Specification and Execution of Transactional Workflows

Marek Rusinkiewicz[*]
University of Houston
Houston, TX 77204-3475
marek@cs.uh.edu

Amit Sheth[†]
Bellcore
Piscataway, NJ 08854
amit@ctt.bellcore.com

Abstract

The basic transaction model has evolved over time to incorporate more complex transaction structures and to selectively modify the atomicity and isolation properties. In this chapter we discuss the application of transaction concepts to activities that involve coordinated execution of multiple tasks (possibly of different types) over different processing entities. Such applications are referred to as transactional workflows. In this chapter we discuss the specification of such workflows and the issues involved in their execution.

1 What is a Workflow?

Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities. A *task* defines some work to be done and can be specified in a number of ways, including a textual description in a file or an email, a form, a message, or a computer program. A *processing entity* that performs the tasks may be a person or a software system (e.g., a mailer, an application program, a database management system). Specification of a workflow involves describing those aspects of its constituent tasks (and the processing entities that execute them) that are relevant to controlling and coordinating their execution. It also requires specification of the relationships among tasks and their execution requirements. These can be specified using a variety of software paradigms (e.g., rules, constraints, or programs). Execution of the multiple tasks by different processing entities may be controlled by a human coordinator or by a software system called a *workflow management system*. Table 1 gives several examples of workflows used in various (computing) environments. In our discussion we will concentrate on workflows involving processing entities that are DBMSs or software application systems.

Many enterprises use multiple information-processing systems that, in most cases, were developed independently to automate different functions. These systems are often independently managed, yet contain related and overlapping data. Certain activities require the participation of multiple application systems and databases. Such activities are characterized by three main components: tasks, processing entities, and the constraints and correctness criteria that are enforced by appropriately coordinating the execution of tasks. When used without additional qualifications, the term workflow will refer to such multi-task activities. While such workflows can be developed using *ad hoc* methods, it is desirable that they maintain at least some of the safeguards of traditional transactions related to the correctness of computations and data integrity. A multidatabase transaction constitutes a special case of a workflow, in which the structuring, isolation and atomicity properties are determined by the underlying transaction model. The term multidatabase transaction will be used to refer to specific types of workflows that operate on multiple database systems and have certain transactional characteristics.

The multi-system workflows considered here cannot be addressed in the context of transaction models developed for the distributed database management systems (DDBMSs). The main problem is the need to preserve the *autonomy* of participating systems. Since many systems used in multi-system workflows were designed for stand-alone operation, they normally do not provide the information and services that would

[*]This work was supported, in part, by the Texas Advanced Research Program Grant number 3652008

[†]Some of the work was performed during the author's short sabbatical at ETH, Zurich.

Workflow Application	Typical Tasks	Typical Processing Entities
mail routing in office computing	email	mailer
loan processing in office computing [13]	form processing	humans, application software
purchase order processing in office computing [18]	form processing	humans, application software, DBMSs
service order processing in telecommunication [1]	transactions, "contracts"	application systems, DBMSs
product life-cycle management in systems manufacturing	transactions	application software, DBMSs

Table 1: Example Workflows

be necessary to execute the distributed transactions while supporting the required transaction semantics. Furthermore, even if such facilities were made available, this may require a complete rewriting of the existing systems and extensive modifications in the applications software (hardly an attractive prospect considering the complexity and expense of such an activity, especially while supporting on-going operations). It is necessary, therefore, to take advantage of the facilities that are provided by the component systems: rather than developing new "global" mechanisms that duplicate the functionality of local systems, we should build a model for managing multi-system workflows that utilize the known task structures, coordination requirements of a collection of tasks, and execution semantics of the systems that execute the tasks.

The remainder of this chapter is organized as follows. In the next section we briefly review related work in the area of multidatabase transaction and workflow models. Section 3 contains a discussion of the issues related to workflow specification. We show how an individual task can be specified and then review the problems of intertask dependencies, atomicity requirements, properties of the entities executing a task and their impact on the execution. This section also includes an example illustrating how simple multidatabase workflows can be specified using multidatabase SQL. Section 4 discusses the execution of workflows. We review the possible approaches to workflow scheduling, including the problems of concurrent execution and recoverability.

2 Related Work

In this section we will briefly discuss the evolution of transaction models. The transaction models discussed in this section can be classified according to various characteristics including transaction structure, intra-transaction concurrency, execution dependencies, visibility, durability, isolation requirements, failure atomicity, and correctness criteria for concurrent execution. In the discussion below, we use the term *traditional transactions* to refer to transactions endowed with the atomicity, consistency, isolation and durability (ACID) properties. *Extended transactions* permit grouping of their operations into hierarchical structures. The term *relaxed transactions* is used to indicate that a given transaction model relaxes (some of) the ACID requirements. We first discuss the the relevant work in extended and relaxed transaction models [14] and then the workflow models.

Extended and Relaxed Transaction Models

An important step in the evolution of a basic transaction model was the extension of the flat (single level) transaction structure to multi-level structures. A *Nested Transaction* [39] is a set of subtransactions that may recursively contain other subtransactions, thus forming a *transaction tree*. A child transaction may start after its parent has started and a parent transaction may terminate only after all its children terminate. If a parent transaction is aborted, all its children are aborted. Nested transactions provide full isolation on the global level, but they permit increased modularity, finer granularity of failure handling, and a higher degree of intra-transaction concurrency than the traditional transactions. *Open Nested Transactions* [51] relax the isolation requirements by making the results of committed subtransactions visible to other concurrently executing nested transactions. They also permit one to model higher level operations and to exploit their application-based semantics, especially the commutativity of operations.

In addition to the extension of internal transaction structure, relaxed transaction models focus on selective relaxation of atomicity or isolation and may not require serializability as a global correctness criterion. They frequently use inter-transaction execution dependencies that constrain scheduling and execution of component transactions. Many of these models were motivated by specific application environments and attempt to exploit application semantics.

Most of the relaxed transaction models use some form of compensation. A subtransaction can commit and release the resources before the (global) transaction successfully completes and commits. If the global transaction later aborts, its failure atomicity may require that the effects of already committed subtransactions be undone by executing *compensating subtransactions*. Relaxing the isolation of multidatabase transactions may cause violation of global consistency (global serializability), since other transactions may observe the effects of subtransactions that will be compensated later [19, 32]. The concept of a *horizon of compensation* in the context of multi-level activities has been proposed in [33]. Under this model a child operation can be compensated only before its parent operation commits. Once the parent operation commits, the only way to undo the effects of a child operation is to compensate the entire parent operation.

The concept of a *Saga* was introduced in [19] to deal with long-lived transactions. A saga consists of a set of ACID subtransactions $\mathrm{T}_1, \ldots, \mathrm{T}_n$ with a predefined order of execution, and a set of compensating subtransactions $\mathrm{CT}_1, \ldots, \mathrm{CT}_{n-1}$, corresponding to $\mathrm{T}_1, \ldots, \mathrm{T}_{n-1}$. A saga completes successfully if the subtransactions $\mathrm{T}_1, \ldots, \mathrm{T}_n$ have committed. If one of the subtransactions, say T_k, fails, then committed subtransactions $\mathrm{T}_1, \ldots, \mathrm{T}_{k-1}$ are undone by executing compensating subtransactions $\mathrm{CT}_{k-1}, \ldots, \mathrm{CT}_1$. Sagas relax the full isolation requirements and increase inter-transaction concurrency. An extension allowing the nesting of sagas has been proposed in [20].

Split- and Join- Transactions [40] were designed for open-ended activities characterized by uncertain, but usually very long-duration, unpredictable development, and interaction with other activities. A transaction may split into two separate transactions (the resources are divided), and later join another transaction (the resources are merged). Split transactions provide a mechanism for direct resource transfer, and provide adaptive recovery (a part of the work may be committed before completion of a transaction).

