

A System for Specifying and Coordinating the Execution of Reliable Distributed Applications

*Frédéric Ranno, Santosh Shrivastava and Stuart Wheeler,
{Frederic.Ranno, Santosh.Shrivastava, Stuart.Wheater}@newcastle.ac.uk
Department of Computing Science,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU,
England.*

Tel: +44 (0) 191 222 7972

Fax: +44 (0) 191 222 8232

Abstract

An increasing number of distributed applications are being constructed by composing them out of existing applications. The resulting applications can be very complex in structure, containing many temporal and dataflow dependencies between their constituent applications. An additional complication is that the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent applications requiring user interactions. In a distributed environment, it is inevitable that long running applications will require support for fault-tolerance and dynamic reconfiguration: machines may fail, services may be moved or withdrawn and application requirements may change. In such an environment it is essential that the structure of applications can be modified to reflect these changes.

This paper describes the design and implementation of a coordination language and the supporting execution environment for expressing the run-time composition of distributed applications and their dependencies. The system supports interoperability as it has been designed and implemented as a set of CORBA services. Dynamic control is made possible because the execution environment is built using a transactional workflow management system where sets of inter-related tasks can be carried out and supervised in a dependable manner. The paper describes how relevant ideas from the fields of workflow management and software architectures can be combined to provide an advanced application construction and execution environment, using distributed objects and middleware technologies.

Keywords: distributed systems, fault-tolerance, coordination language, transactions, transactional workflow systems, CORBA services.

1. Introduction

We describe the design and implementation of a coordination language and the supporting execution environment for expressing the composition of distributed applications and their run-time dependencies. Our work is motivated by the observation that an increasingly large number of distributed applications are constructed by composing them out of existing applications. The resulting applications can be very complex in structure, containing many temporal and dataflow dependencies between their constituent applications. An additional complication is that 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. In a distributed environment, it is inevitable that long running applications will require support for fault-tolerance and dynamic reconfiguration: machines may fail, services may be moved or withdrawn and application requirements may change. In such an environment it is essential that the structure of applications can be modified to reflect these changes.

Our coordination language is specifically designed to express task composition and inter-task dependencies of fault-tolerant distributed applications whose executions could span arbitrarily large durations. Tasks are application specific units of work. Describing an application in terms of tasks and their dependencies captures the *dataflow* and *temporal* structure of an application and is particularly meaningful to users of applications. Fault-tolerance is made possible by the provision of a distributed transactional run-time execution environment. We have in fact implemented a transactional workflow system that enables sets of inter-related tasks to be carried out and supervised in a dependable manner. Workflow scripts, specifying the tasks to be carried out are written using the coordination language to be described here.

Workflows are rule based management software that direct, coordinate and monitor execution of multiple tasks arranged to form complex organisational functions [1,2,3]. Although originally intended for automating business procedures, there is no reason why a suitably designed workflow management system cannot be used for managing and overseeing the execution of any distributed application. Our system is general purpose and supports interoperability: it has been designed and implemented as a set of CORBA services to run on top of a given ORB and makes use of Arjuna distributed transaction processing toolkit [4,5] that has been adapted to be CORBA Object Transaction Service (OTS) compliant. The execution environment provides high level graphical tools that can be run on any Java enabled Web browser.

The paper describes how relevant ideas from the fields of workflow management and software architectures can be combined to provide an advanced coordination language based application construction and fault-tolerant execution environment using distributed objects and middleware technology

2. Approach to fault-tolerance and reconfiguration

In this section we discuss our overall approach for incorporating fault-tolerance and dynamic reconfiguration, into workflow applications. We model the execution of an application as the execution of a collection of interdependent *tasks* (*activities*). A task represents a unit of work to be done (e.g., an atomic transaction that transfers a sum of money from customer account A to customer account B by debiting A and crediting B). Figure 2.1(a) depicts the inter-task dependencies of four tasks (t_1, \dots, t_4); t_2 and t_3 start once t_1 finishes and t_4 starts after both t_2 and t_3 have finished. A dependency could be just a *notification* (temporal) dependency (e.g., a task, say t_2 can start only after t_1 has

terminated) or a *dataflow* dependency (e.g., a task, say t_2 , in addition to notification, needs input data from t_1).

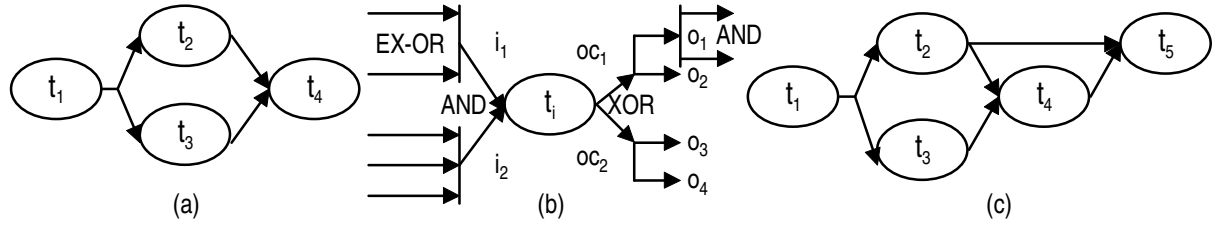


Figure 2.1, Inter task dependencies

The main function of the transactional workflow system is to ensure that tasks are scheduled to run according to their dependencies, even in the presence of faults. We distinguish two levels of fault-tolerance: system level and application level.

