

Semantic WorkFlow Interoperability

F. Casati, S. Ceri, B. Pernici, G. Pozzi

Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. Da Vinci, 32 - I20133 Milano, Italy
ceri/casati/pernici/pozzi@elet.polimi.it

Abstract. A workflow consists of a collection of activities which support a specific business process; classical examples range from claim management in an insurance company to production scheduling in a manufacturing company to patient care management and support within an hospital. In conventional workflow systems, each business process is separately specified and autonomously supported by a workflow management system, which drives and assists computer-supported activities. However, business processes often interact with each other; in particular, activities which are performed in the context of one process may influence activities of a different process in a way that, laying “between” the two processes, is quite difficult to formalize and understand.

Therefore, a new challenging area for research consists of studying the process of workflow interoperability, i.e. to focus on the interactions among workflows which are autonomously and separately specified, yet need each other’s support. This paper is focused on the semantic specification of workflow interoperability, and provides a classification of “modes of interaction” and of the semantic properties of cooperating workflows. We introduce some topological properties of cooperative workflows, e.g. reachability, deadlock and starvation, and discuss “workflow integration”, i.e. links established between workflows that allow us to view an integrated process from its component processes.

1 Introduction

Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities. A task defines some work to be done by a person, by a software system or by both of them. Specification of a workflow (**WF**) involves describing those aspects of its component tasks (and the processing entities that execute them) that are relevant to control and coordinate their execution, as well as the relations between the tasks themselves.

Information in a WF mainly concerns when a certain work task (**WT**) has to start, the criteria for assigning the WT to agents and which WTs it activates after its end; therefore, less emphasis is placed on the specification of the WT itself (which may be informally described and only partially automated), while the focus is on the WTs’ coordination. Connections among WTs can be rather complex, and some of them may be run-time dependent. A Workflow Management System (**WFMS**) permits both to specify WFs and to control their execution. During a WF execution (enactment), a WFMS has to schedule WTs (including

their assignment to users) on the basis of the (static) WF specifications and of the (dynamic) sequence of events signaling the completion of WTs and of generic events produced within the execution environment (including exceptions).

For the specification of WF behavior, we have presented in [8] a conceptual WF model, inspired by a rich literature on WF specification [1, 13, 15, 18, 21, 24, 25]. The model includes a large collection of constructs for specifying WT interactions, enables the specification of accesses to external databases, and supports preconditions and exceptions. All these concepts are formalized in a Workflow Description Language (WFDL), which is presented in [8] together with graphical notation for visualizing flow interconnections.

A critical problem in the conceptual characterization and understanding of WFs concerns the interoperability among different WFs, i.e. the interaction between WF applications which, although autonomously and separately designed, may interact. “Controlled interoperability” may translate into a fruitful interaction, whereas “unplanned interoperability” may cause problems such as deadlock, starvation, nontermination, or failure to terminate in the desired final state.

The problem of WF interoperability has many facets, ranging from technological issues about how to integrate different WFMS of different vendors on different platforms to the purely conceptual issues of specifying how the interaction should occur. This paper is focused on the semantic specification of WF interoperability, i.e. on the abstract specification of conceptual mechanisms by means of which WF interaction may be fully understood, and potential problems may be removed. Thus, we assume that several WFs are autonomously specified by means of the same conceptual model. In this context, all interoperability problems between WFs are confined within a conceptual context and relate to the semantics of WFs. This problem is quite similar to the problem of schema integration in conceptual databases, which occurs when designing applications on federated databases; and, with no surprise, we recognize not only similar problems, but also similar solutions. In particular, this paper progressively introduces a notion of WF integration that serves the purpose of explaining interactions among WF schemata, and a methodology for achieving integrated WF schemata.

1.1 Previous Related Work

Office modeling techniques [6] proposed the first descriptions of WFs as extensions of Petri Nets, flowcharts and production rules. Process modeling in the Software Engineering field brought to a closer relationship between modeling and enactment of processes: several WF specification languages [18, 13] and several process modeling tools to “animate” WFs [2] were introduced.

Interest in connecting WF systems to existing information systems has recently increased, in order to interconnect to existing data and to cope with large volumes of information. The challenge posed by WF management pushes towards removing some of the classical limitations of databases in the context of concurrency and transactional models [21]. Active rules were indicated as a particularly promising operational model for WFs [11, 27]: active rules for WT

enactment are presented in [7, 9]. The field of WF interconnections and interactions, as well as that of exceptions and access to shared databases, still needs going into depth.

1.2 Paper Outline

In order to make this paper self-consistent, Section 2 summarizes the Workflow Conceptual Model presented in [8]. Section 3 introduces two examples of WF schemata that illustrate many of the issues arising in WF interoperability. Section 4 introduces several properties that may help understanding the collective behavior of WF enactments; these include all possible causes of nontermination and three possible interaction schemes: cooperation, competition, and interference. Section 5 discusses WF integration and gives some preliminary guidelines on how to achieve integration. Finally, Section 6 sketches out some final remarks.

2 Conceptual Workflow Model

The conceptual WF model enables the design of *WF schemata*, which contain the specifications of a collection of WTs and of the dependencies between them. This specification indicates which WTs should be executed, in which order, which agent may be in charge of them, and which operations should be performed on external databases. Intertask dependencies are specified using a restricted number of constructs: sequence, alternatives, parallelism such as *fork* and *join* [2, 11, 24]. The behavior of each WT is formally described by listing its preconditions, its actions, and its exceptional conditions during its execution. The peculiar feature of the proposed WF model is to enable, within WTs' conditions, actions, and exceptions, the manipulation of databases by means of standard SQL2 statements.

We call *WF instance* (or "WF case") any execution (enactment) of a WF schema: a WF may control the evolution of a given object which is relevant to an activity. For example, a WF schema may describe the process of a patient that needs a pacemaker and is periodically followed up: a WF case is created whenever a patient is admitted to an hospital and a pacemaker is implanted. Thus, several WF cases of the same WF schema may be active at the same time.

A WF schema is described by means of a **Workflow Description Language** (WFDL), defined in [8]. Each WF schema is composed of two sections: descriptions of flows, showing intertask relationships, and descriptions of WTs; each section starts with definitions of constants, types, variables, and functions. In WFDL, all types are either atomic types or records of atomic types. Definitions in the context of flow descriptions are global (visible to every WT in the WF), while definitions in the context of WTs are local; for details, see [8]. In both cases, variables exist only during an execution of the WF or WT instance, and are not visible outside the scope of a peculiar WT or WF instance.