Flexible Transactions [42, 16] have been proposed as a transaction model suitable for a multidatabase environment. A flexible transaction is a set of tasks, with a set of functionally equivalent subtransactions for each and a set of execution dependencies on the subtransactions, including failure dependencies, success dependencies, or external dependencies. To relax the isolation requirements, flexible transactions use compensation and relax global atomicity requirements by allowing the transaction designer to specify acceptable states for termination of the flexible transaction, in which some subtransactions may be aborted. IPL [8] is a language proposed for the specification of flexible transactions with user-defined atomicity and isolation. It includes features of traditional programming languages such as type specification to support specific data formats that are accepted or produced by subtransactions executing on different software systems, and preference descriptors with logical and algebraic formulae used for controlling commitments of transactions.

Polytransactions [46] have been proposed as a mechanism to support maintenance of interdependent data in a multidatabase environment. It is assumed that interdatabase consistency requirements are specified as a collection of Data Dependency Descriptors (D^3). Each D^3 contains a description of the relationships among data objects, together with consistency requirements and consistency restoration procedures. A polytransaction T^+ is a "transitive closure" of a transaction T with respect to all the D^3s. The main advantage of polytransactions is that they transfer the responsibility for preserving interdatabase consistency from an application programmer to the system.

Reasoning about various transaction models can be simplified using the *ACTA metamodel* [9, 10]. ACTA captures some of the important characteristics of transaction models and using it one can decide whether a particular transaction execution history obeys a given set of dependencies. However, defining a transaction with a particular set of properties and assuring that an execution history will preserve these properties remains a difficult problem.

Workflow Models

A fundamental problem with many transaction models that have been proposed is that they provide a predefined set of properties that may or may not be required by the semantics of a particular activity. Another problem with adopting these models for designing and implementing workflows is that the systems involved in the processing of a workflow (processing entities) may not provide support for facilities implied by a transaction model. Furthermore, the extended and relaxed transaction models are geared mainly towards processing entities that are DBMSs. The desire to overcome these limitations was a motivation for the development of workflow models.

The idea of a workflow can be traced to Job Control Language (JCL) of batch operating systems, such as OS, which allowed the user to specify a job as a collection of steps. Each step was an invocation of a program and the steps were executed in sequence. Some steps could be executed conditionally, for example, only if the previous step was successful or if it failed. This simple idea was subsequently expanded in many products and research prototypes permitting structuring of the activity, and providing control of concurrency and commitment. The extensions allow the designer to specify the data and control flow among tasks and to **selectively choose** transactional characteristics of the activity, based on its semantics.

ConTracts were proposed in [41] as a mechanism for grouping transactions into a multitransaction activity. A ConTract consists of a set of predefined actions (with ACID properties) called *steps*, and an explicitly specified execution plan called a *script*. An execution of a ConTract must be *forward-recoverable*, that is, in the case of a failure the state of the ConTract must be restored and its execution may continue. In addition to the relaxed isolation, ConTracts provide relaxed atomicity so that a ConTract may be interrupted and re-instantiated.

Some issues related to workflows were addressed in the work on *Long-Running Activities* [11, 12]. A Long-Running Activity is modeled as a set of execution units that may consist recursively of other activities or (top) transactions (i.e., transactions that may spawn nested transactions). Control flow and data flow of an activity may be specified statically in the activity's *script*, or dynamically by Event-Condition-Action (ECA) rules. This model includes compensation, communication between execution units, querying the status of an activity, and exception handling.

A recent proposal for a programmable transaction environment also contains several features of workflows, including support for a variety of processing entities and a variety of coordination and correctness requirements [22].

Enforcement of intertask dependencies in workflows is discussed in [3]. Tasks are modeled by providing their states together with significant events corresponding to the state transitions (start, commit, rollback, etc.), that may be forcible, rejectable, or delayable. Intertask dependencies, such as the order dependencies $e_1 < e_2$ and existence dependencies $e_1 \rightarrow e_2$ between significant events of tasks are formally specified using Computation Tree Logic (CTL) and have corresponding dependency automata that can be automatically generated. The dependencies may be enforced by checking relevant dependency automata.

Other terms used in the database and related literature to refer to workflows are task flow, multi-system applications [1], application multiactivities [34], networked applications [13] and long-running activities [12]. Related topics are also discussed in the context of cooperative activities [35] or cooperative problem solving [7].

3 Specification of Workflows

The following are key issues in specifying a workflow:

- Task specification: The execution structure of each task is defined by providing a set of externally observable execution states and a set of transitions between these states. In addition, those characteristics of processing entities that are relevant to the task-execution requirements may be defined.
- Task Coordination Requirements: Coordination requirements are usually expressed as intertask-execution dependencies and data-flow dependencies, as well as the termination conditions of the workflow.
- Execution (Correctness) Requirements: Execution requirements are defined to restrict the execution of the workflow(s) to meet application-specific correctness criteria. These include failure-atomicity

requirements, execution-atomicity requirements (including the visibility rules indicating when the results of a committed task become visible to other concurrently executing workflows), as well as (inter-)workflow concurrency control and recovery requirements.

These issues will be discussed in the following subsections.

3.1 Specification of a Task in a Workflow

A task in a workflow is a unit of work that can be processed a processing entity, such as an application system or a DBMS. A task can be specified independently of the processing entity that can execute it or by considering the capabilities and the behavior of the processing entity. In the latter case, the task is specified for execution by a specific entity or a specific type of processing entities. For example, a task specification may include a precommit state and its execution may be limited to those processing entities that support such a state. We will limit our attention to the case where a task is defined for a specific type of processing entity.

Not all aspects of tasks need to be modeled for the purpose of workflow management. Let us take an example of a transaction executed by a DBMS. From the view point of a workflow, all details of the transaction that describe its sequential processing are unnecessary. Each task performs some operations on its underlying (database) system. Therefore a task is a program (transaction) and it is very important that it be "correct". However, as with the correctness of traditional transactions, on the workflow level we do not model internal operation of the task - we deal only with those aspects of a task that are externally visible.

Hence, a task structure can be defined by providing:

- a set of (externally) visible execution states of a task,
- a set of (legal) transitions between these states, and
- the conditions that enable these transitions (the transition conditions can be used to specify intertask execution requirements).

An abstract model of a task is a state machine (automaton) whose behavior can be defined by providing its *state transition diagram*. In general, each task (and the corresponding automaton) can have a different internal structure resulting in a different state transition diagram. This depends, to a large extent, on the characteristics of the system on which the task is executed. Some of the properties of the processing entities responsible for the execution of a task, like presence or absence of the two-phase commitment interface, will directly affect the task structure and thus the definition of the workflow. Figure 1 shows the structure of some frequently encountered types of tasks.

Other characteristics of a system that executes a task may influence the properties of a task, without affecting its structure. For example, a system executing a task may guarantee *analogous execution and serialization order* [4], which may allow the workflow scheduler to affect the local serialization order of the tasks by controlling their commitment (start, submission) order. Similarly, a system may guarantee *idempotency*[1], thus allowing safe repetition of a task, if a positive acknowledgment is missing or timed out.

Transitions between various states of a task may be affected by various scheduling events. Some of these transitions are controlled by the scheduler responsible for enforcing intertask dependencies. For example, a task can be submitted for execution thus resulting in a state transition from "Initial" to "Executing". Other transitions are controlled by the local system responsible for the execution of the task. For example, an executing task may be unilaterally aborted by its system, thus resulting in the state transition from "Executing" to "Aborted". One or more states of a task may be designated as its termination states. When a task reaches such a state, no further state transitions are allowed. Finally, a task may have various isolation properties. For example, results of an incomplete task may be made visible to other concurrent tasks, or they may be deferred until task commitment. These and other properties have an effect on the concurrency control and recovery mechanism that can be used by the scheduler.

