

A Language for Specifying the Composition of Reliable Distributed Applications

Frédéric Ranno, Santosh K. Shrivastava, Stuart M. Wheeler,
Department of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, England.

Abstract

This paper describes the design of a scripting language aimed at expressing task (unit of computation) composition and inter-task dependencies of distributed applications whose execution could span arbitrary large durations. This work is motivated by the observation that an increasingly large number of distributed applications are constructed by composing them out of existing applications, and are executed in an heterogeneous environment. The resulting applications can be very complex in structure, containing many notification and dataflow dependencies between their constituent applications. The language enables applications to be structured with the properties of modularity, interoperability, dynamic reconfigurability and fault-tolerance.

1. Introduction

A scripting language (co-ordination language) is a very useful application building tool for specifying the composition of applications. The scripting language to be described here has been 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 computation. Our work is motivated by the observation that an increasingly large number of distributed applications are constructed by composing them out of existing applications, which are executed in a heterogeneous environment. The resulting applications can be very complex in structure, containing many notification and dataflow dependencies between their constituent applications. Furthermore, 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will require support for fault-tolerance and dynamic reconfiguration: machines may fail, services may be moved or withdrawn and application requirements may change.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 a scripting language that has been designed to specify the composition of distributed applications with the properties of modularity, interoperability, dynamic reconfigurability and fault-tolerance.

The work described here is claimed to be novel in that scripting language designers have paid little attention to meeting the above requirements in an integrated manner as reported here. We have designed and implemented the language and the supporting execution environment. An overview of the system is presented in [1]. The details of the execution environment, implemented as a collection of CORBA services to form a distributed transactional workflow system are described in [2]. Our workflow system is currently under consideration by the OMG for the development of a workflow standard [3]. This paper concentrates on the language aspects of our work.

Section two of the paper discusses the requirements mentioned above in greater detail; section three then describes our approach to meeting these requirements. Section four describes the main features of the language, and their use is illustrated with the help of a few illustrative examples in section five. Section six describes related work in the area of scripting languages for distributed applications. Concluding remarks, including directions of further work are presented in section seven.

2. Understanding Requirements

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). Fig. 1 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* dependency (shown by a dotted arc, indicating that t_2 can start only after t_1 has terminated) or a *dataflow* dependency (shown by a solid arc, indicating that, say t_3 , needs in addition to notification, input data from t_1). We next discuss the four requirements of dynamic reconfigurability, fault-tolerance, modularity and interoperability.

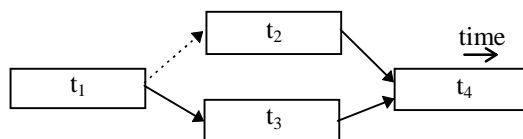


Fig. 1: Inter task dependencies

Fault-tolerance: Notification and dataflow dependencies must be implemented such that tasks

eventually receive their inputs and notifications despite finite number of intervening processor crashes and temporary network related failures. Nevertheless, applications must be prepared to face failure exceptions from the underlying system (e.g., an exception indicating the inability to start a task due to some faulty condition that is refusing to go away, say a network partition that is not healing). Furthermore, facilities to cope with application level exceptions that require error recovery in the form of compensation or task abortion are also needed. 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*. Bearing this in mind, the scripting language should provide facilities for the general case of composing an application out of atomic and non-atomic tasks.

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. As stated earlier, these environmental changes could include machine and network related failures, services being moved or withdrawn, or even the application's functional requirements being changed. Mechanisms that will allow applications to change their internal structures to ensure forward progress are therefore required. It should be therefore possible to change the structure of a running application by adding/deleting tasks, notifications and dependencies.

Modularity: A specification (script) should enforce locality of modifications; only the parts of the script directly affected by a change should need changing. For example, adding an additional input dependency to a task should only affect the script for that task. Further, the language should permit flexible ways of specifying the composition of a task in terms of other primitive tasks.

Interoperability: It should 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.

3. Approach

The 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 behaviour of the workflow application, even in the presence of failures. Fault tolerance requirements of applications have been split into the requirements at the application level itself and at the system level (execution environment). The scripting language provides notations and structures for meeting modularity and application level fault-tolerance requirements, whereas the execution environment is responsible for meeting system level fault tolerance.

Meeting interoperability and dynamic reconfiguration requirements are also the responsibility of the execution environment. We first describe the main features of the scripting language and the execution environment and then discuss how the four requirements have been met.

