Distributed and Parallel Databases, 3, 155–186 (1995)

Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations

NARAYANAN KRISHNAKUMAR nkk@bellcore.com
Information Sciences and Technology Research Lab, Bellcore, MRE 2B324, 445 South Street, Morristown, NJ 07960

AMIT SHETH amit@cs.uga.edu
Large Scale Distributed Information Systems Lab, Dept. of Computer Science, University of Georgia, 415 Graduate Studies Research Center, Athens, GA 30602-7404

Recommended by: Omran Bukhres and e. Kühn

Abstract. The computing environment in most medium-sized and large enterprises involves old main-frame based (legacy) applications and systems as well as new workstation-based distributed computing systems. The objective of the METEOR project is to support multi-system workflow applications that automate enterprise operations. This paper deals with the modeling and specification of workflows in such applications. Tasks in our heterogeneous environment can be submitted through different types of interfaces on different processing entities. We first present a computational model for workflows that captures the behavior of both transactional and non-transactional tasks of different types. We then develop two languages for specifying a workflow at different levels of abstraction: the Workflow Specification Language (WFSL) is a declarative rule-based language used to express the application-level interactions between multiple tasks, while the Task Specification Language (TSL) focuses on the issues related to individual tasks. These languages are designed to address the important issues of inter-task dependencies, data formatting, data exchange, error handling, and recovery. The paper also presents an architecture for the workflow management system that supports the model and the languages.

Keywords: Workflow, Automation, Databases, Heterogeneity, Transactions, Tasks

1. Introduction

The need to improve productivity and cut costs has resulted in the need to significantly reengineer and automate enterprise operations or activities. The objective of the METEOR (Managing End-To-End OpeRations) project[1] is to support and enable flexible automated solutions for enterprise-wide operations (workflows). In this paper, we present a model and the languages for specifying multi-system workflows in METEOR, as well as a discussion of execution support.

A workflow in an enterprise typically involves performing multiple, related tasks, which can be heterogeneous and performed on or by heterogeneous processing entities. The size and complexity of several existing processing entities and the fact that they maintain a real-time inventory implies that they cannot be easily migrated over or modified to enable a more manageable and homogeneous environment. Thus, in METEOR, our approach to automating workflows is necessarily a bottom-up one: we wish (and need) to support the current execution environments as well as evolving ones. Other approaches in the literature to workflow automation have taken one of distributed transaction processing, office

(document or e-mail workflow) automation, or multidatabase transaction perspectives. We wish to support an amalgamation of all of these - both in modeling/specification and implementation. We also attempt to integrate and extend the results of several relevant research efforts in the literature (e.g., [19], [13], [10], [37], [8], [20]).

A practical workflow management system that can enable multi-system applications must deal with the specification and execution support related to:

- different types of (preexisting and new) tasks, the processing entities that execute or perform the tasks, and the interfaces through which the tasks are submitted to the processing entities,
- the coordination requirements between tasks, that are dependent on the execution states of individual tasks and the workflow as a whole as well as the data manipulated by these tasks,
- the data exchange between tasks, that might also involve dealing with different data formats for different tasks, and
- interfacing with existing software systems (e.g., script interpreter/processors, abstract data management and manipulation, data format translators, etc.) that add value to workflow processing and the associated computation.

METEOR deals with all of the above aspects and offers the following features:

- a well-defined model and the languages for specifying the workflows and the tasks,
- a compiler/interpreter for the workflow language (ancillary support may include syntax directed editors and/or graphical user interfaces that support workflow specification, and support for testing the correctness and executability of the defined workflows), and
- run-time components, such as a workflow controller that supervises the progress of the workflow, enforces intertask dependencies and interfaces with existing systems, and task managers that have the responsibility for individual tasks.

We first discuss a workflow model and the languages that support it. The language design is consistent with the need to support appropriate levels of abstractions for different classes of persons who might specify a workflow. At Bellcore and other organizations, for example, one group of persons (systems engineers) focus on the enterprise-level (end-user and application-centric) requirements that the workflows should support, while another group of persons (developers) better understand the systems implementation aspects of the workflows. This dichotomy as well as considerations for modular design and implementation has led us to have two component languages: workflow specification language (WFSL) and task specification language (TSL). It is possible to have graphical languages and tools for more user friendly, customized, and "higher level" workflow specifications (such as those offered by several existing workflow software, see [21]) that can be translated into WFSL and TSL. Furthermore, specifications in WFSL and

TSL can be compiled and/or interpreted into lower level and system specific execution code. In this sense, these languages can be said to offer intermediate-level specification.

The paper is organized as follows. Section 2 discusses in more detail the requirements for workflow support in METEOR based on our interactions with real multi-system applications (both their current implementations and their future planned versions). A high-level description of the operational environment is also discussed since it has influenced the model and the language. Section 3 deals with related work and contrasts the METEOR approach with that of others. In general, METEOR attempts to use several ideas. Section 4 discusses the approach and rationale in the language design. Section 5 and Section 6 discuss many aspects of WFSL and TSL, respectively. Section 7 reviews the architecture of the system, and also briefly deals with the issue of correctness in workflows. Section 8 gives our conclusions.

2. The Environment and Application Requirements

A typical large business operation has several thousand applications. For instance in a telecommunications company, around two hundred of these are large application systems, called Operation Support Systems (OSSs), each with several hundreds of thousands of lines of code and a very large database (e.g., see [1], [26], [27], [21] for representative applications and further discussion about this environment). Historically, end users performed operations on these OSSs using screens. To provide easier access, application programs and scripts have been written to perform operations using terminal emulation or through "contracts" that provide well-defined logical interfaces. Many new applications also access DBMSs directly for the purpose of auxiliary inventory management, monitoring of workflow or task status, statistics maintenance, or error processing. Applications could also involve humans processing some tasks (currently this is often needed for error resolution when OSSs issue Requests for Manual Assists- RMAs). To adequately model such real-world environments, we classify the relevant components of our applications into tasks, processing entities and their physical interfaces, and then discuss how they can be tied together into a workflow.

2.1. Tasks, Interfaces, and Processing Entities

Tasks are operations or a sequence of operations that are submitted for execution at the processing entities using their interfaces. The types of tasks that we currently consider include *user tasks* that involve humans in processing the tasks, and other *application tasks* such as scripts or application programs that may involve terminal emulations to remote systems, client programs or servers invoking application servers, database transactions, and contracts. For this paper, *contract* tasks can be viewed as predefined sets of operations at the OSSs that behave as stored procedures from the invoker's perspective.

The processing entities for the application tasks include application systems (OSSs, legacy and modern application systems), servers supported by client-server and/or trans-

action processing systems (e.g., atop DCE and Encina™), DBMSs (for processing transactions), and script interpreters and compilers (for processing scripts and application programs). Furthermore, with user tasks, humans are the processing entities, who in turn may use other software (office automation software such as spread sheets and document/image processing systems).

The physical interfaces include remote procedure call mechanisms such as DCE RPC to directly make calls to application servers, transactional RPCs to application servers under the control of a transaction monitor (such transaction monitors include ACMS™, Tuxedo™, and Encina), queue managers that deliver requests to application servers (proprietary or vendor product supported, e.g., Tuxedo/Q, Encina's RQS or DEC's MessageQ), workstation-to-mainframe interfaces to allow a workstation-based client program to access a mainframe-based OSS, and interfaces that support user tasks and provide access to graphical user interfaces (such as those associated with document/image processing systems).

2.2. *Requirements for Workflow Management*

We now discuss some of the key requirements of workflow management in our environment that influence the language design and the system architecture.

1. Inter-task dependencies specify how the execution of a task is related to that of others (based on the state of execution of other tasks and their data outputs) and external variables (e.g., time of day). The workflow management system should be able to evaluate and manage several kinds of dependencies between tasks efficiently.

2. Data management in multi-system workflow applications can be very demanding. This involves support for different data formats, transport and storage of this data and complex manipulation of this data by auxiliary systems. Auxiliary systems are existing application programs (including applications that can invoke sub-workflows) or scripts, that can parse and manipulate complex data.

3. Typically, in Bellcore and many large companies, one set of people define the workflows at the conceptual level, describing the functionality of tasks, inputs and outputs and dependencies. The exact details of tasks, interfaces and processing entities is the concern of another set of people who write the code for the tasks. This modular approach to system engineering and design is very important, since it separates several details of an implementation (that might change depending on the implementation environment) from the system specification. The workflow management system should enable this separation and also permit code-reuse.

4. Business processes are rapidly changing. For instance, in the telecommunications world, new telecommunications services are offered very frequently. In general,

Encina is a trademark of Transarc Corp.
ACMS is a trademark of Digital Equipment Corp.
Tuxedo is a trademark of Novell Inc.

as newer systems get incorporated into the business process, there is a need to flexibly and easily modify the workflows. Given that each new workflow most likely incorporates pre-existing tasks, only the conceptual workflow specification has to be written anew. This is yet another reason to separate the workflow specification from the details of the individual tasks.

5. The workflow management system should be able to recognize and handle errors. Errors can be *logical errors* and *system errors*. Logical errors arise at the application level, e.g., if a particular item required from inventory is not in stock, a task attempting to procure this item can complete but with a logical failure. However, if the database is down, there is a system failure since the task cannot execute. The demarcation between logical and system errors is important for modularity. The task programmer and/or the workflow infrastructure handle system errors first, e.g., by retrying the *same* task (see Section 6). A workflow specification deals only with logical errors, but system errors could be elevated to the workflow level as logical errors, e.g., when a task fails repeatedly due to system errors.

6. Workflows can be dynamic, i.e., the entire workflow cannot be determined beforehand. For instance, suppose a customer requires a digital telecommunications link between three of his locations. A special routing task will have to determine possible trunk routes between these locations, and this may in turn result in new tasks that assign appropriate inventory. It is generally not known beforehand how many of these new tasks will be generated, nor what additional control or data flow dependencies they can cause. The workflow management system should allow such incremental changes in the workflow specification at run-time.