[1] We say that a system is idempotent with respect to a task of type T, if the task can be executed one or more times without changing the result. Examples of idempotent tasks are: "set counter c to 0" or "allocate resource number x to the process number y" (but not "increment counter c" or "allocate an instance of resource of type X to process number y").

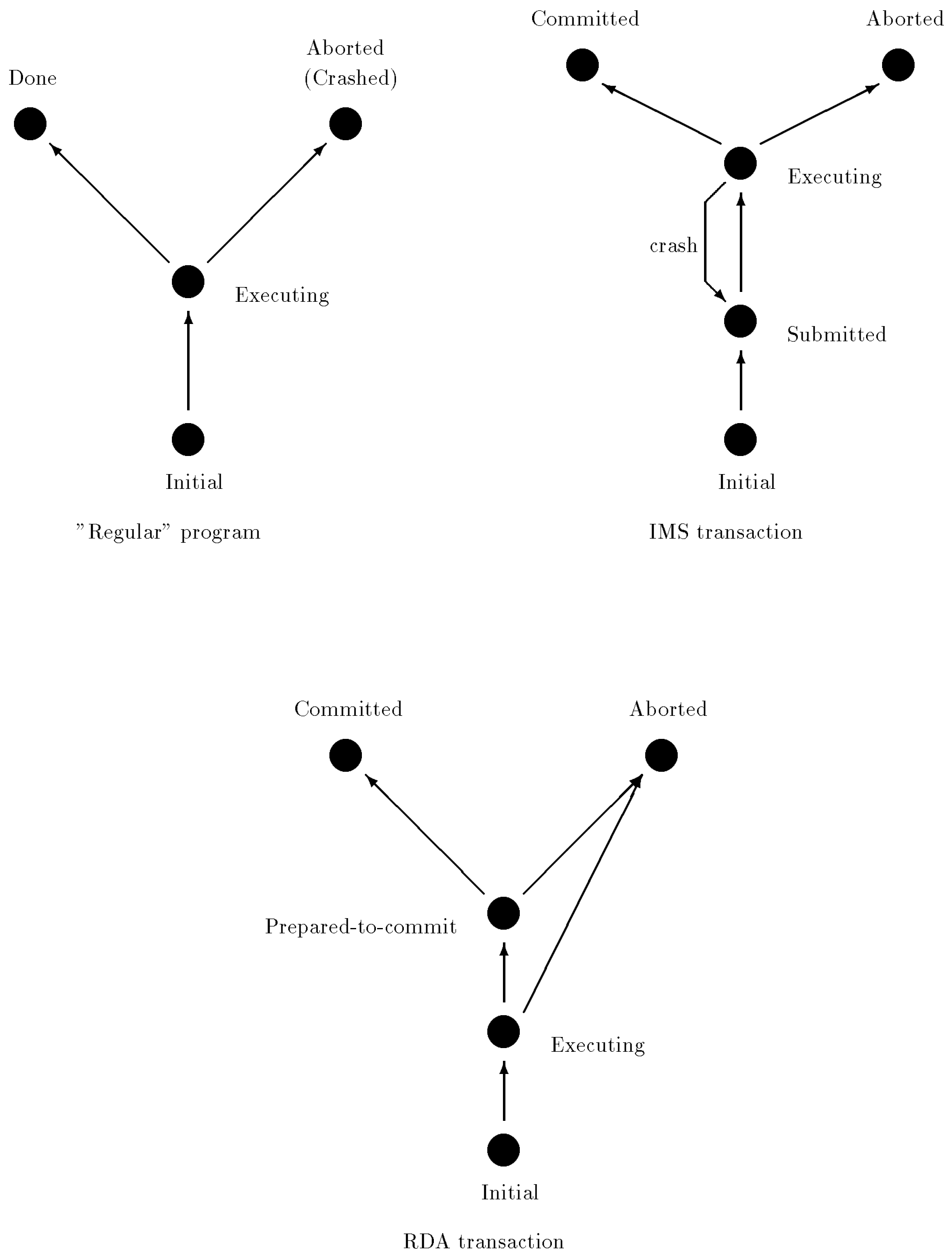

Figure 1: Examples of Task State Transition Diagrams

A partial output of a task may be made available to other concurrently executing tasks or a task may request input from other tasks. We assume that tasks of a workflow can communicate with each other through persistent variables, local to the workflow. These variables may hold parameters for the task program. Different initial parameters for the task may result in different executions of a task. The *data flow* between subtransactions is determined by assigning values to their input and output variables. The execution of a subtransaction has effects on the state of a database and the value of its output variable.

Figure 2 depicts an abstract external view of a task. A task may use parameters stored in its input variable(s), it may retrieve and update data in the local system, store its results in its output variable(s), and may be queried about its execution state.

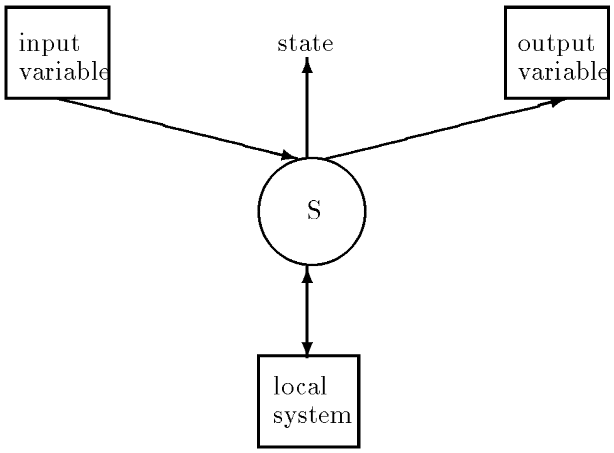

Figure 2: An abstract view of a task in a workflow

At any time the execution state of a workflow can be defined as a collection of states of its constituent tasks and the values of all variables (including temporal). The execution of a workflow begins in an *initial state*. The initial state of a workflow specifies its initialization parameters. Different initial states of a workflow may result in different executions.

3.2 Task Coordination Requirements

Once the tasks constituting a workflow are given, the internal structure of the workflow (or control flow) can be defined by specifying the task coordination requirements, usually as scheduling preconditions for each scheduler-controllable transition in a task. In general they can either be statically defined or determined dynamically during its execution.

- **Statically:** In this case the tasks and dependencies among them are defined before the execution of the workflow starts. Some of the relaxed transactions (e.g. Flexible Transactions [16]) use this approach. A generalization of this strategy is to have a precondition for execution of each task in the workflow, so that all possible tasks in a workflow and their dependencies are known in advance, but only those tasks whose preconditions are satisfied are executed. Such an approach is reported in [1]. The preconditions may be defined through dependencies involving the following:
  - *Execution states* of other tasks. For example, "task t_1 cannot start until task t_2 has ended" or "task t_1 must abort if task t_2 has committed".
  - *Output values* of other tasks. For example "task t_1 can start if task t_2 returns a value greater than 25".
  - *External variables*, that are modified by external events that are not a part of the workflow (but may be related to the events of other tasks in the workflow or other workflows). Examples of such conditions are: "task t_1 cannot be started before 9AM GMT", or "task t_1 must be started within 24 hours of the completion of task t_2".

The terms *execution dependencies*, *data* or *value dependencies* and *temporal dependencies* are used in the literature to refer to various scheduling preconditions. The dependencies can be combined using the regular logical connectors (OR, AND, NOT) to form complex scheduling preconditions.

ConTracts [41], multitransactions [18] and multidatabase transactions [43] support *á priori* specification of dependencies.

- **Dynamically**: In this case, the task dependencies are created during the execution of a workflow, often by executing a set of rules. Examples are long-running activities [12] and polytransactions [46]. The events and conditions affecting the rule processing may change with changes in the execution environment and/or with earlier task executions.