The flow declaration may also refer to persistent data (**DB**) which are shared by all WF agents and possibly by agents of other WFs. These data are usually

defined externally (i.e. they are part of the information system of the organization and their existence is independent of the particular WF being modeled); for the sake of simplicity, we use the relational data model to denote persistent data. DB manipulation and retrieval is the only way of exchanging data with other WFs. The complete description of the conceptual model can be found in [8].

2.1 WorkTask Descriptions

Each WT has five major characteristics:

- *Name*: a string identifying the WT.
- *Description*: few lines in natural language, describing the WT.
- *Preconditions*: a boolean expression of simple conditions which must yield a truth value before the actions in the WT can be executed. Simple conditions may either contain (conventional) boolean expressions in WFDL, or be based on the (boolean) query **exists** on a SQL2 statement, which is interpreted as **false** if the corresponding query is the empty relation, and **true** otherwise.
- *Actions*: sequence of statements in WFDL which serves as a *specification* of the intended behavior of the WT. WFDL actions describe data manipulations of temporary and persistent WF data occurring while the WT is active; therefore, WFDL includes instructions for getting input from agents, for manipulating the content of WT or WF variables, and for retrieving and manipulating shared databases (performed by means of SQL2 **update queries**).

The user executing the WT has full freedom on the way the WT itself should be executed, provided that eventually the actions which are listed in its action part are performed.

- *Exceptions*: in every WT it is possible to specify a set of pairs <Exception, Reaction> to handle abnormal events: every time an exception is raised, the corresponding reaction is performed. An exception is a WFDL predicate, which may include time-related and query predicates. All exceptions are monitored by the WFMS; when they become true (possibly at the start of the WT), the exception is raised. A reaction is next performed by the WFMS to handle the exception. Reactions can be selected among a restricted set of options that includes **end** (imposes the termination of the WT), **cancel** (the WT is canceled), **notify** (a message is escalated to the person responsible for the WT) and a few others; a detailed description of available reactions is available in [8]. A typical exception is raised when a WT is not completed within a specified deadline, and a person is notified.

WTs are graphically represented by boxes, separated in four sections, giving the precondition, the name and description, the action, and the exception, respectively.

Each WT may be fully automated (i.e. when it can be performed by a software program or a machine) or be assigned to one agent. Note that we require

each WT execution to be under control of one specific agent; this is a requirement that should be taken into account in designing WT granularity.

2.2 WorkFlow Descriptions

A WF description consists of several interconnections between WTs, which are formally defined in WFDL and also graphically described.

Two WTs may be directly connected by an edge, with the intuitive meaning that as soon as the first one ends, the second one is ready for execution. In all other cases, connections among WTs are performed by *routing tasks* (RT). RTs are classified into:

- *fork tasks* (FT), for initiating concurrent WT executions
- *join tasks* (JT), for synchronizing WTs after concurrent execution.

A FT is followed by many WTs, called *successors*. FTs are classified as:

- *Total*: after the FT is activated, all successors are ready for execution.
- *Non deterministic*: the fork is associated with a value k ; the FT selects nondeterministically k successors for execution.
- *Conditional*: each successor is associated with a condition; the FT instantaneously evaluates conditions and only successor WTs with a **true** condition are ready for execution.
- *Conditional with mutual exclusion*: it adds to the previous case the constraint that only one condition can be **true**; thus, after the predecessor ends, if no condition or more than one conditions are **true** an exception is risen, otherwise one of the successors is ready for execution.

A JT is preceded by many WTs, called its *predecessors*. JTs are classified as:

- *Total*: JT is activated only after the end of all predecessors.
- *Partial*: the JT is associated with a value k ; after the end of k predecessor WTs the JT is active. Subsequent ends of predecessor WTs have no effect.
- *Iterative*: the JT is associated with a value k ; whenever k predecessor WTs end the JT is active. Iterative JTs are implemented by counting terminated WTs and resetting the counters to zero whenever a successor becomes ready. Iterative join with two predecessors and $k = 1$ is used to describe cycles.

The above values k may be associated with constants, variables, or functions expressed in WFDL; in the last two cases, their value becomes known at execution time.

Each WF schema has one start symbol and several stop symbols; the start symbol has one successor WT and each stop symbol has one predecessor WT. WFDL includes also modularization mechanisms (called “supertask” and “multitask”) which are omitted from this paper [8]. Figure 1 represents the adopted graphical symbology for WF representation.

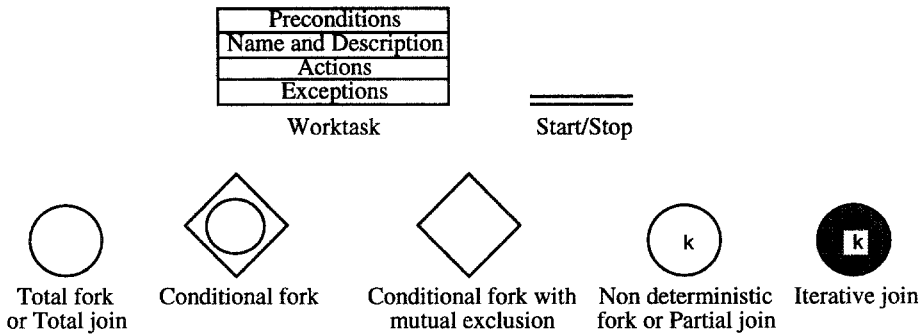


Fig. 1. Graphical symbols for flow interconnections.

2.3 WorkFlow Enactment

A conceptual WF schema describes the legal behaviors of WF executions, called **WF enactment**. An operational semantics of WF enactment is given, by means of active rules, in [9]. Each WF instance, or *case*, is explicitly initiated by an agent and is conducted by assigning WTs to agents according to the WF schema, until the final WTs are reached; at that point summary information about the case is recorded and the case is completed. We assume a WFMS architecture composed of two cooperative environments, one dedicated to WF coordination (*WF management environment*) and one to WT execution (*WT environment*). Details about the architectural environment in which WFs are executed are not of concern in this paper, and can be found in [8, 9].

Figure 2 depicts WT execution by means of a state-transition diagram. A WT becomes *ready* for execution either because it is the first WT of the case or due to the completion of some predecessors; the WF schema allows the WF manager to decide whether a given WT is *ready*. If the WT has no precondition or if the precondition is **true**, then the WT becomes *active*. If instead the WT's precondition is **false**, then the WT's state becomes *inhibited* and the WFMS waits for some external event which may change the truth value of the precondition before the WT becomes *active*. However, exceptional conditions may cause the WT termination from the *inhibited* state.

