

# Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications

S.K. Shrivastava and S.M. Wheeler,  
Department of Computing Science,  
University of Newcastle upon Tyne,  
Newcastle upon Tyne, NE1 7RU,  
England.

## Abstract

*In a distributed environment, it is inevitable that long running applications will require support for dynamic reconfiguration because, for example, machines may fail, services may be moved or withdrawn and user requirements may change. In such an environment it is essential that the structure of running applications can be modified to reflect such changes. A complication is that such long running applications are frequently composed out of existing applications. The resulting application can be very complex in structure, containing many temporal dependencies between constituent applications. This paper describes an approach that supports the dynamic reconfiguration of large scale distributed applications. An application composition and execution environment has been designed and implemented as a transactional workflow system that enables sets of inter-related tasks (applications) to be carried out and supervised in a dependable manner. A task model that is expressive enough to represent temporal dependencies between constituent tasks has been developed. The workflow system maintains this structure and makes it available through transactional operations for performing changes to it. Use of transactions ensure that changes can be carried out atomically with respect running applications. The workflow system is general purpose and open: it has been designed and implemented as a set of CORBA services to run on top of a given ORB.*

## 1. Introduction

We consider long running distributed applications that will require support for dynamic reconfiguration because, for example, machines may fail, services may be moved or withdrawn and user requirements may change. In such an environment it is essential that the structure of applications can be modified to reflect such changes. A complication is that such long running applications are frequently composed out of existing applications. The

resulting application can be very complex in structure, containing many temporal dependencies between constituent applications. The kinds of applications we have in mind can be found within the domain of Internet based applications that covers divergent areas such as electronic retailing, home banking, home entertainment, information and service brokerage etc.

This paper describes an approach that supports the dynamic reconfiguration of large scale distributed applications. The approach is based on techniques from the area of workflow management.

Workflows are rule based management software that direct, co-ordinate and monitor execution of tasks (constituent applications) arranged to form workflow applications representing business processes. We have designed and 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 [1,2,3]. We have developed a *task model* that is expressive enough to represent temporal dependencies between constituent tasks; our application execution environment (consisting of a workflow repository and an execution service) maintains this structure and makes it available through transactional operations for performing changes to it. Use of transactions ensure that changes can be carried out atomically with respect to running applications.

We have selected the transactional workflow approach for coordinating task executions as it provides a natural way of exploiting distributed object and middleware technologies [4,5,6]. The workflow system that we have designed and implemented is general purpose and open: it has been designed and implemented as a set of CORBA services to run on top of a given ORB. Our system specification is currently under consideration by the OMG for the development of a workflow standard [2].

In the next section we discuss requirements of dynamic reconfiguration, and hint at the way we have met those requirements. The following section describes the task model supported by our workflow system and discusses the various possible changes to an application and the

conditions under which they can be performed on-line. Section 4 then describes the relevant features of the workflow system that make dynamic reconfiguration possible. Section 5 relates our work to on-going research on dynamic reconfiguration.

## 2. Requirements

Before focusing on dynamic reconfiguration requirements, we will state two related requirements concerning the development of large-scale distributed applications that have influenced the overall design of our system. The first requirement of *flexible application composition and scalability* is derived from the observation that 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 (that can be arbitrarily distributed) in a uniform manner, irrespective of the languages in which the component applications have been written, and the operating systems of the host platforms. The second requirement of *dependability* is derived from the observation that 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. Furthermore, application level fault-tolerance is necessary to maintain application specific consistency of the application in the face of failure exceptions from the underlying system (e.g., inability to start an application due to some faulty condition that is refusing to go away, say a network partition that is not healing), and application level exceptions that require error recovery in the form of aborts or compensations.

Finally we come to the requirement for *dynamic reconfiguration*. The execution of 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. A long running application is likely, at some point during its execution, to encounter changes to the environment within which it is executing; these environmental changes could include machine and network related failures, services being moved or withdrawn, or the application's functional requirements being changed. Dynamic reconfiguration mechanisms that will allow applications to change their internal structures to ensure forward progress are therefore required.