System level fault-tolerance: This is provided by the workflow management system which records inter-task dependencies in persistent shared objects and uses atomic transactions for implementing notification and dataflow dependencies such that tasks eventually receive their inputs despite finite number of intervening processor crashes and temporary network related failures. The system also ensures automatic (finite number of) retries of tasks that abort due to system level problems (e.g., a transaction aborting owing to a server failure).

Application level fault-tolerance: 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 a task 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 abort or compensation. Individual tasks that make up an application can be *atomic* ('all or nothing' ACID transactions, possibly containing nested transactions within, with properties of: Atomicity, Containment, Isolation and Durability) or *non-atomic*. It has long been accepted that it is not practical (or even possible) to structure a long running application as a single ACID transaction. Hence there has been much research on ways of selectively sacrificing ACID properties by constructing *extended (flexible) transaction* models for long running activities [e.g., 6,7,8]. Recent research has shown that transactional workflow systems are an excellent way of constructing application specific flexible transactions out of individual ACID transactions [1,2,9]. Supporting this requires a rich set of coordination facilities for controlling the flow of information between tasks. Bearing this in mind, our coordination language provides three specific facilities for the general case of composing an application out of atomic and non-atomic tasks:

- *Alternative input sources:* A task can acquire a given input from more than one source. In figure 2.1(b), tasks t_i requires two inputs, i_1 and i_2 , each of which can be received from any one of multiple sources (two for i_1 and three for i_2). This is the principal way of introducing redundant data sources for a task and for a task to control input selection.
- *Alternative outcomes:* A task can terminate in one of several outcomes states (oc_1 or oc_2 for task t_i , figure 2.1(b)), producing distinct outputs (o_1 and o_2 for oc_1). For example, assuming t_i to be atomic, oc_1 could be the outcome for t_i committing and outcome oc_2 could be an indication that t_i has aborted. Each output of such outcomes can be disseminated to multiple destinations (two for o_1 in figure 2.1(b)).
- *Compound tasks:* A task can be composed from other tasks. This is the principal way of composing an application out of other applications.

Dynamic reconfiguration: 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 failures, services being moved or withdrawn,

or even the application's functional requirements being changed. Mechanisms that will allow workflow applications to change their internal structures to ensure forward progress are therefore required. In our system, these mechanisms have been provided by making use of atomic transactions to add and remove one or more tasks from the workflow application and to allow the addition and removal of dependencies between tasks. Use of transactions ensures that changes are carried out atomically with respect to normal processing. Referring to figure 2.1(a), assume that it is necessary to add another task t_5 with dependencies as shown in figure 2.1(c). Provided that tasks t_2 and t_4 have not terminated, our system will permit modification of the relevant shared objects that store inter-task dependencies to reflect new changes. The rest of the paper covers the above topics in greater detail.

3. Coordination language

The coordination language (workflow script language) has been designed to allow the specification of the structure of applications at a level of abstraction which allows the specifier to concentrate on ensuring the correct functional behavior of the workflow application, even in the presence of failures. The characteristics of a set of tasks are defined by a `taskclass` construct specifying a task's input requirements and outcomes, and their associated object references. The `task` construct is used to define an instance of a given `taskclass`. It is also possible to create a `compoundtask` instance of a `taskclass`; this enables a programmer to specify the internal composition of the task in terms of other tasks. The `task/compoundtask` constructs serve three purposes, namely, to indicate: which task class the instance belongs to, the implementation of that task, and the circumstances under which that task can be initiated. The specification of the structure of an application (workflow script) consists of specifying: the classes of input and output objects used by the application, the task classes, task instances and the relationships between the inputs and outputs of these task instances. To ensure that objects are used in a way consistent with their purpose, all objects within a workflow script are associated with a class. This allows the input and output relationships of objects between task instances to be type checked. Task instances manipulate references to input and output objects.

We provide a graphical as well as a textual programming environment. We will illustrate basic features of the language using our graphical notation and later introduce the textual notations.

3.1 Atomic and non-atomic tasks

Tasks (and compound tasks) are instances of task classes. Whereas a `taskclass` specifies just the inputs and the outputs, a particular instance must also specify an implementation (the program that implements the intended behavior of the task). A given implementation could be either atomic (an ACID, possibly nested transaction), or non-atomic. Indeed, there could be several (atomic and/or non-atomic) implementations of a `taskclass` and an application designer can choose the most appropriate one at the application build time or dynamically replace a particular instance by a different one as indicated in the previous section.

