A CORBA Compliant Transactional Workflow System for Internet Applications

Stuart M. Wheater
Santosh K. Shrivastava
Frédéric Ranno

Department of Computing Science
University of Newcastle upon Tyne

Abstract: The paper describes an application composition and execution environment implemented as a transactional workflow system that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. The paper describes how the system meets the requirements of interoperability, scalability, flexible task composition, dependability and dynamic reconfiguration. The system is general purpose and open: it has been designed and implemented as a set of CORBA services, running on top of a given ORB. The system serves as an example of the use of middleware technologies to provide a fault-tolerant execution environment for long running distributed applications.

1 Introduction

The Internet frequently suffers from failures which can affect both the performance and consistency of applications run over it. We are particularly interested in the domain of electronic commerce that covers divergent application areas such as electronic retailing, home banking, home entertainment, information and service brokerage etc. A number of factors need to be taken into account in order to make these applications fault-tolerant.

First, most such applications are rarely built from scratch; rather they are constructed by composing them out of existing applications. It should therefore be possible to compose an application out of component applications in a uniform manner, irrespective of the languages in which the component applications have been written and the operating systems of the host platforms. Second, the resulting applications can be very complex in structure, containing many temporal and data-flow dependencies between their constituent applications. However, constituent applications must be scheduled to run respecting these dependencies, despite the possibility of intervening processor and network failures. Third, the execution of such an application may take a long time to complete, and may contain long periods of inactivity (minutes, hours, days, weeks etc.), often due to the constituent applications requiring user interactions. It should be possible therefore to reconfigure an application dynamically because, for example, machines may fail, services may be moved or withdrawn and user requirements may change.

Bearing the above observations in mind, we present a generic solution to the construction of fault-tolerant distributed applications. We have implemented an application composition and execution environment as a *transactional workflow system* that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. Our system meets the requirements of *interoperability, scalability, flexible task composition, dependability* and *dynamic reconfiguration* implied by the above discussion.

Workflow systems are widely used by organizations that need to automate their business processes, and there are many products available in the market. However, currently available workflow systems are not scaleable, as their structure tends to be monolithic. Further, they offer little support for building fault-tolerant applications, nor can they inter-operate, as they make use of proprietary platforms and protocols. Our system represents a significant departure from these; our system architecture is decentralized and open: it has been designed and implemented as a set of CORBA services, running on top of a given ORB. An overview of our system appears in [Ranno *et al.* 1997]. This paper expands on distributed fault tolerant aspects of our system. Our system architecture is the basis of Nortel's submission to the OMG for a workflow standard [Nortel & Newcastle 1998].

2 Workflow Management Systems

Workflows are rule based management software that direct, coordinate and monitor execution of tasks representing business processes. Tasks (activities) are application specific units of work. A Workflow schema (workflow script) is used explicitly to represent the structure of an application in terms of tasks and temporal dependencies between tasks. An application is executed by instantiating the corresponding workflow schema.

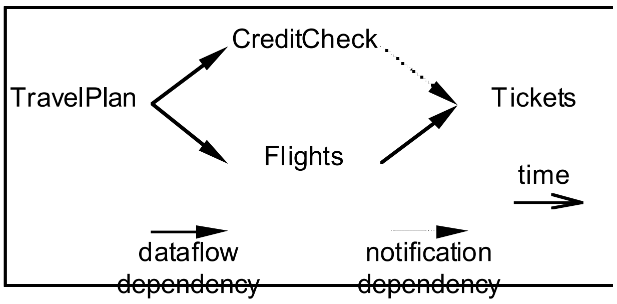

Fig. 1: Inter-task Dependencies

Imagine an electronic travel booking workflow application. Fig. 1 shows its 'activity diagram' depicting the *temporal dependencies* between its four constituent applications (or tasks), *TravelPlan, CreditCheck, Flights* and *Tickets*. Tasks *CreditCheck* and *Flights* execute concurrently, but can only be started after the *TravelPlan* task has terminated and supplied the necessary data, so these two tasks have *dataflow dependencies* on the *TravelPlan* task. Task *Tickets* can only be started after *Flights* task has terminated and supplied the necessary data and task *CreditCheck* has terminated in an 'ok' state. In this case, task *Tickets* has a dataflow dependency on *Flights*, and a restricted form of dataflow dependency (called *notification dependency*) on *CreditCheck.*

There are several organizations involved in the above application (customer organization, travel agency, credit card agency, etc.). Each organization may well possess its own workflow system for carrying out its activities. A specific way of executing this application could be: the travel agency has the application description (workflow script) and is responsible for coordinating the overall execution and it itself executes tasks *TravelPlan* and *Tickets*; its workflow system will invoke *CreditCheck* task and *Flights* task at other organizations.

Clearly, there is a need for a standard way of representing application structure and sending and receiving ‘work items’, if organizations are to cooperate. Standardization efforts are therefore underway. The Workflow Management Coalition (WfMC), an industry-wide consortium of workflow system venders, has proposed a reference model that defines interfaces with the aim of enabling different workflow systems to inter-operate [Lawrence 1997]. Unfortunately, it is a rather centralized model, not suitable for wide-area distribution [Paul *et al.* 1997]. Currently, the Object Management Group (OMG), the consortium of IT vendors and users, is evaluating new proposals for a workflow facility standard.

Our system serves as an example of the use of middleware services to construct a workflow system that provides a fault-tolerant application composition and execution environment for long running distributed applications.