7. We assume that a workflow controller (centralized or distributed) co-ordinates tasks according to the intertask dependencies and other constraints, and maintains the state of the workflow at all times. Suppose the controller fails. It is desirable that *forward recovery* takes place, i.e., when the controller recovers, it resumes the workflow from where it left off. Some incomplete tasks may have to be restarted, but not the entire workflow. This requirement implies that the state of the workflow be stored persistently.

3. Current Approaches

In [14], the authors have described features that a workflow (called operation flow in [14]) model and specification language should support. The METEOR system closely follows this view and supports (with the terms used by [14] in parentheses) features such as: the definition of individual tasks (basic operation definition), a variety of tasks including user tasks (transactions and nonelectronic operations), state-based and value-based intertask dependencies and data management (control and data flow definition), and failure and exception handling (same terms). Currently, no separate or direct support for business rules and constraints, and security and role resolution identified in [14] is provided in

METEOR. These can partly or indirectly be supported by intertask dependencies and by individual interfaces (e.g., security features of DCE).

Approaches in the literature dealing with workflows/activities [15] can be described as those based on multidatabase and relaxed/extended transactions ([19], [1], [37], [18]), active database and rule-based approaches ([12], [13]), combinations of the above two [20], and office and process-automation ([24], [30], [31]). The ACTA model [10] and the DOM model [20] provide frameworks for system specification that capture several of the above approaches. For instance, these models support the specification of complex intra- and inter-transaction state dependencies, and correctness dependencies such as serialization, visibility, co-operation and temporal dependencies. Such models can be used to not only specify workflows but also to check specification correctness. In METEOR, we adopt a subset of this specification framework, partly for simplicity and partly for the reason that the workflows we have studied (primarily in telecommunications) are characterized more by their dynamic nature and need for complex data manipulation, rather than intricate control flow specification based on serializability requirements.

In [19], a language is proposed for describing (possibly nested) multi-transaction activities, and for specifying the flow of data between different *modules*. However, the specification of intertask control dependencies is limited, and it is assumed that all the leaf-level tasks are transactional. The ATM approach ([12], [13]) includes an extended nested transaction model and language for describing long running activities. Such a description includes a procedural static specification of the high-level workflow, and rules (triggers) for the "dynamic" evolution of the workflow. However, the kind of dynamic workflows that we wish to support, where new tasks/inter-task dependencies can be added dynamically, is not discussed. In the ATM model, this would have to be supported by permitting the addition and deletion of rules at run-time. The ATM model also assumes leaf-level transactions, and organizes the workflow activity as an extended nested transaction with deferred and decoupled nested transaction/activities. Furthermore, an activity can be queried as to its status (whether active, committed, aborted or compensated) or that of its subactivities/transactions. The ATM model does support a range of heterogeneous tasks with differing execution state diagrams, however, does not specify how new tasks with new execution structures can be included in the model.

In the ConTract model[37], a long running activity is modeled as a combination of *scripts* and *steps*, where scripts deal with the conceptual workflow and steps deal with individual tasks. The ConTracts model is comprehensive in its treatment of consistency of data, recovery, synchronization and co-operation. Data is passed between steps through contexts, and forward recovery can be handled through making the inputs and outputs of steps persistent. APRICOTS is a prototype implementation of the ConTract model [34] (some of our prototype implementation ideas have been borrowed from APRICOTS). The ConTract model, however, does not allow for dynamic workflows where all possible paths of the workflow do not have to be specified beforehand. Furthermore, there is no support in the ConTract model for the flexible integration of heterogeneous task structures. We believe that the distinction between interface versus processing entity is important in using heterogeneous task structures also (as discussed in the next section).

Another of the early transaction models, the Flex model [18], relaxes the atomicity and isolation properties of transactions. A variety of dependencies are possible between subtransactions including partial orders on execution precedence (internal dependencies) and temporal predicates (external dependencies). Furthermore, a Flex transaction can complete successfully even if some subtransactions fail (defined as acceptable final states). The InterBase project at Purdue ([7]) is based on the Flex model and facilitates the execution of distributed programs with Flex transaction semantics on heterogeneous systems. The InterBase Parallel Language (IPL) ([8]) shares some of our objectives and can deal with different data types, preference descriptions and data dependencies. It also allows the limited dynamic execution of workflows through conditional dependencies. The Transaction Management and Specification Environment(TSME) of the DOM project [20] uses an ACTA-like specification language for users to express properties of extended transactions. The TSME allows the specification of several transactional properties such as serializability, visibility and delegation, but there is no indication of high-level language support for nesting of tasks and the dynamic evolution of workflows. The ASSET system[4] is an interesting tool-kit approach to the management of extended transactions based on the ACTA framework. Language primitives are provided for initiating and committing or aborting transactions, and also for extended functionality such as delegation of resources and formation of dependencies between transactions. ASSET concentrates only on tasks that have behavior corresponding to a transaction or an extended transaction. Additional primitives will have to be added to permit different kinds of task structures, or user-defined nesting of tasks.

The METEOR approach is one of integrating the various techniques above. Our workflow model is unique in that it allows the execution structure of heterogeneous tasks to be specified (as in [2], [35], [33]) and incorporated flexibly into the workflow. Thus several types of interfaces and tasks can be supported easily. By using rules similar to ECA rules [13] and the notion of user-definable compound tasks that are composed of simpler tasks or other compound tasks, we allow workflows to be nested. We also provide a flexible explicit data transfer specification using ideas from [19]. The notion of storing inputs and outputs persistently for the purpose of forward recovery is adapted from ConTracts. A unique feature of METEOR is the support for dynamic workflows, where at run-time several tasks and dependencies can be created anew. The combination of the above features provides for a powerful workflow management system. Note however that METEOR in its current form does have some limitations. We restrict ourselves to simple control dependencies unlike more complex ones ([10], [28], [2], [23], [20]), but allow a more comprehensive set of data dependencies. We do not associate the notions of deferred and decoupled transactions with rules as in the ECA approach; we provide such functionality by using explicit rules. Furthermore, the ConTract model provides synchronization invariants for concurrency control. METEOR does not yet provide this flexibility (see Section 7 for concurrency control issues).

There are many proposals for multi-level and nested transaction models (e.g., [38]) that are relevant to workflow specification to various degrees. A perspective on combining workflow and transaction management was given in [6]. [5] addresses high-level language specification for a restricted class of compensatable transactions. Unlike many of

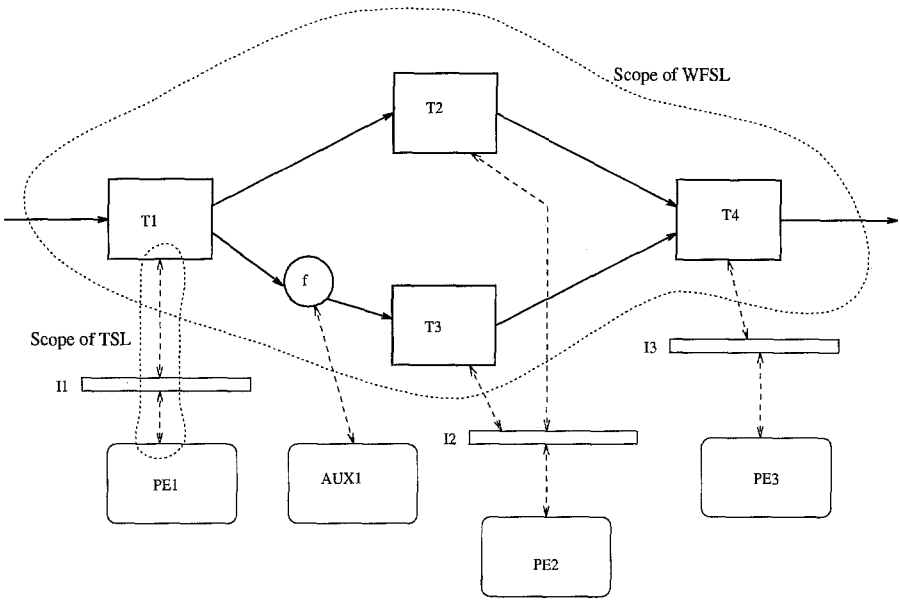


Figure 1. Schematic of a workflow

these models, we do not view the entire application or the activities in only transactional terms, since several tasks can be non-transactional. Other specification and language efforts are the script-based workflow specification in Carnot [36], and the extensions to MSQL to support some features of workflows [33], and the STDL language [3]. The ideas behind TSL have been borrowed from several predecessors with which it shares its objectives and functionality. These include, among others, DOL [32], and the interfaces that support executions of heterogeneous tasks against different processing entities (e.g., LAM in Narada [25], RSI in Interbase [17], and ESS in Carnot [36]). There have also been other approaches for multimedia document flow management [24] that are non-transactional in nature.