A graphical representation of a single workflow task is given in figure 3.1. It depicts a task (called `Task`) that has four inputs (object references `input1`, ..., `input4`); all of these inputs must be present before the task execution can be started. A task terminates in one of the named output states (called *outcomes*). Each outcome of a task is associated with a distinct set of object references, which can be used as inputs by subsequent tasks. This task has two named outcomes: `outcome1` with two output object references, and `outcome2` with one output object reference.

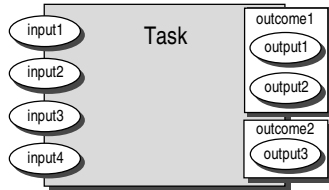


Figure 3.1, A task

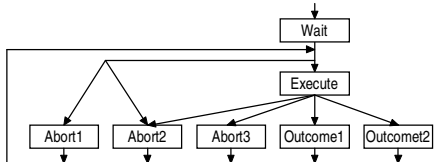


Figure 3.2, Atomic task state transitions

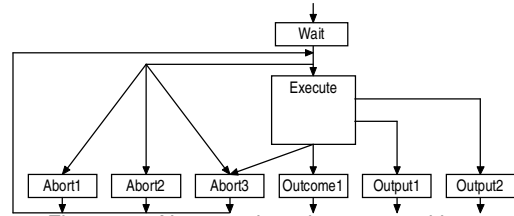


Figure 3.3, Non-atomic task state transitions

Figure 3.2 depicts the state transitions of an atomic task. It is initially in a wait state, awaiting its input dependencies to be satisfied. A task may abort (or be aborted by the execution environment) while waiting for inputs. This could be for reasons such as a timer expiring or the user forcing an abort. The task then terminates in an abort state. We permit an abort to return parameters; hence several abort states are shown, all of these map onto specific abort outcomes of the task specification. For certain abort outcomes, the system may try to restart the task automatically. If the input conditions are met, the task enters the execution state, after which it either commits or aborts. An abort outcome indicates that no changes to the application have been performed. Committed terminations map onto specific non-abort outcomes of the task specification.

Figure 3.3 indicates the state transitions of a non-atomic task. Its state transitions are similar to those of an atomic task, except that a non-atomic task can produce outputs for consumption by other tasks during its execution (in contrast, an atomic task can produce outputs only after it commits). A non-atomic task can also have one or more abort outcomes, even after entering the execution state. However, it will be the task of the implementor of the task to ensure that any results produced have been undone (this would normally be achieved by the implementor providing application specific exception handlers).

3.2 Task coordination

Inter-task dependencies could be either in the form of dataflow (input objects required by a task) or notification (a task must await the termination of a given task). The input of a task can be obtained from one of four sources: from the output of another task (see figure 3.4), from an input of another task, (figure 3.5), from one of its own outputs (if the task is to be repeated, see figure 3.6) or if the task is within a compound task, then from an input of that compound task (figure 3.7). The textual notations used for expressing these dependencies will be introduced in the next subsection.

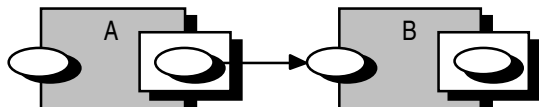


Figure 3.4, Output of task A is used as an input by task B.

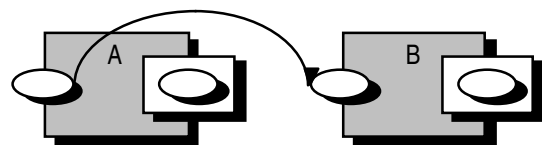


Figure 3.5, The input of task A is also used as an input by task B.

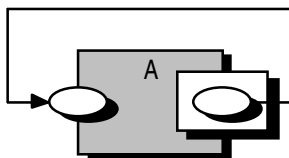


Figure 3.6, Repeating a task: output of task A is also used as its input.

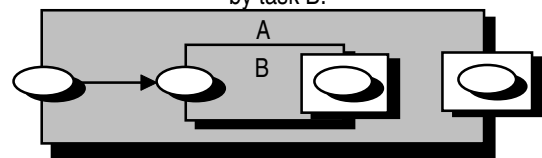


Figure 3.7, Task B receives its input from the input of compound task A.

The input to a task can come from multiple alternative sources; this is depicted in figure 3.8: the input to task C can be obtained from either the output of task A or the output of task B.

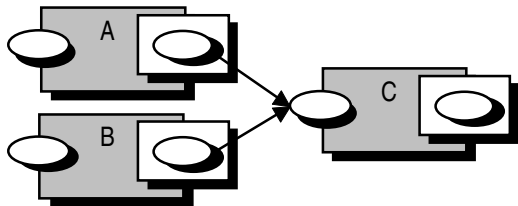


Figure 3.8, Task C can receive its input either from task A or from task B.