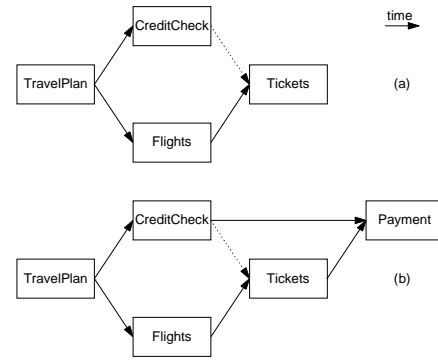


Fig. 1: Travel booking

We discuss a simple example to illustrate our observations. Imagine an electronic travel booking application. Fig. 1(a) 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. This application is widely distributed (e.g., TravelPlan is executed the local branch of the travel agent, CreditCheck is executed at some credit card agency and so forth); furthermore, each constituent application itself can be distributed. It is more than likely that some (or all) of the constituent applications already exist, so it is a matter of composing a new application out of existing ones. Clearly, there is a requirement that the constituent applications must be scheduled to run respecting the dataflow and notification dependencies, despite the possibility of intervening processor and network failures.

Dynamic reconfiguration mechanisms should make it possible to change the structure of a *running application* by adding/deleting: tasks, notifications and dataflow dependencies in a consistent manner. By consistent we mean respecting two conditions: (i) modifications are carried out atomically (either all the changes are performed or none) with respect to the normal processing activities of the application; and (ii) the application is able to execute respecting these changes. The second condition is slightly subtle, and is discussed further. Referring to fig. 1 (a) assume that the task Flights needs to be replaced by a task TrainJourney, with the same input-output dependencies; such a change can be performed provided the Flights task has not yet started. Take another example: assume that electronic payment facilities are available and, it is desirable to extend the application with a new task (Payment) with the dependencies as shown in fig. 1(b).

Once these changes have been performed, the run time system should ensure that Payment task does receive its inputs. So for example, if the changes are performed after CreditCheck has terminated, its outputs should still be made available for consumption by Payment.

We conclude this section by summarizing how we have met the requirements discussed above. Our approach to the construction of an application composition and execution environment has been to build a transactional workflow system [1,2,3]. Workflows are rule based management software that direct, coordinate and monitor execution of tasks arranged to form workflow applications representing business processes. *Tasks* are application specific units of work (e.g., TravelPlan, Payment, fig. 1). 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.

Our workflow system meets the requirement of flexible composition and scalability as follows: our system supports a very flexible task model (described in the next section) that permits tasks to be composed from other tasks, and complex inter-task dependencies to be represented in a simple manner. Secondly, the system has been designed and implemented as a set of CORBA services to run on top of a given ORB. There is no reliance on any centralized facility that could limit the scalability of applications.

Our system has been structured to provide dependability both 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 a workflow script to deal with a variety of exceptional situations [7]. 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.

In our system, dynamic reconfiguration mechanisms have been provided by making use of atomic transactions to add and remove one or more tasks and to allow the addition and removal of dependencies between tasks from a running application (an instance of a workflow schema). Use of transactions ensures that changes are carried out atomically with respect to normal processing.

### 3. Task Model and Dynamic Reconfiguration

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 right hand ‘down stream task’ can start only after the left hand ‘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 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*.

A task is modeled as having a set of *input sets* and a set of *output sets*. In fig. 2, 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. 2, 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 manipulate references to input and output objects. A task is associated with one or more implementations (application code); at run time, a task instance is bound to a specific implementation.

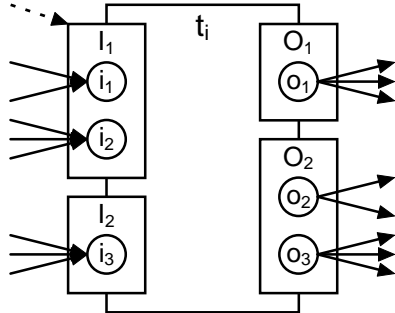


Fig. 2: A task

A schema indicates how the constituent tasks are ‘connected’. We term a source of an input an *input alternative*. In fig. 3 all the input alternatives of a task  $t_3$  are labeled  $s_1, s_2, \dots, s_8$ . 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.