Language Features: The language is based around objects (instances of classes) and tasks (instances of task classes). A task is a unit of computation that requires specified input objects and produces specified output objects. Task instances manipulate references to input and output objects. An important feature of our scripting language is that both objects and tasks are members of classes, and their implementations are external to the script. The implementation of these tasks are specified in an abstract manner which allows the binding to specific implementations to be done at run time; this opens up a way of introducing online upgrade of an application without having to change the corresponding workflow script.

The structure of a task is defined by a taskclass construct specifying a task's inputs and outcomes. The task construct is used to define an instance of a given taskclass. It is also possible to create a *compound task* instance of a taskclass; this enables a programmer to specify the internal composition of the task in terms of other tasks. We provide a graphical as well as a textual programming environment. A graphical representation of a task is given in fig. 2. It depicts a task (called Task) that has two input sets (inputSet1 and inputSet2) with respectively two and one input object references (inputObject1, ..., inputObject3). One of the input sets must have all of its input object references present (input dependencies satisfied) before the task can start. If input dependencies of more than one input set have been satisfied then the input object references from one of these sets (chosen deterministically) are used. A task terminates in one of the named output states (called *outcomes*). Each outcome of a task is associated with a distinct set of output object references, which can be used as input objects by subsequent tasks. This task has two named outcomes: outcome1 and outcome2 with respectively one and two output object references.

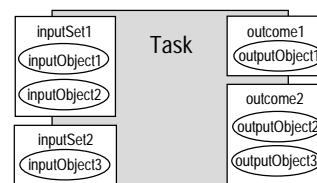


Fig. 2: A task

The language provides a uniform way of specifying atomic and non-atomic tasks. Fig. 3 depicts the state transitions of a non-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. An abort outcome indicates that no changes to the application have been performed. If the input dependencies are met, the task enters the execution state. During execution, a task may produce intermediate outputs called *mark* outputs (e.g., Mark1 and Mark2); the task terminates in one of the named *outcome* states. The state transitions of an atomic task are similar, except that no mark outputs are permitted (an atomic task can produce outputs only after it commits). An atomic task either commits or aborts. Committed terminations map onto specific non-abort outcomes of the task specification. In addition repeat outputs (e.g., Repeat1), could cause the task to re-enter the execution state.

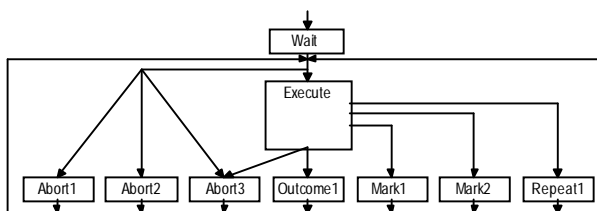


Fig. 3: Task transitions

Execution Environment: The execution environment provides facilities to enable sets of inter-related tasks forming an application to be carried out and supervised in a dependable manner. Further, the environment provides transactional support for the execution of atomic tasks. We have selected the *transactional workflow* approach for coordinating task executions as it provides a natural way of exploiting distributed object and middleware technologies [3,4,5,6].

We will now discuss how the four requirements have been met, starting with modularity.

Modularity: Modularity is ensured through the compound task structure, which provides a flexible way of composing an application out of other applications. For a given task class several tasks and compound tasks can be defined, all providing the same functional behaviour, but with differing non-functional properties; selection of a particular instance is performed at run time. Locality of modifications to the structure of an application is ensured because dependencies within a script are unidirectional, that is only the task which is using outputs and notifications from other tasks is required to declare such dependencies; this means that “up stream” tasks have no knowledge of “down stream” tasks.

Fault-tolerance: We describe how the provision of fault tolerance has been split between application level and system level:

- *Application level:* Notations have been provided to express that a task can acquire a given input from more than one source, and can have alternative input

requirements. This is the principal way of introducing redundant data sources for a task and for a task to control input selection. As fig. 2 shows, a task can terminate in one of several outcomes, producing distinct output objects. This is the principal way of dealing with run time exceptions that could prevent a task from providing ‘normal’ outcome. Finally, the *compoundtask* construct enables an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within a compound task to deal with a variety of exceptional situations.

- *System level:* The workflow management system records inter-task dependencies in persistent shared objects and uses atomic transactions to implement 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).