When a WT is *active*, its execution is controlled within the WT environment. Its state evolves into *executing* as soon as an agent starts operating on the WT (for instance, by opening a window on his screen which corresponds to the WT). The agent can suspend execution, by entering a *suspended* state, and then resume execution.

WT termination is represented by three exclusive states: a WT can be *ended* or *canceled*, in which case WF enactment continues by determining which WT, if any, becomes *ready* due to this event according to the WF schema; the two states are distinguished because in the former case the WT's activities are assumed to be positively accomplished, while in the second case the WT's activities are not accomplished. A WT can also be *refused* by its agent, in which case the WFMS

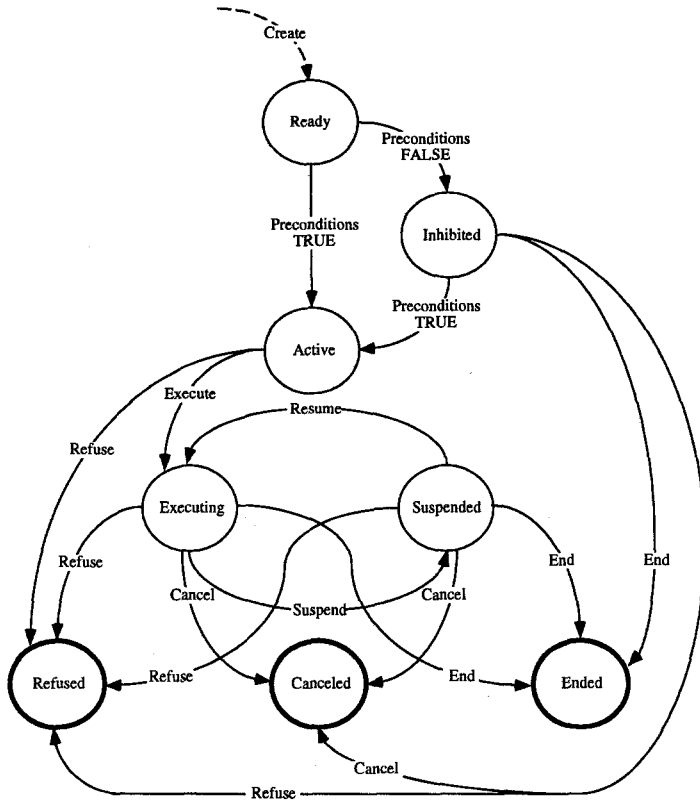


Fig. 2. State-transition diagram describing WT execution (*Refused*, *Canceled*, *Ended* are final states).

must re-assign it to a different agent. When a WT is *active*, *inhibited*, *executing*, or *suspended*, it can be forced to a final state by exceptions which are generated by the WFMS.

3 WorkFlow Interoperability

WF interoperability is concerned with defining the interaction which occurs between distinct cases. Complex interactions may indeed occur between cases of the same WF schema, thus yielding to interoperability issues which are internal to one specific WF application; however, the most difficult problems arise when the interaction occurs between cases from different WF schemata and are caused by applications which are independently and autonomously designed. In this paper, we focus on conceptual interoperability; we assume that each WF schema is described by means of the same WF model, described in Section 2.1 and 2.2, and that its enactment conforms to the rules given in Section 2.3.

The most relevant feature of the conceptual WF model as it concerns inter-

operability is that each case has no access to control variables of other cases, be them of the same WF schema or of a different schema. The only interaction which may occur between cases, documented on WF schemata, is by means of the access to shared databases. This feature occurs in practice, as WFMS are components of enterprise-wide information systems; as we will see, it enables us to identify and focus on a limited number of sources of interaction.

In order to introduce the problems of WF interoperability, we present two examples of tightly integrated WFs. We describe a patient management system, illustrating the sequence of activities performed by patients whose heart disease is cared by means of pacemakers, and a pacemaker management system, illustrating the process of acquiring, charging, implanting, maintaining, and replacing pacemakers. The two processes have a tight interoperability¹.

3.1 Patient Workflow

The precondition to start the WF case, i.e. to admit the patient *Pt*, states that a functioning pacemaker (*PmFree.Status* = ‘‘OK’’) must be available (recorded as a tuple in the table *PmFree*). The patient is then registered (*get Pt*) and assigned a pacemaker chosen among available ones (*Pm1* = *select-one* ...). The chosen pacemaker *Pm1* is no longer available to other patients (*delete from PmFree* ...).

Pacemaker *Pm1* is then implanted into patient *Pt*; this activity, performed by clinicians in the operation room, is reflected in the WF just by a new record in the table *Implanted* indicating that the patient *Pt* is implanted with the pacemaker *Pm1* and will have a follow-up visit after 90 days; two null attributes are reserved for the status of the patient, and status of the pacemaker.

After implant, a follow-up visit occurs every 90 days (adjustments to the visit day may be performed by nurses which directly access records of the *Implanted* table; emergencies are also taken care by nurses that may set the visit day to the current date and then schedule the visit as soon as possible). During a follow-up visit, the status of patients is recorded together with the date of the next visit; an exception may be caused by a failure of the pacemaker, that forces the WT to be suspended and exited with *Status* = *fail*. The *Status* is set to ‘‘Fail’’ also if the patient is not fine (e.g. he needs a more sophisticated pacemaker) or if the patient needs no pacemaker at all.

The WT *Need a pacemaker* states if the patient *Pt* needs a new pacemaker (*get NeedAnotherPM*). If no pacemaker is needed, WT *Explant* explants the old one (*delete from Implanted* ...). If instead a new one is needed, then the WT *Find a new pacemaker* is activated. As a precondition, it looks inside *PmFree* if there is another available functioning pacemaker: it selects a functioning pacemaker (*Pm2* = *select-one* ...) and makes it no longer available to others (*delete from PmFree* ...). The old pacemaker is then explanted (*delete from Implanted* ...), and the patient *Pt* is associated with a new pacemaker

¹ This example was suggested by John Mylopoulos to Stefano Ceri in 1982 while doing research on script languages, long time ago ...

