A Language and Toolkit for the Specification, Execution and Monitoring of Dependable Distributed Applications

Ph.D. Thesis

by Frédéric Ranno

The University of Newcastle upon Tyne

Department of Computing Science

1998

Abstract

This thesis addresses the problem of specifying the composition of distributed applications out of existing applications, possibly legacy ones. With the automation of business processes on the increase, more and more applications of this kind are being constructed. The resulting applications can be quite complex, usually long-lived and are executed in a heterogeneous environment. In a distributed environment, long-lived activities need support for fault tolerance and dynamic reconfiguration. Indeed, it is likely that the environment where they are run will change (nodes may fail, services may be moved elsewhere or withdrawn) during their execution and the specification will have to be modified. There is also a need for modularity, scalability and openness. However, most of the existing systems only consider part of these requirements. A new area of research, called workflow management has been trying to address these issues.

This work first looks at what needs to be addressed to support the specification and execution of these new applications in a heterogeneous, distributed environment. A co-ordination language (scripting language) is developed that fulfils the requirements of specifying the composition and inter-dependencies of distributed applications with the properties of dynamic reconfiguration, fault tolerance, modularity, scalability and openness. The architecture of the overall workflow system and its implementation are then presented. The system has been implemented as a set of CORBA services and the execution environment is built using a transactional workflow management system. Next, the thesis describes the design of a toolkit to specify, execute and monitor distributed applications. The design of the co-ordination language and the toolkit represents the main contribution of the thesis.

Acknowledgements

I am very grateful for all the support and encouragement that I have received during my research. I would like in particular to thank Prof. Santosh Shrivastava for helping me to decide which research topic to pursue, to proof-read this thesis and other papers, for his supervision, help and suggestions regarding this project. I also would like to thank Dr. Stuart Wheater for his help and suggestions as well as for the development of the Workflow Engine for which this toolkit has been designed.

Many thanks also to the other members of the Arjuna group for building the platform that was used to develop and validate this project, especially to Dr. Mark Little who has developed the OTS version of Arjuna. I would also like to address some special thanks to Ms Shirley Craig our departmental librarian, as well as to the staff of the Robinson Library.

I am grateful to Nortel Telecom and in particular to John Warne, Harold Toze, Samantha Merrion, Dave Stringer and A. J. Tunnicliffe to provide us with a real example of a workflow application as well as their ORB.

I would also like to address some special thanks to my family and Sarah for their support during my Ph.D.

My research has been jointly funded by the UK Engineering and Physical Sciences Research Council (EPSRC award 94315028) and by CaberNet, the ESPRIT network of excellence in distributed computing systems architectures (European HCM Research Fellowship contract ERBCHBGCT93). Northern Telecom (Nortel) and the European LTR project C3DS (project no. 24962) also sponsor the workflow project.

Contents

List of figures

Chapter 1

Introduction

1.1- Motivation

In a competitive market driven environment, enterprises must improve their productivity and efficiency. Automation of business processes is therefore considered important. In this thesis, a business process is defined as a set of organised activities aiming at reaching a common business goal (for example, the activities needing to be performed by "MyOwnComputers Ltd" to build a new computer given its parts form a business process). More and more such processes are being automated. By automating their business processes, companies try to gain efficiency and effectiveness.

In the last few years, global networks, such as the Internet, have grown considerably in scale mainly due to the advances in telecommunication, and this has made it possible to access information within seconds. More and more business processes are growing in complexity, and are also becoming distributed. Automating the processes can also speed up access to the information. For instance, you can keep the details of your products and their availability in a database and have an instantaneous idea of the state of your stock.

Moreover, the information access has become worldwide and as a result processes are now usually crossing enterprise-level boundaries. A typical example of such a process is ticket bookings via a travel agency: the travel agent starts a booking process that involves his company as well as some air-lines, railways companies, ferry companies, bus companies... Most travel agencies now directly buy the tickets via computers from their providers. Their system acts as a client that contacts some servers where they can check what is available, when and at what price. If the customer is happy with what was found, then the proposed travel arrangements can be booked.

Other examples of applications being automated can be found in the field of electronic commerce. Electronic commerce is a relatively new concept that deals with buying and selling goods on the net. Typically that involves browsing an electronic catalogue, put the articles you want to buy in an electronic basket, get the bill, pay it and then wait for the delivery. The most famous example of a company trading by electronic commerce is amazon.com [3], the "electronic" bookstore, from which you can order books at American prices from all over the world by using your computer connected to the Internet. Another example is the emergence of virtual shopping malls, such as the BarclaySquare [5] created by Barclays, where customers can buy from companies such as Argos, Interflora or Thomas Cook. The customer selects the goods that he wants to buy and then clicks on a retailer's cash register to pay for the chosen items. The payment is then carried out thanks to electronic money stored in an electronic wallet linked to a BarclayCard account. The area of electronic commerce is growing fast, and is estimated to be worth already up to $15 billion and is expected to reach $200 billion in the year 2000 [62].