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

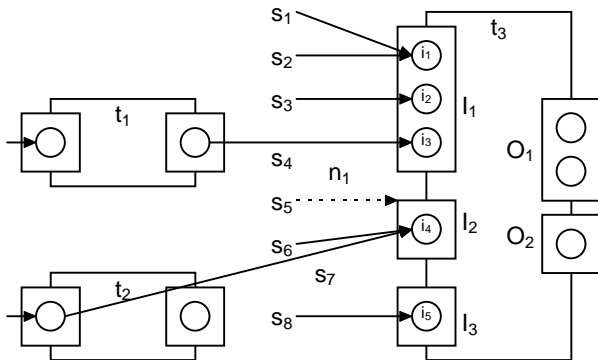


Fig. 3: A workflow schema indicating inter-task dependencies

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) or a compound task composed from simple 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.

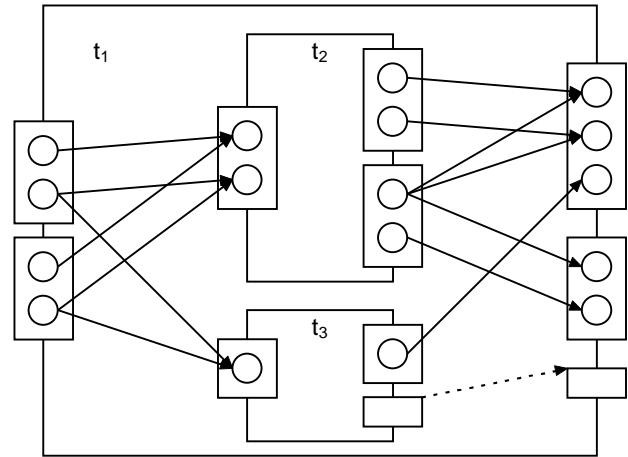


Fig. 4: A compound task

Our workflow system stores workflow schemas in a repository and it is possible to modify the structure of a stored schema; so off-line reconfiguration is supported. Here we focus on dynamic reconfiguration of a running application (modifications to an instance of a workflow schema). Below is the list of possible changes that can be performed on a workflow instance:

- The implementation bound to a simple task can be changed.
- Tasks can be added or removed from workflow instances.
- The constituent tasks of a compound task can be changed.
- Input alternatives can be added and removed from a task.
- The priority associated with input alternatives of a task can be changed.
- Output alternatives can be added and removed from a compound task.
- The priority associated with output alternatives of a compound task can be changed.

These changes must be performed consistently, by which we mean respecting two conditions: (i) modifications to a workflow schema instance are carried out atomically (either all changes are performed or none) with respect to the normal processing activities; and (ii) the application is able to execute respecting these changes.

We use transactions to respect condition one. In addition, the following restrictions need to be observed to respect condition two.

- R1: The implementation bound to a simple task can be changed, provided the task is in a wait state.
- R2: Input alternatives cannot be added, removed or in anyway modified for tasks that are in state *active* or *complete*.
- R3: Output alternatives cannot be added, removed or in anyway modified for a compoundtask that is in a state *complete*.

To illustrate the use of this approach to supporting dynamic reconfiguration, an example application will be described which comes from the field of electronic commerce.

The application involves the processing of a customer's order. It has been modeled as a compound task *processOrderApplication* which contains four constituent task instances: *paymentAuthorisation*, *checkStock*, *dispatch* and *paymentCapture*. The relationship between the tasks is shown in fig. 5. For the sake of simplicity input (output) sets are shown to contain at most one object each (an empty output set represents an aborted task termination).

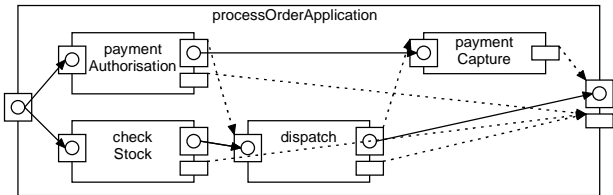


Fig. 5: Customer order processing

The *processOrderApplication* is started when an external input which is the customer's order is obtained. To process an order, *paymentAuthorisation* and *checkStock* tasks are executed concurrently. If both complete successfully then *dispatch* task is started and if that task is successful the *paymentCapture* task is started. If the *paymentAuthorisation*, *checkStock* or *dispatch* tasks fails, the *processOrderApplication* task will fail, this is represented, for all four, tasks, by the production of the second output set which contains no outputs.