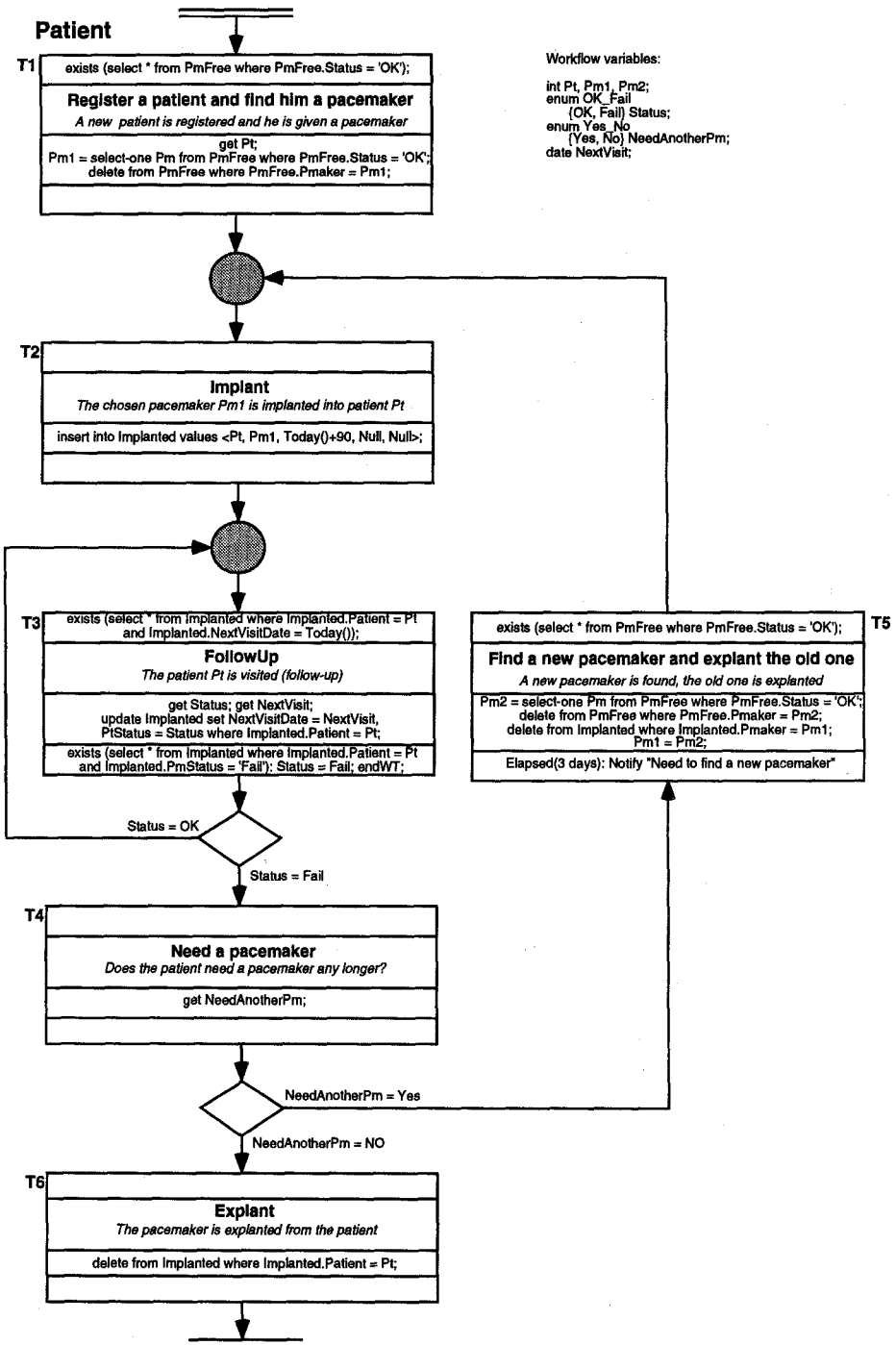


Fig. 3. Patient Workflow.

(by means of the assignment $Pm1 = Pm2$, which allows us to refer to the pacemaker currently implanted in the patient Pt by means of the variable $Pm1$). As an exception, when the patient Pt has been waiting for a new pacemaker for more than three days, a message is notified to the WT agent. Finally, the new pacemaker is implanted and the follow-up cycles starts again.

Note that the *Patient* WF requires several local variables: Pt , $Pm1$, $Pm2$, $Status$, $NextVisit$, $NeedAnotherPm$. Values of these variables cannot be accessed outside a specific WF case; indeed, the WFDL imposes that these variables be atomic so as to associate each case with a row of values, one for each variables, in a specific table describing the case evolution. This table can be accessed by WT execution environments if needed, although normally variables are used as parameters within procedure calls which activate the WTs.

All communication with the external environment, consisting of other cases, possibly enactments of different schemata, and of generic other applications, is performed by means of database tables $PmFree$ and $Implanted$. In particular, $PmFree$ is manipulated by the *Pacemaker* WF, which is described next.

3.2 Pacemaker Workflow

The *Pacemaker* WF gives the dual view of this application from the pacemaker's perspective. A pacemaker Pm is assembled, tested and inserted into $PmFree$. If the pacemaker is defective ($Status = Fail$), WT *Trash* trashes it and the WF ends.

After Pm is implanted into a patient, it is checked by WT *Check*, performed by the technicians who are responsible of the pacemaker functioning. Normally, each patient gets an appointment on the same date, and then is seen by clinicians and by technicians. Thus, in the same way as the *Follow-up* WT of the *Patient* WF, the precondition of WT *Check* requires that Pm is implanted and today is the next visit day. The result of the *Check* is encoded in the variable $status$, now referring to the pacemaker, which is entered into the suitable record of the *Implanted* table. $Status$ of the pacemaker is set to "OK" if Pm is functioning correctly: otherwise $Status$ is set to "Fail". If $Status$ is "OK", WT *Check* is re-activated, where its precondition force next check to take place only at the next fixed day: if $Status$ is "Fail", WT *Check* ends.

An exception of WT *Check* may force $Status$ to "Fail", regardless of the functionality of Pm . In fact, if Pt is not fine, the exception is raised and the WT is forced to completion. In both cases (either a pacemaker failure or a change of the patient's care), technicians have no responsibility on the subsequent pacemaker explant, and their next WT aims at recharging and testing Pm after it has been explanted: as a precondition, Pm must be not implanted (`not exists (select * ...)`). After the recharge and the test, if the $Status$ of Pm is "OK", the pacemaker waits for another implant into another patient (WT *Implant* of WF *Patient*): otherwise it is trashed and removed from the $PmFree$ table.