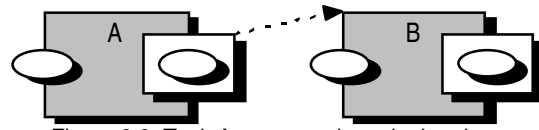


Figure 3.9, Task A must terminate in the given outcome state before B can start.

A notification dependency is depicted in figure 3.9: task A must terminate in the given outcome state before task B can start.

3.3 Composing an application

To illustrate the main features of our language, we will use a hypothetical example. It will illustrate how task classes are specified and how an application can be composed using tasks and compound tasks. The example is an application which does customer order processing. We assume that several functionally equivalent implementations of this application, satisfying differing non-functional requirements are required (e.g., an implementation built from scratch, a fault-tolerant implementation built using existing applications etc.). We therefore construct a taskclass, `CustomerOrder` that specifies the inputs and the outcomes: any instance of this class will take a `customer` and the `item` which the customer wishes to buy, and produce, if successful, an `order` (see figure 3.10). Figure 3.11, shows a particular implementation, `ProcessCustomer`, which is a compoundtask instance of `CustomerOrder`; it consists of four task instances, namely `CreditCheck` (labeled `CC`), `GetCustomerInfo` (labeled `GCI`), `GetSalesperson` (labeled `GSP`) and `StandardOrder` (labeled `SO`).

Due to space constraints the example will concentrate mainly on the taskclass `CreateOrder`. We will show two implementations of this class: a task instance (`StandardOrder`, labeled `SO` in the figure) and a compoundtask instance, `ReliableOrder` that is fault tolerant.

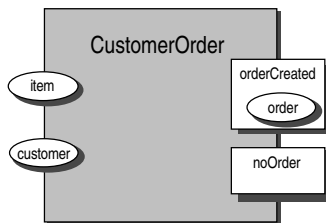


Figure 3.10, Task class CustomerOrder.

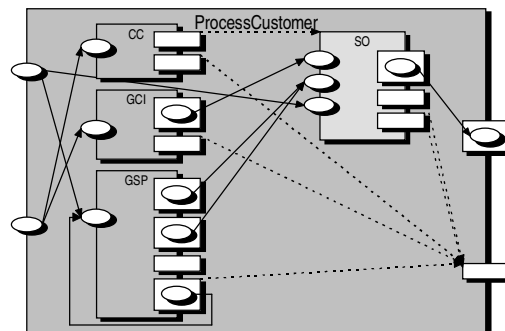


Figure 3.11, A compound task instance of CustomerOrder.

The application uses five classes of objects for input and output, which are specified within the workflow script as follows:

```
class Customer; class CustomerInfo; class Item; class Order; class Salesperson;
```

The workflow system does not require any detailed knowledge of the workflow objects, their names being sufficient. We show the textual representation of the task class `CreateOrder`. This task class has three input object references `customerInfo`, `salesperson` and `item` of object classes `CustomerInfo`, `Salesperson` and `Item`, respectively; further, the definition contains three outcomes, `orderCreated`, `noOrder` and `failed`. The outcome `orderCreated` outputs a single output object

reference order of object class Order. The outcome noOrder on the other hand does not produce any object references. The outcome failed is specified to be an abort outcome which means that if this is the outcome of the task, then the task will not have produced any effects; this particular outcome does not produce any output object references.

```
taskclass CreateOrder {
  inputs {
    customerInfo of class CustomerInfo;
    salesPerson of class Salesperson;
    item of class Item
  };
  outputs {
    outcome orderCreated {
      order of class Order
    };
    outcome noOrder { }
    abort outcome failed { }
  };
};

taskclass SelectSalesPerson {
  inputs {
    item of class Item
  };
  outputs {
    outcome successful {
      salesperson of class Salesperson
    };
    repeat salespersonBusy {
      item of class Item
    };
    outcome requiresSupervisor {
      supervisor of class Salesperson
    };
    abort outcome failed { }
  };
};
```

The task GetSalesperson (labeled GSP in figure 3.11) is an instance of class SelectSalesperson shown below.

In this class, the output salespersonBusy is a repeat outcome which means that if an instance of this task class produces the outcome salespersonBusy it will be restarted with the item from this outcome as input. This will be done automatically by the workflow execution environment.