3 System Overview

3.1 Meeting the Application Requirements

We discuss how our system has been designed to meet the requirements stated earlier, namely: interoperability, scalability, flexible task composition, dependability and dynamic reconfiguration.

- *Interoperability*: The system has been structured as a set of CORBA services to run on top of a CORBA-compliant ORB thereby supporting interoperability including the incorporation of existing applications and services.

- *Scalability*: There is no reliance on any centralized service that could limit the scalability of workflow applications.

- *Flexible Task Composition*: The system provides a uniform way of composing a complex task out of transactional and non-transactional tasks. This is possible because the system supports a simple yet powerful *task model* permitting a task to perform application specific input selection (e.g., obtain a given input from one of several sources) and terminate in one of several outcomes, producing distinct outputs.

- *Dependability*: The system has been structured to provide dependability at *application level* and *system level*. Support for application level dependability has been provided through flexible task composition mentioned above that enables an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within an application to deal with a variety of exceptional situations. The system provides support for system level dependability by recording inter-task dependencies in transactional shared objects and by using transactions to implement the delivery of task outputs such that destination tasks receive their inputs despite finite number of intervening machine crashes and temporary network related failures.

- *Dynamic Reconfiguration*: The task model referred to earlier is expressive enough to represent temporal (dataflow and notification) dependencies between constituent tasks. Our application execution environment is *reflective*, as it maintains this structure and makes it available through transactional operations for performing changes to it (such as addition and removal of tasks as well as addition and removal of dependencies between tasks). Thus the system directly provides support for dynamic modification of workflows (*ad hoc workflows*) [Shrivastava & Wheater 1998]. The use of transactions ensures that changes to schemas and instances are carried out atomically with respect to normal processing.

3.2 System Architecture

The workflow management system structure is shown in fig. 2. Here the big box represents the structure of the entire distributed workflow system (and not the software layers of a single node); the small box represents any node with a Java capable browser. The most important components of the system are the two transactional services, the workflow repository service and the workflow execution service.

Workflow Repository Service: The repository service stores workflow schemas and provides operations for initializing, modifying and inspecting schemas. A schema is represented according to the model described in the next section, in terms of tasks and dependencies. We have designed a scripting language that provides high-level notations (textual as well as graphical) for the specification of schemas. The scripting language has been specifically designed to express task composition and inter-task dependencies of fault-tolerant distributed applications whose executions could span arbitrarily large durations [Ranno *et al.* 1998].

Workflow Execution Service: The workflow execution service coordinates the execution of a workflow instance: it records inter-task dependencies of a schema in persistent atomic objects and uses atomic transactions for propagating coordination information to ensure that tasks are scheduled to run respecting their dependencies. Its design is discussed in section 5.

These two facilities make use of CORBA Object Transaction Service (OTS). The implementation for OTS used for the workflow management facility is OTSArjuna, which is an OTS compliant version of Arjuna distributed transaction system built by us [Parrington *et al.* 1995]. In our system, application control and management tools required for functions such as instantiating workflow applications, monitoring and dynamic reconfiguration etc., (collectively referred to as administrative applications) themselves can be implemented as workflow applications. Thus the administrative applications can be made fault-tolerant without any extra effort. A graphical user interface to the these administrative applications has been provided by making use of Java applets which can be loaded and run by any Java capable Web browser.

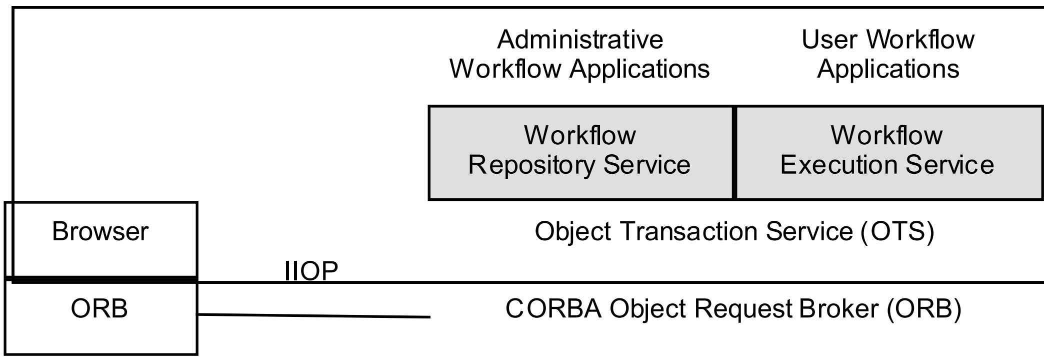
Administrative Workflow Applications
User Workflow Applications
Workflow Repository Service
Workflow Execution Service
Browser
Object Transaction Service (OTS)
IIOP
ORB
CORBA Object Request Broker (ORB)

Fig. 2: Workflow Management System Structure

4 Repository Service

A workflow schema must be expressive enough to be able to represent temporal dependencies of applications. The schema represents a workflow application as a collection of tasks and their dependencies. A task is an application specific unit of activity that requires specified input objects and produces specified output objects. As indicated earlier, dependency could be just a *notification* dependency (indicating that the 'down-stream' task can start only after the 'up-stream' task has terminated) or a *dataflow* dependency (indicating that the 'down-stream' task requires in addition to notification, input data from the 'up stream' task). We next present the *task model*, highlighting first some of its features that enable flexible ways of composing an application:

- *Alternative inputs*: A task can start in one of several initial states, representing distinct ways in which the task can be started, each associated with a distinct set of input objects. This is useful to introduce time related processing (e.g., a set of 'normal' inputs and a set for an exceptional input such as a timer enabling a task to wait for normal inputs with a timeout).