A variety of computer systems for automating the task of scheduling and executing applications have been developed. These systems are known as workflow management systems and the applications are called workflows. As we can see from the previous examples, workflows can usually be divided into smaller units of work carried out by participants that can be among others human beings, their agents (programs acting on their behalf) or even computer programs. The participants have to collaborate to reach a common aim, the achievement of the global process. This collaboration is usually carried out by exchanging data and by ensuring that the dependencies between units of work are respected. Such dependencies can usually be divided in two different types: temporal dependencies (e.g. a unit of work has to be executed before another one) and data-flow dependencies (e.g. a unit of work needs some data coming from another unit of work before starting). Similarly an increasing number of applications are being built as a set of existing applications collaborating among them. The resulting applications are usually rather complex and have a lot of dependencies between their constituents. Another problem inherent to this new kind of applications is that the execution of such applications may span long periods of time. Indeed, it may include long periods of inactivity, usually due to component tasks requiring interactions with human users. Furthermore, these applications are not only long-lived; they are also usually executed in a heterogeneous environment, across company-level boundaries.

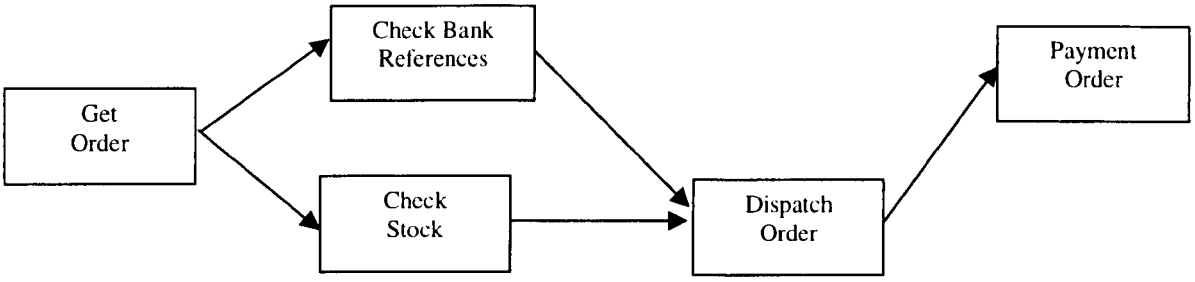


Figure 1.1: Example of a workflow application

In figure 1.1, an example of workflow application in the area of commerce is shown. It depicts the process of ordering, dispatching and paying an order. Initially, customer orders some items (unit of work *get order*), then the stock as well as the bank references of this customer are checked in parallel (units of work *check stock* and *check bank reference*). If both checks are successful, the order is then dispatched (unit of work *dispatch order*), which triggers the payment for the order (unit of work *payment order*).

In a distributed environment, long-lived activities do need support for fault tolerance and dynamic reconfiguration. On one hand, the risk of encountering faults obviously increases proportionally to the duration of the application. On the other hand, it is likely that the environment in which they are run will change (nodes may fail, services may be moved elsewhere or withdrawn) during their execution and the specification will have to be modified. As a result, it is inevitable that workflows will need some support for fault tolerance and dynamic reconfiguration. Furthermore, to be trusted workflow need to be reliable and ensure correctness of the process.

As a result, an important problem for companies looking to automate their business processes is to have a system to help them to specify, execute and monitor the resulting application in an efficient manner. Moreover, this system should incorporate some tools for the management of these processes, including monitoring, configuration and consistency checking tools.

A lot of systems have been proposed in an effort to meet these requirements. The solutions have ranged from email based system to extended database systems. In an email-based system [15], some extra information is added to the email to specify the routing of the message. This kind of system can be quite useful in an office environment where a form has to go through a particular route (corresponding to a sequence of processing steps) to be accepted. Such a system is however quite limited in scope as it only provides routing information. In particular

there is no real support for inter-tasks dependencies. In database systems, different extensions have been proposed to extend their capabilities. The first of these extensions was to add some Event-Condition-Actions rules as objects in the database, and managed by the database management system. When an event occurs (such as an object becoming available or the completion of an activity), its associated condition is evaluated and if true, the associated action is scheduled for execution. The resulting systems are known as active database management systems. They allow the creation of multi-steps applications, where each step is a transaction, and where access to data and synchronisation among transactions are managed by the system. Another interesting approach is represented by the transactional workflow systems that are closely related to work on extending transaction models. The idea behind extended transaction models is to relax part of the ACID (Atomicity, Consistency, Isolation and Durability) properties of conventional transactions. ACID transactions are well suited to provide fault tolerance and consistency for short-lived interactions with shared objects; in contrast, extended transactions are intended to provide similar functionality for long-lived applications. A transactional workflow system provides facilities to define an application as a set of co-ordinated transactions. By developing an underlying system for supporting flexible transaction facilities, an efficient, reliable workflow management system can be developed. Such a model is described in [76]. This system could be based on one of the extended transactions model based on the relaxation of the Isolation property of the Atomic Transaction.