The internal structure of a compound task can be modified without affecting the tasks which supply it with inputs or use it for inputs. In this case it would be possible to change the payment and stock management policies, for example, causing payment capture even if the item is not presently in stock (a regrettable practice), or the addition of a task which could check the stock levels of the suppliers of the company, and arrange direct dispatch from them.

In fig. 6 the relationship between the task instances is shown that would result from the reconfiguration of the application to allow direct dispatch from a supplier. This reconfiguration involves the inclusion of an additional task, *directDispatch*, and additional dependencies between tasks, for example, the input of the order being needed by the *directDispatch* task. This reconfiguration can be performed dynamically, provided none of the restrictions *R1*, *R2* and *R3* are not violated. Here *R2* and *R3* are relevant; provided compound task *processOrderApplication* has not terminated (*R3*), and *paymentCapture* has not started (*R2*) task *directDispatch* can be added.

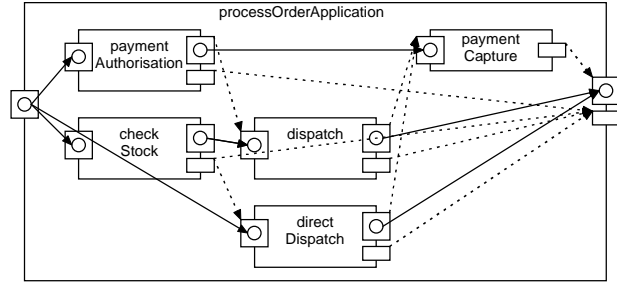


Fig. 6: Reconfigured structure of process order application

#### 4. System Support for Dynamic Reconfiguration

This section presents details of the workflow management system, concentrating on the aspects relevant to dynamic reconfiguration. The workflow management system has been constructed as two transactional services, a workflow *repository* service and a workflow *execution* service (see fig. 7). These two facilities make use 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 [8]. In our system, dynamic reconfiguration facility itself is implemented as a workflow application. In general, there will be several such workflow administration applications for managing user applications. These administration applications can be for instantiating workflow applications, monitoring an application etc. Graphical user interface to these administrative applications has been provided by making use of Java applets which can be loaded and run by any Java capable Web browser.

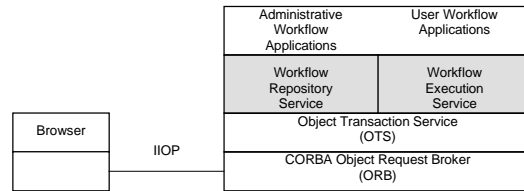


Fig. 7: Workflow management facility structure

*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 previous section, in terms of tasks, compound tasks and dependencies. We have designed a scripting language that provides high-level notations (textual as well as graphic) 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 [7].

*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. 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 of 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*, (mentioned earlier) 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. 8. For example, task controller  $tc_3$  will coordinate 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  $t_3$ .

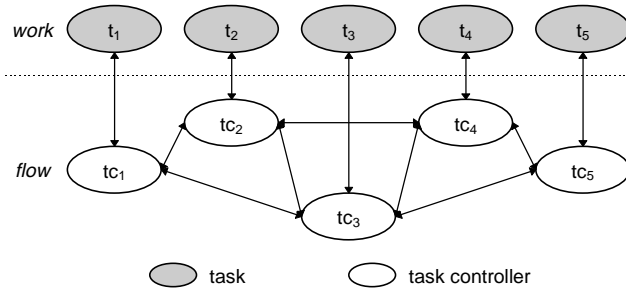


Fig. 8: Tasks and task controllers

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 (*Tasks* 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. *TaskControl* objects provide two operations (implemented as transactions): *requestNotification* and *notification*.

The state transition diagram for a task controller is shown in fig. 9. The *TaskControl* object provides a *GetStatus* 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 controller ( $tc_j$ ) it must ‘register’ with  $tc_j$

by invoking *requestNotification* 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 *notification* operation of  $tc_i$  to inform input availability.

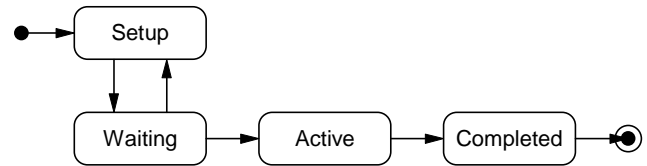


Fig. 9: Task controller state diagram

Once a task controller has been setup, it enters the waiting state. The waiting, active and complete states correspond respectively to the waiting, active and complete 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. Fig. 10 shows the interaction for some of task controllers and tasks of fig. 9.