- *Alternative input sources*: A task can acquire a given input from more than one source. This is the principal way of introducing redundant data sources for a task and for a task to control input selection.

- *Alternative outputs*: A task can terminate in one of several output states, producing distinct outcomes. Assume that a task is an atomic transaction that transfers a sum of money from customer account *A* to customer account *B* by debiting *A* and crediting *B*. Then one outcome could be the result of the task committing and the other outcome could be an indication that the task has aborted.

- *Compound tasks*: A task can be composed from other tasks. This is the principal way of composing an application out of other applications. Individual tasks that make up an application can be *atomic* ('all or nothing' ACID transactions, possibly containing nested transactions within, with properties of: Atomicity, Consistency, Isolation and Durability) or *non-atomic*.

- *Genesis tasks*: A genesis task represents a place holder for a task structure, and is used for on demand instantiation. For a complex application (modeled as a set of compound tasks) this enables instantiation of only those components which are strictly necessary. This also allows the execution of repetitive tasks.

A task is modeled as having a set of *input sets* and a set of *output sets*. In fig. 3, task t_i is represented as having two input sets I_1 and I_2, and two output sets O_1 and O_2. A task instance begins its life in a *wait* state, awaiting the availability of one of its input sets. The execution of a task is triggered (the state changes to *active*) by the availability of an input set, only the first available input set will trigger the task, the subsequent availability of other input sets will not trigger the task (if multiple input sets became available simultaneously, then the input set with the highest priority is chosen for processing). For an input set to be available it must have received all of its constituent input objects (i.e., indicating that all dataflow and notification dependencies have been satisfied). For example, in fig. 3, input set I_1 requires three dependencies to be satisfied: objects i_1 and i_2 must become available (dataflow dependencies) and one notification must be signaled (notifications are modeled as data-less input objects). A given input can be obtained from more than one source (e.g., three for i_3 in set I_2). If multiple sources of an input become available simultaneously, then the source with the highest priority is selected.

A task terminates (the state changes to *complete*) producing output objects belonging to exactly one of a set of output sets (O_1 or O_2 for task t_i). An output set consists of a (possibly empty) set of output objects (o_2 and o_3 for output set O_2).

Task instances, which represent applications, manipulate references to input and output objects. Such tasks are associated with one or more implementations (application code); at run time, a task instance is bound to a specific implementation.

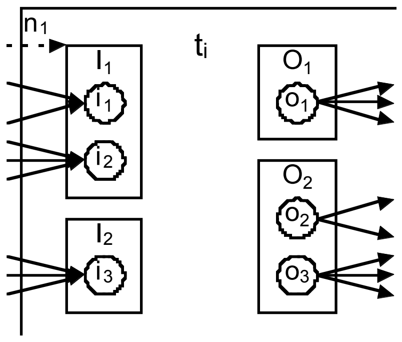

Fig. 3: A Task

A schema indicates how the constituent tasks are ‘connected’. We term a source of an input an *input alternative.* In fig. 4 all the input alternatives of a task t_3 are labeled s_1, s_2, …, s_8. An example of an input having multiple input alternatives is i_1, this has two input alternatives s_1 and s_2. Note that the source of an input alternative could be from an output set (e.g., s_4) or from an input set (e.g., s_7); the latter represents the case when an input is consumed by more than one task.

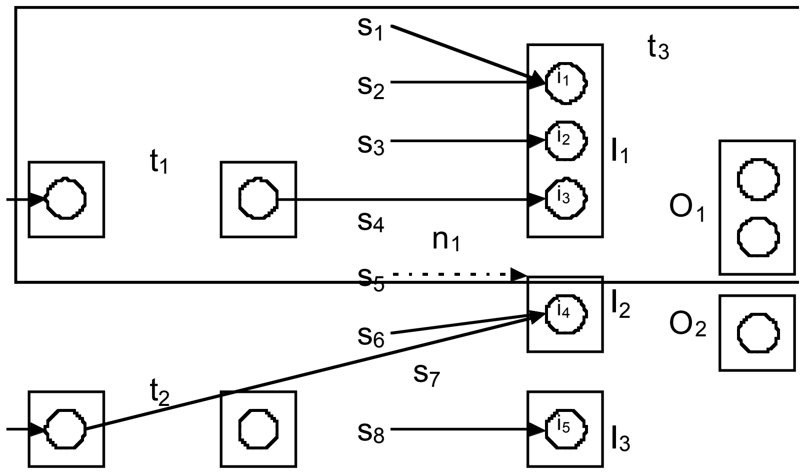

Fig. 4: A Workflow Schema Indicating Inter-task Dependencies

The notification dependencies are represented by dotted lines, for example, s_5 is a notification alternative for notification dependency n_1.

To allow applications to be recursively structured, the task model allows a task to be realized as a collection of tasks, this task is called a *compound task*. A task can either be a *simple task* (primitive task), a genesis task or a compound task composed from simple, genesis and compound tasks. A compound task undergoes the same state transitions as a simple task. The figure below illustrates a compound task, t_1, composed of tasks t_2 and t_3. A given output of a compound task can come from one or more internal sources (*output alternatives*). If multiple sources of an output become available simultaneously, then the source with the highest priority is selected.

