ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

From Database Programming to Business Process Programming (Invited Talk)

Umeshwar Dayal (Hewlett-Packard Laboratories)

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

From Database Programming to Business Process Programming

Umeshwar Dayal

Hewlett-Packard Laboratories
Palo Alto, California, USA

Qiming Chen

Hewlett-Packard Laboratories
Palo Alto, California, USA

Abstract

Over the past decade, database programming has focused on languages and methodologies for developing data-intensive applications, and on techniques for efficiently implementing them. Research in this area has yielded languages with rich type systems, orthogonal persistence, support for collection types, and other advanced features. The commercial impact of this research, however, has been rather limited. Database applications are a multi-billion dollar business worldwide, but most applications are still developed using rudimentary tools.

The goal of this paper is to urge the database programming research community to rise to the next challenge: business process programming, a field that is in its infancy, and where there is the potential for significant impact. Competitive pressures are forcing enterprises to re-engineer and automate their business processes. Such processes are of long duration, complex, and error-prone. They consist of multiple steps that need to be coordinated; that may be executed by application programs, machines, or humans in different roles across different organizations; and that may execute in heterogeneous, distributed computing environments.

This paper introduces topics in business process programming that we believe will be of interest to the database programming community. These range from issues in modeling and specifying business processes, including new transactional models for capturing their execution semantics, to problems of efficient, scalable implementation in the heterogeneous distributed environments prevalent in typical enterprises.

1 Introduction

Research in database programming, as reflected in the Proceedings of the five DBPL Workshops held so far, has focused primarily on new languages and methodologies for developing data-intensive applications. This has resulted in the design and implementation of database programming languages with rich type systems that are extensible, provide orthogonal persistence, and support a variety of collection types to facilitate the manipulation of persistent bulk data stored in a database.

What impact has research on database programming languages had on the multi-billion dollar database applications industry worldwide? Relatively little, unfortunately. Most of these applications are still being developed in fourth generation languages or even third generation languages (COBOL or C) plus SQL. The elusive research goal of orthogonal persistence has not been attained in practice as the “impedance mismatch” between the host programming language (in which the application logic is coded) and the database manipulation language continues. While object-oriented database systems supporting Persistent C++ or Smalltalk with varying degrees of orthogonal persistence have appeared, the technology has not taken off commercially, and these systems represent a tiny fraction of the market.

So, what can the DBPL research community do to increase commercial impact? The next big challenge for the information technology industry is the management and automation of business processes. Business process management, or “workflow management”, recently has generated a lot of research activity in different sub-disciplines of computer science. Also, several dozen commercial products have appeared, and an industry group, the Workflow Management Coalition (WfMC), has published a reference architecture for such products [28]. This paper argues that there are many important research problems in this complex area to which the DBPL research community can make important contributions.

Business process management cannot be handled by means of conventional database techniques alone. Business processes typically are of long duration, complex, and error-prone. A business process, in general, is a set of activities with a common goal. It is built by linking together diverse activities, specifying the flow of data and control among them, and specifying business rules that have to be enforced during the execution of the process. Business processes often consist of multilevel, long-duration and collaborative activities, and involve many users, information

systems, applications and tools in heterogeneous and distributed environments. Examples of business processes include purchase or service order processing, insurance claims processing, loan application processing, making travel arrangements, planning and scheduling, design and manufacturing processes, and medical and laboratory protocols. Consider the process of opening a new sales office for a company's division; this includes several sequential and parallel activities, such as submitting a proposal, determining the location, negotiating office rental price with owners, getting approval, etc. These activities may be transactional or non-transactional, and may range from the execution of computer programs to human actions such as attending meetings or signing forms. Also, some of the activities themselves may be complex processes. Today, the logic for linking the activities that comprise a business process are either expressed informally in operating manuals or in people's heads, or embedded in application programs.

As enterprises seek to become more competitive, they are turning to business process management technology to help re-engineer and automate critical business processes. The Workflow Management Coalition defines business process management as "a technology that allows companies to deploy both old and new business process applications that are faster and more flexible to build by making the business rules and flow of work logic independent of the business application" [28]. A business process management system (or workflow system) controls and automates the interactions between activities, programs, data and users. The ability to integrate these components sets business process programming apart from database programming: *programming in the large* versus programming individual applications.

