intranet based Workflow Automation*, 1997. Available at http://www.staffware.com/home/whitepapers/data/globalwp.htm.

[29] *Staffware2000 White Paper*, 1998. Available at http://www.staffware.com/home/products/Staffware2000WP.zip.

[30] Sun Microsystems, Inc. Remote Method Invocation Specification. Available at http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html, 1999.

[31] W. van der Aalst. Interorganizational workflows: an approach based on message sequence charts and Petri nets. *Information Systems*, 1999. to appear.

[32] R. van Stiphout, T. D. Meijler, A. Aerts, D. Hammer, and R. le Comte. TREX: Workflow transaction by means of exceptions. In *Proceedings of the EDBT Workshop on Workflow Management Systems*, Valencia, Spain, Mar. 1998.

[33] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. Available at http://www.smarts.com/products.html, 1997.