Workflow management Systems have been defined by the Workflow Management Coalition * as "a system which provides procedural automation of a business process by management of the sequence of work activities and the invocation of appropriate human and/or IT resources associated with the various activity steps" [78]. Often seen as "collections of tasks organised to accomplish some business processes" [24], workflows are intended to enable quick acquisition of services and resources. Workflow management systems are good management tools as they can co-ordinate and control business processes.

A workflow management system can be divided in several components [29] as depicted in figure 1.2. The main division is between the build-time and the run-time components. The build time components allow you to specify the workflow. They can usually be divided into two sets:

---

* Workflow Management Coalition: industrial consortium set up to create an industry standard for Workflow Management Systems.

the conceptual model and the tools provided to facilitate the construction of workflow specifications. The conceptual model itself is based on a workflow language, specified by its syntax and semantics. The Workflow community [13] has been working on languages to specify workflow applications, while the Software Design community [39] [35] [6] has been working on Architecture Description languages. The other build time components, the build time tools allow to check your specifications, simulate the execution of these specifications and animate them.

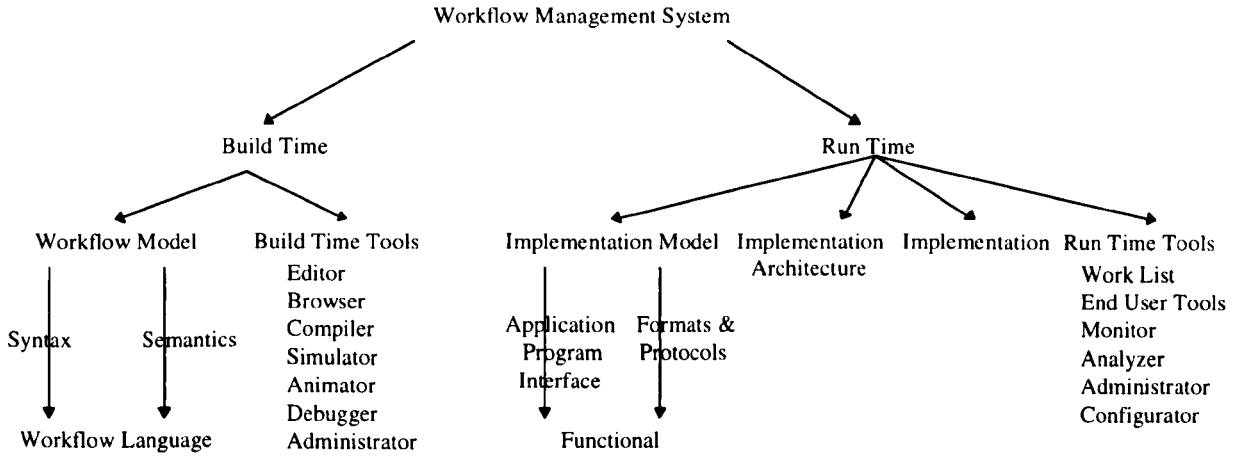


Figure 1.2: The components of a workflow management system

The run-time part of a workflow management system is itself sub-divided into four components: the implementation model, the implementation architecture, the implementation and the run-time tools. The implementation model is the conceptual basis for the run-time part of the workflow system. It defines the functional components (tasks...) and the protocols between them by specifying the application program interfaces and the formats and protocols. The implementation architecture is the enactment of the functional components or in other words how the functional components are mapped to active elements of the underlying system. The implementation is the instantiation of this architecture. In order to support this implementation, some run-time tools have to be provided. For instance, monitoring tools are needed to find out what is going on and control the progress of the workflow. Analysis tools can be provided to find out the history of a workflow execution. Work lists can also be provided when you have human participants to let them know what they have to do. Given that the requirements can change during the execution, some administration and configuration tools have to be provided to handle those changes.

1.2- Objectives

There is no real consensus on what a workflow is or on what kind of features a workflow management system has to provide. A lot of different definitions [20] have been proposed for workflows ranging from a business process, a software automating such a process, its specification or a software that supports the co-ordination of people implementing such a process. In this thesis, we define a workflow as *a set of tasks arranged to form (complex) organisational functions,* where a task is defined as *an application specific unit of work.* A workflow management system is defined as the *software system that can be used to specify and co-ordinate the execution and monitor of organisation functions.*

The aim of this thesis is to provide an efficient build and run-time environment to specify, execute and monitor reliable workflow applications constructed out of other existing applications and workflows. It should allow the co-ordination of inter-dependent distributed applications with the properties of fault tolerance, dynamic reconfiguration, modularity, scalability and openness. Each of these requirements will now be further described starting with co-ordination.