We next discuss how an instance of a taskclass can be specified textually, by illustrating the case of task StandardOrder of class CreateOrder. Assume that the particular implementation of this task (executable binary) has been named createStdOrder. We have to specify the sources or each of the three inputs. The specification indicates that input customerInfo can be obtained from the task GetCustomerInfo if the outcome of this task is successful. On the other hand salesperson can be obtained from any of two possibilities: salesperson from the task GetSalesperson if that task has an outcome of successful or supervisor from the task GetSalesperson if that task has an outcome of requiresSupervisor. Order of input alternatives is significant: the first in the list being given preference over the second one and so on. The input item is specified to be obtained from input item to the task ProcessCustomer. The final requirement of the initiation of the task StandardOrder is that the task CreditCheck has achieved the outcome creditworthy (the reader should compare these notations with the graphical ones depicted in figure 3.11).

```
task StandardOrder of taskclass CreateOrder {
  code createStdOrder;
  inputs {
    input customerInfo from {
      customerInfo of task GetCustomerInfo if outcome successful
    };
    input salesperson from {
      salesperson of task GetSalesperson if outcome successful;
      supervisor of task GetSalesperson if outcome requiresSupervisor
    };
    input item from {
      item of task ProcessCustomer
    };
    notification from {
      task CreditCheck if outcome creditworthy
    }
  };
};
```

Next we describe how a compound task instance is specified. The specification of the compound task has three parts: specification of inputs, specification of the task instances it

contains, and finally the specification of the relationship between the outputs of the tasks and its own outputs. The specification of inputs is identical in structure to that of `task` construct; the specification of task instances is also performed identically to that described previously. As an example of a compound task we will specify a replacement for the task `StandardOrder` described before with a compound task called `ReliableOrder`. The compound task `ReliableOrder` contains two task instances `Primary` and `Secondary` both of task class `CreateOrder`. Its purpose is to provide a more reliable way of processing an order: if the primary task fails, the secondary is initiated. The structure of this compound task is represented in figure 3.12.

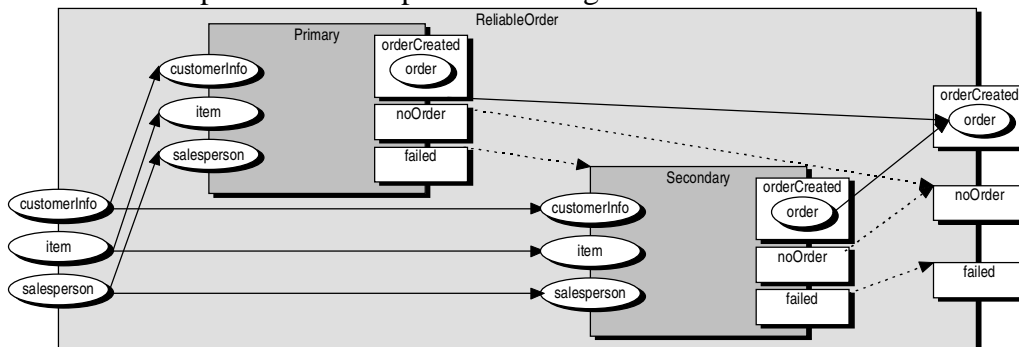


Figure 3.12, Structure of compound task

One difference between the specification of a task and a compound task is that for a compound task the mapping from the outputs of internal task instances to the compound tasks outputs must be specified. In the specification below the `orderCreated` of outcome `orderCreated` of the compound task can be the `order` object reference coming either from `Primary` or `Secondary` (if there is choice, then like input alternatives, the first in the list is given preference over the second one and so on). Outcome `failed` is activated if the task `Secondary` has outcome `failed`.

```
compoundtask ReliableCreateOrder of taskclass CreateOrder {
  inputs {
    input customerInfo from { . . . };
    input salesperson from { . . . };
    input item from { . . . }
  };
  task Primary of taskclass CreateOrder { . . . };
  task Secondary of taskclass CreateOrder {
    notification from { task Primary if outcome failed }
  };
  outputs {
    outcome orderCreated {
      output order from {
        order of task Secondary if outcome orderCreated;
        order of task Secondary if outcome orderCreated
      };
    };
    outcome noOrder {
      notification from {
        task Primary if outcome noOrder;
        task Secondary if outcome noOrder
      }
    };
    abort outcome failed {
      notification from { task Secondary if outcome failed }
    };
  };
};
```

4. Transactional workflow toolkit

The main function of the transactional workflow toolkit is to ensure that tasks are scheduled to run according to their dependencies. The workflow toolkit consists of five main components:

- The *workflow execution service* is responsible for reliably coordinating the execution of the tasks which constitute a workflow application. The service manages to achieve such coordination despite temporary machine or network failures.
- The *workflow specification service* is responsible for maintaining the specifications of workflow applications in a form which is accessible by workflow applications.
- The *graphical user interface* (GUI) allows users of the system to specify, execute and monitor workflow applications, using an interface which is easy to use and understand.
- *Script servers* are used to store the specifications of workflow applications in their text form. These specifications need not be complete, so unlike the workflow specification service, the script servers can be used to store workflow specifications during their construction.
- *Workflow administration tasks* are workflow tasks for managing other tasks. These tasks can for example, be used to instantiate workflow applications, perform dynamic reconfiguration or terminate tasks whose outputs are no longer required.