Dynamic Reconfiguration: 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 fig. 1, assume that it is necessary to add another task t5 with dependencies from t2 and t4. Our system will permit modifications of the relevant shared objects that store inter-task dependencies to reflect new changes, this is described in more detail in [7].

Interoperability: 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. Thus our execution environment is open and fully distributed. The overall architecture of workflow system is shown in fig. 4, and described in [1,2,3]. 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. 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 [8]. 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 these administrative applications has been provided by making use of Java applets which can be loaded and run by any Java capable Web browser.

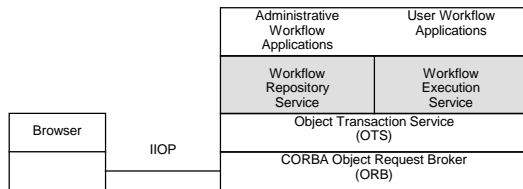


Fig. 4: Workflow management system structure.

Workflow Repository Service: The repository service stores workflow scripts (schema) and provides operations for initializing, modifying and inspecting scripts. A script is represented in terms of tasks, compound tasks and dependencies.

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.

4. The Scripting language

In this section, we will describe the syntax and semantics of the language constructs which we have used within our workflow scripting language. We will describe how object and task classes can be specified along with how tasks instances are declared and how an implementation can be specified which is composed out of existing tasks.

4.1 Object classes

All objects within a workflow script are associated to classes, this allows the checking of the types of objects references which are passed between tasks. To introduce the name of a class of objects into a script the class construct is used (e.g., a class *Account* will be defined as: class *Account*). The member operations and variables of a class are not required to be specified, just the name of the class. The workflow script only controls and co-ordinates the passing of object references between tasks, it does not make use of member variables or operations of the object references.

4.2 Task classes

A task class declaration is introduced into a script using the `taskclass` construct, containing a list of the inputs and outputs of the task class. An example of the construct

is given below, in which a task class *PaymentAuthorisation* is defined:

```
taskclass PaymentAuthorisation
{
  inputs
  {
    ...
  };
  outputs
  {
    ...
  }
}
```

Inputs. A task class can be specified to have multiple input sets. 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). An input set is a list of input objects references and their associated classes. The task described below has one input set *main* with two input objects: *item* and *account*, respectively of classes *Item* and *Account*.

```
inputs
{
  input main
  {
    item of class Item;
    account of class Account
  }
}
```

Outputs. A task class can be specified to have multiple outcomes (or outputs), each of them having an associated set of named output object references. An output will belong to one of four types: *outcome*, *abort outcome*, *repeat outcome* and *mark* (see also fig. 2).

- *outcome*: this output type indicates the final output that this task could produce.
- *abort outcome*: this output type indicates that the task has terminated without producing any side effects. It also specifies that the task is atomic.
- *repeat outcome*: this output type indicates that this output should be used to restart the task. Object references of this output are not usable by any other tasks as object input.
- *mark*: this output type indicates that this output may be produced once during the execution of a task. Mark outputs provide an ‘early release’ mechanism for task results. A task which produces a *mark* output can’t subsequently produce an *abort outcome* (as the task has already produced results).

In the example below the output called *dispatchCompleted* of output type *outcome* has one output object reference *dispatchNote* which is of class *DispatchNote*.

```
outcome dispatchCompleted
{
  dispatchNote of class DispatchNote
}
```

4.3 Task

To create a task instance of a task class, it is necessary to specify all the input sources and the details of the implementation. In the example below a task instance called *paymentCapture* is being declared, of the task class *PaymentCapture*. Using the construct implementation, some run-time information on the task is introduced as a set of couple (keyword value). Some possible keywords are code, location, agent, deadline, priority... The implementation of the task instance has been specified using the keyword code to be *SETPaymentCapture*.

```
task paymentCapture of taskclass PaymentCapture
{
  implementation { "code" is "SETPaymentCapture";
  inputs { ... }
}
```

The name of the implementation can refer to either the code itself (executable), or some script (this allows application scripts to be used as the implementation of a task instance). Next we describe how input sources are specified.

Input selection (dataflow dependencies). For each input object reference required by a task instance, a set of alternative sources can be specified. The order of the alternatives is significant; in the event that multiple alternatives are available the first available alternative in the list will be used. An input can be obtained from two different sources: from the output of another task instance, or from an input to another task instance.