4. METEOR's Integrated Approach

In this section, we discuss the model, and the basic design decisions behind two languages for specifying multi-system workflow applications. Tasks in a workflow can have different functionality, and can further be differentiated on the basis of the interface and/or the processing entity they are executed on. The METEOR workflow model in Section 4.1 provides the basis for separating the conceptual workflow specification from the details of individual tasks and is an adaptation of the model outlined in [35] and [33]. WFSL is based on the features of the workflow model. TSL supports the detailed specification of each task and its interaction with the interfaces/processing entities in a

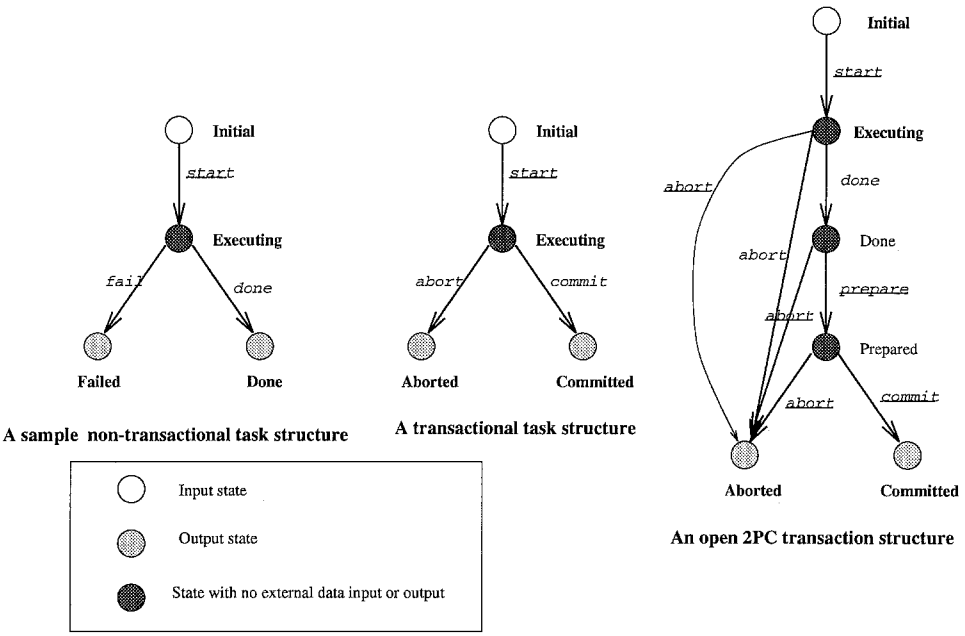
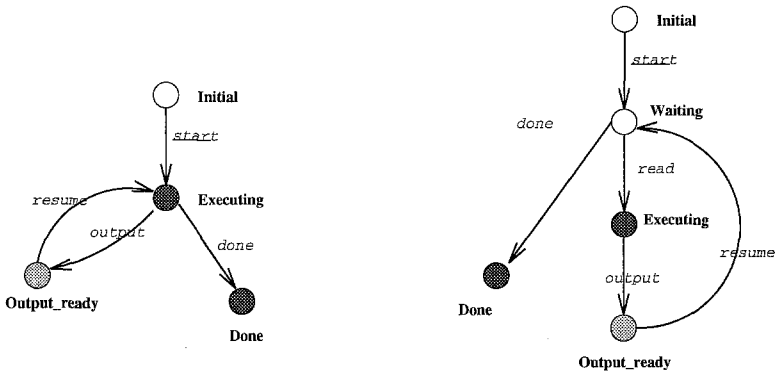


Figure 2. Some Task Structures

distributed environment (see Figure 1). At an architectural level, the WFSL specification would be executed by a workflow controller, whereas each task specification would be executed by a task manager (see Section 7).

4.1. The Workflow Model

A workflow comprises multiple tasks. The workflow specification might not be concerned with the details of the tasks, however, it would have to at least deal with the (externally visible) starting and completion events of tasks, the inputs and outputs of tasks and how they relate to other tasks. In general, we represent the execution behavior of each task using task structures. Three frequently occurring task structures in our environment are shown in Figure 2. Each task structure has an initial state, and on the `start` transition moves into the `executing` state. There could be one or more transitions after this, and in the three structures, the task eventually enters a *terminating state* (such as `done`, `failed`, `committed`, `aborted`). We distinguish between *controllable* transitions that can be enabled by the workflow controller, and *non-controllable* transitions that are enabled by the processing entity (this aspect of specification is especially important for scheduling, see [2] for more details). For instance, the `done` transition from the `executing` state to the `done` state is enabled only by the processing entity, whereas the `start` transition is enabled by the workflow controller. All the transitions that are *controllable* in these tasks



Note : Both tasks above have a Failed state that is not shown.
All states except the Initial and Done state have a fail transition to this state.

Figure 3. Task Structures for Special Non-transactional Tasks

are underlined. Task structures also indicate which states can possibly have data inputs or data outputs or both. In Figure 2, the initial state can receive data inputs, and outputs can be produced in the `done`, `failed`, `committed` or `aborted` states.

In our environment, a user task or script is characterized by the non-transactional task structure shown in Figure 2. The task structure of a contract (stored procedure) is typically a transactional task (with ACID properties), which has an aborted or committed final state. The third type of task structure shown is that of a transaction supported by DBMSs that provide an open two-phase commit (2PC) feature [9]. Notice the transitions from `prepared` to `aborted` or from `prepared` to `committed` are controllable. Furthermore, there are two kinds of abort transitions from the `executing` state: one that is controllable (used when the workflow controller decides to abort the transaction), and another that is not controllable and initiated by the processing entity.

There could be other task structures that model non-transactional computation. For instance, consider two tasks, one (a *producer* task) that produces a stream of output and the other (a *consumer* task) reads and processes that stream of output and produces another output stream (the output of one task is "pipe"d to the other). The execution structures for these tasks is shown in Figure 3. Note that inputs and outputs are not necessarily associated only with the start and termination of tasks: the consumer task can read input in the `waiting` state and produce output in the `output_ready` state, while the producer task produces its output in the `output_ready` state.

The generic task structures in Figure 2 account only for possible interface or processing entity system errors elevated to logical errors as transitions to the `failed` or `aborted` states. A task in the `done` or `committed` state reflects normal execution of a task from the system (as opposed to the application) perspective. The task can be further deemed successful or unsuccessful, based on application-specific criteria which can be included in

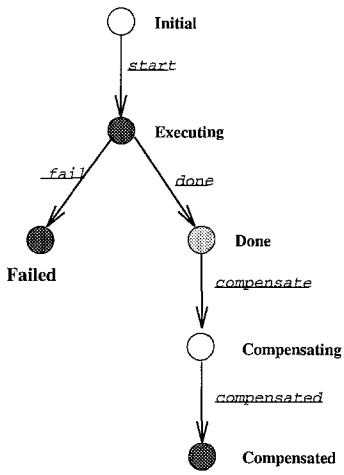


Figure 4. Compound Non-transactional Task with Compensation

the workflow specification. Another error that is possible from the view of the workflow controller is as follows. The workflow controller might enable a controllable transition, but the transition fails due to some internal error, e.g., a task is unable to start due to a lack of enough swap space. It is assumed that for every such failure of an enabled controllable transition, the task makes a transition to a corresponding `_err` state, e.g. `start_err`. (Since handling the errors of each transition might make the specification very tedious, another approach that has been explored in a Bellcore prototype [11] is to specify a *default* state such as `failed` or `aborted` that is reached when such an error happens.)

Observe that a task structure does not determine the means of execution nor the functionality of the task, but only a high-level description of the (visible) state transitions. Two tasks might have the same structure, but have different functionality or can be executed atop different interfaces and/or processing entities. Usually, the task structure is determined by the states that are observable while the task executes at the processing entity. Sometimes it is possible that the interface does not support the same view: suppose a database transaction is submitted through a persistent queue. The queue might only offer a non-transactional view of the task, i.e., the task has been submitted, and the task has either failed or has been done (and not that the task committed or aborted). This issue is discussed further in Section 6.

The tasks described thus far are *simple tasks* in our model, i.e., a simple task is a physical unit of work that executes at a processing entity. Our model also includes *compound tasks*, which can be composed of simple tasks and other compound tasks. Compound tasks are logical units of work that are not executed against processing entities, but meant to specify co-ordination and data flow requirements between sub-tasks (see Section 5.4 for examples). All transitions of a compound task are controllable since the workflow specification of the task determines all the transitions. For example, consider

a non-transactional compound task, comprising several tasks, that is compensatable. The task structure for the compound task corresponds to Figure 4. The `start` and `compensate` transitions are initiated by the workflow controller due to external events, whereas the `fail`, `done` and `compensated` transitions would be initiated by the controller due to appropriate states reached by the sub-tasks. A workflow can itself be specified as a compound task in our model, thus allowing a seamless integration of different workflows by nesting them within a super-workflow. Such flexible specification of compound tasks are a unique feature of the METEOR model.

Tasks are not the only executable entities in a workflow. Often the data transferred between tasks has to be formatted appropriately using *filters*. Filters are repeatable (and side effect free) functions that need not be recoverable and need not be characterized by a task structure. Such functions can either be executed locally as part of the workflow management system or remotely from shared libraries on auxiliary systems.

Thus a workflow specification primarily contains: (a) simple tasks, filters, and compound tasks, (b) the task structure of each task in the workflow, (c) the typed inputs and outputs of each task/filter types, and how they are related to the outputs and inputs of other task types, and (d) the preconditions for each controllable transition in each task.

4.2. The METEOR Language Design and Rationale

We briefly review the design of the METEOR languages and the functionality that they support. WFSL, which is used to describe the conceptual workflow specification, has been designed based on the METEOR workflow model. WFSL is a declarative rule-based language that can define compound tasks. Note however that WFSL need not serve as the language used by the workflow designer to program in. A graphical interface could alternatively be used to generate a WFSL specification automatically.

In WFSL, the workflow designer can declare a set of task types, based on their structure and then define classes of tasks with differing structures. Each task class has a task structure type and a set of typed inputs and outputs. Inputs and outputs can be scalar or array datatypes, and can also be streams.

After defining instances of each class, the workflow designer can link up the task instances using *rules*. Each WFSL rule has two components: a *control part* and an optional *data transfer* part. The control part denotes the preconditions for a single controllable task transition. Whenever a task makes a state transition, we say that an *event* occurs. The preconditions can include references to events and/or data outputs of other tasks, as well as program variables. The data transfer part indicates which outputs of other tasks or variables (possibly passing through filters) are input to that task, i.e., for instance, the output $o1$ of task $T1$ is fed through a filter f for reformatting and then fed into the input $i1$ of task $T2$. The data transfer specification in WFSL is very flexible. For instance, the data outputs of several tasks that become available at different times can be fed into one task. We can also specify alternative inputs to tasks, i.e., it is possible that under one circumstance, the output of task $T1$ feeds into the input of a second task $T2$, but under some other circumstance, the output of a third task $T3$ feeds into the input of $T2$.