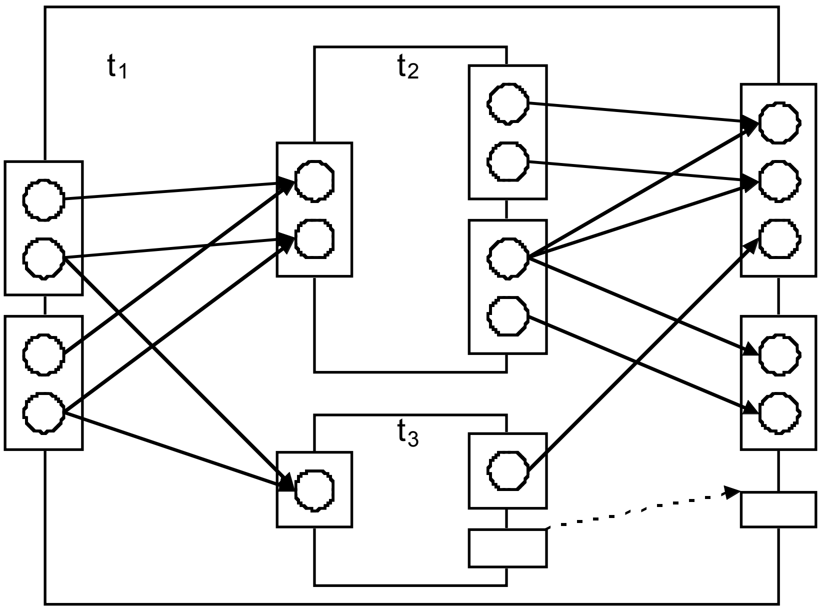

Fig. 5: A Compound Task

The workflow system stores workflow schemas in a repository and it is possible to modify the structure of a stored schema using transactions. For a complete specification of the repository service interfaces see [Nortel & Newcastle 1998].

5 Execution Service

The workflow execution service coordinates the execution of a workflow instance: it records inter-task dependencies of a schema in persistent atomic objects and uses atomic transactions for propagating coordination information to ensure that tasks are scheduled to run respecting their dependencies. The dependency information is maintained and managed by *task controllers*. Each task within a workflow application has a single dedicated task controller. The purpose of a task controller is to receive notifications of outputs (and inputs) from other task controllers and use this information to determine when its associated task can be started. The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers. Each task controller maintains a persistent, atomic object, *TaskControl* that is used for recording task dependencies. A task controller is an active entity, a process, that contains an instance of a *TaskControl* object (however, to simplify subsequent descriptions, no distinction between the two will be made). The structure is shown in fig. 6. For example, task controller tc_3 will co-ordinate with tc_1 and tc_2 to determine when t_3 can be started and propagate to tc_4 and tc_5 the results of t_3.

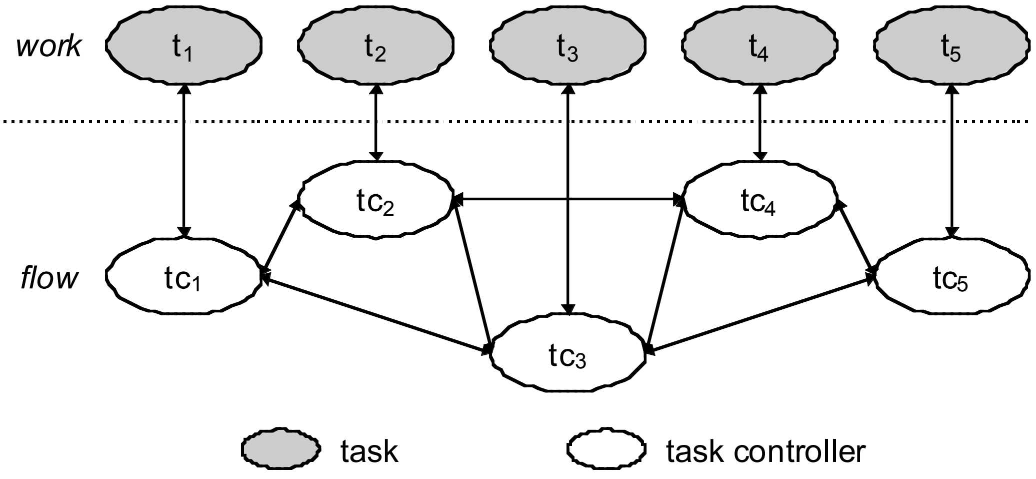

Fig. 6: Tasks and Task Controllers

In addition to *TaskControl*, the workflow execution service maintains two other key objects: instances of *Resource*, and instances of transactional objects *Task*. Objects whose references are to be passed between workflow tasks are derived from *Resource*. *Task* objects represent the workflow tasks which make up a workflow application (*Task*s are ‘wrapper’ objects to real application tasks). The most important operation contained within the *Task* interface is *start*, which takes as parameters: a reference to a *TaskControl* and a sequence of *Resource* references. The *TaskControl* reference is that of the controller of the task, and the sequence of *Resource* objects are the input parameters to the workflow task.

The *TaskControl* object provides a *get_status* operation that returns its current state. During the initial setup phase, operations can be performed on the task controller to set inter-task dependency information. If task controller (tc_i) depends on an input from the input or output set of some task controller (tc_j) it must 'register' with tc_j by invoking *request_notification* operation of tc_j (a complementary, 'unregister' operation is available for deregistering). When the relevant input/output object of tc_j becomes available, tc_j invokes the *notification* operation of tc_i to inform input availability.

Once a task controller has been setup, it enters the waiting state. The waiting, active and completed states correspond respectively to the waiting, active and completed states of a task. The task controller uses the *start* operation to start its task. Upon termination, a task invokes the *notification* operation of its controller to pass the results.