In the example below a task instance *t1* is specifying that its input object reference *i1* can be satisfied by any of: task *t2*'s input object *i3* from the input set *main*, task *t3*'s output object *o1* if *t3*'s outcome is *oc1* or task *t3*'s output object *o2* if *t3*'s outcome is *oc2*. Input object reference *i2* can only be satisfied by *t4*'s output object *o1* if its outcome is *oc1*.

```
task t1 of taskclass tc1
{
  inputs
  {
    ...
    input main
    {
      inputobject i1 from
      {
        i3 of task t2 if input main;
        o1 of task t3 if output oc1;
        o2 of task t3 if output oc2
      };
      inputobject i2 from
      {
        o1 of task t4 if output oc1
      }
    }
  }
}
```

Notification (temporal dependencies). Each notification dependency takes the form:

```
notification from { ... }
```

As before, a notification dependency can have a number of alternate sources. In the example below a task

instance *t1* is specifying that it can't be started until either *t2* or *t3* has produced outcome *oc1* and either *t2* or *t4* has produced outcome *oc2*.

```
task t1 of taskclass tc1
{
  inputs
  {
    input main
    {
      notification from
      {
        task t2 if output oc1;
        task t3 if output oc1
      };
      notification from
      {
        task t2 if output oc2;
        task t4 if output oc2
      }
    }
  }
}
```

4.4 Compound tasks

Compound tasks can be used to specify task implementations which can be used within other scripts or as a structuring device to compose a complex task out of other tasks.

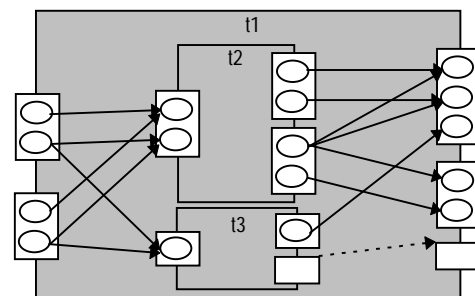


Fig. 5: A compound task

An instance of a compound task which is being used to form a complex task, must specify its input alternatives in the same way that a task instance does; then constituent task instances are specified. Unlike task instances, a compound task instance must provide a mapping between the object references used/produced by its constituent tasks and its outputs. Fig. 5 depicts a compound task *t1* that is having two constituent tasks *t2* and *t3*. The skeletal structure of compound task *t1* is given below, and shows how *t2* and *t3* are introduced.

```
compoundtask t1 of taskclass tc1
{
  inputs
  {
    input main { ... };
    input alternative {...}
  };
  task t2 of taskclass tc2 { ... };
  task t3 of taskclass tc3 { ... };
  ...
  outputs { ... }
}
```

For each output produced by the compound task instance, a set of alternative sources (coming from constituent tasks) can be specified. The list is contained

within the construct given below, *outputname* being the name of the output object reference whose alternative sources are being specified.

```
outputobject outputname from { ... }
```

The notation for specifying the alternatives is the same as that described for input alternative selections in section 4.3.1.

For a compound task which is to be used to specify a task implementation, the inputs are not specified. The inputs will be specified by the task instance that is naming the compound task as its code.

4.5. Task templates

The construct *tasktemplate* has been added to enable the parametrisation of task definitions. It specifies the parameters expected and follows the same rules of the task and *compoundtask* constructs. The following task template takes two parameters *param1* and *param2*., that are used to specify as argument from which tasks input object references are coming.

```
tasktemplate task tasktemplename of taskclass taskclassname
{
  parameters
  {
    param1;
    param2
  };
  implementation {...};
  inputs
  {
    input main
    {
      i1 of task param1 if output success;
      i2 of task param2 if input main
    }
  }
}
```

A task *taskname* can then be instantiated:

```
taskname of tasktemplate tasktemplename(argument1, argument2)
```

5. Examples

Three example applications will be presented to demonstrate the features of our scripting language. The first example is from the telecommunications domain, concerned with network management. The remaining two are from the area of electronic commerce.

5.1 Network Management

One important aspect of network management is concerned with maintaining the quality of services in the presence of failures. If component failures are detected, it is necessary to analyse the impact of the failures on the services to customers and take any corrective actions. Let us assume that this is achieved by monitoring an alarm source object which provides access to the alarms triggered by problems in the network, such as loss of a link or bandwidth degradation. The alarms are correlated by an alarm correlator application to deduce the nature of the fault which is causing the problems. Its output is then

analysed to find which services are or will be impacted by the fault. This information is then used by a service impact resolution application that determines which (if any) of the impacted services on the network are to be restructured to reduce the impact of the fault; this may involve disconnecting less profitable services or rescheduling some services, etc.