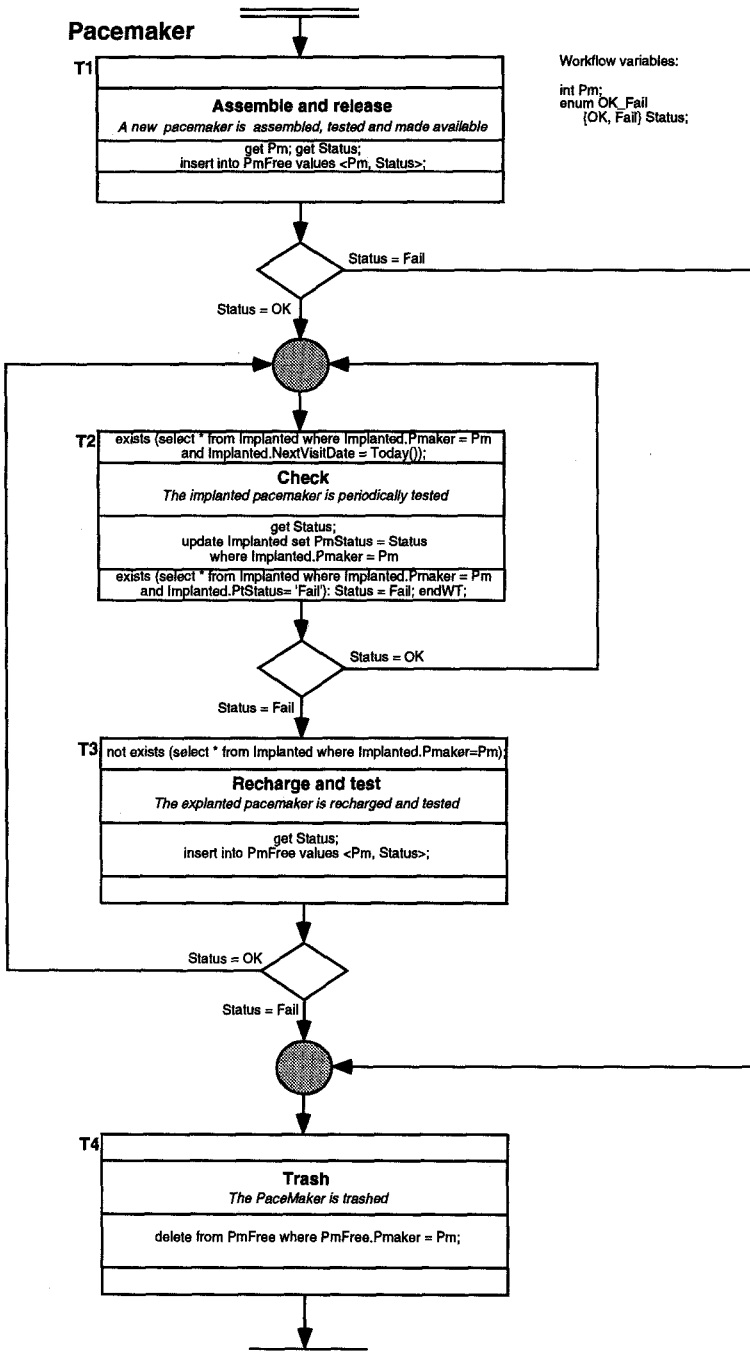


Fig. 4. Pacemaker Workflow.

4 Properties of Interacting Workflows

The above examples may be used in order to introduce several interoperability problems that may exist between WFs. Indeed, we will see that the above WFs exhibit a nice cooperation pattern, which can be formally studied and proved.

4.1 Properties of Enactments of a Single WF Schema

We first define properties that characterize the enactment of a single WF schema. Recall that there can be multiple concurrent case executions of the same WF schema, and that one case may cause multiple concurrent executions of the same WT. Therefore, we identify each case enactment by a *case-id* C_i , which is assigned at the start of the case by the WFMS, and each WT execution E_{ijk} by a triple *case-id*, *task-id*, *execution-number*, where the execution number k is progressively assigned for each execution of a given WT T_j within a given case C_i .

The following properties describe WT execution during a given case enactment:

- **Reachability**: a WT is reachable if it can be set to the *active* state during the course of a given case enactment. When a WF schema is compiled, all WTs should be reachable (or else they don't contribute to the WF); however, during enactment in general the set of reachable WTs for a given case reduces progressively.
- **Potential Termination**: a case potentially terminates iff at least one of its stop symbols is reachable. This property should always hold during enactment, otherwise the schema is incorrectly specified.
- **Mutual Exclusion**: two WTs of the same case are in mutual exclusion if they cannot be in the *executing* state at the same time.
- **Potential Parallelism**: two WTs of the same case are potentially parallel if they can be in the *executing* state at the same time.
- **Potential Precedence**: this relationship is a partial order between WTs $T_h <_{pp} T_k$ which holds when, in an arbitrary case enactment, T_h can be completed (*ended* or *canceled*) before T_k is made *ready*.
- **Absolute Precedence**: this relationship is a partial order between WTs $T_h <_p T_k$ which holds when, in all case enactments, T_h must be completed (*ended* or *canceled*) before T_k is made *ready*. If T_h and T_k can be activated multiple times, then absolute precedence applies to each WT execution for each case; therefore, $\forall i, j, E_{ihj}$ should be completed before E_{ikj} is *ready*.

For example, in the *Patient* WF all WTs are reachable; the WT T_1 becomes unreachable once it is completed; no WTs can be potentially parallel; T_1 absolutely precedes T_6 , and T_2 absolutely precedes T_4 and T_5 .

4.2 Properties of Enactments of Different WF Schemata

Concurrent enactments of cases from different WF schemata are characterized by one additional identifier, the *schema-number*, which needs to be added in order to properly identify WTs (T_j^s), cases (C_i^s), and WT executions (E_{ijk}^s).

- **Condition-Action Dependency:** this relationship is a partial order between WTs $T_h^p <_{ca} T_k^q$, $p \neq q$, which holds when it is required that the WT execution T_h^p be *ended* within an arbitrary case before the precondition of T_k^q becomes true in another arbitrary case. The left hand side of the precondition may include multiple WTs when termination of anyone of them may cause T_k^q to become *ready*.
- **Case Binding:** it is a link established between case identifiers i and j when the condition-action dependency $T_h^p <_{ca} T_k^q$ occurs between specific WT executions E_{ihm}^p and E_{jkn}^q .
- **Activation Precedence:** this relationship is a partial order between WTs $T_h^p <_{ap} T_k^q$, $p \neq q$, which holds when T_h^p can become *active* before T_k^q .
- **Pseudo-Simultaneous Execution:** it is established between two WTs T_h^p and T_k^q , $p \neq q$, when it holds both $T_h^p <_{ap} T_k^q$ and $T_k^q <_{ap} T_h^p$ and they are not in mutual exclusion.
- **Task Binding:** it is a link established between case identifiers i and j when the pseudo-simultaneous execution of T_h^p and T_k^q occurs between specific WT executions E_{ihm}^p and E_{jkn}^q .