In Section 2, we briefly describe the requirements for business process management, and the extent to which the existing commercial workflow products meet these requirements. Then, in the remainder of the paper, we describe an approach to business process management that we are taking at Hewlett-Packard Laboratories. In Section 3, we describe our process model; in Section 4, we describe the transactional semantics associated with our process model; and in Section 5, we describe failure handling. In Section 6, we illustrate our approach via a fairly complex example.

2 Business Process Management

We first describe the requirements for modeling, specifying, and managing business processes. Then, we review the state of commercial practice, and point out the strengths and limitations of commercial products. This should underscore the key areas for research contribution.

2.1 Requirements

We characterize the requirements for business process management in terms of a variety of services.

- Process definition services
- Process enactment services
- Distribution services
- Organization services
- Monitoring and administration services

Process definition entails the specification of the activities comprising the process; the control and data flows among them; and the business rules, policies and constraints relevant to the process. Processes are often defined as graphs, with the nodes representing the activities of the process, and the directed edges representing control and data flows. Some models specify control and data flows declaratively through rules or dependencies. Rules specify that events signaled by activities cause downstream activities to be triggered. Dependencies prescribe temporal relationships among activities (e.g., activity a_1 cannot start until a_2 terminates). Some models use procedural specification in a scripting language (typically a block structured concurrent programming language). For complex processes, it is useful to support abstractions for hierarchical structuring. Some models allow processes to be recursively defined in terms of subprocesses. It is important to allow flexible binding of subprocesses: at process definition time, at process instantiation time, or even later at the time the subprocess executes. This flexibility is important in that it allows subprocess libraries to be constructed using reusable templates. It also supports autonomy in that the subprocess can be redefined without affecting the larger process; this is especially important if process execution crosses organization boundaries.

Process enactment includes services for instantiating processes, orchestrating the control and data flows among the activities of a process, and enforcing business rules, policies, and constraints. Since processes may access shared persistent data, they must be subject to transactional control. However, executing an entire complex process as a single transaction is unnecessary and inefficient. Some models support extended transaction semantics for modeling a process as a hierarchical collection of related transactions. Finally, services for exception and failure handling are required. The execution of a business process may fail to complete. This requires system support for determining the scope of the failure, rolling back the process within that scope (such as withdrawing an office rental application in the above example), and rolling forward the process via alternative paths.

Various distribution services are required. These include interaction with users and communication with application systems; naming and directory services for locating resources required for executing activities; and interoperation with other process management systems.

Organization services support role resolution, the assignment of resources to activities, and authorization and access control.

Finally, services for monitoring and administration of processes are required. These include recording the history and status of process executions, queries over the status information, and dynamic modification of processes.

The architecture of a typical process management system includes several components: a worklist handler for managing interactions between a user and the system via worklists containing the activities he is assigned or allowed to perform; one or more database servers for maintaining process-related data (this is often separate from the “business data objects” used by the activities); a workflow server for providing flow control; an organization server for identifying and locating the appropriate users or invoked applications; and a transport service for communication between users, external applications and cooperative workflow systems in a heterogeneous and distributed environment [26, 10, 11, 5, 8, 4, 21, 23, 20, 16, 18, 2, 7]. A reference architecture has been defined by the Workflow Management Coalition [28].

2.2 Commercial Practice

Over two dozen workflow products are on the market today. The early products were aimed at “automating” clerical tasks in administrative and other office procedures. These systems, therefore, were message-centred or document-centred.

Message-centred systems essentially are extensions of electronic mail. Work is routed between workers through the flow of structured messages. Flows are either statically defined or may be dynamic based on message content. Forms-based utilities are often provided for creating, approving, and signing forms. Such systems are limited to manual tasks and provide no recovery beyond reliable message delivery. Also, they do not provide status monitoring or reporting.

Document-centred systems are extensions of image (document or file) management systems, and typically support the flow of folders between users. They were developed for applications such as loan or claims processing. Documents and accompanying routing lists may be stored in a database or file system. Such systems, too, are limited to manual tasks and provide limited recovery (of the document database or files).