3.3 Failure-Atomicity Requirements of a Workflow

Using the understanding of semantics of a workflow and of the multisystem consistency constraints, the workflow designer may specify the *failure atomicity* requirements of the workflow. The traditional notion of failure atomicity would require that a failure of any task results in the failure of the workflow. However, a workflow can, in many cases, survive the failure of one of its tasks, for example, by executing a functionally equivalent task at another site. Therefore, we should allow the designer to define failure-atomicity requirements of a workflow. The system must guarantee that every execution of a workflow will terminate in a state that satisfies the failure-atomicity requirements defined by the designer. We will call those states *acceptable termination states* of a workflow. All other execution states of a workflow constitute a set of *non-acceptable termination states*, in which the failure-atomicity may be violated.

An acceptable termination state can be designated as committed or aborted. A *committed acceptable termination state* is an execution state in which the objectives of a workflow have been achieved. In contrast, an *aborted acceptable termination state* is a valid termination state in which a workflow has failed to achieve its objectives. If an aborted acceptable termination state has been reached, all undesirable effects of the partial execution of the workflow must be undone in accordance with its failure-atomicity requirements.

In general, a task can commit and release its resources before the workflow reaches a termination state. However, if the multitask transaction later aborts, its failure-atomicity may require that the effects of already completed tasks (e.g., committed subtransactions) be undone by executing *compensating tasks (subtransactions)* [25]. The notions of acceptable termination states and scheduling dependencies can be used to express the semantics of compensation without resorting to special constructs as required in other transaction models. The semantics of compensation requires that a compensating transaction eventually completes its execution successfully, possibly after a number of resubmissions. In the model described here, this property of compensating transactions can be defined by appropriately specifying their scheduling preconditions.

3.4 Execution Atomicity Requirements of a Workflow

Similarly to the failure-atomicity requirements, the designer can specify *execution-atomicity* requirements of a workflow. The traditional transaction model would require that a whole workflow constitute an execution-atomic unit. Therefore, an interleaved execution of workflows should have the same effects as if they were executed serially, in some order. Relaxing execution atomicity of transactions in centralized databases has been discussed in [37]. In [17], a transaction is divided into execution-atomic steps and interleaving with other concurrent transactions is allowed only between these steps. In the workflow context, tasks are usually natural execution-atomic steps, since they execute on separate processing entities.

However, sometimes the data integrity constraints span the boundaries of individual databases and, as a consequence, the tasks accessing interrelated data must constitute an execution atomic unit. For example, consider a workflow that transfers money between accounts in two different banks. To avoid inconsistent retrievals, tasks (subtransactions) accessing databases of those banks should constitute an execution-atomic unit with respect to other concurrent transactions.

3.5 Specification of Multidatabase Workflows in Extended SQL

Multidatabase SQL (MSQL) [36] is an extension of the SQL query language proposed as an access language for loosely coupled multidatabase systems. Since SQL is both an official and a *de facto* standard for relational databases, it is reasonable to think of MSQL as a testbed for a new emerging standard for multidatabase environments. The basic idea is that of providing SQL with new functions for non-procedural manipulation of data in different and mutually non-integrated relational databases. In this subsection we will briefly discuss the recent extensions to MSQL proposed in [48] and show how they can be used to specify failure atomicity requirements of multidatabase workflows.

MSQL allows the user to change the states of multiple databases. Therefore the semantics of such multiple updates must be carefully defined. The following example helps in understanding the problems involved in the implementation of such updates in a loosely coupled environment. Let us consider a multidatabase system providing access to databases of airlines that store information about availability of seats on different flights and databases of car rental companies that store information about the availability of their cars. Let us suppose that we want to *raise the fares of flights from Houston to San Antonio on Continental, Delta and United by 10%.* This update can be specified by the following MSQL statement:

```
USE       continental delta united
UPDATE    flights
SET       rate = rate * 1.1
WHERE     source = 'Houston' AND
          destination = 'San Antonio'
```

In the above example, the USE statement specifies the scope of the query or update identifying the databases to be accessed[2]. The multiple update is decomposed into three subtransactions to be executed by the three Local Database Systems (LDBS) of Continental, Delta and United. We assume that the LDBSs are autonomous and heterogeneous, hence they may use different two-phase commitment (2PC) protocols; some may not support 2PC on may not provide a visible 2PC interface. Each system may be forced to abort its local subquery for reasons such as local conflicts, failure, or deadlock. The result of a multiple update may leave the multidatabase in a state that is inconsistent from the global user point of view. The only possibility to check if the multiple update was consistent would be to access each of the involved LDBSs and see what has happened. To address this problem the USE statement illustrated above has been extended to allow the user to specify the desired level of consistency for the execution of a particular multiple update. The multiple update shown above can be modified as follows:

```
USE       continental VITAL delta united VITAL
UPDATE    flights
SET       rate = rate * 1.1
WHERE     source = 'Houston' AND
          destination = 'San Antonio'
```

The semantics of VITAL designators are similar to those defined in [18] for sub-sagas. Databases in the query scope are designated as VITAL or NON VITAL (default). All VITAL subqueries must either commit or abort, so that the desired multidatabase consistency is maintained. A multiple query is successful when all VITAL subqueries commit. It fails when all VITAL subqueries are rolled back. The execution is considered incorrect if some VITAL subqueries are committed and some others are not. All NON VITAL subqueries can be executed in auto-commit mode, since their results have no effect on the success or failure of the global multiple query. If all subqueries are NON VITAL, the multiple query is always successful. The set of VITAL databases is called the *vital set*. Failure atomicity is enforced with respect to the vital set.

The described semantics of the VITAL designators are not applicable in cases in which the user wants to include in the vital set databases that do not support 2PC. If two or more such databases are VITAL it

[2]Since naming and schema heterogeneities may exist in such an environment, MSQL provides mechanisms for their resolution [36].

is not possible to enforce failure atomicity with respect to the vital set. Nothing can be done if one of them commits and another aborts the related subquery, and global consistency is violated. A possible solution to this problem is the use of compensation. The extended MSQL allows the specification of compensating actions for individual data-manipulation statements. For each VITAL database in the scope of the query that does not support 2PC, the user must provide a COMP clause in which the needed compensating actions are specified. For example, assuming that the Continental database does not provide 2PC, the previous multiple update can be rewritten in the following way:

```
USE      continental VITAL delta united VITAL
UPDATE   flights
SET      rate = rate * 1.1
WHERE    source = 'Houston' AND
         destination = 'San Antonio'
COMP     continental
         UPDATE   flights
         SET      rate = rate / 1.1
         WHERE    source = 'Houston' AND
                  destination = 'San Antonio'
```

With the specification of the compensating action for the local update to the Continental database, the original semantics of the VITAL designator are preserved. If the Continental update is aborted the United update can be rolled back. If the United update is aborted, the Continental update can be compensated.

The introduction of VITAL designators and compensation is a step in the direction of the specification of multidatabase transactions in relational environments. MSQL queries which specify VITAL subqueries and compensating actions constitute small transactional units. The natural next step is the specification of more complex transactions. In [48] we describe how multidatabase transactions can be specified in MSQL. The main idea is to expand the COMMIT statement to allow the specification of the failure atomicity requirements of a transactions. For example, we can specify acceptable combination of commitment of tasks by using the following syntax

```
COMMIT (when)
(Continental AND National) (or)
(Delta AND Avis)
```

In this example, the global transaction corresponding to the whole workflow will be committed only if either the transactions submitted to Continental and National databases commit, or if the transactions submitted to Delta and Avis databases commit. In all other cases the global transaction will be aborted.

4 Execution of Workflows

A workflow-management system must permit specification and scheduling of intertask dependencies. In addition, concurrency and recovery may be supported, in which case it may be possible to integrate the scheduler enforcing intertask dependencies with a relaxed transaction management system.