Given that the only information shared by different WF cases is stored within database tables, condition-action dependencies normally are established by means of queries in the precondition of T_k^q and database inserts or updates in the action of T_h^p . Similarly, the presence of the same precondition in two WTs T_k^q and T_h^p indicates that they could be pseudo-simultaneous, as the same external event may cause their simultaneous activation.

Bindings established between cases are normally associated with data sharing among the executions E_{ihm}^p and E_{jkn}^q . Although in many simple cases bindings between executions are *one-to-one* (e.g. one case C_i^p is bound to one case C_j^q and viceversa), indeed bindings may be also *one-to-many* and *many-to-many*.

In the *Patient* and *Pacemaker* WFs, condition-action dependencies are established from WTs $\{Pacemaker.T1, Pacemaker.T3\}$ to WTs *Patient.T1* and *Patient.T5*. These dependencies establish case bindings, which intuitively correspond to the possibility of progressing both WFs which occur when a pair (*Pacemaker-Patient*) is identified. Such bindings are *one-to-one*; however, they evolve dynamically, as the same patient can dynamically receive multiple pacemakers, and similarly the same pacemaker may be implanted within many patients.

WTs *Patient.T3* and *Pacemaker.T2* (corresponding to the *Patient's FollowUp* and the *Pacemaker's Check*) are pseudo-simultaneous, due to the fact that they have the same precondition. Correspondingly, whenever they are executed a task binding is established between two executions of *Patient.T3* and *Pacemaker.T2*,

respectively. These WT bindings indicate that indeed the two WTs of the different WF schemata are candidate for integration.

4.3 Properties Concerning Termination

The next properties concern the termination of cases in presence of interaction between cases.

- **Sufficiency:** A set of WF schemata $A, B \dots N$ is sufficient on schema P iff all the WTs of P are condition-action dependent only from WTs of $A, B \dots N$. Formally, if $task(F)$ denotes the set of WTs of a WF F , $dep(T_h^p) = \{T_k^q : T_h^p <_{ca} T_k^q\}$ denotes the set of all WTs on which T_h^p depends, and the symbol $*$ denotes transitive closure, then the set $A, B \dots N$ is sufficient on a WF schema P if:
 $\forall T_i \in task(P) (dep^*(T_i) \in (task(A) \cup task(B) \dots task(N)))$
- **Starvation:** A WT T is in starvation, and thus the cases in which T is *ready* may be indefinitely held in wait, when the WTs of the set $dep(T)$ on which T depends are not evolving by providing sufficient data so as to satisfy the preconditions for all the above cases.
- **Deadlock:** two interacting WFs A and B could be in deadlock when they have two WTs T_a and T_b such that the former depends on the latter and viceversa, i.e. both $T_a \in dep(T_b)$ and $T_b \in dep(T_a)$ hold at the same time.

For instance, the set $\{Patient-Pacemaker\}$ is sufficient (since all the condition-action dependencies of *Patient* are related to *Pacemaker* and viceversa); when no *Pacemaker* is available, WTs *Patient.T1* and *Patient.T5* are in starvation.

As an example of deadlock, consider two simple WFs *Invite* and *Propose*. The *Invite* WT is performed by a conference's chairman, who waits for someone to send him a proposal of a tutorial before inviting someone to give the tutorial. The *Propose* WT is performed by the company of potential speakers, expecting tutorial invitations before sending the corresponding abstracts. The two WFs are in deadlock, since they can indefinitely wait for each other. Note that a deadlock between WFs can be broken by external events; for instance, a colleague of the conference chairman could send the speaker's abstract to the chairman and propose him for the tutorial which an activity that is outside of the WF specifications.

4.4 Workflow Interactions

The following classification indicates the possible interaction schemes that may be identified between different WF schemata.

- **Cooperation:** Two WFs A and B cooperate iff their interaction is desired and foreseen. In this case, the WF administrator designs the WF schemata so that at given points A produces some data and, possibly at different points, B uses those data. Typical examples of cooperation are the producer-consumer and the client-server environments.

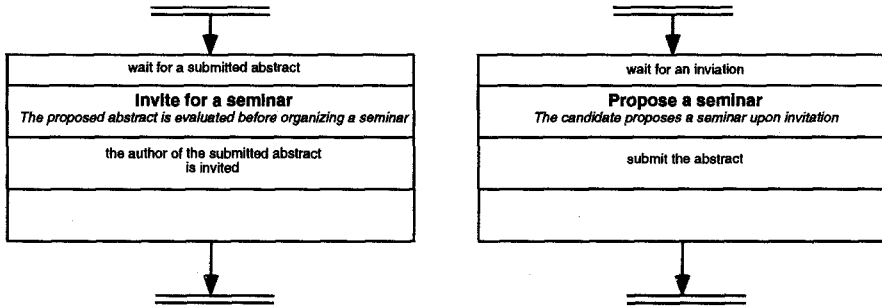


Fig. 5. Workflow deadlock: the two WFs are waiting for each other.

- *Competition*: Two WFs A and B compete iff their interaction is not desired but foreseen. In this case, the two WFs A and B could be executed correctly in a separated way, but a contemporary execution is not possible due to the limited number of available resources. A typical example of competition takes place when WFs A and B need two resources, R1 and R2, and the resource pool ResPool has one item only.
- *Interference*: Two WFs A and B interfere iff the interaction is neither desired nor foreseen. In this case, both A and B are correctly designed for a stand-alone execution but their synchronization is badly designed.

Clearly, in the *Patient-Pacemaker* example the cases of *Patient* and of *Pace-maker* cooperate, while the cases of *Patient* compete with each other for the assignment of pacemakers required by each patient.

5 Workflow Integration

WF integration aims at building an integrated frame representing two or more cooperative WF schemata, which indicates explicitly case and WT bindings. WF integration may lead to the identification and removal of interferences between WFs, and therefore can cause the redesign of some features of the component WFs. It is important to note, however, that WF integration is hereby proposed as a conceptual device, whereas each WF application remains managed by the WF management environment as explained in Section 2. Thus, the result of WF schemata integration is not another WF schema but only a frame in which interactions of the original schemata can be better understood.