A novel feature of our system is that task controllers of an application can be grouped in an arbitrary manner. Fig. 7 show a possible configurations.

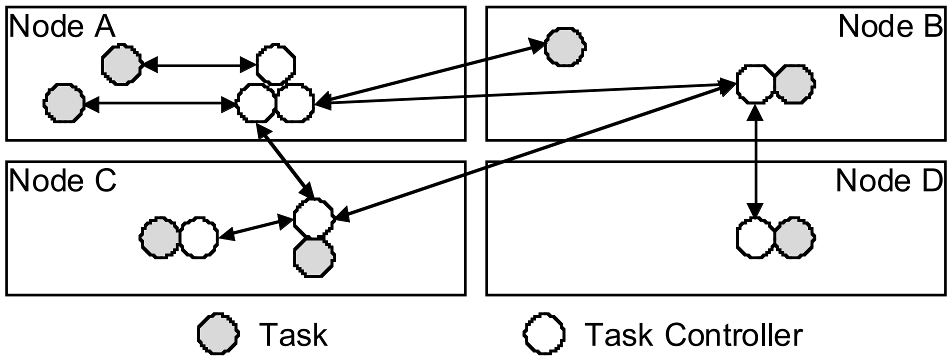

Fig. 7: Task Coordination

Nodes *B*, *C* and *D* are depicted, in fig. 7, as using a distributed coordination scheme, where a controller is co-located with the corresponding *Task* object, whereas a centralized scheme is being used by node *A*, where all the controllers have been grouped together at a given machine. A suitable configuration can be selected using the workflow administration application that is responsible for instantiating a schema (see below). The choice of a given schema could depend on various factors (e.g., dependability, performance, monitoring, administrative convenience etc.), and is left to the users and administrators. If dependability is crucial to the workflow application, then the task controllers can be placed on multiple machines so that the failure of a single machine will have a minimal effect on the progress on the workflow application. If the monitoring of the progress of the workflow application is more important than its dependability, then the task controllers can be grouped on the monitoring machine so reducing communications overhead. In most cases the placement policy for the task controllers within the workflow application will be a compromise between these two extremes.

5.1 Instantiating a workflow schema

A workflow schema stored in the repository contains almost all the information required to create an instance for execution by the execution service; the only additional information required is its initial inputs. The creation of a workflow instance involves six steps:

- *creating task controller objects*: for each simple, genesis and compound task within the schema, a task controller will be created. The placement of task controllers will depend on the various factors indicated earlier and is a user level choice.

- *creating task objects*: for each simple task within the schema, a *Task* object will be created and bound to appropriate implementation (application code of the task). If there are several conformant implementations, then the choice is left to the user.

- *assigning tasks to their task controllers*: tasks need to be assigned to task controllers so that the initiation and termination of the tasks can be controlled. A task is assigned to a task controller by invoking the *set_task* operation of the task controller with an object reference to the task as a parameter.

- *assigning task controllers of genesis tasks with task definitions*: task controllers of genesis tasks need to be assigned a task definition, this task definition being instantiated by the task controller, when the task is started. A task controller is assigned a task definition by invoking the *set_task_definitions* operation of the task controller with an object reference to the task definition as a parameter.

- *linking task controllers to form the structure of the workflow schema*: Task controllers must be initialized with inter-task dependency information contained in the schema. Task controllers possess operations such as *set_input_alternative* and *set_output_alternative* (*set_output_alternative* only appropriate to task controllers of compound tasks); these are performed on a ‘down-stream’ task controller to initialize it with the information about the “source” of a dependency (‘up-stream’ task controller). Once this information has been provided, a ‘down-stream’ task controller will invoke the *request_notification* operations on all the source ‘up-stream’ task controllers to register itself as a sink of dependencies.

- *providing the initial input*: the execution of a workflow instance will not start until the input conditions of the “root” task controller have been satisfied. This will be an application specific activity.

5.2 Workflow execution

The execution of a workflow application is controlled by the exchange of notifications between task controllers and tasks. Notifications are generated when a controller of a task, either simple, genesis or compound, selects an input set or produces an output set. As stated earlier, these notifications are sent from an ‘up-stream’ task controller to a ‘down-stream’ task controller by the former invoking the *notification* operation on the latter. The parameters of the *notification* invocation contains information indicating: the source of the notification, the identity of the input/output set which caused the notification, and the objects which constitutes the input/output set contents. Referring to fig. 4, assume that the input set maintained by the task controller for task t_2 becomes ready; in this case, the task controller will invoke the *notification* operation on the controller of t_3.

A novel dependability feature of our system is that for an atomic task (i.e., the task is performed as transaction), its task controller can provide the guarantee that the task as well as some or all of its notifications are performed atomically. Fig. 8(a) shows a schema. Assume that task *t1* is not transactional; then dependencies *A* and *B* will generate two notification transactions (*A* and *B*, fig. 8(b)) as we have discussed already. If however, task *t1* is transactional, then if desired, the execution of this transaction as well as the two notification transactions can be enclosed within an outermost transaction *T* (fig. 8(c)). Thus it can be arranged that the effects of the task's execution will only be committed if all of the notifications are completed successfully. This allows the successful completion of an atomic task to be predicated on the completion of a set of "required notifications". Such a facility may well be attractive in electronic commerce applications.