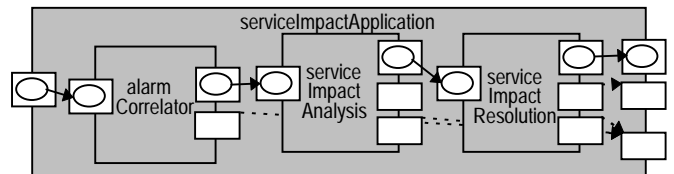


Fig. 6: Structure of service impact application

Our approach to modelling this application consists of a compound task *serviceImpactApplication* which contains three constituent tasks: *alarmCorrelator*, *serviceImpactAnalysis* and *serviceImpactResolution*. The resulting application structure is given in fig. 6. In real life, this would be a widely distributed application, and if executed within the execution environment provided by our workflow system, will benefit from the reliable dependency and notification services.

For this application four classes of objects are used: *AlarmsSource*, *FaultReport*, *ServiceImpactReports* and *ResolutionReport*, and specified in the following manner.

```
class AlarmsSource;
class FaultReport;
class ServiceImpactReports;
class ResolutionReport;
```

For each of the tasks instance within the application a task class must be defined. The application is represented by the compound task *serviceImpactApplication* which is of task class *ServiceImpactApplication*, and is defined below.

```
taskclass ServiceImpactApplication
{
  inputs
  {
    input main
    {
      alarmsSource of class AlarmsSource
    }
  };
  outputs
  {
    outcome resolved
    {
      resolutionReport of class ResolutionReport
    };
    outcome notResolved
    {
    };
    outcome serviceImpactApplicationFailure
    {
    }
  }
};
```

The compound task which implements the service impact application is specified below; it shows the three constituent task instances and their interdependencies. It also shows the circumstances under which the outputs of the compound task will be produced.

```

compoundtask serviceImpactApplication of taskclass ServiceImpactApplication
{
  task alarmCorrelator of taskclass AlarmCorrelator
  {
    implementation { "code" is "refAlarmCorrelator";
    inputs
    {
      input main
      {
        inputobject alarmSource from
        {
          alarmsSource of task serviceImpactApplication if input main
        }
      }
    }
  };
  task serviceImpactAnalysis of taskclass ServiceImpactAnalysis
  {
    implementation { "code" is "refServiceImpactAnalysis" };
    inputs
    {
      input main
      {
        inputobject faultReport from
        {
          faultReport of task alarmCorrelator if output foundFault
        }
      }
    }
  };
  task serviceImpactResolution of taskclass ServiceImpactResolution
  {
    implementation { "code" is "refServiceImpactResolution" };
    inputs
    {
      input main
      {
        inputobject serviceImpactReports from
        {
          serviceImpactReports of task serviceImpactAnalysis
        }
      }
    }
    outputs
    {
      outcome resolved
      {
        outputobject resolutionReport from
        {
          resolutionReport of task serviceImpactResolution if output foundResolution
        }
      };
      outcome notResolved
      {
        notification from
        {
          task serviceImpactResolution if output foundNoResolution
        }
      };
      outcome serviceImpactApplicationFailure
      {
        notification from
        {
          task alarmCorrelator if output alarmCorrelatorFailure;
          task serviceImpactAnalysis if output serviceImpactAnalysisFailure;
          task serviceImpactResolution if output serviceImpactResolutionFailure
        }
      }
    }
  };
}
}

```

The behaviour of the implementation of the service impact application can be configured by choosing appropriate implementations of the constituent tasks instances. This will be done by binding the names *refAlarmCorrelator*, *refServiceImpactAnalysis* and *refServiceImpactResolution* to suitable implementations

at instantiation time. This means that the compound task *serviceImpactApplication* can be used as a template application which can be instantiated to detect, analyse and resolve many different scenarios by providing sets of appropriate implementations for constituent tasks.

5.2 Electronic order processing

This application involves the processing of a customer's order. It has been modelled as a compound task *processOrderApplication* which contains four constituent task instances: *paymentAuthorisation*, *checkStock*, *dispatch* and *paymentCapture*. The relationship between the task instances is shown in fig. 7.