More recently, we have started to see the emergence of process-centered systems aimed at high-end business-critical processes such as order processing or customer care. These systems provide tools for modeling processes as collections of activities or tasks, specifying control and data flow, and defining business rules and constraints. They support manual and automated activities, provide some recovery and exception handling, and integrate into distributed computing frameworks such as CORBA. For instance, they may support the encapsulation of legacy database systems, file systems, and application systems as business objects using CORBA interfaces, and use CORBA RPC as the transport mechanism to launch applications. They use persistent databases to record history and status and to support the querying of this information. They also support organization databases, and tools for monitoring and managing processes.

The process models supported are still quite limited, however. In particular, there is little support for hierarchical modeling of complex processes, and this limits scalability to enterprise-wide or inter-enterprise processes. Also, failure and exception handling are rather limited, and there still are no generally accepted transactional models for interrelated collections of long-duration activities.

In summary, business process management relies on several relevant sub-disciplines of computer science: software process modeling, groupware and collaborative work, distributed computing, distributed database systems, and transaction management. The challenge is to integrate these technologies into a coherent framework that provides a rich

process model, flexible yet precise transactional semantics, integrated data sharing and failure handling mechanisms and capabilities for incorporating activities, data objects and users.

3 Open WorkFlow Manager Process Model

In the rest of this paper, we describe work we have done on an Open WorkFlow Manager (OWFM). This section describes our process model.

Most workflow systems are activity-net based, where activities for a process or subprocess are modeled by graphs that reflect the forking and joining of control flow. Also, in some of these models, data flow can be separated from control flow. A flat model, however, makes it difficult to provide a well understood foundation for data sharing and failure recovery, and does not scale up to complex processes. However, it is also insufficient to represent a process purely by a tree structure, since the tree structure only captures the containment relationship between a parent activity and its child activity, but not any relationships that associate activities at the same level.

To support multilevel process management flexibly, we introduce a process model that is characterized by multilevel blocks and hyper-activity/block coupling. This model provides one way to materialize the conceptual model provided by the Workflow Management Coalition.

3.1 Specification Model

A business process is a top-level block that contains multilevel nested blocks representing either subprocesses or nested process modules [1, 7]. A block contains a set of activities linked to represent control flow and data flow. Conditional branching, parallel forking and joining, as well as looping (iteration) are allowed. A block is *represented* by a *hyper-activity* T at the next higher level; the block is denoted by B^T , and T is referred to as the *parent* of the activities directly contained in B^T . Since a component activity can be a hyper-activity representing another inner block, blocks or hyperactivities can be nested to arbitrary levels.

A block, B^T , has an associated logical data container called **Block Data Container (BDC)** that is denoted by Π_{B^T} . The BDC of the top-level block of an activity hierarchy is actually the process data container. BDC's may be implemented in terms of multi-level databases with controlled access [19], or in terms of other repository facilities. Data held in the BDC of a block is visible to the activities in that block and in any inner blocks; therefore, an activity can **checkout** data from the BDC of any higher level block. An activity can acquire data by accessing external data sources, or by **checking out** data from the BDC of a block containing it directly or transitively. Upon termination, an activity **checks in** its results into the BDC of its parent block or of some ancestor. We will describe these options for commit control later.

For the purpose of failure handling, an activity at any level may be paired with a *contingency activity* $\tilde{T}$, which is used when rolling forward along an alternative execution path; and T may also be paired with a *compensation activity* $\bar{T}$ that can logically eliminate its effects. A compensation or contingency activity can be flat or hierarchical and may be structured differently from the original activity. A contingency activity may in turn have its own contingency activity.

Finally, the model includes a *multilevel authorization resolution* facility to allow user **intervention** of processes and subprocesses by authorized users (supervisors), i.e., to dynamically modify flow control, change user assignments, partially rollback processes, and so on. To allow users to balance process protection and subprocess autonomy in a way suitable to their applications, a *supervision protocol* controls upward delegation of intervention privileges along the process hierarchy.

3.2 Other Specification Styles

In addition to the explicit specification of process and activity structure described so far, we support three additional mechanisms for modifying the control and data flow within a process: events, dependencies, and self-repair protocols.