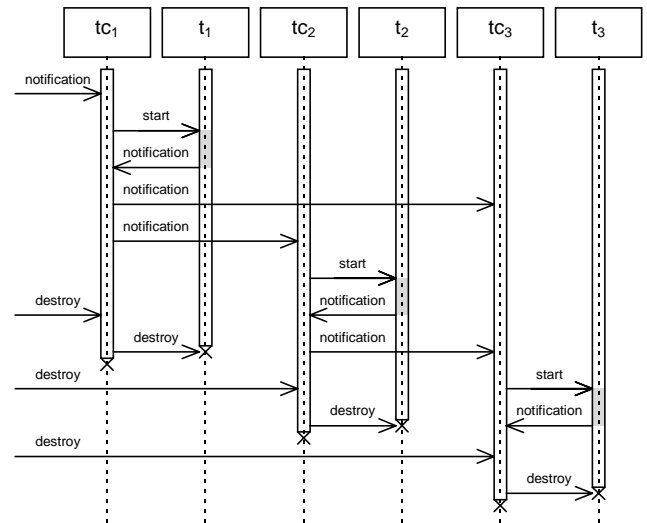


Fig. 10: Interactions between task controllers and tasks

*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 placement and initial inputs. The creation of a workflow instance involves five steps:

- *creating task controller objects:* for each simple and compound task within the schema, a task controller will be created. The placement of task controllers will depend on factors such as the dependability, performance and monitoring required by the workflow application, and is a user level choice. If dependability is crucial to the workflow application, then the task controllers can be placed on distinct 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. What is important to note is that the workflow system permits arbitrary choice to be made.

- *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. Also, the binding can be changed at any time (provided of course, restriction  $R1$  holds).
- *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.
- *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 operation 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 *requestNotification* 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.

*Workflow execution*: The execution of a workflow application is controlled by the exchange of notifications between task controllers and tasks, as illustrated in fig. 10. Notifications are generated when a controller of a task, either simple 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. 3, 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  $t_1$ .

*Example*: To illustrate the sequence of invocations required to reconfigure a workflow application, we will describe the sequence of operations which would be required to reconfigure the *processOrderApplication* workflow application from the structure in fig. 5 to that of fig. 6. The sequence of events could be as follows:

1. The reconfiguration routine is passed an object reference to the task controller which is controlling the *processOrderApplication* compound task.
2. A transaction is begun, within which subsequent operation are performed (these operations check that restrictions  $R1$ ,  $R2$  and  $R3$  are being followed by checking the status of the appropriate task controllers, else the transaction is aborted).
3. The *get\_output\_alternative* operation will be used on the task controller of *processOrderApplication* to find the task controllers which contribute to the output of the task. These task controllers will have the *get\_input\_alternative* operation used to find other task controllers. This process will be repeated until the entire structure of the *processOrderApplication* workflow application is found.
4. A task controller and task will be created for *directDispatch*, the task being assigned to the task controller.
5. The *set\_input\_alternative* operation is invoked on the task controller of *directDispatch* to specify that it requires an input from *processOrderApplication* and to specify that it requires a notification from *checkStock* (this will cause *request\_notification* to be invoked on the task controllers of *processOrderApplication* and *checkStock*).
6. The *set\_output\_alternative* operation is invoked on the task controller of *processOrderApplication* to specify that it requires an output from *directDispatch* (this will cause *request\_notification* to be invoked on the task controller of *directDispatch*).
7. The *set\_input\_alternative* operation is invoked on the task controller of *paymentCapture* to specify that it requires an input from *directDispatch* (this will cause *request\_notification* to be invoked).
8. The transaction is ended, if the reconfiguration was successful the transaction will be committed, but if for any reason the reconfiguration was not completely successful (e.g., some  $R_i$  is violated) the transaction will be aborted. The effect of aborting the transaction is to recover/undo the effects of steps 3 to 7.

To summarize our approach: we have developed a task model that is expressive enough to represent temporal (dataflow and notification) dependencies between constituent tasks; our application execution environment (workflow repository and execution service) maintains this structure and makes it available through transactional operations for performing changes to it.