A workflow management system consists of a scheduler and task agents. A task agent controls the execution of a task by a processing entity; there is one task agent per task. A scheduler is a program that processes workflows by submitting various tasks for execution, monitoring various events, and evaluating conditions related to intertask dependencies. A scheduler may submit a task for execution (to a task agent) or request that a previously submitted task be aborted. In the case of multidatabase transactions, the tasks are subtransactions and the processing entities are local DBMSs. In accordance with the workflow specifications, the scheduler enforces the scheduling dependencies and is responsible for assuring that that a tasks reaches an acceptable termination state.

There are three architectural approaches to the development of a workflow management system. A centralized approach has a single scheduler that schedules the tasks for all concurrently executing workflows.

The partially distributed approach is to have one (instance of) a scheduler for each workflow. When the issues of concurrent execution can be separated from the scheduling function, the latter option is a natural choice. A fully distributed approach has no scheduler, but the task agents coordinate their execution by communicating with each other to satisfy task dependencies and other workflow execution requirements.

4.1 Scheduling of a Workflow

We first discuss the objectives or a scheduler and then review some approaches and prototypes.

4.1.1 The Objectives of a Scheduler

The main objectives of a scheduler are to assure:

- **Correctness of the scheduling.** The scheduling process cannot violate any of the dependencies provided in a workflow specification. Additionally, the scheduler is limited by constraints imposed by the global concurrency control, since uncontrolled interleaving of tasks belonging to different workflows may lead to incorrect results. Determining if the temporal scheduling dependencies can be satisfied is particularly difficult [23]. The scheduler must be aware that in the presence of temporal dependencies the logical value of scheduling predicates can change dynamically, without any action of the system. At the same time, these dependencies limit the possible actions of the scheduler (e.g., by specifying that a task must not start before 10:00 am).

- **Safety.** The scheduler must guarantee that a workflow will terminate in one of the specified acceptable termination states. Before attempting to execute a workflow, the scheduler should examine it to check whether it may terminate in a non-acceptable state. If the scheduler cannot guarantee that a workflow will terminate in an acceptable state, it must reject such specifications without attempting to execute the workflow.

  As an example, let us consider a workflow consisting of two tasks represented by subtransactions S_1 and S_2, and the usual failure-atomicity requirements indicating that either both subtransactions are committed or none of them is. If we assume that S_1 and S_2 do not provide prepared-to-commit state and do not have compensating transactions, three execution strategies are possible:

  1. execute S_1 first and if S_1 commits, then submit S_2,
  2. as above, but try S_2 first and then S_1, or
  3. try to execute both subtransactions concurrently.

  In cases (1) and (2), if the second subtransaction aborts, the workflow is in a non-acceptable termination state. The same is true for (3) if one subtransaction commits and the other aborts. Therefore, such a workflow specification should be considered *unsafe* and rejected. Similarly, if in the course of processing a workflow he scheduler discovers that there is no safe continuation, the workflow should be immediately aborted.

- **Optimal scheduling policy.** A scheduler should achieve an acceptable termination state in the "optimal" way. However, the meaning of *optimal* can vary from application to application. One possibility is to define it as achieving the goal in the shortest possible time. Alternatively, we may associate a cost function with the execution of every task. The objective of a scheduler would then be to execute the entire workflow with the minimal possible cost. If the probabilities of tasks' commitment are known in advance, the scheduler can use them to find an execution strategy which yields the maximal probability of a global commit.

- **Handling of Failures.** A scheduler should be able to reach an acceptable termination state even in the case of a failure. For example, the scheduler could continue processing after failure and recovery, as if "nothing happened," thus providing forward recoverability. Otherwise, the scheduler could abort the whole workflow (i.e., reach one of the global abort states). Both approaches require that state information be preserved in the case of a failure, since even in the latter case some subtransactions may need to be committed or even submitted for execution (e.g., compensating subtransactions). Therefore, the scheduler should log on a secure storage all the information about its state that it would need to recover and proceed.

4.1.2 Scheduling approaches

Several schedulers for multidatabase transactions are described in the literature. However, most of the proposed solutions address only some of the issues identified above. Therefore, they can be useful only in special, restricted cases. With the exception of [3], all the schedulers were primarily developed for multidatabase transactions, a special type of workflows. Although the problem has attracted the attention of many researchers, no comprehensive and practical solution exists yet. We briefly review some of the prototypes and approaches below.

- **A scheduler based on the Predicate Petri Nets model** ([16]). This scheduler was written for Flexible Transactions. The scheduler uses Predicate Petri Nets to identify a set of subtransactions schedulable in a given state. The construction of the Petri Net reflects in its structure the precedence predicates associated with subtransactions. However, this scheduler does not address safety nor optimality issues. Therefore, it cannot guarantee that a multidatabase transaction will terminate in an acceptable termination state.
- **An executor for Flexible Transactions in a logically parallel language L.0** ([6, 2]). This scheduler for Flexible Transactions achieves the maximal available parallelism among subtransactions, hence it can execute a transaction in the shortest time. However, the execution can be quite expensive (in a case when only one subtransaction out of N should be committed, the program will execute all N transactions and then compensate $N-1$ of them). This method assumes that all subtransactions are compensable. If this assumption does not hold, the safety of scheduling is not guaranteed and a transaction can stop in a non-acceptable termination state.
- **A Scheduler as an interpreter of multidatabase transaction specification language.** The underlying idea is to map the transaction specifications into a set of production rules or logic clauses. Such a specification can then be interpreted as a pseudocode, to directly control processing of the multidatabase transaction. The responsibilities of the transaction designer are much broader in this case, since the high-level transaction specifications must be translated into a logic program, which is a tedious and error-prone task.

  An example of such an approach for Flexible Transactions is described in [31], where the Vienna Parallel Logic (VPL) language is used for multidatabase transaction specifications. A multidatabase transaction is specified as a set of executable VPL queries. The language is powerful enough to express both serial and parallel executions, explicit commitment, and to specify data exchange between subtransactions. As a Prolog-based language VPL provides backtracking, which in this case means compensating and/or aborting subtransactions. The solution tree is searched until the terminating predicate is satisfied or the tree is traversed. Therefore, if a solution exists, it will be found, although no guarantees concerning its optimality can be given. The quality of the solution (including its correctness and safety of the execution strategy) depends to a large extent on the programmer who wrote the specifications.
- **A Scheduler as an interpreter of Event-Condition-Action (ECA) rules [11, 12]**. The authors describe the execution of long-running activities. The scheduler executes a script augmented by the actions that may be triggered as a result of ECA rules. A similar approach is discussed in [22], where the intertask dependencies in multidatabase transactions are implemented using (ECA) rules.
- **Scheduling as a game versus Nature.** An approach under which the scheduling process is modeled as a game of the scheduler against its environment represented by the LDBSs is described in [43]. The LDBSs are considered to be Nature, i.e., a stochastic, non-hostile player. A *move* in this game means changing the state of one or more subtransactions. Some changes can be done by the scheduler while others depend on accessed LDBSs. For example, the scheduler can submit a subtransaction to execute, thus changing its state from *Initial* to *Executing*. The LDBS can abort an executing subtransaction,

changing its state from *Executing* to *Aborted*. The scheduler wins when the multidatabase transaction reaches an acceptable termination state. This method exploits the maximal available parallelism and generally leads to the shortest execution time. The disadvantage of this approach is that it may lead to some transactions' being executed unnecessarily, to be compensated later.

- **Scheduler as a finite-state automaton** ([29]). In this model the scheduler uses a finite-state automaton to analyze dependencies among subtransactions. The scheduler can use protocol analyzing tools to determine reachability of an acceptable state. This approach would guarantee a correct and safe processing strategy. If optimality criteria could be considered as yet another kind of dependency, and implemented in the same way, it would also provide the optimal schedule. In the current state of development, this method suffers from high computational complexity due to the state explosion. Therefore, multidatabase transactions composed of a large number of subtransactions cannot be processed in this way. This scheduler has a partially distributed architecture.