Co-ordination

Several types of approaches have been presented to specify the interactions between component tasks. An active field of research has been on rule-based approaches, where the co-ordination is expressed by Event-Condition-Action rules. In this model, the specified actions are triggered by some events under certain conditions. For instance, in the electronic commerce field, it is likely that the company selling the goods will want to be paid before delivering the goods that a customer has ordered. This would be modelled by specifying that the task "delivering goods" can only begin when the task "pay for goods ordered" has been successfully completed. Approaches based on Petri-nets, extended SQL queries or email-based have also been proposed. Another important approach is the Architecture Description Language (ADL) based specifications which allow the description of the structure of the components of a software system. These components communicate through connectors, and are providing services for the other components. Those services are provided and obtained via ports. However current ADLs do not capture the computation structure (run-time behaviour) of an application. Our goal is to provide a way to capture this computational structure. The language presented in this thesis does that by describing the application as a set of tasks (units of work) that are linked among them by temporal dependencies. These dependencies are of

two different types, notification dependencies specifying the temporal constraints among tasks and input dependencies (data-flow dependencies) specifying the mapping of the inputs and outputs of a task with the data available in the system.

Dependability

Organisations that are automating their processes using workflows must be able to rely upon the workflow management system. Indeed when automating processes, the human responsibility for these processes can be partly discharged to the corresponding workflows. Of course, to be trusted, workflows need to be reliable and ensure correctness of the process. If they are not fault-tolerant, the consequences could be disastrous: just imagine a financial transaction badly completed in the financial field, the consequences can be catastrophic.

Furthermore, systems become bigger, go across organisation boundaries, are run within a distributed environment, and on top of that particular problems arise from co-ordinating applications. Among other, this includes:

- *Network splits:* the network can be partitioned due to hardware faults for instance.
- *Machine failures:* machines can crash, users can reboot them unexpectedly, hardware faults can occur.
- *Machine removal:* machines can be removed from the set of resources available for the workflow. This is becoming more and more an issue with the emerging technology of mobile computing.
- *Services may be withdrawn:* one of the best examples of this problem can be encountered on the Internet. When you make a query on resources using a search tool, it is common to retrieve a lot of links on web resources that have vanished since their registration.
- *Services may be moved:* in order to optimise the access or to provide a better quality of service, services may be moved elsewhere. For instance, as technology evolves, new machines with better performance may be introduced and services moved on them.
- *Tasks may fail:* there are a lot of reasons why tasks may fail including bad programming (bug-free applications and complex applications do not go well together). The resources that are needed by those tasks may not be available.

The system proposed must aim at being able to cope with these problems as automatically as possible. The automation of business processes should indeed increase the quality of service.

The way to recover for the workflow often has to be based on compensation as unrecoverable actions may have taken place. For instance in the electronic commerce field, you may send a bill corresponding to the goods ordered. Once this activity has been completed, it is impossible to "erase" it. You can only compensate it by sending a message invalidating the previous one.

Dynamic reconfiguration

As was stated earlier, the applications considered are likely to be long-lived. As a result, they are likely to find out during their execution that the environment in which they are executing has changed or the user requirements may change. As a result they have to evolve to cope with those changes. It is an accepted fact that it's not viable to abort them and that some kind of support for run-time reconfiguration should be provided to cope with those changes. Our model should allow some additions and removals of component applications and dependencies during run-time without having to stop the application.

Scalability

Workflow technology aims at the integration of applications. As a result, scalability becomes a real issue as the applications modelled grow in size. A scalable Workflow System will be a Workflow that keeps the communication between participants as small as possible and do no rely on any centralised services. Indeed if we were to rely upon such a service, it would introduce some bottlenecks which would not be scalable. A fully distributed system architecture should be adopted. It seems that the distributed Object-Oriented transaction systems can be a good way to cope with this requirement as well as with those of heterogeneity, distribution and flexibility.

Modularity

The specification of the application should be modular. Indeed, changing the specification of a task should have as little impact on the rest of the application as possible. The changes involved should only be made where the task is actually defined. Moreover it should also be possible to provide a way to compose an application out of other applications similarly constructed. The language presented in this thesis achieves this by hiding from the "downstream" tasks the "upstream" ones, or in other words, the tasks supposed to be executed at a certain time have no knowledge what so ever of the tasks depending on them and executing later.

Openness

Nowadays, systems have to be as open as possible and not rely on proprietary components. As a result, the system adopted should deal with component application in the same way irrespectively of the language in which they have been written. The construction of the applications should be made in a uniform manner. This has been achieved in the model presented by using CORBA (see figure 1.3).