Integrated frames collect WTs from various WF schemata, called its *component WFs*. They have the following features:

- There can be multiple start tasks, one for each component WF.
- New special tasks *link* and *unlink* are introduced to denote case bindings and unbinding. A link node has several incoming edges, representing component WFs, and one outgoing edge, representing the integrated frame; an unlink node has one incoming edge, representing the integrated frame, and several

outgoing edges, representing component WFs. Syntactically, along each path connecting a link node to a stop symbol there must be at least an unlink node.

- Most WTs are taken unchanged from anyone of the component schemata; however, they may be replicated on the integrated schema if this facilitates the schema merging.
- Bound WTs from component WFs are merged into one integrated WT, graphically represented by double frames. The name of bound WTs is given by the composition of names of the original WTs, preconditions and exceptions are composed by means of disjunction (to indicate that anyone of them applies to the bound WT) and actions are listed in sequence (to indicate that all actions of the bound WTs are eventually performed).

From a methodological viewpoint, the integrated frame is built by detecting condition-action dependencies and activation precedences, and deducing case and WT bindings from them. Case bindings are represented by link nodes. Once WF schemata are linked, the designer should consider each component WF and understand which WTs are performed within WF by preserving the binding (be them bound WTs or not), thus identifying a dual unlink node, that represents the logical point in the execution of the WF where case bindings is changed or no longer effective. Link-unlink nodes are paired; if component WFs remain linked until they reach their stop task, the unlink is simply added as the last node before all stop tasks.

5.1 Patient - Pacemaker Integration

In Figure 6, we illustrate the integration of the *Patient-Pacemaker* WFs. As already indicated in Section 3, condition-action dependencies are established from WTs $\{Pacemaker.T1, Pacemaker.T3\}$ to WTs *Patient.T1* and *Patient.T5*. These dependencies establish the *one-to-one* case binding between the WFs, which intuitively matches the fact that a given patient has a given pacemaker implanted². Case binding is represented by a link node which immediately follows the two WTs where the condition-action dependency is recognized; the link node delimits a portion of the integrated schema representing the binding of cases, which is represented by a gray shade in Figure 6.

After the implant operation, which is performed within the WF *Patient*, we recognize a WT which is candidate for merging. Indeed, as already noticed, WTs

² Note that in the proposed example case binding is very concrete case, as it corresponds to an “implant” operation (and similarly the unbinding corresponds to an “explant” operation). Normally, case binding is less evident, and it corresponds to using shared data which become relevant to several WF schemata. For instance, claim management from the viewpoint of an insurance company and of a lawyer company, where binding is due to the processing of the same claim; or reviewing process seen from the viewpoint of the editor and the referees, where binding is due to the processing of the same paper.

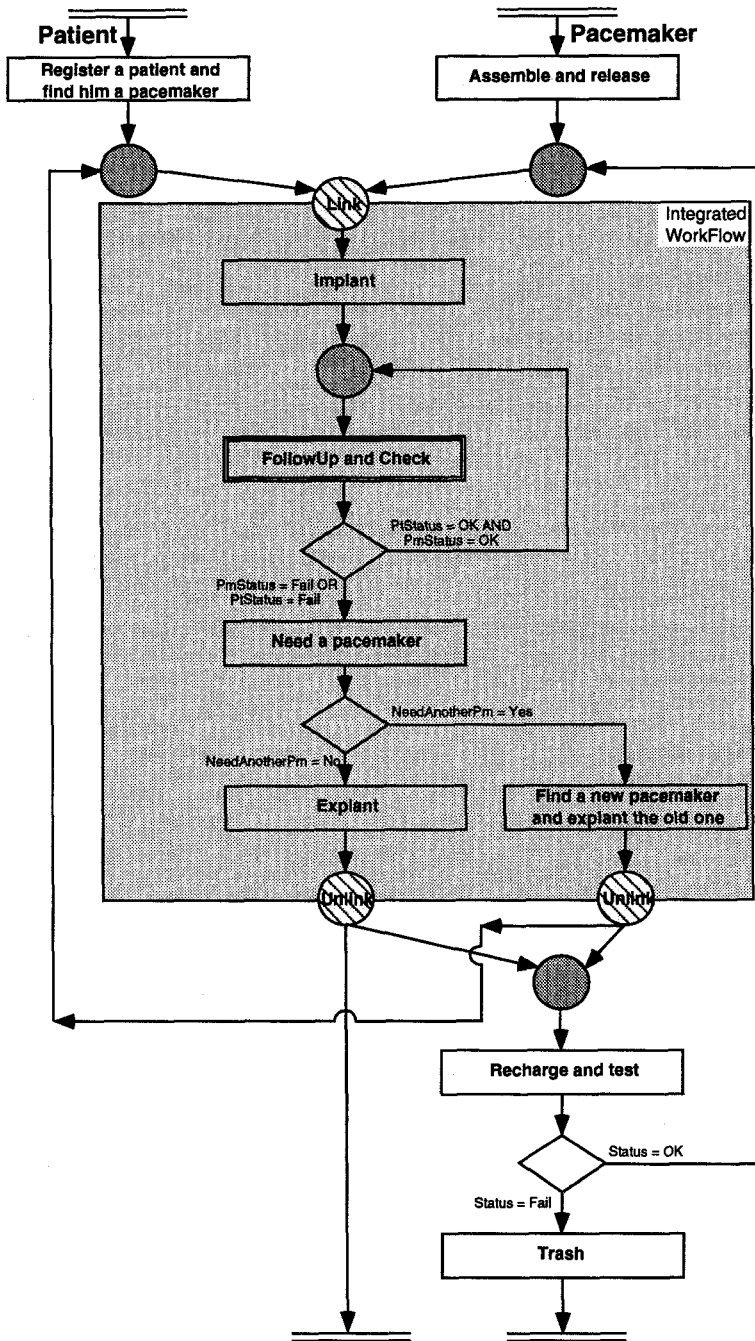


Fig. 6. Frame resulting from the integration of Patient and Pacemaker workflows

Patient.T3 and *Pacemaker.T2*, representing the patient's follow-up and the pacemaker's check, are pseudo simultaneous, have the same precondition (indicating that the patient is being visited during the current day), and physically happen one after the other, as the patient sees the doctor and the pacemaker technician during the same visit³. Their merge generates the *FollowUp and Check* WT of the integrated schema, which is associated with:

- The same precondition as its components (corresponding to the disjunction of identical predicates).
- The sequencing of follow-up and check activities.
- The removal of exceptions (we recognize that exceptions can actually be removed as they express within each WF the failure of the other WF; thus, WT variables **PtStatus** and **PmStatus** can simply be tested in disjunction at the end of the WT).