The event-driven mechanism is based on Event-Condition-Action (ECA) rules [22, 15]. The basic semantics of rule execution is that when the specified event is detected, the condition is evaluated; and if the condition is satisfied, then the action (which can be an activity or even an activity hierarchy), is triggered. As described in [22], we distinguish three event-action coupling modes: *immediate*, *deferred* and *detached*, which provide flexible ways to modify activity hierarchies. The use of ECA rules allows process specifications to be dynamically modified or overridden. Rules can be used for intra-process activity triggering, inter-process synchronization and reacting to general system failures, resource reallocations, security violations, and so on.

Dependencies are used to enforce acceptable activity execution orders, and once translated into dependency graphs, they can also be used to “trigger” activities. State dependencies between activities may occur fortuitously, or they may be explicitly specified. A dependency between two activities T_j and T_i can be represented by $T_i \rightarrow_{\tau} T_j$ where T_j, T_i are in states *start*, *commit*, *abort* and so on; and τ is the dependency type. For example, the *fail dependency* denoted by $T_i \rightarrow_{fail} T_j$ means that in case T_i has failed, if T_j has not completed, then T_j is canceled and any completed block component of T_j is compensated for; else T_j is compensated for. Informally, the fail dependency asserts that if T_i fails, then T_j must fail too; and this dependency underlies the semantics of failure handling across activity hierarchies or subhierarchies.

Self-repair mechanisms include protocols for exception handling and compensation. These affect process execution by erasing the effects of non-fatal failures through roll back and roll forward. These mechanisms will be described in more detail in Section 5.

3.3 Execution Model

It is important to separate the *specification model* and the *execution model* of a business process. The former provides a template for the structure of the process and control flow within it. The latter is dynamically constructed at run time, and expresses the actual progress of the process instance and records the history of its execution. The execution history is used in failure recovery, and can also be used by visualization tools to animate the process.

To execute a business process, an instance of its top-level activity is created based on its specification. However, the instance of a component activity is created only when it should be executed. Thus, as shown in Figure 1, activities forming conditional branches or even loops in process specification are described linearly based on the actual execution of the multiple activity instances. Activities triggered by rules and enforced dependencies also have an impact on the construction of a process instance.

Figure 1: Specification Model vs. Execution Model

Thus, the instances of an activity’s children activities simply form either a sequential execution or a parallel execution. A multilevel business process instance is built recursively with sequential and parallel activity instances (for brevity, an activity instance is hereafter simply called an activity).

An activity at any level undergoes state transitions at run time; it is initiated in the *start* state, remains *active* for some time, and then exits by either reaching the end of its work or a failure; accordingly, it terminates either in the *commit* state or in the *abort* state.

Rolling back an activity hierarchy, either fully or partially, is not based on its specification, but on its instance execution history. Therefore, when control flow forms branches, only the executed activities are considered, and when control flow forms a loop each executed instance of activities is taken into account. As activities are parameterized (through the associated data containers), appropriate information can be passed from the instance of an activity to the instance of its compensation activity. For example, when activity *reserve-flight* is associated with the compensation activity *cancel-reservation*, the passenger’s identifier can be a parameter for both; and in this way a loop including multiple executions of *reserve-flight* can be compensated for by multiple executions of *cancel-reservation* with the proper parameters. Further, activities involved in a loop may be “re-entered” with different parameters, and “re-entering” actually means re-creating an instance. Thus, each execution of the activity involved in a loop can be considered as an individual activity instance associated with its own compensation activity instance. Those instances are distinguishable by their parameters.

The hyper-activity/block coupling allows a process to be handled at different abstract levels, and offers significant flexibility in layering control. For example, compensating the collective effects of a block of activities at a higher level

can avoid having to define compensation activities for each of them, since the latter is sometimes impossible.

4 Transactional Semantics

Business processes may access shared, persistent data, and thus must be subject to transactional semantics. However, executing a complex process as a single transaction is unnecessary in most cases. Instead, organizing a process hierarchically allows its effects on data objects to be controlled level by level with relaxed transactional semantics.