We follow the ConTracts model in the requirement of forward recovery, and the rule-based nature of WFSL is consistent with this requirement. Every next step of the workflow is determined by an evaluation of relevant rules when an event occurs, and a rule which has all its preconditions satisfied fires. Note that by this declarative approach, we have removed the need for a program counter. If all inputs, outputs and task states were made persistent, then during recovery, the workflow controller can start up with its state intact and resume evaluating rules. (In an imperative program implementing the workflow, the memory image of the program will have to be checkpointed instead. The logical checkpointing in the declarative approach is usually more efficient than the memory image checkpointing.) Furthermore, if the workflow is dynamic, the declarative nature of WFSL allows it to be incrementally interpreted during run-time.

We now give a high-level description of TSL which is used to specify simple tasks. One of the key objectives of TSL is the minimal rewriting of existing tasks. TSL provides a wrapper for code describing interaction with an interface to a processing entity and essentially comprises a set of macros that can be embedded in a host language like C or C++. The main functionality of the TSL macros is to indicate points in the task execution at which the workflow controller can be informed about the current logical state of the task (and thereby points where the state of the task can be made persistent). This functionality also allows the workflow to deal with legacy applications without changing their code. If the legacy application is batched, then the TSL program consists of (a) a call to a macro indicating that the application is about to execute, (b) a call to an interface that submits or calls the legacy application, and (c) a call to a macro when the application completes execution (see Section 6). If instead the legacy application is interactive, then the TSL code will have to include functionality to interpret the intermediate results, provide input and also convey the state to the workflow controller. The task is essentially written in the host language, embedded sub-languages supported by the processing entity, and TSL macros.

As part of TSL, we also prescribe how new task programs should be written. The interface is made explicit in the task specification, and this allows the task programmer to specify error handling for interface or processing entity (system) errors. For instance, the task submission to the interface might time out, or might need to be re-submitted for up to n retries, or a different interface might be chosen, and so on. (Note that a system error can eventually develop into a logical error seen at the workflow level. For instance, a task program can try to re-submit a task several times with repeated system errors before it gives up. The workflow specification in WFSL would then have to handle the error by possibly submitting an *alternative* task.)

In summary, WFSL deals with the enterprise or application issues. It is used to specify the workflows, including all task types and classes in a workflow, all intertask dependencies, filter calls, and application level failure recovery and error handling issues (including possible application specific compensation using alternate tasks). TSL provides a homogeneous view of simple tasks to the workflow controller, and deals with interface-specific details. TSL thereby also deals with task level failure recovery and error handling that are interface and/or processing entity specific.

5. WFSL

We describe this language by using some representative examples. We focus on how the tasks can be placed together in a workflow. For brevity, details of the languages are not discussed, but can be found in [29].

5.1. Task type and instance declarations

The following example declares a simple task type that has the structure of a transaction. (We will use the names SIMPLE_NON_TRANSACTIONAL, SIMPLE_TRANSACTIONAL and TRANSACTIONAL_OPEN2PC to refer to the task structures in Figure 2.) This task has one controllable transition from initial to executing, and before the transition occurs, input(s) can be received in the initial state. The two non-controllable transitions can produce output in the state at the end of the transition. Furthermore, recall that an additional start_err state is defined implicitly to account for the (possible) failure of the controllable start transition.

```
simpleTaskType SIMPLE_TRANSACTIONAL
{
    CONTROLLABLE            start(initial,executing) input ;
    NOT_CONTROLLABLE        abort(executing,aborted) output ;
    NOT_CONTROLLABLE        commit(executing,committed) output ;
}
```

The following statement defines task classes called Loop_Assignment and Circuit_Facility_Check that are SIMPLE_TRANSACTIONAL tasks and which have input and outputs of type FCIF. (FCIF (Flexible Computer Interface Format) is a de facto standard for data transfer between OSSs in several telecommunications environments. Messages in the FCIF format have a tree structure, where the non-leaf nodes have tags and the leaf nodes hold the data, which is assumed to be defined as in C or C++.) Furthermore, the input is received in the initial state and the output is externalized in only the committed state.

```
simpleTaskClass Loop_Assignment, Circuit_Facility_Check SIMPLE_TRANSACTIONAL
      (input@{initial} FCIF input1 , output@{committed} FCIF output1) ;
```

Several instances of the same task class can be used within a workflow, as follows, where L1 and L2 are two instances of Loop_Assignment:

```
Loop_Assignment L1, L2 ;
```

The definition of a compound task is given entirely within WFSL (unlike a simple task, whose code is written in another language and compiled/interpreted independently). An entire workflow can be represented as a compound task. In our environment, the compound task usually associated with a workflow has a non-transactional task structure. We refer to this task structure by COMPOUND_NON_TRANSACTIONAL (Figure 5). Compound task types and classes are specified like simple task types and classes. Compound tasks

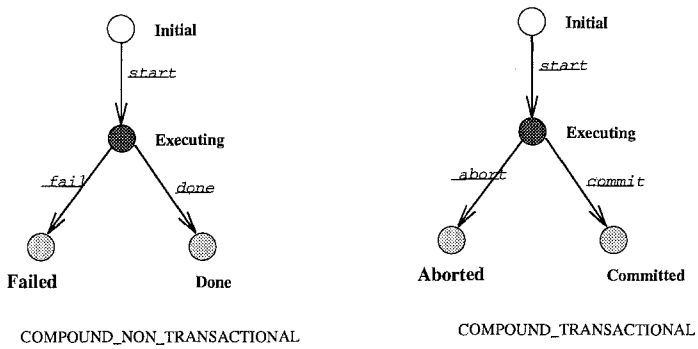


Figure 5. Some Compound Task Structures

can also have a transactional task structure provided all tasks within them are transactional with open two-phase commit. For instance, within a large workflow, there might be two simple tasks to be executed as a transactional unit. Then the two can be grouped together into a transactional compound task type COMPOUND_TRANSACTIONAL (Figure 5). We discuss such an example in Section 5.4. Specification of transactional tasks and transactional compound tasks can support the development of *transactional workflows* ([35], [33], [21]).

5.2. *Intertask Dependencies*

Intertask dependencies determine how the tasks (instances of task classes) in the workflow are coordinated for execution. Other coordination aspects, such as concurrency control among concurrent workflows, will be ignored for now and briefly discussed in Section 7. Two general types of dependencies are of interest: *state dependencies* and *value dependencies*.

A state dependency specifies how a controllable transition of a task depends on the current observable states of other tasks. A state dependency is specified as a rule consisting of < left hand side > evaluator < right hand side >. Several complex dependencies have been defined in the literature ([28], [10], [2], [20]). However, in the environments in which related tasks in a workflow execute on autonomously developed and often "closed" systems, we find that the BEGIN-dependency, SERIAL-dependency, BEGIN-ON-COMMIT-dependency and BEGIN-ON-ABORT dependency of [10] are the most relevant.

We express the dependencies such as those mentioned above using the evaluator ENABLES, such that the left hand side includes a predicate over task states and the right hand side refers to a controllable transition. For instance, the following state dependency specifies that the start transition of L2 can be enabled only after L1 has entered the done state.

```
[L1,done] ENABLES [L2, start]
```

ENABLES [2] is defined as follows : *the event(s) corresponding to the (controllable) transition(s) identified on the right hand side is enabled if and only if the event(s) leading to the state(s) identified by the left hand side have occurred.* Other evaluators can be incorporated into WFSL as the need arises (the main impact would be on the scheduling algorithms within the workflow controller: see Section 7).

The approach taken above is similar to E-C-A rules [12], where the occurrence of an event (transition) and the satisfaction of a condition leads to an action being triggered. However, ECA rules use events on the left hand side, in contrast to states as in our case. There are two small differences between the two approaches. First, the semantics of having reached a state implies additional actions of having produced the output objects (if appropriate) and logged them; approaches using transitions between the states on the left hand side may or may not imply such actions. Second, since several events could lead to the same state, one can minimize the number of rules by using the state instead of naming each event.

A value dependency may optionally be associated with a state dependency to further constrain the latter. The data that are referred to in the value dependency can be data items that are the output of some task, program variables that keep track of some data items in the workflow, constants, or the results of filter evaluations. When used in a value dependency, a filter can also be used to determine the logical success or failure of a previously terminated task (typically a `done` nontransactional task or a `committed` transactional task). For example, the above state dependency can be further constrained using a program variable `outvalL4` and by using a filter function called "success" which determines whether `L1` has logically (i.e., at the application level) completed successfully:

```
[L1,done] & (success(L1.output1) = TRUE) & (outvalL4 > 5)
                    ENABLES [L2, start]
```

The usual boolean and arithmetic operators can be used in value dependencies.

5.3. *Input and Output Assignments*

In a workflow, each task has to get data input(s) from either the output of some other task, constants, program variables or the input to the entire workflow (compound task). Thus, the workflow specification includes the association of inputs and outputs of each task to those of some other tasks. As an example, `output1` of task `L1` fed to `input1` of task `L2` is specified as:

```
L1.output1 -> L2.input1;
```

A program variable can be assigned to in a similar fashion. Program variables in a WFSL program can be global or local to a compound task. For the purpose of simplicity and to avoid problems of parallel assignment due to concurrent tasks, we assume that each program variable can be assigned to at most once during run-time and that the value of a program variable is undefined before it is assigned to. Program variables are different from the input and output variables of tasks, since input and output variables

of tasks can be assigned to several times, as in the case when a task is restarted at the workflow level (see Section 5.4 for an example). We do not yet support the capability to address different versions. Hence the WFSL program allows at most one version of a particular task instance to be executing at any time, and any reference to a input/output variable of a task indicates that of the latest version. A local program variable or an instance of a task class defined within a compound task can be referred to only within that compound task. [3]

The input and output assignments of a task are given in conjunction with the intertask dependencies, since it is crucial to determine when the outputs referred to in these assignments are made available. We assume that only the parts of control dependencies that refer to the states of tasks are evaluated before the (optional) data part of the dependency corresponding to the output of the state can be evaluated. Furthermore, the data dependency is enabled only when the control dependency evaluates to true.