- **Scheduling and enforcing intertask dependencies using temporal propositional logic** [3]. In the Carnot project, carried out at MCC in collaboration with Bellcore and the University of Houston, each task is modeled as a collection of significant events (start, commit, rollback, etc.), that may be forcible, rejectable, or delayable. Transaction semantics is defined using order dependencies $e_1 < e_2$ and existence dependencies $e_1 \rightarrow e_2$ between significant events of tasks. Intertask dependencies are specified as constraints on the occurrence and temporal order of significant events of the related tasks. A temporal propositional logic called Computational Tree Logic (CTL) is used to specify dependencies discussed in [10, 30]. This allows automatic generation of automata that enforce the dependencies. By accepting, rejecting, or delaying requests, the scheduler can enforce all dependencies. The scheduler is provably correct and safe. This scheduler has a centralized architecture and high computational cost. Hence it is not appropriate for managing many intertask dependencies without additional optimization.

4.2 Concurrent Execution of Workflows

We assume that each task of a workflow is executed under the control of an individual processing entity (e.g., a DBMS) that provides local concurrency control to the extent required by the semantics of each task. This guarantees that each processing system is left in a (locally) consistent state; however, it may not be sufficient to guarantee global correctness of concurrent and interleaved execution of workflows.

Assuming that each workflow is executed correctly, a concurrent execution of multiple workflows is correct if it is in some sense equivalent to running these workflows one at a time, without interference. In the absence of any additional information about the constraints that exist among the states of the multiple systems involved in the execution of a workflow and about the properties of the workflows, assuring such equivalence requires enforcement of global serializability and global commitment of workflows. If additional information (e.g., the failure- and isolation-atomicity requirements of each workflow) is available, weaker correctness criteria for concurrent execution of workflows may become applicable. In the discussion below, we first review basic concepts of multidatabase transaction management that are applicable in workflow management and then discuss the possible extensions. In [28], we discuss application and system semantics in a real-world environment that allows the use of simple and efficient concurrency control and recovery methods.

Global serializability. Global serializability requires tasks belonging to different workflows to have the same relative serialization order at all sites on which they execute. To assure that global serializability is not violated, local histories must be validated by the workflow-management system. The problems of determining local serialization order were discussed in the literature on multidatabase transactions [5]. The main difficulty is caused by the possibility of *indirect conflicts* that may be caused by the local tasks executed outside of global workflows. A possible mechanism for detection of inter-workflow conflicts may be based on the ticket concept proposed in [24].

Global commitment. We say that a workflow can become *globally committed* when it reaches an acceptable termination state. To assure failure atomicity of a workflow, recovery procedures must deal with problems caused by the autonomy of the systems involved in processing a workflow. The difficulties arise because the local systems cannot distinguish locally uncommitted tasks that belong to globally committed workflows from uncommitted local tasks. If a local system provides basic transaction management mechanisms, after a failure its local recovery procedures rollback all locally uncommitted tasks, even if they belong to globally committed workflows. In addition, workflows which have a locally committed task cannot be rolled back. Workflow-recovery actions at each local system constitute new transactions, which from the point of view of the local systems have no connection to the failed tasks they are supposed to complete.

These issues have been discussed in the context of multidatabase transactions and some solutions proposed in the literature [53, 21, 38] can be expanded to workflow management.

Relaxing Global Serializability Requirements. In the discussion below, we assume that the specifications of workflows provided by the designer and the initial parameters are correct, that is, a workflow executed in isolation does not violate consistency. The workflow scheduler guarantees that the execution of every workflow proceeds in accordance with its specification and local concurrency controllers guarantee that interleaved execution of tasks preserves local consistency constraints. The workflow concurrency control mechanism has to guarantee that global histories are correct, that is, preserve *multi-system consistency constraints*. Below we define a class of global histories that are considered correct under the workflow model described above.

Tasks of various workflows issue operations that may locally conflict with operations of other workflows or local transactions. The execution order of committed conflicting operations of local transactions or tasks (in general transactions) results in the serialization precedence between transactions that have issued them (denoted by $\prec$). The serialization precedence relation between transactions is transitive and can be used to define *M-serializability*, a correctness criterion for concurrent execution of workflows.

Informally, a global history is M-serializable if every local history is serializable and no two workflows W_i and W_j have tasks T_{ik}, T_{il} belonging to the same execution-atomic unit of W_i and tasks T_{jk}, and T_{jl} belonging to the same execution-atomic unit of W_j, such that $T_{ik} \prec T_{jk}$ and $T_{jl} \prec T_{il}$.

Global serializability requires that the serialization order of workflows be *compatible* at all processing entities. That is, if a task T_i of W_i precedes task T_j of W_j at a processing entity then at no other entities can a task of W_j precede a task of W_i. M-serializability requires only that tasks belonging to the same execution-atomic unit of a workflow have compatible serialization orders at all local sites they access. M-serializability allows some global histories that would be rejected under global serializability. This is because operations of a task T_i are not related to operations of tasks that belong to execution-atomic units other than that of T_i. Therefore, the serialization precedence of T_i imposes a serialization precedence only on tasks that belong to the same execution-atomic unit as T_i and not on all tasks of W_i, as in the case of global serializability.

As an example, consider the history in Figure 3. Let us assume that workflow W_1 consists of two execution atomic units $\{T_{11}, T_{12}, T_{14}\}$ and $\{T_{13}\}$, while the execution atomic units of W_2 are $\{T_{22}, T_{24}\}$ and $\{T_{23}\}$. Then, the serialization precedence between T_{12} and T_{22} established at processing entity 2 has no effect on the serialization precedence at processing entity 3. On the other hand, if the tasks T_{14} and T_{24} were to conflict, T_{24} would have to be serialized after T_{14}. A formal definition of M-serializability and a mechanism to ensure it are presented in [43].

4.3 Recovery of a Workflow

The objective of failure recovery in workflow management is to enforce the failure atomicity of workflows. The recovery procedures must make sure that if a failure occurs in any of the workflow-processing components (including the scheduler), the workflow eventually reaches an acceptable state – possibly using compensation. We may assume that the processing entities involved in the workflow have their own local recovery systems and handle their local failures. Therefore, we will discuss here only the handling of failures of the workflow execution controllers (scheduler(s) and the concurrency controller).

In order to recover the execution-environment context, the failure-recovery routines need to restore the state information at the time of failure, including the information about the execution states of each task and information about the scheduling dependencies of the concurrency controller. Therefore the appropriate status information must be logged on stable storage [28].