FollowUp and Check is executed until the **Status** of either the patient or the pacemaker is not **OK**. Then an explant becomes required. If the patient needs a new pacemaker, a new one is found before scheduling the explant operation, otherwise the explant has no prerequisites. In both cases, the explant operation brings as a consequence the unlink of the WF schemata, and delimits the “merged” area of the integrated WF schema.

6 Conclusions

The problems of WF integration have many aspects in common with schema integration in conceptual database design [4]. Indeed, schema similarities are searched, and entities are sometimes integrated by merging their contributions from various schemata. Name and type conflicts are common to schema integration and WF integration, and may lead to WT renaming and restructuring which are dual to schema integration activities. Properties such as transformational, mapping, and behavioral equivalence of conceptual schemata can also be extended to WFs. Therefore, we believe that this paper has opened a new promising area for research, which can be suitably attached by reusing competence already established for conceptual database design.

Another interesting analogy links WF interoperability to object-oriented interoperability. In the latter context, objects are described together with their messages (events) which may cause a change in their internal state; objects' coordination is achieved through the message passing mechanism, and the interoperability problem consists in understanding and/or designing a global system evolution. Some proposals in this direction are contained in the F-ORM model (Functionalities in Objects with Roles Model [12]), in which object modeling processes are introduced to facilitate coordination between interacting objects

³ In the concrete case study that we are considering, the pacemaker checking is sometimes performed by a clinician.

aiming at achieving a common goal. The concepts of cooperation in object-orientation systems could thus be another source of design principles for formalizing WF integration, while keeping each WF relatively independent during the execution of its own WTs.

In future work we will deepen our analysis of theoretical properties of integrated WF schemata and will use such theory in the development of tools assisting the designer in WF analysis and integration.

References

1. Attie P., Singh M., Sheth A., Rusinkiewicz M., "Specifying and Enforcing Intertask Dependencies", *Proc. 19th VLDB Conf.*, Dublin, Ireland, 1993.
2. Bandinelli S., Fuggetta A., Ghezzi C., "Software Process Model Evolution in the SPADE Environment", *IEEE Transactions on Software Engineering*, December 1993.
3. Baralis E., Ceri S., Paraboschi S., "Modularization Techniques in Active Rule Design", Tech. Report IDEA.WP.003.01, Politecnico di Milano, Milano, Italy, Dec. 1994.
4. Batini C., Ceri S., Navathe S.B.: *"Conceptual Database Design: An Entity-Relationship Approach"*, The Benjamin-Cummings P.C., Redwood City, Ca, 1992.
5. Batini C., Lenzerini M., Navathe S.B.: "Comparison of Methodologies for Database Schema Integration", *ACM Computing Surveys*, 18:4, 323-364, Dec. 1986.
6. Bracchi G., Pernici B., "The Design Requirements of Office Systems", *ACM Trans. on Office Information Systems*, 2(2), April 1984.
7. Bussler C., Jablonski S., "Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems", *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)*, Houston, Texas, February 1994.
8. Casati F., Ceri S., Pernici B., Pozzi G., "Conceptual Modeling of Workflows", in *Object-Oriented and Entity-Relationship Approach* Int. Conf., Gold Coast, Australia, December 1995.
9. Casati F., Ceri S., Pernici B., Pozzi G., "Deriving Active Rules for Workflow Enactment", submitted for publication, 1995.
10. Ceri S., Fraternali P., Paraboschi S., Tanca L., "Active Rule Management in Chimera", in [27]
11. Dayal U., Hsu M., Ladin R., "Organizing Long-running Activities with Triggers and Transactions", *Proc. ACM SIGMOD*, 1990.
12. De Antonellis V., Pernici B., "Reusing Specifications through Refinement Levels", *Data and Knowledge Eng.*, vol. 15, 1995
13. Ellis C., Nutt G., "Modeling and Enactment of Workflow Systems", in *Application and Theory of Petri Nets*, M. Ajmone Marsan Ed., Lecture Notes in Computer Science 691, New York: Springer Verlag, 1993.
14. Ellis S., Keddara, K. and Rozenberg, G., "Dynamic Change within Workflow Systems", *ACM Conf. on Organizational Computing Systems (COOCS 95)*, 1995.
15. Forst A., Kuhn E., Bukhres O., "General Purpose Work flow Languages", *Distributed and Parallel Databases*, vol. 3, n. 2, April 1995.
16. Francalanci C., Pernici B., "View Integration: a Survey of Current Developments", *Internal Report n. 93.053*, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, 1993.

17. Fraternali P., Tanca L.: "A Toolkit for the Design of Active Database Semantics", Dipartimento di Elettronica e Informazione, Politecnico di Milano, Technical Report 93-078, 1993.
18. Georgakopoulos D., Hornick M., Sheth A., "An Overview of Workflow Management: from Process Modeling to Workflow Automation Infrastructure", *Distributed and Parallel Databases*, vol. 3, n. 2, April 1995.
19. Gonzales-Quel L. A., Gonzales S., Perez M., "IDEA Technology Assessment Based on Workflow Applications. The Royal Life Application", *IDEA Rep. IDEA.DE.21S.001.01*, November 1994.
20. Hewlett Packard Company, "Workflow Module Programming Guide", Palo Alto, Ca, July 1994.
21. Hsu M. (ed.), Special Issue on Workflow and Extended Transaction Systems, *Data Engineering Bulletin*, 16(2), June 1993.
22. Krishnakumar N., Sheth A., "Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations", *Distributed and Parallel Databases*, vol. 3, n. 2, April 1995.
23. Paton N.W., Diaz O., Williams M., Campin J., Dinn A., and Jaime A: "Dimensions of Active Behaviour", in *Rules in Database Systems*, Norman W. Paton and M. Howard Williams (Eds), Springer-Verlag, 1993.
24. Rusinkiewicz M., Sheth A., "Specification and Execution of Transaction Workflows", in *Modern Database Systems: the Object Model, Interoperability, and beyond*, Kim W. (ed.), Addison-Wesley, 1994.
25. Sheth A., Rusinkiewicz M., "On Transactional Workflows", *Data Engineering Bulletin*, 16(2), June 1993.
26. Ullman, J.D., "Principles of Database and Knowledge Base Systems", 2 Volumes, Computer Science Press, Rockville, Maryland, 1989.
27. Widom J., Ceri S., "Active Database Systems", Morgan-Kaufmann, San Mateo, Ca, May 1995.