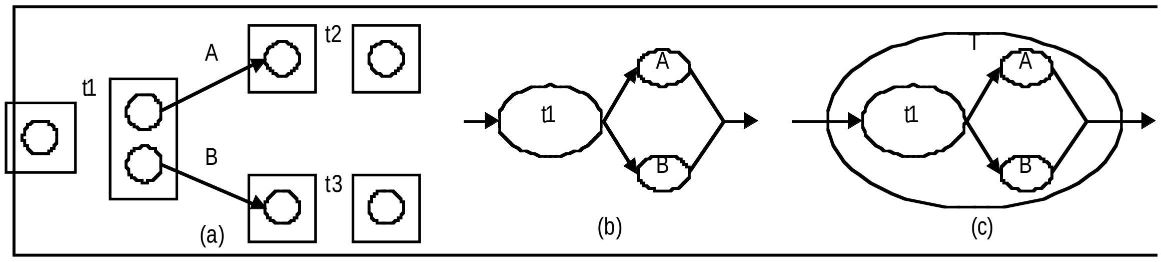

Fig. 8: Atomic Notifications

The actual algorithm used by a task controller to control its task (either simple, genesis or compound) is very simple; an important feature of this algorithm is that it can be restarted after failure. The following pseudo code describes the algorithm.

The following variables represent the state required to control a task, all variables being persistent and manipulated by transactions:

```
Boolean                         isSimpleTask;
sequence Alternative<>          inputAlternatives;
Boolean                         inputSetSelected;
ObjectSet                       inputSet;
sequence NotificationRequester<> inputSetNotificationRequesters;
TaskDefinition                  taskDefinition;
Boolean                         instantiateDefinition;
sequence Alternative<>          outputAlternatives;
Boolean                         outputSetProduced;
ObjectSet                       outputSet;
sequence NotificationRequester<> outputSetNotificationRequesters;
```

The first part of the algorithm involves collecting notifications to form an input set. This corresponds to the activity of a task controller in its 'waiting' state; the controller is repeatedly checking if any of its input sets is complete. Note that error handling in not included in the pseudo code.

```
// Collect notifications to form input set
if not inputSetSelected then
    loop
        transaction
            inputSetSelected := CheckForInputSet(inputAlternatives);
            if inputSetSelected then
                inputSet := BuildInputSet(inputAlternatives)
            endif
        endtransaction
    when inputSetSelected exit
        ProcessNotification(inputAlternatives)
    endloop
endif;
```

The second part of the algorithm involves sending notifications to all task controllers which are interested in the contents of the input set (this would be the case for shared inputs, e.g., input set of task *t2*, fig. 4 or if the input set belongs to a compound task, fig. 5).

```
// Send notifications to requesters of input set
loop
    when not ExistUnsentNotification(inputSetNotificationRequesters) exit
    transaction
        NotificationRequester notificationRequester;
        notificationRequester := GetUnsentNotification(inputSetNotificationRequesters);
        SendNotification(notificationRequester, inputSet);
        MarkAsSent(notificationRequester)
    endtransaction
endloop;
```

The third part of the algorithm involves obtaining the output set (the controller is in state ‘active’). This is done in one of three ways depending on whether the task is a simple task, a genesis task or a compound task. For a simple task this involves: beginning a transaction, starting the task with the input set, collecting the results some time later, then committing the transaction (assuming all has gone correctly). For a genesis task, this involves instantiating the task structure in its associated task definition, starting their executions and obtaining the results. For a compound task, this simply involves collecting notifications to form an output set.

```
// Obtain output set
if not outputSetProduced then
    if isSimpleTask then
        // Invoke task and obtain output
        transaction
            StartTask(inputSet);
            CollectResults(outputSet);
            outputSetProduced := true
        endtransaction
    else if isGenesisTask then
        // Instantiate task defintions and obtain output
        if not instantiateDefinition then
            transaction
                InstantiateDefinition(taskDefinition);
                instantiateDefinition := TRUE
            endtransaction
        endif;
        transaction
            StartSubTaskController(inputSet);
            CollectResults(outputSet);
            outputSetProduced := true
        endtransaction
    else // isCompoundTask
        // Collect notifications to form output set
        loop
            transaction
                outputSetProduced:= CheckForOutputSet(outputAlternatives);
                if outputSetProduced then
                    outputSet := BuildOutputSet(outputAlternatives)
                endif
            endtransaction
        when outputSetProduced exit
            ProcessNotification(outputAlternatives)
        endloop
    endif
endif;
```

The fourth and final part of the algorithm involves sending notifications to all task controllers which are interested in the contents of the output set (the task controller switches to state ‘complete’).

```
// Send notifications to requesters of output set
loop
    when not ExistUnsentNotification(outputSetNotificationRequesters) exit
    transaction
        NotificationRequester notificationRequester;
        notificationRequester := GetUnsentNotification(outputSetNotificationRequesters);
        SendNotification(notificationRequester, outputSet);
        MarkAsSent(notificationRequester)
    endtransaction
endloop;
```

In summary, our system provides a very flexible and dependable task coordination facility that need not have any centralized control. Because the system is built using the underlying transactional layer, no additional recovery facilities are required for reliable task scheduling: provided failed nodes eventually recover and network partitions eventually heal, task notifications will be eventually completed. For a complete specification of the execution service interfaces, see [Nortel & Newcastle 1998].

6 Administration Applications

As stated earlier, in our system, application control and management tools required for functions such as instantiating workflow applications, monitoring and dynamic reconfiguration etc., (collectively referred to as administrative applications) themselves can be implemented as workflow applications. This is made possible because the repository and execution services provide operations to examine and modify the structure of schemas and instances respectively. A graphical user interface (GUI) has been provided for these applications.