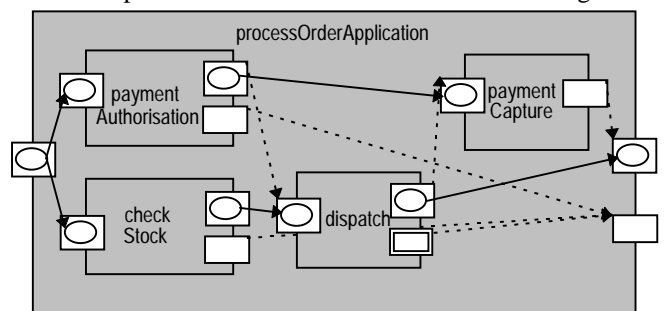


Fig. 7: Structure of process order application.

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.

```

taskclass Dispatch
{
  inputs
  {
    input main
    {
      order of class Order
    }
  };
  outputs
  {
    outcome dispatchCompleted
    {
      dispatch of class DispatchNote
    };
    abort outcome dispatchFailed
  };
}

```

Note that the *dispatchFailed* output is of type abort outcome (represented by a box with a double line border), meaning that if an instance of the task class *Dispatch* produces this output then the instance will not have caused any effects (this can be ensured by binding the instance to a transactional implementation).

The script below captures the overall structure.

```

compoundtask processOrderApplication of taskclass ProcessOrderApplication
{
  task paymentAuthorisation of taskclass PaymentAuthorisation
  {
    implementation { "code" is "refPaymentAuthorisation" };
    inputs
    {
      input main

```

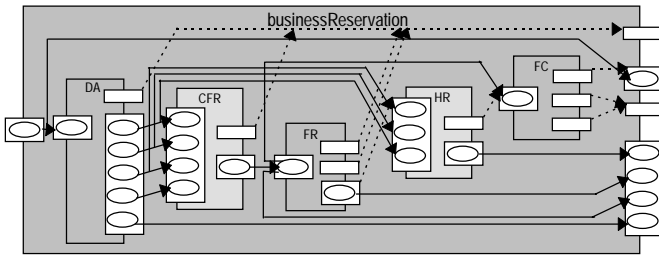



Fig. 9: Structure of businessReservation compound task.

This compound task is then detailed in fig. 9: an input object called *user* is used as input for the *dataAcquisition* (DA) task, which outputs the dates of the trip as well as the location and the maximal cost of the flight to the *checkFlightReservation* task (CFR) which is itself a compound task which internally starts a dynamic task containing several parallel requests to try to find a flight fulfilling the requested conditions. If it succeeds, a task *flightReservation* (FR) is then started which reserves that flight, after which it transmits the resulting cost of the flight to the external world via a mark outcome (represented by a box with dotted border) and starts a task *hotelReservation* (HR) which either fails, triggering the execution of the compensating task *flightCancellation* (FC). Should one of the first three tasks described fail or abort, the whole compound task is aborted.

```
compoundtask tripReservation of taskclass TripReservation
{
  ...
  compoundtask businessReservation of taskclass BusinessReservation
  {
    inputs
    {
      input main
      {
        inputobject user from
        {
          user of task tripReservation if input main;
          user of task businessReservation if output retry
        }
      }
    }
  }
};
```

We can see in this part of code that the task *businessReservation* can accept two inputs: either an input coming from its embedding task *tripReservation* or an input from its repeat outcome.

```
task flightReservation of taskclass FlightReservation
{
  ...
};
task flightCancellation of taskclass FlightCancellation
{
  ...
  inputs
  {
    input main
    {
      notification from
      {
        task hotelReservation if output failed
      };
    };
    inputobject plane from
    {
      plane of task flightReservation
    }
  }
};
```

```
}
};
```

This task is an example of a compensation task. If the *flightReservation* succeeds but the task *hotelReservation* fails the task *flightCancellation* is used to compensate *flightReservation*.

```
compoundtask tripReservation of taskclass TripReservation
{
  outputs
  {
    ...
    mark toPay
    {
      outputobject cost from
      {
        cost of task businessReservation if output success
      }
    }
  }
};
```

The script above is an example of a mark output; during the execution of the *tripReservation* the output *toPay* can be produced.

6. Related work

Scripting languages have long been used for composing applications out of existing applications. Tcl or Perl are well known recent examples of general purpose languages. Our work is in the relatively new area of composing reliable distributed applications. Here the state of art is represented by the work in the transactional workflow area.