Many extended transaction models support nested structures and relaxed atomicity. For example, closed nested transactions [25, 17], allow concurrent subtransactions but strictly enforce atomicity; by allowing a subtransaction only to commit to its parent transaction tentatively, its partial results are not externalized until the top-level transaction commits. Since such atomicity is too rigid for engineering applications, the open nested transaction approach [29, 11, 3] relaxed it to enhance concurrency of subtransaction execution. By allowing a subtransaction to commit to the database independently of its parent transaction, the visibility of its effects can be broadened, but top-level atomicity and protection are sacrificed.

Similarly, Sagas [14] relax isolation for long-lived transactions, allowing their serial component transactions to be interleaved with each other. While a Saga is not atomic, it cannot be executed partially; also, it lacks the flexibility to retry an aborted component, to try an alternative component or even to ignore a component failure. These issues have been tackled by nested Sagas and ACTA [13, 9] but only independently committed (open) component transactions are considered. ATM [11] supports activity hierarchies with both (closed) transactions and (open) activities, but sacrifices atomicity of the top-level activity while improving inter-sub-activity parallelism; also, commit scopes other than closed and open are not considered. Therefore these mechanisms only provide two extremes in the tradeoff between maintaining atomicity and enabling concurrency.

4.1 Commit Control in Open Work Flow Manager

To adequately balance atomicity and concurrency and to control visibility and protection at selected levels of a transaction hierarchy, we have extended the notions of closed and open transactions by allowing an activity to commit to a selected ancestor independently of its parent, making its results visible to that ancestor and thus improving the concurrency in the transaction subtree beneath that ancestor [7, 6]. Such an activity is called a *scoped activity*. This extension provides synergy between advanced transaction models and workflow models for nested activity modeling.

The introduction of BDC provides data access control along the block hierarchy. Upon termination, an activity can

- *commit to parent* by checking in its results to the BDC of the containing block, making its effects visible to the activities inside that block;
- *commit to top* or *commit to an ancestor* by checking in its results to the corresponding BDC, making its effects visible to all activities inside that block; or
- *commit to database* by checking in its results into an external database, making its effects permanent and visible to public (i.e., to activities inside all top-level blocks).

As we described in [7], such multilevel commit scope control extends the closed/open nested activity model to accommodate applications that require improved process-wide concurrency without sacrificing top-level atomicity. In [6] we have presented a further extension that allows a subactivity to commit to an arbitrary ancestor. For simplicity, we do not discuss this extension in this paper.

5 Failure Handling

Transactional systems handle transaction failures by undoing (rolling back) in-progress transactions. For handling process failures, there are two complications. First, an in-progress process may actually have committed some transactions, so these cannot be undone; rather, they have to be compensated for. Second, requiring that the failure of any step in a business process cause the entire process to fail is expensive, because rolling back a process could potentially undo or compensate a lot of work. We allow activities inside a process or subprocess to be designated *vital* to indicate that the failure of any of them should cause the entire process to fail.

Our model supports a combination of *backward failure recovery* and *forward failure recovery*. This allows a process to partially roll back to a semantically accepted restart point, then continue to “roll forward”, making further progress.

5.1 Failure Handling Inside A Block

Handling failure inside a block or in a flat process is based on ideas found in advanced transaction models such as Sagas [14] for backward recovery (i.e., how to undo the effects of certain operations), or in Flex transactions [30] for forward recovery (i.e., how to select different paths of execution [12]).

In the following cases, we say that there exists an *in-block failure recovery plan* wrt T_f 's failure, if:

- (a) T_f is *non-vital* to the containing block **B**, so its cancellation can be ignored, and the block execution can continue;
- (b) T_f can be replaced by a contingency transaction acting as its exception handler, so upon its cancellation the block execution can continue by retrying the contingency transaction [11, 27, 29];
- (c) T_f is *retriable*, that is, it will eventually succeed if retried a sufficient number of times [24, 30];
- (d) the failure caused by T_f is recoverable within the block **B**, since the block execution can be rolled back to a transaction from which an alternative path is available (such a transaction is called *pivot* in the Flex Transaction approach [24, 30], and is implemented in [12]), and logically undoing any transaction on the rollback path does not cause block **B** to fail;
- (e) there exists a *modifiable* transaction, T_m , specified by the business process designer (e.g. modifying a design, a request, a schedule, etc.) preceding the failed transaction, T_f , such that the block execution can be logically rolled back (without causing **B** to fail) to T_m , then restarted by redoing T_m .