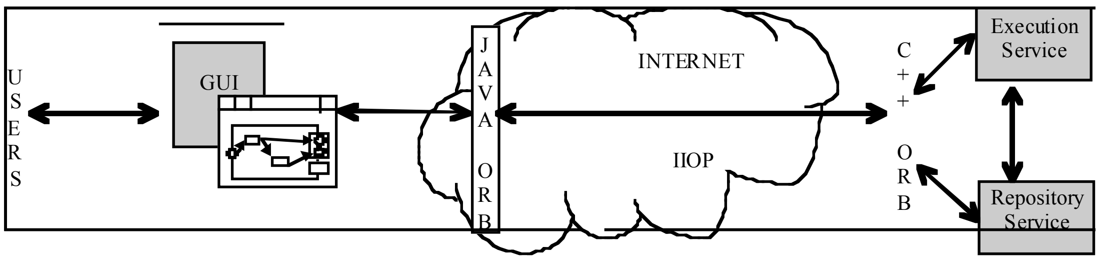

Fig. 9: Graphical User Interface

The GUI has been implemented as a Java applet and as a result it is platform independent, and can be loaded and run by any Java capable Web browser. This component of the toolkit is important as it makes it easier to use the workflow system, enabling a user to specify, execute and control workflow applications with minimal effort.

One of the purposes of the GUI is to help the designer compose the specification of a workflow application. The specification of a workflow application can be performed either from scratch by specifying all of the application components (in our script notation [Ranno *et al.* 1998], these are object classes, task classes, tasks and compound tasks) and the relationships between them or by composing the application out of existing components which have been already specified. Existing component specifications can be obtained from the workflow repository.

The GUI can be used to view graphical representations of tasks and their instances, which can then be modified, using a forms style interface to input the required modifications. Fig. 10

shows a screen dump from the Java applet showing the graphical representation of a task. Here compound tasks are represented by double line rectangles.

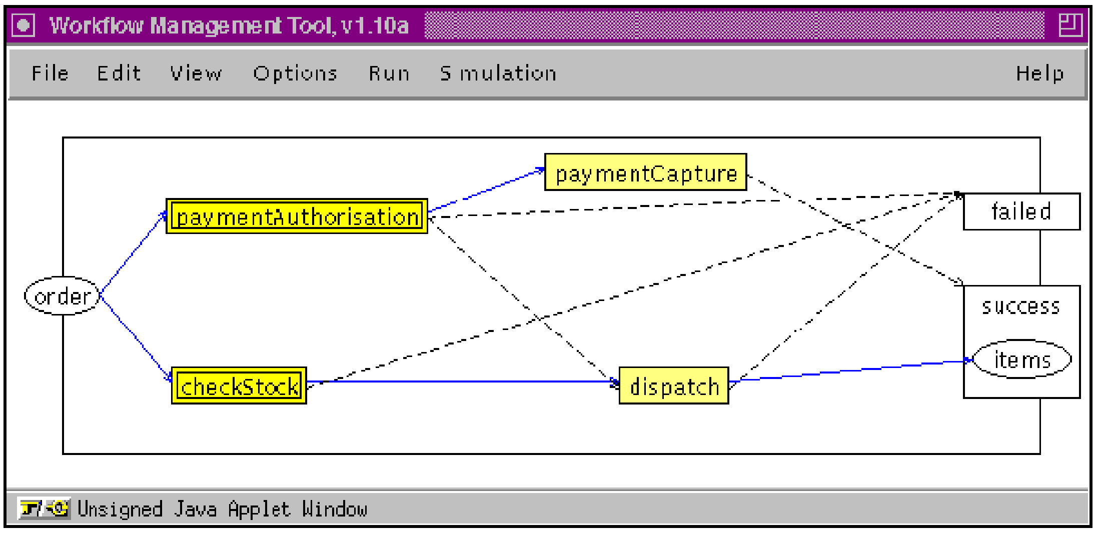

Fig. 10: Graphical Representation of a Task

A navigation system is also available that lets users zoom in and out of a specification. Zooming in on a compound task lets users see its component tasks, while zooming in on a simple task displays its task class as well as all the dependencies it is involved in. The zoom-out is the reverse action: zooming out lets users see the embedding workflow.

Consistency checking and simulation tools are also available. The script semantics can be expressed as Petri nets. So Petri net tools can be used to check the specification. We have done this using B(PN)^2 Petri net tool kit [Best & Grahlmann 1995] available to us. This tool kit can be used for checking for safeness, reachability, deadlocks and liveness. The simulator allows animating of the script in a variety of ways.

The GUI can be used for observing the execution of an application. This is possible because the GUI can access the *TaskControl* objects of the workflow execution service and hence can display the starting and completion states of tasks (recall that *TaskControl* objects store information on dependencies and task states). In addition to this simple monitoring, the GUI can also be used for driving the workflow administrative applications for dynamically modifying the execution of an application by forcing certain tasks to abort (when possible) or even by adding/removing tasks and dependencies [Shrivastava & Wheater 1998].

7 Related Work

The workflow approach to coordinating task executions provides a natural way of exploiting distributed object and middleware technologies [Georgakopoulos *et al.* 1995] [Warne 1995]. However, currently available workflow systems are not scaleable, as their structure tends to be monolithic. There is therefore much research activity on the construction of decentralized workflow architectures. Systems such as ours, RainMan [Paul *et al.* 1997] and ORBWork from the METEOR research group [Das *et al.* 1997] represent a new generation of (research) systems that can work in arbitrarily distributed environments. We briefly compare and contrast these two systems with ours.