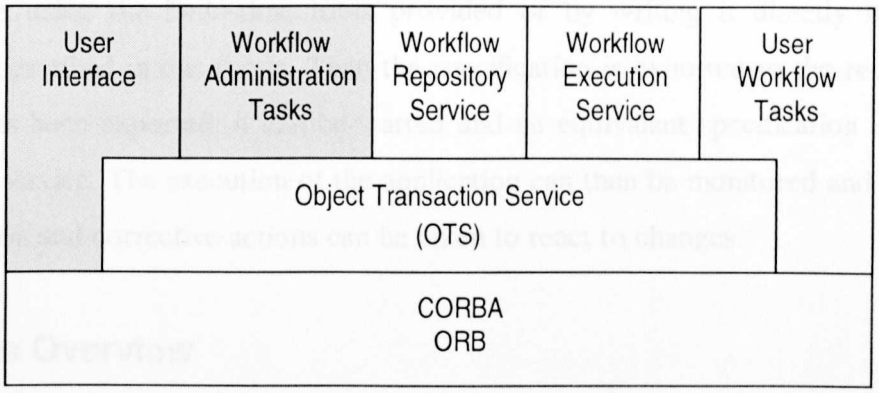


Figure 1.3: Software structure of the toolkit

It should also be possible to use the system regardless of the host from which you are connecting. This has been achieved by using Java to implement the Graphic User Interface (GUI) for the system. It provides remote access to the system from wherever a web browser supporting Java is available.

The software structure of the toolkit is depicted on figure 1.3. The toolkit is divided into five components: the Java user interface, the workflow administration tasks, the workflow repository service, the workflow execution service and the User Workflow Tasks. The User Interface was implemented as a Java applet and provides some tools to specify a workflow, including a graphical editor, some consistency check tools, some animator and simulation tools for build time. It also provides some monitoring and analysing tools for run-time as well as some tools allowing the dynamic reconfiguration of the workflow specification or schema. Once specified, the workflow schema is stored in the workflow repository. Using the GUI, a workflow schema can be instantiated and the workflow execution service is used to support the execution of the application. The tasks are mapped to user workflow tasks just before being executed. Dynamic reconfiguration can then be achieved by using administration tasks. Most of the service components within the toolkit are provided via Common Object Request Broker Architecture (CORBA) interfaces defined using the CORBA Interface Definition

Language. The components using those services are CORBA clients, which allow interoperability. The underlying transactional system is the CORBA-compliant object transaction service of Arjuna [55]. The design and implementation of the toolkit have been a team effort. The author is responsible for the work on the co-ordination language and the shaded components in figure 1.3, which form the main part of the thesis.

The usual way of constructing a complex application using the system is to specify the application using the build-time tools provided or by writing it directly in the workflow language described in this thesis. Then the specification is exported to the repository service. Once it has been exported, it can be started and an equivalent specification is created in the execution service. The execution of the application can then be monitored and analysed during its execution and corrective actions can be taken to react to changes.

1.3- Thesis Overview

This work first looks at what needs to be addressed to support the specification, execution and monitoring of reliable workflow applications constructed out of other applications in a heterogeneous, distributed environment, before proposing a language and a toolkit to support these requirements. A new architecture combining the advantages of different existing solutions and aiming at providing a flexible reliable model supporting the new workflow technology will also be presented. The language presented in this thesis is novel in that it provides a simple yet flexible way to specify the temporal structure of complex applications created out of other applications while being ideally suited to incorporate dynamic changes at run-time. It also provides a novel and uniform way to provide support for fault-tolerance to these applications.

The thesis is structured as follows. In chapter 2, we present and discuss the state of the art of the different fields considered by our system, and define the requirements associated to those fields that our language has to fulfil. The architecture of the overall workflow system and its implementation are then presented in chapter 3. The system has been implemented as a set of CORBA services and the execution environment is built using a transactional workflow management system. In chapter 4, we specify our language and present the equivalent graphical notation. Then in chapter 5, we show using examples how workflow applications can be modelled using our system. Finally, in chapters 6 and 7, we describe a toolkit that supports the specification, the execution and the monitoring of workflows as well as describe how to

generate some equivalent specifications based on Petri-nets. External tools for consistency checking can use these specifications. The thesis then finishes with some conclusions and references.

In this thesis, we will adopt the following lexicographical conventions. Whenever a code sample is given, we indicate the language keywords by using bold letters. These examples are written in Courier font. Bold in conjunction with double quotes is used to denotate language keywords, whenever they are included in "normal" text. Important words are emphasised using italic, while names of languages and products are written as upper case words.

Chapter 2

Related work

In this chapter, we will present the state of the art for Workflow Management Systems. Keeping in mind the description of the components of a workflow management system as depicted in figure 1.2, we will start with the presentation of two architectures proposed as models to build workflow management systems. Then, we will present in turn the build-time and the run-time part of the workflow management systems. The build-time part will be first presented including different approaches taken to specify workflow applications, as well as the tools provided to help design the application specification. The main focus for the build time part will be on the languages chosen, as the features supported by the workflow management systems depend mainly on the expressive power of these languages. We will then turn our attention towards the run-time environment with a special focus on what has been done in the transactional workflow field.

2.1- Architectures

In this part, we will introduce major architectures that have been proposed for workflow management systems. We will first describe the proposal of the Workflow Management Coalition, and then we will carry on with a different approach based on flexible transactions proposed by ANSA.