The restriction on cases (d) and (e) is essential: logically undoing any transaction T_k on the rollback path from T_f , must not cause block **B** to fail. If this restriction is unsatisfied, then even if there is a compensation transaction defined for T_k , we cannot ensure the existence of an in-block failure recovery plan wrt T_f , since undoing T_k would cause the failure to be passed to the higher-level block anyway.

When there exists an in-block failure recovery plan wrt T_f 's failure, the failure may be recoverable within block **B**. Otherwise, failure recovery by partially undoing transactions within block **B** only, is impossible.

However, in the presence of multilevel commit scopes, different pieces of partial results of a block may be visible to different scopes, and should be compensated in matched scopes to ensure consistent failure semantics. For example, when a seminar announcement is made accessible to a department, the seminar cancellation notification should be made accessible to the same department, rather than to a smaller or a larger scope.

5.2 Failure Handling in an Activity Hierarchy

Failure handling for nested activities has the following specific characteristics compared with that for flat or two-level transactions.

- In an activity hierarchy, the failure of a sub-activity may cause its parent or higher level ancestors to abort. The highest ancestor affected by such *abort-up* represents the scope of logical rollback, and possibly, the restart point for rolling forward through an alternative path.
- Different from compensating atomic transactions discussed in previous work, compensating an activity hierarchy may be made *directly*, or *indirectly* through compensating its children activities (e.g. *make-application* may be compensated for directly by *withdraw-application*, or indirectly by compensating each step). A compensation activity may either be flat or nested, and can be constructed differently from the activity to be compensated for. Further, an activity hierarchy should be *partially compensatable* for its already committed sub-activities, regardless of its own commit status.
- It is reasonable to try to logically roll back an activity subtree at the highest applicable level. Rolling back an activity subtree from its root can halt all its branches promptly; and compensating a higher-level activity is potentially more general than compensating a lower-level activity. For example, when activity *make-application* is directly compensated for by *withdraw-application*, the collective effects of *make-application* can be eliminated regardless of its internal steps (some of those steps may not even be compensatable).

Activity nesting provides failure protection in two general situations. First, when a failed activity is *non-vital* to its parent, the failure can be ignored, and the parent activity can continue. Second, when a failed activity can be replaced by a contingency activity acting as its exception handler, the process can continue by retrying the contingency activity. Therefore, when an activity T fails, the "abort-up" chain terminates at the closest ancestor of T that is non-vital, associated with a contingency activity, or *without a parent* (e.g. the top-level activity of a process, a contingency activity, or a compensation activity). We call such an activity the *Undo Root* of T .

Failure recovery consists in

- (a) bottom-up searching for the Undo Root of the originally failed activity, and
- (b) top-down logically rolling back the activity subtree under that Undo Root, using *compensate* and *abort* operations.

In top-down undoing, scoped activities with effects internal to the subtree under the Undo Root should be aborted and activities with effects externalized over the Undo Root should be compensated for. However, a scoped activity may or may not necessarily be compensated for directly since its effects may be logically eliminated by a compensation applied at a higher level. For an activity T that needs to be compensated for, if T is associated with a compensation activity $\bar{T}$, then T and the whole activity hierarchy beneath T can be directly compensated for by executing $\bar{T}$ with the same commit scope as T ; otherwise the children of T are processed, and the recursion may spread down along the activity hierarchy, with each branch ending up with a direct compensation or abortion. We have described this process in [6].

6 An Example

Figure 2 shows an example of a complex business process implemented on our OWFM prototype. This example illustrates a trip planning process for attending a conference. It demonstrates an open nested activity hierarchy, flexible flow control, activity dependencies, activity triggering with various modes, exception handling, activity compensation and recovery.