The relationships between the components of the workflow toolkit are depicted in figure 4.1.

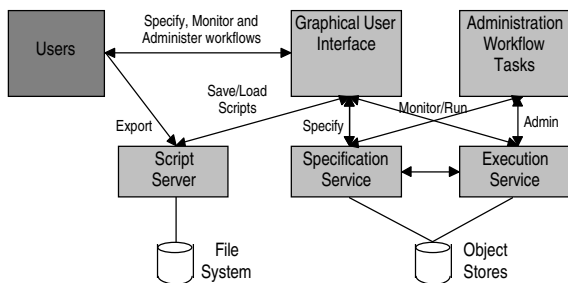


Figure 4.1 Relationships between the different components of the workflow toolkit.

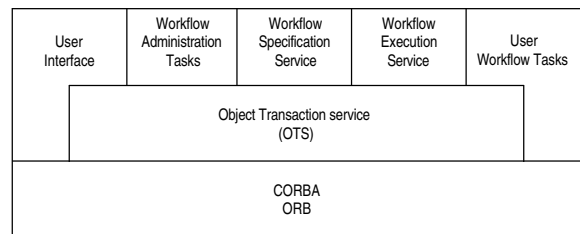


Figure 4.2 Software structure of workflow toolkit.

To allow interoperability with other systems, most of the service components within the workflow toolkit are provided through CORBA interfaces, which are defined using the CORBA Interface Definition Language (IDL). Workflow toolkit components which make use of these services are constructed as CORBA clients of these services. Figure 4.2 shows the resulting software structure of the workflow toolkit. The underlying transaction support has been provided by the Arjuna distributed transaction system [4,5] which has been adapted to be CORBA object transaction service (OTS) compliant. Thus our system is open and fully distributed. Our current implementation uses the proprietary object request broker (ORB) of our industrial sponsor; ports on commercially available ORBs is fairly straightforward and in progress.

In the following section we will briefly describe the workflow execution service. The other major component of the workflow system is the GUI which allows users to specify, monitor and control the execution of workflow applications. The GUI makes use of the CORBA service provided by the rest of the workflow system and being implemented in Java is platform independent, this means that it can be run on any Java enabled Web browser. The workflow system, including the GUI, is described in more detail in [11].

4.1 Execution service

The workflow execution service provides the system level fault tolerance described in section 2; it records inter-task dependencies in persistent shared objects and uses atomic transactions for propagating coordination information. The coordination information is maintained 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 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, which is used for recording task dependencies (see below). This structure is shown in figure 4.3.

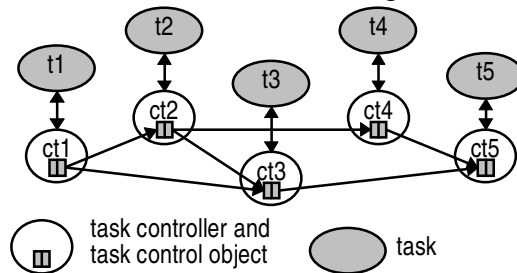


Figure 4.3, Shows the relationship between tasks and task controllers within a workflow application.

The workflow execution service maintains a set of CORBA objects, the key ones are: TaskObject, Task and TaskControl. Objects whose references are to be passed between workflow tasks are derived from TaskObject. Task objects represent the workflow tasks which make up a workflow application. The most important operation contained within the Task interface is start, which takes as parameters: a reference to a TaskControl and a sequence of TaskObject references. The TaskControl reference is that of the controller of the task, and is used for notifying the task's controller of the outputs of the task. The sequence of TaskObject objects are the input parameters to the workflow task.

TaskControl objects provide the most important service within the workflow execution environment; they collectively maintain the structure of the workflow application and information about the status of tasks. TaskControl objects are used for implementing a specialised transactional event management system. The system allows the propagation of information about events which occur within the workflow application, for example, the completion of a task. To support this function two operations (implemented as transactions) are provided by TaskControl: requestNotification and notify. The requestNotification operation is invoked on a TaskControl instance by other TaskControl instances which are interested in an event (typically an output) produced by the task associated with the invoked TaskControl object; the events being propagated using the notify operation. The other purpose of a TaskControl object is to control the starting of the associated task.

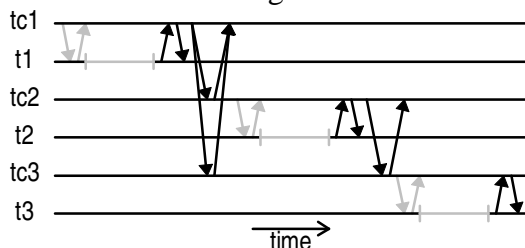


Figure 4.4, Shows the interaction between task controllers and tasks within a workflow application.