The following rule indicates that L2 starts when L1 completes successfully and the value of outvalL4 is more than 5, and that the input for L2 is available when L1 transitions to done.

```
[L1,done] & (success(L1. output1) = TRUE) & (outvalL4 > 5)
    ENABLES [L2, start] % L1.output1 -> L2.input1 ;
```

5.4. *Examples*

We discuss two examples to demonstrate some of the workflow specification features. The emphasis in this section is not on the language but how the METEOR model flexibly allows the integration of transactional and non-transactional simple and compound tasks. The first example shows how the repeated occurrence of an error can be handled. The second example shows how a transactional compound task can be constructed and placed within another compound task.

The first example has three non-transactional tasks, A, B, and C (Figure 6). When A is done, C is started. If A encounters a system error, then B is started. When B is done, then A is started up again. Notice that the relationship between A and B is similar to that between a task and the corresponding failure handling task - when the task fails, the failure handling task takes over, and once the latter completes, the original task takes over. The workflow fails if C fails or B fails.

The workflow specification in Figure 7 defines the control and data flow between tasks A, B and C (the line numbers are just for reference and are not part of the syntax). Notice the use of a filter to massage the input to the workflow. Line 5 indicates that the workflow can make a transition to the done state when C reaches the done state. The workflow enters the failed state if C has failed or if B has failed (line 6), or if there was some problem during the initiation of the tasks (line 7). Notice that the following scenario can occur : A moves to failed, B is started and then completes, then A is started, and so on (the assumption is that this terminates). When we refer to a state of A on the left hand side of any of the rules (e.g., in line 4), the reference is to the currently "active" instance when the rule is evaluated.

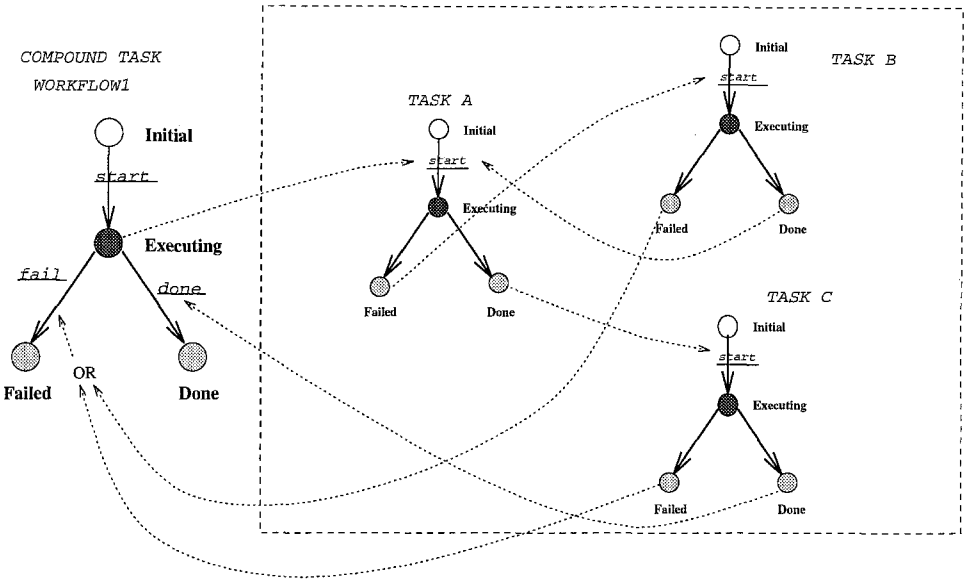


Figure 6. Workflow Example 1: Control Flow

We now review the second example (Figure 8). The workflow is indicated in lines 10 to 14 . WORKFLOW1 contains a simple non-transactional task A and a compound transaction BC (described in lines 1-9). BC is composed of two simple open-2pc transactions B and C that are done as a transactional unit. If either A fails or the transactional unit aborts, then the workflow is said to have failed. The compound task BC has rules for a (non-standard) two-phase commit between B and C. Rule 1 generates a global transaction id, that is assumed to be propagated to the resource managers through the B and C. After B and C have reached the done state, the workflow designer wishes to verify some data conditions before deciding to commit the transaction (rules 4 and 5). This shows the flexibility of WFSL. However, if all the workflow designer wants is a standard two-phase commit between B and C, we can provide syntactic sugar to let the designer specify this without using the rules. Rules 13 and 14 influence the fail transition of WORKFLOW1.

We have thereby illustrated in this section how WFSL can be used to specify workflows that contain both transactional and non-transactional units.

5.5. *Advanced Features: Dynamic aspects of workflows*

Suppose a specification of a workflow includes a set of tasks and their associated dependencies. It is possible that based on some predicates which may include states or outputs of tasks or external variables, some of those tasks are executed and some others are not. We do not consider this scenario a dynamic workflow since the specification is static

```
typedef ... FCIF ;
typedef struct
{
        int field_one ;
        char field_two;
           ...
} * SPECIAL_REC ;
const NULL = 0 ;
simpleTaskType SIMPLE_NON_TRANSACTIONAL{ ... } ;
compoundTaskType COMPOUND_NON_TRANSACTIONAL{ ... } ;
simpleTaskClass A_class SIMPLE_NON_TRANSACTIONAL
  (input@{initial} FCIF input1 , output@{done} SPECIAL_REC output1);
simpleTaskClass B_class SIMPLE_NON_TRANSACTIONAL
  (input@{initial} SPECIAL_REC input1 , output@{done} FCIF output1);
simpleTaskClass C_class SIMPLE_NON_TRANSACTIONAL
  (input@{initial} SPECIAL_REC input1 , output@{done} SPE-
CIAL_REC output1);
Filter FCIF f1(FCIF) ;
compoundTaskClass WORKFLOW1 COMPOUND_NON_TRANSACTIONAL
  (input@{initial} FCIF input1 , output{done,failed} SPECIAL_REC out-
put1 );
{
        A_class A ; B_class B ; C_class C ;
        SPECIAL_REC var1 ;
1       [WORKFLOW1,executing] ENABLES [A,start] %
                f1(WORKFLOW1.input1) -> A.input1 ;
2       [A,failed] ENABLES [B,start] %
                A.output1 -> B.input1 ;
3       [B,done] ENABLES [A,start] %
                B.output1 -> A.input1 ;
4       [A,done] ENABLES [C,start] %
                A.output1 -> C.input1 ;
5       [C,done] ENABLES [WORKFLOW1,done] %
                C.output1 -> WORKFLOW1.output1 ;
6       [C,failed]) | [B,failed] ENABLES [WORKFLOW1,fail] %
                A.output1 -> WORKFLOW1.output1 ;
7        [A,start_err] | [B,start_err] | [C,start_err]
                ENABLES [WORKFLOW1,fail] %
                NULL -> WORKFLOW1.output1 ;
}
WORKFLOW1 WF1 ;
```

Figure 7. Workflow Specification Example 1

```
typedef char[2000] str ;
constant int ERROR = 0; constant int PARTIAL_SUCCESS = 1 ;
... ;
simpleTaskClass A_class SIMPLE_NON_TRANSACTIONAL
  (input@{initial} str input1 , output@{done} str output1) ;
simpleTaskClass TID_class SIMPLE_NON_TRANSACTIONAL
  (output@{done} int output1) ;
simpleTaskClass B_class TRANSACTIONAL_OPEN2PC
  (in-
put@{initial} int i1,input@{initial} TID t1,output@{done} int output1);
simpleTaskClass C_class TRANSACTIONAL_OPEN2PC
  (input int i1, TID t1 ; output@{done} int output1) ;
Filter int f1(str) ; Filter int f2(str) ;
compoundTaskClass TRANS_BC COMPOUND_TRANSACTIONAL
   (input@{initial} str input1) ;
{  B_class B ; C_class C ; TID_class genTID ;
1  [TRANS_BC,executing] ENABLES [genTID,start] ;
2  [genTID,done] ENABLES [B,start] %
       f1(TRANS_BC.input1) -> B.i1, genTID.output1 -> B.t1 ;
3  [TRANS_BC,executing] ENABLES [C,start] %
       f2(TRANS_BC.input1) -> C.i1, genTID.output1 -> C.t1 ;
4  [B,done] & [C,done] & (B.output1 > C.output1) ENABLES
       [B,prepare] & [C,prepare] ;
5  [B,done] & [C,done] & (B.output1 <= C.output1) ENABLES
       [B,abort] & [C,abort] ;
6  [B,prepared] & [C,prepared] ENABLES [B,commit] & [C,commit] ;
7  [B,committed] & [C,committed] ENABLES [TRANS_BC,commit];
8  [B,aborted] ENABLES [C,abort] & [TRANS_BC,abort];
9  [C,aborted] ENABLES [B,abort] & [TRANS_BC,abort] ;
... }
compoundTaskClass WORKFLOW1 COMPOUND_NON_TRANSACTIONAL
   (input@{initial} str input1 , output@{failed,done} str output1,
      output@{failed} int output2 ) ;
{  A_class A ; TRANS_BC BC ;
10 [WORKFLOW1, executing] ENABLES [A,start] %
              WORKFLOW1.input1 -> A.input1;
11 [A,done] & (success(A.output1) = TRUE) ENABLES
       [BC,start] % A.output1 -> BC.input1 ;
12 [BC,committed] ENABLES [WORKFLOW1,done] %
                     A.output1 -> WORKFLOW1.output1;
13 ([A,done] & (success(A.output1) = FALSE)) | [A,failed] ENABLES
       [WORKFLOW1,fail] % ERROR -> WORKFLOW1.output2 ;
14 [BC,aborted] ENABLES [WORKFLOW,fail] % A.output1 -> WORKFLOW1.output1,
              PARTIAL_SUCCESS -> WORKFLOW1.output2 ;
...}
```

Figure 8. Workflow Specification Example 2

even though the run-time behavior of the workflow is dynamic (conditional execution of statements can be defined using predicates in intertask dependencies).