The top-level activity, *trip-planning*, is broken down into several sequential subactivities (their execution order is indicated by arrows): *trip-authorization*, *travel-reservation*, *notification*, *billing*, *expense-approval*, and *document filing*. Each subactivity may be further broken down, thus forming an **activity hierarchy**. Activities at leaf levels may be explicitly assigned to particular users. These subactivities may be executed sequentially, concurrently, or conditionally based on the system state, the user's option, or the prespecified priorities. For instance

- activity *notification* has four subactivities to be executed in parallel, for notifying the manager, co-workers, system administrator, and the conference organizers about the trip;
- activity *flight-reservation*, a subactivity of *travel-reservation*, has three optional subactivities that allow the traveler to choose from three airlines;
- activity *hotel-reservation* has two subactivities ordered by prespecified priorities, where Hilton is the first choice, and Ramada is considered only if Hilton is unavailable;
- activity *car-rental-reservation*, has conditional branches, where the selection of car-rental company is based on the selection of airline, e.g., if NW airline is chosen, Hertz car-rental will be selected, and if UA airline is chosen, National car-rental will be selected.

Activity dependencies describe the relationships between activities and can be used as constraints on activity firing. For example, under activity *travel-reservation*, there exist three subactivities for flight, hotel, and car-rental reservations. whose order of execution is not specified explicitly, but enforced by the following activity dependencies:

- *commit dependency (CD)*: the commit of *hotel-reservation* depends on the commit of *flight-reservation*; the commit of *car-rental-reservation* depends on the commit of *hotel-reservation*; (in fact, the conditional branching of *car-rental-reservation* is also expressible by commit dependencies);
- *exclusion dependency (EX)*: if Ramada hotel is taken, car-rental is unnecessary (we may assume that the conference is held at the Ramada hotel).

Activities may be triggered under *immediate*, *deferred* or *detached* modes. For example, the following activities are triggered upon successful termination of activity *expense-approval*:

- activity *travel-record* is triggered as an immediate child of *expense-approval*;
- activity *rearrange-appointments* is triggered as a deferred activity to be executed at the end of the top-level activity, *trip-planning*;
- activity *data-survey* is triggered as a detached, independent activity.

An activity that is unable to commit, may invoke a **contingency activity**. For example, if activity *expense-approval* cannot commit, its pre-defined contingency activity, *corporate-approval* is invoked, and if it succeeds, activity *expense-approval* is considered to be successful as well, and then the process continues to roll forward.

An activity may be associated with a predefined **compensation activity** that logically undoes the effect of that activity. For example, *flight-reservation* may be compensated for by *cancel-reservation*. In Figure 2, compensation activities are indicated by dashed-line boxes.

7 Conclusions

As enterprises try to re-engineer and automate critical business processes, there is the need for middleware for developing, executing, and managing business processes. Numerous workflow products have appeared on the market, but these do not scale to enterprise process management. In particular, these products provide limited capabilities for process modeling, and limited execution and failure semantics. The database research community has produced several extended transaction models, but more research is needed to describe the formal properties of these models, and to devise efficient implementation techniques.

Business process programming goes beyond the programming of individual database applications, since it deals with the linking of many applications into a complex process. A business process is a container for a set of activities and links between them. At a minimum, the specification of a process includes attributes such as name, create date and so on; *activities* (or nodes); *links* that link activities hierarchically and horizontally, either forward or backward; *rules*; *dependencies*; *process data*; and *supervisors* who can intervene during process execution. An activity specification may include: *attributes* such as name, create date, and possibly others; the *user* to whom the activity is assigned; the *action* to be performed; a *commit scope*; a *subset of process data*, or other visibility control information; *triggers with coupling modes*; *dependencies*, and so on.

According to the Workflow Management Coalition, the specification of a business process environment should include the following: *process groups* containing processes in specific application domains; *processes*; *invoked applications*; *process data*; *roles* that are users capable of doing certain work. Our richer model adds to these information about event services, workflow domains, transport mechanisms, resource managers, process execution status monitors, and user groups.

These requirements point to the need for business process programming languages and environments with rich formally defined semantics, and accompanying methodologies and tools; but successful solutions will have to integrate many technologies. The challenge lies in dealing with heterogeneous, distributed and legacy applications, and in supporting an acceptable degree of availability, scalability, and reliability. The goal of this paper is to urge the database programming research community to rise to the challenge of business process programming, a field that is in its infancy, and where there is the potential for significant impact.

References

- [1] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Gunthor, and M. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. IFIP WG8.1 Working Conference on Information System Development for Decentralised Organizations*, 1995.
- [2] D. Barbara, S. Mehrota, and M. Rusinkiewicz. INCAS: A computation model for dynamic workflows in autonomous distributed environments. *Technical Report, Matsushita Information Technology Laboratory*, 1994.