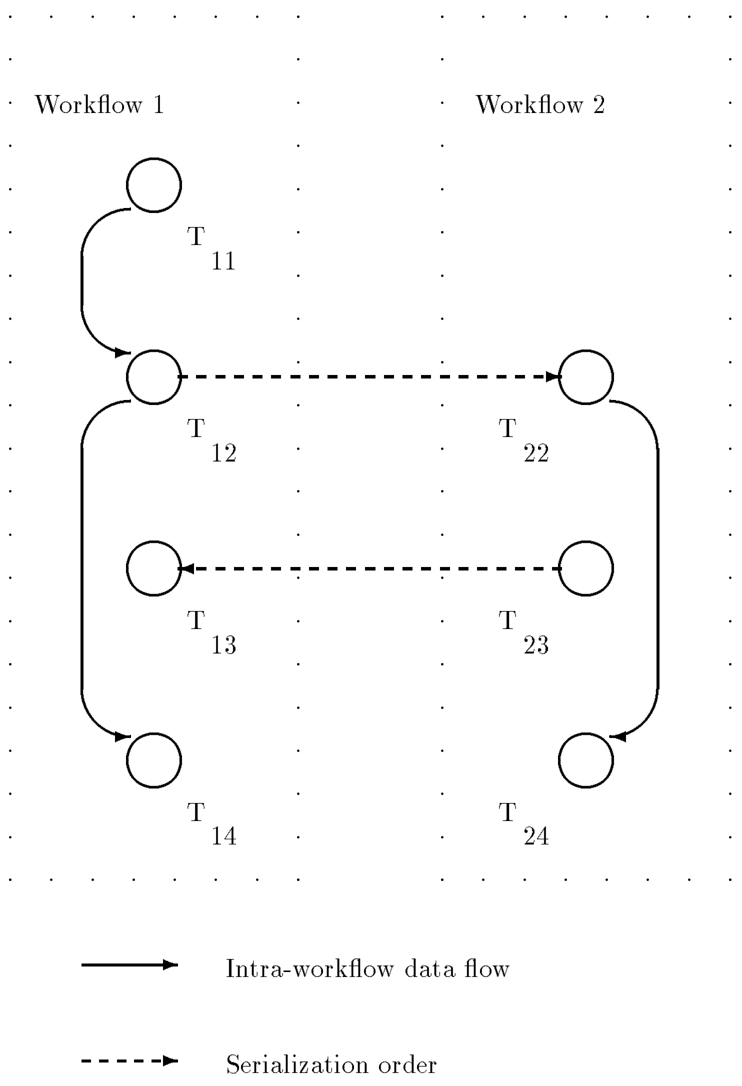

Figure 3: Example of M-serializable Execution of Workflows

The failure of the concurrency controller may be detected using timeout mechanisms, and the status information can be reconstructed by scanning the logs. Assuming forward recoverability [41] and idempotency of the processing entities, the execution of the workflow may continue by resubmitting the tasks that were not completed.

We also need to consider the contents of the request queues. If a mechanism providing the functionality of persistent pipes is used (e.g., in DEC VMS system, the queues may be implemented using mailboxes – persistent message queues accessed as virtual I/O devices), the messages are stored in permanent storage and are not lost in case of failure. If the queues are in volatile storage (e.g. under UNIX the queues may be implemented using sockets), instead of recovering the contents of the queue, we let schedulers re-send their requests.

5 Support for Workflow Execution

Three basic approaches to the implementation of a workflow-management system are: (a) an embedded approach that requires and exploits significant support from the processing entities for specification and enforcement dependencies, for example, an approach requiring processing entities to support some active data-management features, (b) a layering approach that implements workflow-control facilities on top of uniform application-level interfaces to entities by developing modules to support inter-task dependencies and other workflow specifications, and (c) an approach that provides limited workflow-management facilities in an environment consisting of transaction monitors and event monitoring/synchronizing facilities (e.g., similar to those proposed in CORBA [47]).

The first approach is aligned to the work reported in [12]. Variants of the second approach are used in the Narada [27], Interbase [15], and Carnot [49] projects. The facilities provided by these projects may be used to implement relaxed transaction and workflow capabilities. GTE's Distributed Object Management (DOM) project [22] seems to follow a combination of the first two approaches by distinguishing between native DOM objects and data managed by non-DOM systems that support transactions (i.e., DBMSs), and do not support transactions ("transactionless systems").

Work on execution of multi-system applications in a heterogeneous computing environment has been carried out in the Omnibase project since 1985 [26]. The basic contributions of this project to the workflow management were the development of a task specification language DOL and a distributed execution environment called Narada. The software architecture based on the concept of a task-specification language was later adapted and expanded in numerous projects including ERNIE at Bellcore, Interbase at Purdue [44], and Carnot at MCC [52].

A DOL-based system consists of the *resource directory*, the *DOL engine* and a collection of *Local Access Managers* (LAMs). The resource directory contains up-to-date information about all the services known to the DOL engine. The information includes physical addresses, communication protocols, login information and data-transfer methods. The DOL engine is responsible for executing DOL programs by communicating with various services. It provides task activation, task synchronization, conditional execution, and data exchange between LAMs, as well as commitment control for local and global tasks. It is also responsible for opening and closing connections to the participating LAMs, setting up communication channels between them, sending them their local commands and monitoring their execution status. A LAM acts as a proxy user for the processing entity it manages, and provides an abstraction of the service it encompasses. LAMs preserve the autonomy of the processing entities. The DOL engine communicates with LAMs, which are responsible for executing the requested local commands.

DOL programs can be written directly by end-users or can be automatically generated by another software system. The DOL engine receives a DOL program and produces an execution plan for it. It consults the resource directory to get relevant information about invocation of the corresponding LAMs. The DOL engine executes the DOL program by invoking the LAMs that support the corresponding services. These services can be database systems, knowledge based systems, software packages or other DOL engines. When each LAM executes its commands successfully, the output may be forwarded to another LAM. The DOL engine communicates with all LAMs using the same high-level communication protocol. This simplifies the architecture and allows addition of new services without modifying the other system modules.

The Interbase project extended the concept of LAM in DOL to what is called Remote System Interface (RSI). RSIs were developed for a variety of (heterogeneous) software systems to provide, in addition to the LAM functionalities, some of the novel features of IPL, such as more sophisticated format-translation and information-exchange facilities and coordination of concurrent workflows (termed "global applications" in [15]).

Carnot goes further in developing support for distributed services and applications, including relaxed transactions and workflows [52, 49]. Its main component, called an Extensible Services Switch (ESS), uses a language called Rosette based on an Actor model. Rosette supports interpretive control of distributed applications, high concurrency, and ultra-lightweight processes. Since control programs or scripts are interpreted, one site may send a script to another site to affect the local control of a distributed application. Furthermore, an ESS supports many distributed processing abstractions, such as ByteStreams and TreeSpaces, for easier development and efficient implementation of workflow and relaxed transaction management. ESSs that communicate with different communication resources (e.g., TCP/IP, SNA, X.25), information resources (e.g commercial relational and object-oriented DBMSs), and application services (e.g., X.500, X.400) have been built.

6 Summary

Hierarchical structures of extended transaction models such as nested transactions [39] and multi-level transactions [50] are often too rigid for workflow applications. A basic problem with the development of workflow management systems based on a particular transaction model is that a predefined set of properties provided by the model may or may not be required by the semantics of a workflow. For example, intertask dependencies required by the semantics of a workflow may be more complex than those supported by a given transaction model or correctness guarantees (such as global serializability) that are provided may be stronger than it is needed.

Another problem with adopting the extended transaction models for designing and implementing workflows is that the systems involved in the processing of a workflow (processing entities) may not provide support for facilities implied by a transaction model. For example, multidatabase transaction models are geared towards processing entities that are DBMSs, hence they are not applicable if the entities do not provide local transaction management facilities. While we may need to define workflows that also incorporate non-DBMS processing entities, it is clear that such workflow systems may not be able to provide global data consistency guaranteed by transactions.

The desire to overcome these limitations was a motivation for the development of workflow models. In our opinion, a comprehensive transactional workflow system should support multitask, multisystem activities where: (a) different tasks may have different execution behavior and properties, (b) the tasks may be executed on different procesing entities, (c) application or user defined coordination of the execution of different tasks, including data exchange is provided, and (d) application or user defined failure- and execution-atomicity are supported.

A basic type of transactional workflow system uses transactions as the tasks and exploits support for transaction execution provided by systems (which is similar to extending a relaxed transaction model with intertask dependencies). The workflows may additionally require concurrency control to support execution atomicity and coordination requirements with respect to concurrent execution of multiple workflows. When the tasks are transactions and entities that execute them are DBMSs, it is possible and desirable to borrow from the extensive recent work on extended and relaxed transaction models. Further work is needed when the tasks are not transactions and/or the entities do not provide transaction execution support, but workflow failure and execution atomicity requirements exist.

While it may be possible to express some workflow correctness criteria as intertask dependencies, it may be desirable to incorporate them into a workflow model. In the future, we expect an evolution towards application development models that provide the extended transaction and workflow capabilities to suit the needs of complex applications accessing heterogeneous existing systems.