Now consider the following telecommunications example. Suppose a customer requires a digital link between three of his locations. Usually, such a link would require the following steps: (a) assignment of loop inventory from each location to the local central office, (b) the assignment of the central office equipment that connects with the loop inventory, and then (c) the assignment of trunk equipment and the associated central office equipment that connects the central offices together. The (a) steps for all three locations can be done independent of each other and similarly the (b) steps. The (c) step however first involves generating a route for the trunk equipment that would connect the central offices together. This might result in further assignments of equipment at intermediate central offices, and for some specific digital technologies, these assignments have to be done in a specific order (together with data flow between these assignments) to ensure proper connectivity. Note that since the number of possible tasks in the above example is finite, the entire workflow above can be coded up in a static specification using appropriate pre-conditions. However, enumerating all the conditions in a workflow that contains all possible tasks is messy and tedious, and the workflow specification is likely to be very long and incomprehensible. In addition, note that some complex procedures like the generation of the routes between the central offices of the customer locations are usually done by specialized software. The approach we take below allows us to incorporate these procedures as tasks in the workflow.

We summarize the situations under which we believe a workflow is dynamic or which are tedious to capture using a static "all-encompassing" specification:

1. A variable number of new instances of task types are added depending upon the values of certain data items in the workflow.

2. New flow of data are added between pre-existing tasks, between new tasks and between pre-existing and new tasks.

3. New control and data dependencies are added.

The following features in the language support the above requirements:

1. Arrays of task type instances, and syntactic sugar to refer to all the tasks or a subset of the tasks.

2. General control modifiers that process the current graphs representing control and data flows (see Section 7) and their inputs to add new tasks and dependencies to the existing graphs (in the trivial case, the entire workflow can be a control modifier with one node, but then this is a true closed legacy application!).

We handle arrays of task instances as follows. Suppose we have a `simpleTaskType` ARRAYCONTROL:

```
simpleTaskClass ACF_Class ARRAY_CONTROL
   (input@{initial} FCIF input1 ;
    output@{done} int numouts, FCIF output1[MAXNUM]) ;
```

Class ACF_Class takes an FCIF input and produces two outputs: a number and an array of FCIF outputs. Suppose the former is the number of instances that need to be executed to handle the multiple outputs being generated by the task. A program fragment indicating how the information output by an instance of class ACF_Class is used by the workflow designer as follows. We introduce the following two predicates in the language: (forall i in a..b) and (exists i in a..b) to quantify over elements in a set, where the semantics of these predicates are as usual.

```
    ...
    ACF_Class ACF ; A_Class A[MAXNUM] ;
    B_Class B;
    ...
1   [ACF,done] ENABLES
        (forall i in {1..ACF.numouts} [A[i],start]) %
        (forall i in {1..ACF.numouts} {ACF.output1[i] -> A[i].input1}) ;
2   (forall i in {1..ACF.numouts} [A[i],done] ) ENABLES
        [B,start] %
        ACF.numouts -> B.input1,
        (forall in {1..ACF.numouts} A[i].output1 -> B.inputs2[i]) ;
3   [B,done] ENABLES [WF1,done] % B.output1 -> WF1.output1 ;
4   (exists i in {1..ACF.numouts} [A,failed]) ENABLES [WF1,fail] %
    ...
```

Line 1 above indicates that when ACF reaches the done state, multiple instances of A are started, and each one of the outputs of ACF is input to one A. Line 2 shows how B is enabled when all the instances of A have reached the done state. Each of the outputs of the instances of A are fed as an array input to B, along with the number of inputs. Line 4 indicates that the enclosing WF1 fails if any of the A's reach the failed state. This example thereby shows how one can generate multiple task instances of the same class, such that a restricted class of new dependencies and data flow can be added between new and pre-existing task instances.

A more complex case is when arbitrary new dependencies and data flow can be generated and inserted into the workflow program. We take the approach that this involves application-specific code that could arbitrarily modify the entire workflow. We therefore introduce another control class called controlClass that indicates to the workflow controller that in addition to its outputs, this task produces a new workflow specification that has to be re-interpreted by the workflow controller. There is a naming problem that arises here. The software that generates a modification to the workflow should know about the current workflow program, and produce a new program that includes the same names for task instances and possibly other new names for new tasks. If this software is pre-existing code, a translator is required to translate the current graph specification to the format accepted by the software and a reverse translator to convert back to WFSL. It is assumed that no changes can be made to the declaration or state of pre-existing task instances, except involving them in more control and data dependencies. Once the control modifier finishes execution, the workflow controller reinterprets the WFSL program,

```
Contract (FCIF_id1,FCIF_id2)
FCIF FCIF_id1,FCIF_id2 ;
{   int temp_qms_stat, iter ;
    extern int did_commit() ;/*Function determining if
                              contract committed*/

    TASK_EXECUTING() ;
    iter = MAX_TRIES ; /* Try QMS call up to MAX_TRIES
                          times if system failure*/
    do
    {   EXEC QMS send_and_recv (FCIF_id1, FCIF_id2) ;
        temp_qms_stat = qms_status ;
        iter-- ;
    }
    while (temp_qms_stat == (QMS_FAILURE || OSS_DOWN_FAILURE)|| iter > 0)
    if (iter ==0) TASK_ABORTED() ; /* Abort due to system failure */
    else if (did_commit(FCIF_id2) = TRUE)
             TASK_COMMITTED(FCIF_id2) ;
        else TASK_ABORTED() ; /* Abort due to actual abort
                                 of contract at OSS */
}
```

Figure 9. Contract Task

and starts executing the workflow specification as before. An example is given in the Appendix, and more details of how this is implemented is given in Section 7.

6. TSL

In this section we briefly describe how a task program is written. A task programmer includes the following kinds of statements in his/her task program:

1. Interface specific statements : This includes statements to identify the interface and to handle errors at the interface or processing entity.

2. Processing entity specific task statements : These are statements of the task that need to be executed against the processing entity. For instance, these might be embedded SQL statements for databases, or contract specifications for OSSs.

3. Statements for revealing the task (structure) state to the workflow controlling entity : The task programmer explicitly includes macros within the program such as `TASK_EXECUTING()` and `TASK_DONE()` to indicate to the workflow controller that the task has reached a particular state.

We consider an example task program that submits a contract to an OSS through a queued message system called QMS (Figure 9). QMS is accessed by a transactional RPC call from a client. We assume that the access to QMS is made partially transparent using Embedded QMS calls (like Embedded SQL calls). We assume a `send_and_recv` construct in Embedded QMS to send a message to QMS and block until the correlated reply comes back, or there is a system failure. Suppose the workflow designer assumed `Contract` to be transactional, and the contract execution at the processing entity is also transactional. However, since the submission of the contract is through the interface, it is possible that the task completes successfully as far as the interface is concerned (i.e., without a system error), but the task could have committed or aborted. Thus, if the task does return successfully, the function `did_commit()` in the task program is used to determine whether the contract committed or aborted. Notice how the task takes into account the system errors of the interface and the processing entity. Furthermore, the workflow controller does not know any details of the interface that this task is using. Existing applications which are callable can be wrapped in such task code. The details of the TSL macros are not covered in this paper due to lack of space.

7. Architecture and Workflow Execution Support

In this section, we sketch the run-time architecture of a workflow management system that can support the model and the languages that we have discussed. A prototype of such a system has been implemented at Bellcore [4]. The run-time components (Figure 10) include the following (we assume a message passing architecture, but that need not be the case):

- A workflow controller that co-ordinates the execution of the workflow based on the specification in WFSL. If the workflow is static (*i.e.*, there are no arrays of tasks, or no `controlClass` declaration), the workflow specification in WFSL can be compiled into a controller executable. However, if the workflow can evolve dynamically, the workflow specification has to be interpreted.
- Task managers that are responsible for starting up the TSL programs and perform supervisory roles during forward recovery.
- A communications infrastructure for two purposes: (a) the controller and the task managers have to interact reliably with one another, for which we use the messaging facilities of a TP Monitor, and (b) native interfaces that allow task manager programs to interact with processing entities.
- Filter function libraries that enable the massaging of data from one format to another, extract useful information from complex formatted data, and the like. Filters can be invoked locally within the workflow controller or remotely at auxiliary systems (using the messaging infrastructure).
- A recovery management system that logs the state of individual tasks and the inputs and outputs of tasks for the purpose of forward recovery.

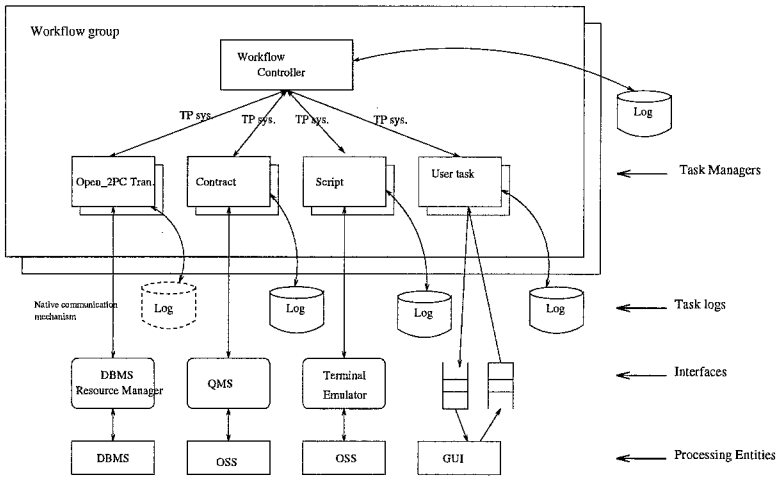


Figure 10. Run-time architecture

We now discuss the execution of the workflow controller. We first deal with the simple case when the specification is interpreted. A workflow controller is started for each instance of a workflow. The controller first reads in the specification of the workflow and interprets the specification. The internal data structure for the workflow is two sets of graphs, one for data flow and one for control flow. When the workflow instance starts, it is assumed to be in the `executing` state. The controller then starts evaluating the rules and decides on a controllable transition to enable. This transition could be for a compound task or for a simple task. The controller then determines whether the transition is a start transition. If so, for a simple task, it needs to start (or connect to) a task manager that runs the task program (a directory service that maps task names to the names of executables at specific machines). For a compound task, the controller needs to record the start of the compound task and continue to evaluate the rules of the "started" compound task. If the transition is not a start transition, the controller enables the appropriate transition by either sending a message to the appropriate task manager (for a simple task) or by simply recording the transition (for a compound task). In both cases, the enabling of the transition is logged along with the inputs associated with that event. When the task program changes its logical state (an event), it reports the current state through the TSL macros back to the controller, which logs it. If the event involves a transition to a state in which data can be made available to the controller, then the data is logged and also sent to the controller. The controller then evaluates relevant rules (involving the event in the pre-conditions, if any) and determines which controllable transitions can be enabled again. (Several strategies have been discussed in the literature, e.g., [2], [33], to determine how inter-task dependencies can be evaluated.) This continues until the workflow has logically succeeded or logically failed.