RainMan is a distributed workflow system based on a generic workflow framework called RainMaker. The RainMaker framework defines a model in which *sources* generate service requests and *performers* manage the execution of service requests. The RainMan workflow system implementation, in Java, uses the interfaces specified by the RainMaker framework. There is a *builder* application that acts as both an interactive graphical environment for specifying workflows (as directed acyclic graphs) and as a *source* from which service requests are generated (directed acyclic graphs interpreter). The *builder* application represents a central point of co-ordination of the workflow, but is also a single point of failure.

The sources and performers of RainMan/RainMaker represent respectively, task controllers and tasks of our system. The centralized coordination configuration of our system corresponds to the ‘builder’ approach. As we have indicated, our system can support arbitrary placement of task controllers, so can be made immune from a central point of failure. There is no support for fault tolerance in RainMan.

ORBWork is a CORBA based workflow enactment system for the $METEOR_2$ workflow model. In the $METEOR_2$ model the workflow system’s runtime is divided into two types of components: *task managers* and *tasks*. The purpose of a *task manager* is to control/schedule the execution of a *task* within a workflow application. The specification of the control/schedule is stored in Workflow Intermediate Language (WIL). The WIL is used to automatically generate code fragments (C++) which are combined with ORBWork task manager code to create programs which can be used to control the workflow application. $METEOR_2$ model distinguishes different types of *tasks* components depending on their behaviour, for example, transactional, non transactions or user tasks. For each behavioural set an appropriate *task manager* is specified, with an appropriate IDL interface.

The task managers and tasks of ORBWork correspond to task controllers and tasks respectively of our system. However, unlike our system, ORBWork does not implement a transactional task coordination facility: task managers and tasks are not transactional CORBA objects. Therefore, as we have stated earlier, our system does not need special recovery facilities to deal with failures. Naturally, recovery facilities need to be implemented in ORBWork; the cited paper describes its design.

There are two features of our system that distinguishes it from the rest: i) The use of a transactional task coordination facility means that the system naturally provides a fault-tolerant ‘job scheduling’ environment. ii) Our system is reflective: computation structure is maintained by the system at run time and exposed in a careful manner for dynamic control. The execution service directly maintains the structure of an application within task controllers making it available through transactional operations. This makes the provision of system monitoring and dynamic workflows relatively easy.

8 Concluding Remarks

We have described the design and implementation of an application composition and execution environment that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. The system meets the requirements of interoperability, scalability, flexible task composition, dependability and dynamic reconfiguration. Our system architecture is decentralized and open: it has been designed and implemented as a set of CORBA services to run on top of a given ORB. Wide-spread acceptance of CORBA and Java middleware technologies make our system ideally suited to building Internet applications.

Acknowledgements

This work has been supported in part by grants from Nortel Technology, Engineering and Physical Sciences Research Council, ESPRIT CaberNet and ESPRIT LTR Project C3DS (project no. 24962). Mark Little has played a key role in making the OTS compliant version of Arjuna. Support from colleagues Samantha Merrion, Dave Stringer, Harold Toze and John Warne at Nortel Technology, Harlow is gratefully acknowledged.

References

[Best & Grahlmann 1995] E. Best and G. Grahlmann, "PEP: Programming Environment based on Petri nets", Documentation and User Guide, Version 1.4, Universitat Hildersheim, 1995.

[Das *et al.* 1997] S. Das, K. Kochut, J. Miller, A. Seth and D. Worah, "ORBWork: A reliable distributed CORBA-based workflow enactment system for METEOR2", Tech. Report No. UGA-CS-TR 97-001, Dept. of Computer Science, University of Georgia, 1997.

[Georgakopoulos *et al.* 1995] D. Georgakopoulos, M. Hornick and A. Sheth, "An overview of workflow management: from process modeling to workflow automation infrastructure", Intl. Journal on distributed and parallel databases, 3(2), pp. 119-153, 1995.

[Lawrence 1997] Lawrence P., (ed.) "WfMC Workflow Handbook", John Wiley & Sons Ltd, 1997.

[Nortel & Newcastle 1998] Nortel and University of Newcastle upon Tyne, "Workflow Management Facility Specification", Revised submission, OMG document bom/98-03-01, 1998.

[Parrington *et al.* 1995] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, 1995.

[Paul *et al.* 1997] S. Paul, E. Park, and J. Chaar, "RainMan: a Workflow System for the Internet", Proc. of USENIX Symp. on Internet Technologies and Systems, 1997.

[Ranno *et al.* 1997] F. Ranno, S.M. Wheater, and S.K. Shrivastava, "A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications", IFIP Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany, 1997.

[Ranno *et al.* 1998] F. Ranno, S.K. Shrivastava, and S.M. Wheater, "A Language for Specifying the Composition of Reliable Distributed Applications", The 18th International

Conference on Distributed Computing Systems (ICDCS '98), Amsterdam, The Netherlands, 1998.

[Shrivastava & Wheater 1998] S.K. Shrivastava, and S.M. Wheater, “Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications”, The 4th International Conference on Configurable Distributed Systems (CDS'98), Annapolis, Maryland, USA, 1998.

[Warne 1995] J.P. Warne, “Flexible Transaction Framework for Dependable Workflows”, ANSA Report No. 1217, 1995.