2.1.1 The Workflow management Coalition Architecture

Created in 1993, and consisting of more than 200 members, the Coalition has proposed a framework for the establishment of workflow standards. This framework includes five interfaces for interoperability and standardisation of communication. The aim is to have a common set of interfaces that will allow multiple workflow products to coexist and inter-

operate within a user's environment. In [78], the reader will be able to find some further information on the technical details. Three levels of compatibility (levels A, B and C) with the framework presented have been defined to provide some flexibility to the workflow product manufacturers. For instance, for the read/write interface of workflow process definition [80] (API1 between the workflow engine and the process definition program), the specification is in fact a set of interfaces and the number of interfaces supported gives the level of compatibility with the overall specification. It has to be noticed that a level of compatibility C implies a level of compatibility A, as we have a relation of inclusion between subsequent level of compatibility. In other words, features supported for level A included in features that have been to be supported to get compatibility level B themselves included in what is needed for level C

All workflow systems contain a number of generic components, which interact in a variety of ways. To achieve interoperability between workflow products a standardised set of interfaces and data interchange formats is necessary [79]. Then these interfaces can be used as references when building interoperability scenarios. For instance processes expected to be shared by several users from potentially different organisations using different workflow engine can be specified using a tool of one workflow system and exported afterwards to the others users regardless of the workflow system that they are using. Similarly workflow client applications should be able to receive tasks generated by other workflow systems providing that they follow the standards. The major components and interfaces identified by this model are listed below, and depicted on figure 2.1:

- Reference Model (core component) - Specify a framework for workflow systems, identifying their characteristics, functions and interfaces.
- Process Definition Tools Interface (1) - Define a standard interface between the process definition tools and the workflow engine(s).
- Workflow Client Application Interface (2) - Define standards for the workflow engine to maintain work items which the workflow client presents to the user.
- Invoked Application Interface (3) - A standard interface to allow the workflow engine to invoke a variety of applications. This interface has still to be specified.
- Workflow Interoperability Interface (4) - Definition of a variety of interoperability models and the standards applicable to each
- Administration & Monitoring Tools Interface (5) - Definition of monitoring and control

functions.

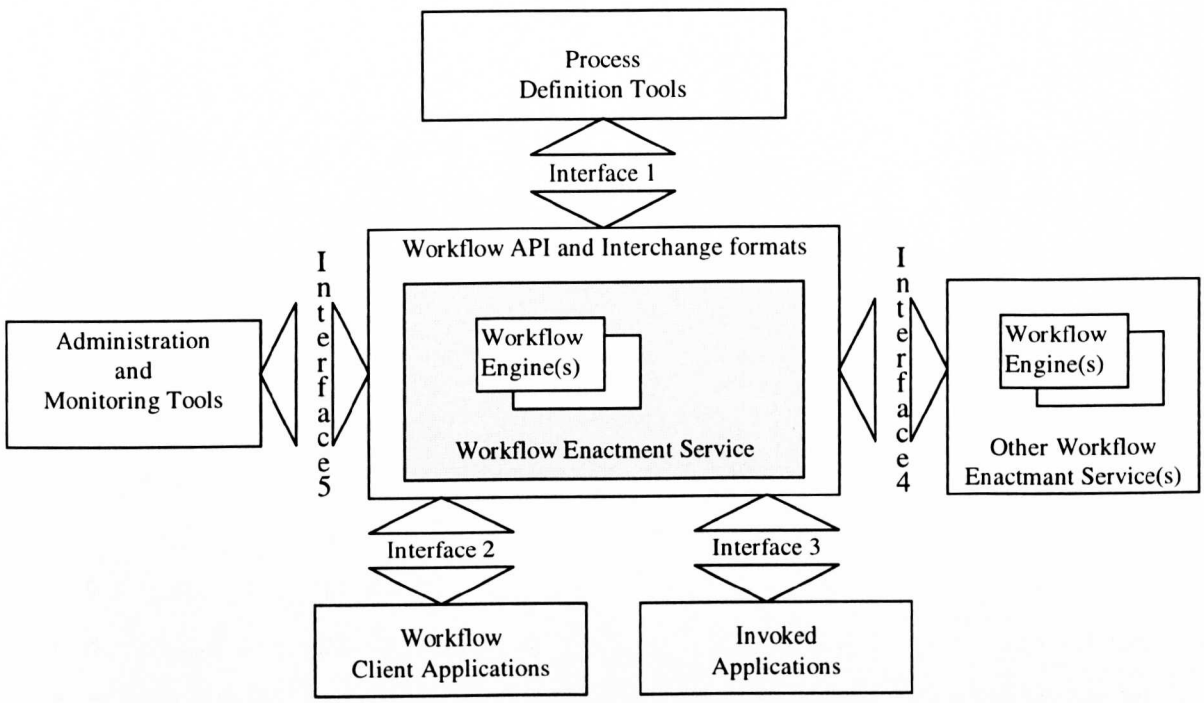


Figure 2.1: Components and interfaces of the WfMC model.

We will now consider in turn the major components in the following sections.