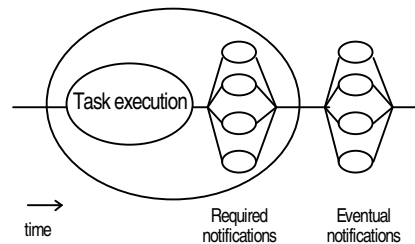


Figure 4.5, Execution of an atomic workflow task.

To illustrate how tasks and task controllers interact, the significant messages exchanged between some of the tasks and task controllers within figure 4.3 are shown in figure 4.4. The pairs of arrows represent a remote procedure call. Two types of calls are shown: notify transaction operations (in black) and task start instructions (in grey). The grey line on a task line represents the task in a state of execution.

As the figure illustrates, when a task terminates, the associated task controller executes the required notify operations on the task controllers of the dependent tasks. Referring to t1

and `tc1`, first `t1` invokes the `notify` operation of the `TaskControl` object of `tc1` to inform `tc1` of the termination and also to deposit any output object references `t1` has produced; then `tc1` invokes the `notify` operations at `tc2` and `tc3`. A novel feature of our system is that for an atomic task (i.e., the task is performed as transaction), the execution service can provide the guarantee that the task as well as some or all of its notifications are together performed atomically (see figure 4.5). It is possible to nest some (or all) of the notifications within an outer most transaction which also includes the task transaction (the task transaction itself could be nested of course, but this is not depicted in the figure). Thus it can be arranged that the effects of the task's execution will only be committed if all of the notifications within the encompassing transaction are completed successfully. This allows the successful completion of an atomic task to be predicated on the completion of a set of "required notifications". The remaining notifications are performed "eventually". If a task is not atomic, then, naturally all notifications can only be performed using the eventual notification scheme.

The second novel feature of our system is that task controllers of an application can be grouped in an arbitrary manner. Figures 4.6 and 4.7 show two possible configurations. A distributed coordination scheme is depicted in figure 4.6, where a controller is co-located with the corresponding `Task` object, whereas a centralised scheme is shown in figure 4.7, where all the controllers have been grouped together at a given machine.

A suitable configuration can be selected using the workflow administration task that is responsible for instantiating an application. The choice of a given scheme could depend on various factors (e.g., degree of fault tolerance required, administrative convenience etc.), and is left to the users and administrators.

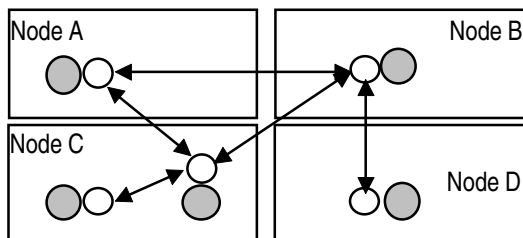


Figure 4.6, Distributed coordination

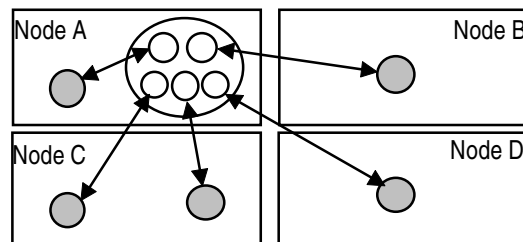


Figure 4.7, Centralised coordination

5. Related work

In this section we will concentrate mainly on the coordination language aspects of our work, and relate it to recent research work on architecture description languages (ADLs) and workflow scripting languages.

Workflow scripting languages use entities such as tasks (transactional or non-transactional) and their termination conditions (e.g., `commit`, `abort`) for expressing inter-task dependencies. The language developed by the METEOR group [12] is a representative example of this class of languages that are good at expressing temporal structure of an application. Such languages use the 'event-condition-action' paradigm for application building. Event conditions specify firing rules for enabling the corresponding actions (tasks). We have chosen not to follow this way of application structuring as we have not found it sufficiently modular. Our approach for composing applications is closer to those of ADLs. But as we discuss below, our language like workflow scripting languages, concentrates on representing the temporal structure of an application, rather than its software component structure as current ADLs do.

ADL-based specifications are intended to describe the structure of the components of a software system and their inter-relationships [13,14,15,16,17]. It is common to model an application as a set of components communicating through connectors. Typically, an

application is composed from components, where a component provides services to other components. A component within an application 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.

ADLs however capture the software structure of an application from the perspective of an application builder and not from the perspective of the *users of the application*. For the users, meaningful specification of an application has to be in terms of the tasks (activities) the application performs. This requires describing the temporal structure of an application [18]. Our language captures this structure in terms of tasks and their dependencies by specifying input output requirements. Another advantage of describing application structure in terms of tasks is that it directly enables application level fault tolerance requirements to be specified and controlled: we do this using transactions and compound tasks.