In the workflow community, several techniques from other domain of computing science have been used as the basis to languages for specifying task interactions. For instance, workflow scripts can be rule based, specifying actions to be taken in the event of a given condition becoming true. The METEOR project [6] has developed such a language. Some other projects have chosen to base their languages on an extension of Petri nets which enable them to model the control flow using tokens [9]. Projects from the database community use built-in SQL statements [10].

Our approach to the design of the language has been to specify an application's structure rather than list the events - conditions - actions that make up the application. Our approach is closer to those of architecture description languages (ADLs). ADL-based specifications are intended to describe the structure of the components of a software system and their inter-relationships [11,12,13,14,15]. 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 service 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. Currently available ADLs however do not capture the *computation structure* of an application. We claim that this has to be in terms of the tasks (activities) the application performs. This requires describing the temporal structure of an application. 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 the `compoundtask` construct that enables an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within a compound task to deal with a variety of exceptional situations. Whilst our language is ideally suited to expressing these requirements, it cannot specify the details of mappings of tasks onto the software components: this would be the function of an ADL. The combination an ADL 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.

7. Concluding Remarks

We have described a scripting language that together with the supporting execution environment provides fault-tolerance, dynamic reconfiguration, modularity, interoperability. The language has been designed to support the construction of reliable, long-lived applications, which can also be used as components within other larger applications. The language and the execution environment have been implemented. Future work will include building realistic applications.

At present our language does not support sub-typing of object or task; we are at present investigating what advantages their addition would bring. For example, the addition of sub-typing of object would be straightforward and would allow the specification of "building block" tasks which operated on standard super-types.

8. 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). 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, John Warne, Dave Stringer is gratefully acknowledged.

References

- [1] F. Ranno, S.M. Wheeler and S.K. Shrivastava, "A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications", Conf. on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany, September 97.
- [2] S.M. Wheeler, F. Ranno and S.K. Shrivastava, "A CORBA Compliant Transactional Workflow System for Internet Applications," to be published.
- [3] Nortel Technology and Newcastle University submission to OMG Business Object Domain Task Force (BODTF), OMG document bom/98-03-01 (also available at <http://arjuna.ncl.ac.uk/WorkflowSystem/98-03-01.ps>).
- [4] J.P. Warne, "Flexible transaction framework for dependable workflows", ANSA Report No. 1217, 1995.
- [5] M. Rusinkiewicz and A. Sheth, "Specification and execution of transactional workflows", Modern database systems (W. Kim, ed.), pp. 592-620, ACM Press, 1995.
- [6] D. Georgakopoulos, M. Hornick and A. Sheth, "An overview of workflow management: from process modelling to workflow automation infrastructure", International Journal on distributed and parallel databases, 3(2), pp. 119-153, April 1995.
- [7] S.K. Shrivastava and S.M. Wheeler, "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," The 4th International Conference on Configurable Distributed Systems (CDS'98), Annapolis, Maryland, USA, May 4-6, 1998.
- [8] 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 95.
- [9] A. Schill, C. Mittasch, "Workflow management systems on top of the OSF DCE and OMG CORBA", Distributed System Engineering Journal, December 1996.
- [10] L. Russell, O. Wolfson, C. Yu, "Information flow in the DAMA project beyond database managers: information flow managers", Distributed System Engineering Journal, December 1996.
- [11] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying Distributed Software Architectures", Proceedings of the 5th European Software Engineering Conference, Barcelona, September 1995.
- [12] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", SIGSOFT 96, ACM Software Engineering Notes, Vol 21 No. 6, November 1996.
- [13] J. Bishop and R. Faria, "Connectors in Configuration Programming Languages: are They Necessary?", Proc. of 3rd IEEE International Conference on Configurable Distributed Systems, Annapolis, pp. 11-18, May 1996.
- [14] L. Bellissard, S. Ben Atallah, F. Boyer, M. Riveill, "Distributed Application Configuration", Proc. of 16th IEEE International Conference on Distributed Computing Systems, Hong-Kong, pp. 579-585, May 1996.
- [15] V. Issarny and C. Bidan, "Aster: A Framework for Sound Customisation of Distributed Runtime Systems", Proceedings of 16th IEEE International Conference on Distributed Computing Systems, Hong-Kong, May 1996.