## 5. Related Work

Workflow systems are widely used by organizations that need to automate their business processes. 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 and RainMan [6] represent new generation of (research) systems that can work in arbitrarily distributed environments. Our system is noteworthy for its use of transactions for inter-task coordination. Given that we use transactions for inter-task coordination, it is but natural to go a step beyond and use them for dynamic workflow modification. And that is what we have described here. To the best of our knowledge, we are not aware of any working distributed system that supports this form of functionality.

Current research on configurable distributed systems is focused on configuration management of software systems (e.g., [9,10,11,12]). The configuration of a software system is expressed as a set of components communicating through connectors. Special *architecture description languages* (ADLs) are used for describing system structure. The composition of a system is expressed in terms of components, where a component provides services to other components. A component within a system can be either a simple component, or composed out of a group of other components. The components provide and obtain services through ports. The interaction between ports can take many forms, for example, buffered message passing, one-to-many event dissemination, or synchronous request-reply communication. Dynamic configuration management in this context is usually taken to mean on-line changes to component inter-connections and/or replacement of one or more software components.

There appears to be much commonality between ADL based configuration management and the approach described here that uses workflow schema (workflow scripts) for describing application structure, and it is probable that a unified approach can be developed. At the time of writing however, we have much better understanding of the differences between them! ADLs and workflow scripts each concentrate on describing different aspects of a system structure. Whereas an ADL describes the 'system build', a script describes the temporal interactions between application level units (tasks). Dynamic reconfiguration of an application becomes conceptually very simple once the temporal structure is available. Indeed it is hard to see how dynamic reconfiguration can be performed if such a structure (or information about it) is not maintained by a system. Whilst our task model is ideally suited to expressing temporal dependencies, it does not specify the details of

mappings of tasks onto the underlying software components. Clearly there is a relationship between the two approaches that needs exploring.

## Acknowledgements

This work has been supported in part by grants from EPSRC (grant GR/L 73708) and Nortel Technology, Harlow. This work has benefited from frequent discussions with Frédéric Ranno (at Newcastle) and from colleagues at Nortel Technology, Samantha Merrion, Dave Stringer, Harold Toze and John Warne.

## References

- [1] F. Ranno, S.M. Wheeler 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, September 1997.
- [2] Nortel & University of Newcastle upon Tyne, "Workflow Management Facility Specification", Initial submission, OMG document number bom/97-08-04, August 1997.
- [3] S.M. Wheeler, F. Ranno and S.K. Shrivastava, "A CORBA Compliant Transactional Workflow System for Internet Applications", to be published.
- [4] J.P. Warne, "Flexible transaction framework for dependable workflows", ANSA Report No. 1217, 1995.
- [5] D. Georgakopoulos, M. Hornick and A. Sheth, "An overview of workflow management: from process modelling to workflow automation infrastructure", Intl. Journal on distributed and parallel databases, 3(2), pp. 119-153, April 1995.
- [6] S. Paul, E. Park and J. Chaar, "RainMan: a Workflow System for the Internet", Proc. of USENIX Symp. on Internet Technologies and Systems, 1997.
- [7] F. Ranno, S.K. Shrivastava and S.M. Wheeler, "A Language for Specifying the Composition of Reliable Distributed Applications", to be published.
- [8] G.D. Parrington, S.K. Shrivastava, S.M. Wheeler and M.C. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.
- [9] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", SIGSOFT 96, ACM Software Engineering Notes, Vol 21 No. 6, Nov. 1996.
- [10] L. Bellissard, S. Ben Atallah, F. Boyer, M. Riveill, "Distributed Application Configuration", Proc. of 16th IEEE Intl. Conf. on Distributed Computing Systems, Hong-Kong, pp. 579-585, May 1996.
- [11] V. Issarny and C. Bidan, "Aster: A Corba-based Software Interconnection System Supporting Distributed System Customization", Proc. of 3<sup>rd</sup> IEEE Intl. Conf. on Configurable Distributed Systems, Annapolis, pp. 194-201, May 1996.
- [12] J. Oueichek and X. Rousset de Pina, "Dynamic Configuration Management in the Guide Object-oriented Distributed System", Proc. of 3<sup>rd</sup> IEEE Intl. Conf. on Configurable Distributed Systems, Annapolis, pp. 28-35, May 1996.