- [3] A. Buchmann, M. Ozsu, M. Hornick, D. Georgakopoulos, and F.A. Manola. A transaction model for active distributed object systems. in A. Elmagarmid (ed) *Transaction Models for Advanced Database Applications*, Morgan-Kaufmann, 1992.
- [4] O. Bulkhres, J. Chen, A. Elmagarmid, X. Liu, and G. Mullen. InterBase: A multidatabase prototype system. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1993.
- [5] Q. Chen. A rule-based object/task modelling approach. In *Proc. ACM-SIGMOD'86, SIGMOD Rec. Vol.15, No.2*, 1986.
- [6] Q. Chen and U. Dayal. Commit scope control in nested transactions. *Proc. 5th International Conference on Extending Data Base Technology (EDBT'96)*, 1996.
- [7] Q. Chen and U. Dayal. A transactional nested process management system. *Proc. 12th International Conference on Data Engineering (ICDE'96)*, 1996.
- [8] Q. Chen and Y. Kambayashi. Coordination of data and knowledge base systems in distributed environment. In *Proc. IFIP TC2/WG2.6 Conf. on Semantics of Interoperatable Database Systems, DS-5*, 1992.
- [9] P. K. Chrysanthis and K. Ramamritham. ACTA: The SAGA continues. in A. Elmagarmid (ed) *Transaction Models for Advanced Database Applications*, Morgan-Kaufmann, 1992.
- [10] U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1990.
- [11] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long running activities. In *Proc. 15th International Conference on Very Large Data Bases*, 1991.
- [12] G.Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Gunthor, and C. Mohan. Failure handling in large scale workflow management systems. *Research Report, RJ 9913, IBM Almaden Research Center*, 1994.
- [13] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested sagas. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1), 1991.
- [14] H. Garcia-Monina and K. Salem. SAGAS. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1987.
- [15] N. Gehani, H. Jagadish, and O. Shmueliu. Event specification in an active object oriented database. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1992.
- [16] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Malona. Specification and management of extended transactions in a programmable transaction environment. In *Proc. 10th International Conference on Data Engineering*, 1994.
- [17] J. Gray and A. Reuter. Transaction processing: Concepts and techniques. *Morgan Kaufmann Publishers*, 1993.
- [18] M. Hsu. Special issue on workflow and extended transaction systems. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(2), 1993.
- [19] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. In *Proc. 10th International Conference on Very Large Data Bases*, 1984.
- [20] N. Krishnakumar and A. Sheth. Specification of workflows with heterogeneous tasks in METEOR. In *Proc. 20th International Conference on Very Large Data Bases*, 1994.
- [21] F. Leymann and D. Roller. Business process management with flowmark. In *Proc. IEEE Computer Society Conference Compcon'94*, 1994.
- [22] D. McCarthy and U. Dayal. The architecture of an active database system. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1989.

- [23] D. McCarthy and S. Sarin. Workflow and transactions in InConcert. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(2), 1993.
- [24] S. Mehrotra, R. Rastogi, A. Silberschatz, and H.F. Korth. A transaction model for multidatabase systems. In *Proc. of the International Conference on Distributed Computing Systems*, 1992.
- [25] E. Moss. Nested transactions. In *MIT Press*, 1985.
- [26] M. Nodine and S. Zdonik. Cooperative transaction hierarchy: A transaction model to support design applications. In *Proc. 16th International Conference on Very Large Data Bases*, 1990.
- [27] K. Rothermel and C. Mohan. Aries/NT: A recovery method based on write-ahead logging for nested transactions. In *Proc. 15th International Conference on Very Large Data Bases*, 1989.
- [28] Workgroup1A The Workflow Management Coalition. The workflow reference model. *Workflow Management Coalition Document*, 1995.
- [29] G. Weikum and H. Schek. Concepts and applications of multilevel transactions and open nested transactions. in *A. Elmagarmid (ed) Transaction Models for Advanced Database Applications, Morgan-Kaufmann*, 1992.
- [30] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1994.

Figure 2: Trip-planning Example