In our environment, several of the systems use closed transaction monitors with queued message inputs, so one cannot include them in an atomic transaction [22]. Transactional

compound tasks in our environment are limited to transactions that run on XA-compliant databases, i.e., the database libraries support `xa_...` calls [22]. Transactional compound tasks are executed as follows. The individual databases are registered as resource managers with the workflow controller when the task manager for each corresponding transaction is started. The workflow controller functions as the transaction manager: the workflow specification has the responsibility of generating a global transaction identifier (`trid`) for the whole transaction and then the workflow controller co-ordinates a two-phase commit between the involved resource managers, through the task managers. Essentially, the macro code for the tasks (running within the task manager process) issues the `xa_...` calls for prepare and commit (or abort), when these controllable transitions are enabled by the workflow controller. In case the databases are not XA-compliant but an open transaction manager governs them, it is possible to use the gateway techniques prescribed in [22] to interact with the database.

The next issue is that of incorporating dynamism into workflows. Our approach included syntax for specifying the enabling of transitions of arrays of tasks, by quantifying over a set of integers respresenting the indices of those tasks. On the start transitions of the tasks, the workflow controller instantiates the control graph with a fixed number of concurrently executing task instances of a particular class whose input/output behaviors are similar. The `forall` and `exist` predicates are used not only as a means of quantification, but also as syntactic sugar. If a `forall` quantifies over an entire rule (both control and data flow), then several independent instances of the rule are assumed by the workflow controller, for each index quantified in the `forall`. This allows several instances of the same rule to fire independently of one another. For instance, consider the following code fragment that enables the compensating transaction for `A[i]` if `A[i]` has already committed and the compound task to which `A` belongs has to be compensated. The input to `compA[i]` is the same as that for `A[i]`.

```
(forall i in {1..num} [WF,compensate] & [A[i],committed]
    ENABLES [compA[i],start] % A[i].input1 -> compA[i].input1 ;
```

In this case, if `num` is 2, then two rules for `A[1]` and `A[2]` respectively are assumed by the workflow controller. This allows the two compensations to be executed independent of one another. If the `forall` quantifies over only the control part of a rule, then the workflow controller assumes several rules with differing control parts based on the indices referred to in the quantification, and with same data parts. If the `forall` quantifies only over the left hand side of the control part of a rule, then a logical AND of all the instances of the left hand side is taken (logical OR for `exists`).

For more general dynamism, the `controlClass` can be used. We assume that since this can modify the graphs of control and data flow dependencies, at most one such instance of this class can be executing at any point of time. When such an instance is started, the workflow controller provides as an additional input to the task a representation of the two graphs. When the task completes execution, it provides the workflow controller with two new graphs depicting the new control flow and the new data flow. After instantiating these graphs, the workflow controller resumes the evaluation of the various control and data dependencies between tasks. The `controlClass` can also produce typed outputs that

can be used in the workflow. Thus after the graph has been modified, the instance is treated as just any other task when it is done and it has outputs that can be consumed in the workflow. (see Appendix for a dynamic workflow example).

To support forward recovery in case the workflow controller or the task managers crash, the task states, and inputs and outputs are logged. If either the workflow controller or the task managers fail, then when they restart, a recovery mechanism is used to recover their states at the time of failure. Any task manager that fails and recovers must interact with the workflow controller to resynchronize with the controller (it is possible that another task manager has been started up in the meantime to handle the same task). The recovery mechanisms used are documented in [27] and [11]. Notice however that interactions between task managers and the interfaces to processing entities might not be reliable. It is therefore possible that the same tasks may be resubmitted to a processing entity (possibly if the task manager detects a communications failure). It is imperative that either the task be idempotent, i.e, the task can be repeated with the same effect, or some other mechanism prevents the task from being executed again. OSSs and their interfaces simulate the idempotence property - any task that is submitted to the interface has a unique identifier and the OSSs do not execute it if already executed. Consider now a standard DBMS transaction. If the logging of the start of the task at the workflow controller and the database access of the task is run as part of the same (top-level) "transaction", then the task needs to be resubmitted only if it is not recorded in the workflow controller log. If the task is non-transactional, then either the task should itself be idempotent or the task manager must be able to verify that the resubmitted task has or has not been done. If neither is true, human assistance would become necessary.

We now briefly address the issues related to concurrency control and enforcement of inter-task dependencies. Traditionally, the correctness of a database system has been defined in terms of the serializability of transactions and this has been ensured by using the ACID properties. Appropriate concurrency control and recovery techniques are used to implement ACID transactions. In multi-database systems, several application-specific relaxed notions of serializability and isolation have been developed to accommodate the semantics and architecture of the application. Our environment with its mix of transactional and non-transactional tasks presents yet another difficult example where serializability and isolation of workflows cannot be preserved. For transactional tasks, the unit of atomicity and isolation is the task itself since the systems at which these tasks execute ensure isolation at the system. Non-transactional tasks do not have any isolation or serializability properties, so entire workflows cannot be serializable with respect to one another. In the case of workflows which only deal with transactional tasks, we have proposed elsewhere concurrency control and recovery mechanisms that make use of application-specific properties like limited commutativity and relaxed isolation [26].

Another requirement is that the specification of the workflow be correct, i.e., the rules guarantee progress and safety properties that the workflow will eventually terminate. In [2], attributes (*forcible*, *rejectable* and *delayable*) are associated with controllable transitions to establish safety and termination properties of a workflow. Such attributes are needed by the workflow controller for the correct enforcement of inter-task dependencies during run-time (also see [20] and [21] for related work.)

8. Conclusions

We have discussed the issue of supporting workflow management in environments that involve heterogeneous tasks such as applications that access remote servers, scripts, database transactions as well as human tasks. The tasks may be submitted through a variety of interface systems such as those based on persistent queues, RPCs, transactional RPCs, etc. The processing entities that process the tasks include script interpreters, application systems with their own databases, DBMSs, and humans (who may in turn be supported by GUI interfaces and use other application software). We discussed a workflow model and "intermediate" languages used in the METEOR project.

The language WFSL specifies the application-level issues of workflows. WFSL is characterized by *task type* declarations that capture the observable behavior of the task using *task structures* and data inputs and outputs, *task class* and instance declarations to support specification reuse, *inter-task dependencies* and *data exchange statements* that express task coordination and data flow requirements, and *filters* for data manipulation. TSL is used to specify individual tasks while accommodating some heterogeneities related to interfaces and processing entities in a distributed environment. TSL supports macros that allow tasks to reveal their task structures to the workflow management system, and also enable logging for recovery. Furthermore, task programs are designed to handle interface or processing entity related system errors.

We have also briefly discussed a system architecture for the workflow management aspect of METEOR. A prototype of METEOR was completed and used to demonstrate the basic functionality to prospective clients by using a real multi-system application, and to get further requirements. A rudimentary graphical specification interface was also implemented. One approach to the workflow controller part of the system was discussed in [2] and later implemented at MCC, while another was completed at Bellcore. A more comprehensive prototype system is currently being implemented by the collaborators in the METEOR project. We have also initiated an effort to develop a comprehensive graphical environment for specifying, testing, simulating and maintaining the workflows.

Several issues need further work. WFSL has been extended to accommodate some applications that have dynamic workflows and need more expressiveness from the language aspect. The extensions until now have dealt with specifying more flexible data or task types. While we support complex value dependencies, we have restricted the "control" aspect of WFSL (based on application needs) by using only the ENABLES evaluator. We plan to extend WFSL to include more evaluators as and when the need arises and particularly when non-transactional tasks are involved. We discussed an example that showed how a compound task can be composed of two transactional tasks with open two-phase commit by writing rules for the commit explicitly. If this was a "vanilla" transaction, then we could provide some syntactic sugar in WFSL to relieve the workflow designer from specifying the rules. We are looking at examples to see what kind of syntactic sugar might be needed. We have attempted to address some instances of heterogeneity in our environment. Several other heterogeneities exist such as those based on transaction and concurrency control mechanisms, semantic heterogeneity, and the like. We need to address such issues. Support for specifying business rules and roles is also being con-

sidered motivated by requirements for workflow automation in healthcare environments. We are also investigating how a workflow in WFSL can be evaluated for the properties of progress and safety.

Acknowledgments

The METEOR technical team at Bellcore that included the authors and Henri Weinberg and Chris Wood, and our long time collaborator Prof. Marek Rusinkiewicz have had significant influence on this work. Andrzej Cichocki implemented a prototype system based on the architecture discussed in this paper, while Henri Weinberg implemented a limited graphical workflow specification prototype. Linda Ness's technical inputs, comments and guidance, and our past collaborations with MCC (M. Huhns, M. Singh, C. Tomlinson and others), Ameritech and ETH, Zurich is also acknowledged. Our special thanks to all application experts and developers at Bellcore for their participations in a task force, for sharing their future application requirements, and for informally reviewing our work.