Acknowledgements

We acknowledge our collaboration with the members of the Carnot Project at MCC (P. Attie, P. Cannata, M. Huhns, M. Singh, C. Tomlinson, D. Woelk) and discussions with N. Krishnakumar, L. Ness at Bellcore, G. Weikum, H-J. Schek at ETH-Zurich, and G. Karabatis, M. Bregolin at the University of Houston.

References

[1] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceedings of the 18th International Conference on Very Large Data Bases*, August 1992.

[2] M. Ansari, M. Rusinkiewicz, L. Ness, and A. Sheth. Executing Multidatabase Transactions. In *Proceedings of the 25th International Conference on System Sciences*, Hawaii, January 1992.

[3] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th VLDB Conference*, 1993.

[4] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17, September 1991.

[5] Y. Breitbart and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1988.

[6] E. Cameron, L. Ness, and A. Sheth. An Executor for Multidatabase Transactions which Achieves Maximal Parallelism. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, April, 1991.

[7] S. Chakravarthy, S. Navathe, K. Karlapalem, and A. Tanaka. Meeting the Cooperative Problem Solving Challenge: A Database-Centered Approach. In *Cooperating Knowledge Based Systems 1990*, Ed. S.M.Deen, Springer-Verlag, 1990.

[8] Jiansan Chen, Omran A. Bukhres, and Ahmed K. Elmagarmid. IPL: A Multidatabase Transaction Specification Language. In *Proc. of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA, May 1993.

[9] P. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.

[10] P. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Model. In *Proceedings of 17th International Conference on VLDB*, 1991.

[11] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.

[12] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proceedings of the 17th VLDB Conference*, September 1991.

[13] E. Dyson. Workflow. In *Forbes*, November 1992, p. 192.

[14] A. Elmagarmid, editor. *Transaction Models for Advanced Database Applications.* Morgan-Kaufmann, February 1992.

[15] A. Elmagarmid, J. Chen, and O. Bukhres. Remote System Interfaces : An Approach to Overcome Heterogeneous Barriers and Retain Local Autonomy in the Integration of Heterogeneous Systems. *the International Journal on Intelligent and Cooperative Information Systems*, 1993. (to appear).

[16] A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[17] A. Farrag and M. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[18] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating Multi-transaction Activities. Technical Report CS-TR-247-90, Princeton University, February 1990.

[19] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[20] H. Garcia-Molina, K. Salem, D. Gawlick, J. Klein, and K. Kleissner. Modeling Long-Running Activities as Nested Sagas. *Data Engineering Bulletin*, 14 (1), March 1991.

[21] D. Georgakopoulos. Multidatabase Recoverability and Recovery. In *Proceedings of First International Workshop on Interoperability in Multidatabase Systems*, April 1991.

[22] D. Georgakopoulos, M. Hornick, and P. Krychniak. An Environment for Specification and Management of Extended Transactions in DOMS. Technical Report September, GTE Laboratories Inc., 1992.

[23] D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin. Chronological Scheduling of Transactions with Temporal Dependencies. Technical Report UH-CS-91-03, Dept. of Computer Science, University of Houston, February 1991.

[24] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. Using Ticket-based Methods to Enforce the Serializability of Multidatabase Transactions. *to appear in IEEE Transactions on Knowledge and Data Engineering, 1993.*

[25] J.N. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on VLDB*, September 1981.

[26] M. Rusinkiewicz et al. OMNIBASE: Design and Implementation of a Multidatabase System, *IEEE CS Distributed Processing Newsletter*, Volume 10, Number 2, 1988

[27] Y. Halabi, M. Ansari, R. Batra, W. Jin, G. Karabatis, P. Krychniak, M. Rusinkiewicz, and L. Suardi. Narada: An Environment for Specification and Execution of Multi-System Applications. In *Proceedings of the Second International Conference on Systems Integration*, 1992.

[28] W. Jin, L. Ness, M. Rusinkiewicz, A. Sheth. Concurrency Control and Recovery of Multidatabase Work Flows in Telecommunication Applications. In *Proceedings of the SIGMOD Conference*, May 1993.

[29] W. Jin, N. Krishnakumar, L. Ness, M. Rusinkiewicz, and A. Sheth. Supporting Telecommunications Applications with Multidatabase Transactions. Bellcore Technical Memorandum, submitted for publication, September 1993.

[30] J. Klein. Advanced Rule Driven Transaction Management. In *IEEE COMPCON*, 1991.

[31] E. Kuehn, F. Puntigam, and A. Elmagarmid. Transaction Specification in Multidatabase Systems Based on Parallel Logic Programming. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, April, 1991.

[32] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[33] P. Krychniak, M. Rusinkiewicz, A. Sheth, and G. Thomas. Bounding the Effects of Compensation under Relaxed Multi-level Serializability. Technical Report UH-CS-92-06, Dept. of Computer Science, University of Houston, March 1992.

[34] L. Kalinichenko. A Declarative Framework for Capturing Dynamic Behavior in Heterogeneous Interoperable Information Resource Environment. In *Proceedings of the 3rd RIDE Intl. Workshop on Interoperability in Multidatabase Systems (IMS'93)*, April 1993.

[35] K. Lee, W. Mansfield, and A. Sheth. A Framework for Controlling Cooperative Agents. In *IEEE Computer*, July 1993.

[36] W. Litwin. MSQL: A Multidatabase Language. *Information Sciences*, 1990.

[37] N. Lynch. Multi-level Atomicity - a New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4), December 1983.

[38] S. Mehrotra, R. Rastogi, Y. Breitbart, H. Korth, and A. Silberschatz. Ensuring Transaction Atomicity in Multidatabase Systems. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.

[39] J.E.B Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT Press, Cambridge, MA, 1985.

[40] C. Pu. Superdatabases for Composition of Heterogeneous Databases. In *IEEE Proceedings of the 4th International Conference on VLDB*, 1988.

[41] A. Reuter. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. In *Proceedings of the 3rd Intl. Workshop on High Performance Transaction Systems,* September 1989.

[42] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin. Extending the Transaction Model to Capture more Meaning. *SIGMOD Record*, 19, 1990.

[43] M. Rusinkiewicz, A. Cichocki and P. Krychniak. Towards a Model for Multidatabase Transactions. *International Journal of Intelligent and Cooperative Information Systems*, Vol 1, No. 3, 1992

[44] M. Rusinkiewicz, S. Osterman, A. Elmagarmid, and K. Loa. The Distributed Operational Language for Specifying Multisystem Applications. In *Proceedings of the First International Conference on Systems Integration*, 1990.

[45] A. Sheth and L. Kalinichenko. Information Modeling in Multidatabase Systems: Beyond Data Modeling. In *Proceedings of the 1st International Conference on Information and Knowledge Management*, November 1992.

[46] A. Sheth, M. Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. In [14].

[47] R. Soley. A Common Architecture for Integrating Distributed Applications. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, January 1993.

[48] L. Suardi, M. Rusinkiewicz, and W. Litwin. Execution of extended multidatabase SQL. in *Proceedings on 9th International Conference on Data Engineering*, 1993.

[49] C. Tomlinson, P. Attie, P. Cannata, G. Meridith, A. Sheth, M. Singh, D. Woelk. Workflow Support in Carnot. In *Data Engineering Bulletin*, June 1993.

[50] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM TODS*, 16(1), March 1991.

[51] G. Weikum and H.-J.Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In [14].

[52] D. Woelk, P. Attie, P. Cannata, G. Meridith, A. Sheth, M. Singh, and C. Tomlinson. Task Scheduling using Intertask Dependencies in Carnot. In *Proceedings of the SIGMOD Conference*, May 1993.

[53] A. Wolski and J. Veijalainen. 2PC Agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. In *Proceedings of PARBASE-90 Conference*, February 1990.