Consider the interaction of a client C with three servers S_1 , S_2 and S_3 . Let us assume that the client C first interacts with server S_1 , then based on the results of this interaction C either interacts with server S_2 or server S_3 . Using an ADL such as Darwin, it would be possible to describe the fact that C will interact with S_1 and possibly with S_2 and S_3 , but it would not be possible to describe the temporal aspects of the interaction between the client and the servers, and which sets of interactions constitute application level units of work (tasks), and which interactions are to be made fault-tolerant. Whilst our language is ideally suited to stating precisely these requirements, it cannot specify the details of mappings of tasks onto the software components. The combination an ADL such as Darwin [13] with our language will permit both structural and temporal specifications of an application to be expressed in a uniform manner. This is suggested as a direction for future research.

6. Concluding remarks

Our system is intended to provide a fault-tolerant execution environment for long running distributed applications, such as telecommunication, banking and business processes. We have selected the transactional workflow approach for coordinating task executions, as it provides a natural way of exploiting distributed object and middleware technologies. Our system serves as an example that this is possible. Future work on coordination will include producing rigorous specifications of the semantics of the language, producing example scripts illustrating how extended transaction models can be composed and building demonstrator applications.

7. Acknowledgments

This work has been supported in part by grants from Nortel Technology, Engineering and Physical Sciences Research Council and ESPRIT CaberNet. Two members of the Arjuna research group, Graham Parrington (now with Sun Microsystems) and Mark Little have played key roles in making the OTS compliant version of Arjuna. Support from colleagues at Nortel Technology, Samantha Merrion, Harold Toze and John Warne is gratefully acknowledged.

References

- [1] J.P. Warne, "Flexible transaction framework for dependable workflows", ANSA Report No. 1217, 1995.
- [2] M. Rusinkiewicz and A. Sheth, "Specification and execution of transactional workflows", Modern database systems (W. Kim, ed.), pp. 592-620, ACM Press, 1995.

- [3] 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, April 1995.
- [4] S.K. Shrivastava and D. McCue, "Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 4, pp. 421-432, April 1994.
- [5] G.D. Parrington, S.K. Shrivastava, S.M. Wheeler and M. Little, "The design and implementation of Arjuna", *USENIX Computing Systems Journal*, vol. 8 (3), pp. 255-308, Summer 1995.
- [6] H. Wächter and A. Reuter "The ConTract model" in "Transaction Models for advanced Database applications", (ed. A. Elmagarmid), Chapter 7, pp. 220-262, Morgan-Kaufman, February 92.
- [7] H. Garcia-Molina and K. Salem, "Sagas", *Proc. 1987 SIGMOD Intl. Conf. on the Management of Data*, pp. 249-259, May 1987.
- [8] S.K. Shrivastava and S.M. Wheeler, "Implementing fault-tolerant distributed applications using objects and multi-coloured actions", *Proc. of 10th Intl. Conf. on Distributed Computing Systems*, Paris, pp. 203-210, June 1990.
- [9] G. Alonso et al., "Advanced transaction models in workflow contexts", *Proc. of 12th Intl. Conf. on Data Engineering*, New Orleans, February, 1996.
- [10] D. Georgakopoulos, M. Hornick and F. Manola, "Customising transaction models and mechanisms in a programmable environment supporting reliable workflow automation", *IEEE Trans. on Knowledge and Data Eng.*, Vol. 8(4), pp. 630-649, August 1996.
- [11] S.M. Wheeler, F. Ranno and S.K. Shrivastava, "Newflow: A distributed transactional workflow system", Technical report, University of Newcastle upon Tyne, 1997.
- [12] N. Krishnakumar, A. Sheth, "Managing heterogeneous multi-system tasks to support enterprise-wide operations", *Intl. Journal on distributed and parallel databases*, 3(2), November 1994.
- [13] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying Distributed Software Architectures", *Proc. of 5th European Software Engineering Conference, ESEC '95*, Barcelona, September 1995.
- [14] J Magee and J. Kramer, "Dynamic Structure in Software Architectures", *SIGSOFT 96, ACM Software Engineering Notes*, Vol 21 No. 6, Nov. 1996.
- [15] M. Radestock and S. Eisenbach, "Semantics of a Higher-Order Coordination Language", *Coordination Languages and Models, First Intl. Conf.*, Cesena, Italy, LNCS, Vol. 1061, Springer Verlag, April 1996.
- [16] 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.
- [17] V. Issarny and C. Bidan, "Aster: A Framework for Sound Customisation of Distributed Runtime Systems", *Proc. of 16th IEEE Intl. Conf. on Distributed Computing Systems*, Hong-Kong, May 1996.
- [18] P. Wegner, "Coordination as constrained interaction", *Coordination Languages and Models, First Intl. Conf.*, Cesena, Italy, LNCS, Vol. 1061, Springer Verlag, pp. 28-33, April 1996.