Core component- Workflow Enactment Service

The workflow enactment service provides the run-time environment in which one or more workflow processes are executed. This may involve more than one actual workflow engine. The enactment service is distinct from the application and end-user tools, which are used to process items of work. A wide range of industry standard or application specific tools can therefore be integrated with the workflow enactment service to provide a complete workflow management system. This integration takes two forms:

- The invoked application interface, which enables the workflow engine directly to activate a specific application to undertake a particular activity. This would typically be server-based and require no user action, for example to invoke an email application or passing data to a mainframe system.
- The workflow client application interface through which the workflow engine interacts with a separate workflow client application and responsible for organising work on behalf of a particular user.

API1- Process Definition Tools

A variety of tools may be used to analyse, model, and describe a business process. The workflow model is not concerned with the particular nature of such tools, and currently each of them is in a form tailored for the particular workflow management software for which it was designed. One of the interfaces proposed by the Coalition enables more flexibility in this area. This interface is termed the process definition import/export interface and is aiming at providing a common interchange format for the following types of information:

- Process start and termination conditions
- Identification of activities within the process, including associated applications and workflow relevant data.
- Identification of data types and access paths
- Definition of transition conditions and flow rules
- Information for resource allocation decisions

All workflow management tools should provide their workflow definition schema, that is given to the workflow engine which can afterwards modify, delete or add some new definitions.

API2- Workflow Client Applications

The workflow client application is the software entity presenting the end user with his or her work items, and that may invoke application tools, which present to the user the task and the data relating to it. It also allows the user to take actions before passing the case back to the workflow enactment service. The workflow client application may be supplied as part of a workflow management system, may be a third party product (such as an email product) or written specially for a given application. There is thus the need for flexible means of communication between a workflow enactment service and the workflow client application, which would provide a series of functions for connecting to the service as well as obtaining and processing items of work.

API3- Invoked Applications (not yet fully specified)

There is a requirement for workflow systems to deal with a range of invoked applications; for example, to invoke an email service, a fax service, document management services or

existing user applications. The Coalition sees value in the development of standards for the invocation of such applications by building "tool agents" which will provide the interface to invoke applications. In addition it is believed that it may be possible to develop a set of APIs which will allow other developers to build "workflow enabled" applications which can be invoked directly from the workflow engine. The specification of this API is expected to be merge soon with the API2 specification.

API4- Workflow Interoperability

A key objective of the Coalition is to define standards that will allow workflow systems produced by different vendors to pass work items between one another. Workflow products are diverse in nature ranging from those used for ad-hoc routing of tasks or data to those aimed at highly regularised production processes, each product having its own particular strengths. In its drive for interoperability standards the Coalition is determined not to force workflow product vendors to choose between providing a strong product focused on the needs of its customers and giving up those strengths just to provide interoperability. Interoperability can work at a number of levels from simple task passing through to workflow management systems with complete interchange of process definitions, workflow relevant data and a common look and feel. The greatest level of integration is unlikely to be available generally as it relies on a commonality of approach by a wide range of developers deep in their products where it is likely that innovation is rife. The following interoperability approaches have been identified and are being investigated:

- Level 1 - Coexistence: ability for a number of workflow systems to reside on the same hardware and software base
- Level 2 - Unique Gateways: developed to allow specific workflow systems to move work between themselves
- Level 2A - Common Gateway API: an enhancement of Unique Gateways
- Level 3 - Limited Common API: a subset of workflow product functionality is reduced to an open API; for example: connect, request task, and completion of task function calls
- Level 4 - Complete Workflow API: all aspects of workflow system behaviour are embodied via an open API
- Level 5 - Shared Definition Format: each workflow product can use the same process definitions at run time

- Level 6 - Protocol Compatibility: all APIs including transmission of definitions, work items, and recovery is standard
- Level 7 - Common Look and Feel: workflow product components appearance and method of operation are very similar

API5- Administration & Monitoring Tools

A common interface standard, which will allow one vendor's status monitoring application to work with one or more vendor's workflow enactment service engines. Firstly it will allow a complete view of the status of work flowing through the organisation regardless of which system it is in, and secondly this will allow the customer to choose the best monitoring tool for their purposes.

Action Technologies Inc., DST Systems, IBM, ICL, Plexus, SAP AG, Staffware and InConcert Inc. have demonstrated working prototypes based on the WfMC standards, IBM being the only one validating all specified interfaces (November 97).

The main problem with the WfMC architecture is a client-server architecture where the workflow server is responsible for process execution, auditing, management of the organisational directory and distribution of activities to appropriate participants. It also manages and hosts the work lists of the participants. Centralising all these functions in a single logical entity results in a monolithic system architecture, that is neither flexible nor scalable. For instance, it's the workflow server and not the service provider that decides the task implementation (cf. interface 4). A side effect is that binding tasks to implementation is made rather early. A critical view of the WfMC architecture can be found in [64] and [65].