Appendix

A dynamic workflow example

The following example performs service provisioning, *i.e.* the assignment of inventory for a digital link between several customer locations. The input to the workflow program is a service order. For each customer location specified, the loop inventory from the customer location to the nearest central office facility is first assigned, and then inventory at each central office for connecting to the assigned loop. Then a routing task figures out what trunk lines and intermediate central offices are needed to connect all the customer locations (`MAXNUM` is the maximum number of such locations assignable on a service order) and their adjoining central offices together. The input is initially parsed by the `LoopC` task, which determines how many customer locations need to be connected and what their adjoining central offices are. This task also produces an appropriate FCIF output for each loop assignment task, `Loop`. After assigning the loops, the `Route` task is started that determines intermediate trunk lines and additional central offices. Furthermore, the loop assignments determine in what order the trunks and central offices are assigned. In short, `Route` is an instance of modifyControlClass, that *adds* data flow and dependencies between the tasks. When `Route` is done, the central office assignments can be done. If any of the steps abort, the workflow is compensated, *i.e.* all other steps in the workflow that have committed are compensated too. Notice the complex data flow between the tasks.

```
...
simpleTaskClass LoopCtrlClass ARRAY_CONTROL
  (input@{initial} FCIF input1,output@{done} int numouts,
      output@{done} FCIF output[MAXNUM]) ;
```

```
modifyControlClass RouteClass MODIFY_CONTROL
  (input@{initial} FCIF inputs[MAXNUMS],
   output@{done} FCIF outputs[MAXNUMS]);
simpleTaskClass LoopClass SIMPLE_TRANSACTIONAL
  (input@{initial} FCIF inputs[MAXNUM],
   output@{committed,aborted} FCIF output1 ) ;
simpleTaskClass compLoopClass SIMPLE_TRANSACTIONAL ... ;
simpleTaskClass SwitchClass SIMPLE_TRANSACTIONAL ... ;
simpleTaskClass compSwitchClass SIMPLE_TRANSACTIONAL ... ;

compoundTaskClass WF1 COMPOUND_NON_TRANSACTIONAL
  (input@{initial} FCIF input1 , output@{done} FCIF outputs[MAXNUM]);
{ LoopCtrlClass LoopC ;
  LoopClass Loop[MAXNUM] ;
  compLoopClass compLoop[MAXNUM] ;
  SwitchClass Switch[MAXNUM] ;
  compSwitchClass compSwitch[MAXNUM] ;
  RouteClass Route ;

  [WF1,executing] ENABLES [LoopC,start] % WF1.input1 -> LoopC.input1;
  [LoopC,done] ENABLES (forall in {1..LoopC.numouts} [Loop[i],start])%
    (forall i in {1..LoopC.numouts} LoopC.output1[i] -> Loop[i].input1);
  (forall i in {1..LoopC.numouts}[Loop[i],committed]) ENABLES [Route,start]%
    (forall i in {1..LoopC.numouts} Loop[i].output1 -> Route.inputs[i]);
  [Route,done] ENABLES (forall i in {1..LoopC.numouts} [Switch[i],start])%
    (forall i in {1..LoopC.numouts} Route.outputs[i] -> Switch[i].input1) ;
  (forall i in {1..LoopC.numouts} [Switch[i],committed]) -> [WF1,done] %
    (forall i in {1..LoopC.numouts} Switch[i].output1 -> WF1.outputs[i]);
    // Failure cases when some task aborts, and the other
    // tasks have to be compensated for
  [LoopC,failed] ENABLES [WF1,compensated] ; // No task to compensate
  [Route,failed] ENABLES [WF1,compensate] ;
  (exists i in {1..LoopC.numouts} [Loop[i],aborted] | [Switch[i],aborted])
    ENABLES [WF1,compensate] ;
  (forall i in {1..LoopC.numouts} [WF1,compensate] & [Loop[i],committed]
    ENABLES [compLoop[i],start] % Loop[i].input1 -> compLoop[i].input1 );
  (forall i in {1..LoopC.numouts} [WF1,compensate] &  [Switch[i],committed]
    ENABLES [compSwitch[i],start]%Loop[i].input1 -> compSwitch[i].input1);
  (forall i in {1..LoopC.numouts} ([compLoop[i],committed] |
    [Loop[i],aborted]) & ([compSwitch[i],committed] | [Switch[i],aborted]))
    ENABLES [WF1,compensated] ;
  ...
}
```

Notes

1. The METEOR project was initiated at Bellcore. Current collaborators include the University of Georgia and the Univerity of Houston.
2. ENABLES is a combination of the "$\rightarrow$" and "$<$" primitives [28]: If e_1 and e_2 are transitions and e_1 leads to the state s_1, then s_1 ENABLES e_2 is equivalent to $e_1 \rightarrow enable(e_2)$, $enable(e_2) \rightarrow e_1$ and $e_1 < e_2$, where $enable(e)$ indicates the action of the workflow controller enabling e.
3. The ConTracts model does provide support for versioning, and also creates a new version of a context every time a new output is produced by a task.
4. The prototype supports all key components and features discussed here, but (a) implemented limited types of task managers, and (b) several implementation choices allowed a simple implementation but at the expense of performance.

References

1. M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proc. of the 18th VLDB Conference*, August 1992.
2. P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proc. of the 19th VLDB Conference*, 1993.
3. P. Bernstein, P. Gyllstorm and T. Wimberg. STDL - A Portable Language for Transaction Processing. In *Proc. of the 19th VLDB Conference*, 1993.
4. A. Biliris, S. Dar, N. Gehani, H.V. Jagadish and K. Ramamritham ASSET: A system for supporting extended transactions. In *Proc. of the 1994 ACM SIGMOD Conference on Management of Data*, 1994.
5. M. Bregolin. Master's thesis, University of Houston, 1993
6. Y. Breitbart, A. Deacon, H.-J. Schek, A.Sheth, and G. Weikum Merging Application-centric and Data-Centric Approaches to Support Transaction-oriented Multi-system Workflows. In *SIGMOD Record*, September 1993
7. O. Bukhres, J. Chen, W. Du, A. Elmagarmid and R. Pezzoli InterBase: An Exceution Environment for Heterogeneous Software Systems. In *IEEE Computer*, Vol. 26 No. 8, August 1993.
8. J. Chen, O. Bukhres, and A. Elmagarmid. IPL: A Multidatabase Transaction Specification Language. In *Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, Pittsburgh, PA, May 1993.
9. J. Chen, Bukhres, O.A. and Sharif-Askary, J. A Customized Multidatabase Transaction Management Strategy. In *4th International Conference on Database and Expert Systems Applications*, September 6-8, 1993, Prague, Czech Republic.
10. P. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proc. of the 17th VLDB Conference*, 1991.
11. A. Cichocki. A prototype of a workflow execution controller Summer intern report and documentation, Databases and Formal Methods Research Group, Bellcore, 1994.
12. U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1990.
13. U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-Running Activities. In *Proc. of the 17th VLDB Conference*, September 1991.
14. U. Dayal and M. Shan. Issues in Operation Flow Management for Long Running Activities. In [15].
15. M. Hsu, editor. Special Issue on workflow and Extended Transaction Systems, 16 (2), June 1993.
16. A. Elmagarmid, editor. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, February 1992.
17. 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 Intl. Journal on Intelligent and Cooperative Information Systems*, 1993.
18. A.K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proc. of the 16th VLDB Conference*, 1990.
19. 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.

20. D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proc. of the Intl. Conf. on Data Engineering*, February 1994.
21. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *in this issue.*
22. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques* Morgan Kaufman, 1993.
23. R. Gunthor. Extended Transaction Processing Based on Dependency Rules. In *Proc. of the RIDE-IMS '93: Intl. Workshop on Multidatabase Systems*, April 1993.
24. M. Hsu, R. Obermarck, and R. Vuurboom. Integration and Interoperability of a Multimedia Workflow Model and Execution. In [15].
25. Y. Halabi et. al. Narada: An Environment for Specification and Execution of Multi-System Applications. In *Proc. of the 2nd Intl. Conf. on Systems Integration*, 1992.
26. W. Jin, L. Ness, M. Rusinkiewicz, A. Sheth. Concurrency Control and Recovery of Multidatabase Work Flows in Telecommunication Applications. In *Proc. of ACM SIGMOD Conf. on Management of Data*, May 1993.
27. W. Jin, N. Krishnakumar, L. Ness, M. Rusinkiewicz, and A. Sheth. Multidatabase transactions in the telecommunications environment: Modeling, Concurrency Control and Recovery Issues Bellcore Technical Memorandum, September 1993.
28. J. Klein. Advanced Rule Driven Transaction Management. In *Proc. of the IEEE COMPCON*, 1991.
29. N. Krishnakumar and A. Sheth. Specifying Multi-system Workflow Applications in METEOR. Comp. Sc. Tech. Rep. TR-CS-02, Univ. of Georgia, September 1994.
30. D. McCarthy and S. Sarin. Workflow and Transaction in InConcert. In [15].
31. R. Medina-Mora, H. Wong, and P. Flores. ActionWorkflow™ as the Enterprise Integration Technology. In [15].
32. M. Rusinkiewicz, S. Osterman, A. Elmagarmid, and K. Loa. The Distributed Operational Language for Specifying Multisystem Applications. In *Proc. of the 1st Intl. Conf. on Systems Integration*, 1990.
33. M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In Modern Database Systems: The Object Model, Interoperability, and beyond, W. Kim, Ed., Addison-Wesley/ ACM Press, 1994.
34. F. Schwenkreis. APRICOTS - A prototype implementation of a ConTract system: Management of the control flow and the communications system In Proc. 12th Symposium on Reliable Distributed Systems, 1993.
35. A. Sheth and M. Rusinkiewicz. On Transactional Workflows, In [15].
36. C. Tomlinson et. al. Workflow Support in Carnot. In [15].
37. H. Wachter and A. Reuter. The ConTract Model. Chapter 7, In [16], 1992.
38. G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. Chapter 13, In [16].

ActionWorkflow is a trademark of Action Technologies, Inc.