2.1.2 The ANSA framework

In [76], ANSA presents a flexible transaction framework for dependable workflows. In order to succeed, workflow will need to operate with the ability to cope with system failures and provide dependable services. Atomic transactions are good for encapsulating short-lived interactions, but they don't scale to long-lived activities. Flexible transactions are an answer to that problem. Each of these flexible transactions are formed by a collection of ACID transactions with a set of execution dependencies between them and a set of rules describing

the flow of resources. This proposal adopts the operational features of atomic transactions. The resulting workflow is as a result a dependable flexible transaction with built-in mechanisms for failure detection and automatic error recovery. In this model, a workflow is seen as a collection of steps, each of which being modelled as an application specific flexible transaction. These steps are organised to carry out a business process. A scheduler is in charge of controlling the order of the execution of those steps, as well as the control flow between the steps and the synchronisation. The rules used to control the execution are described using some scripting co-ordination language.

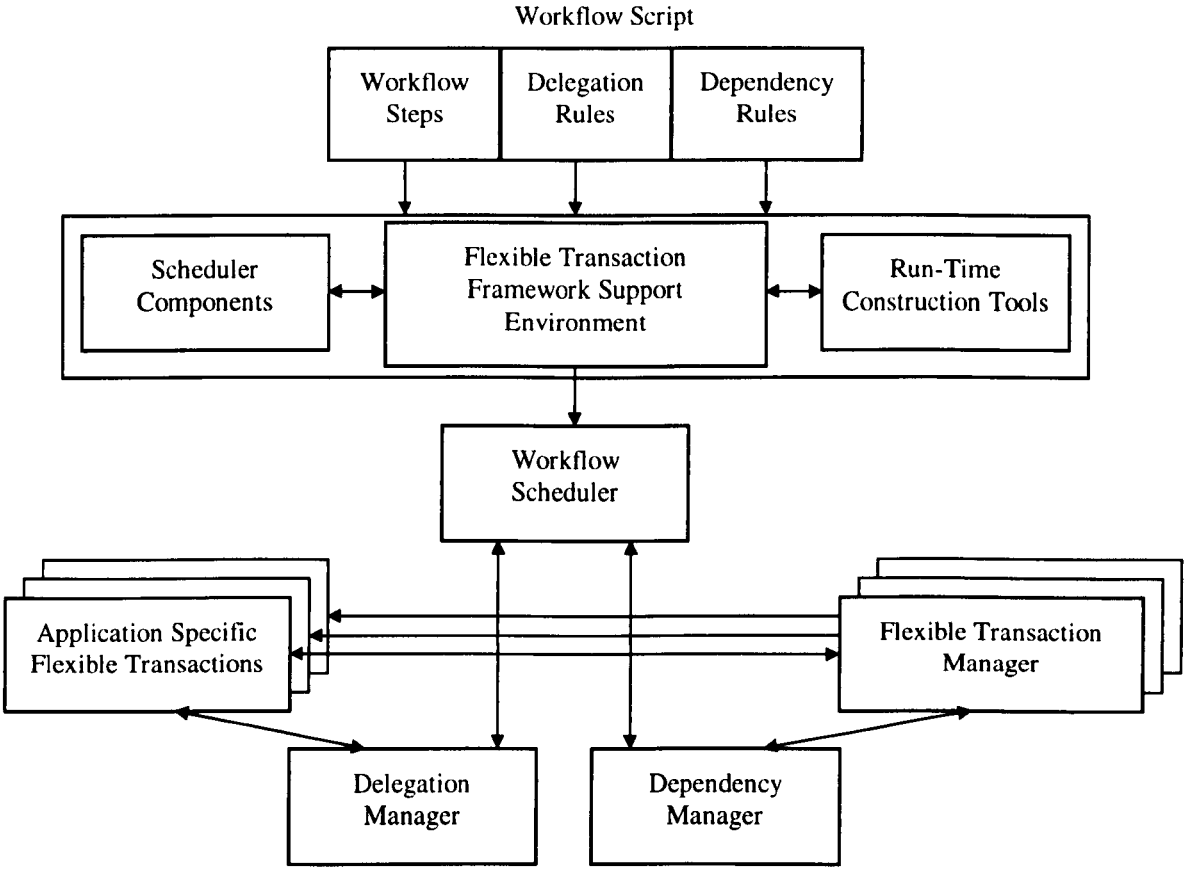


Figure 2.2: Architectural model

The architectural model is depicted in figure 2.2 and will be now described. There are several components: the workflow script, the scheduler, the delegation and dependency managers and the flexible transaction framework support environment.

The workflow script itself sub-divided in three parts:

- The workflow steps each of them describing an instance of a particular application-specific flexible transaction
- The dependency rules for the control of the order and synchronisation of the steps

- The delegation rules describing how the object resources are shared between steps.

Using this script the flexible transaction framework support environment interacting with some run-time building tools and some scheduler components (libraries), generates a workflow scheduler, which is the run-time equivalent of the script specification. This scheduler is itself a flexible transaction interacting with a delegation and a dependency manager to control the execution of the workflow application as a set of flexible transaction steps. The scheduler, delegation and dependency managers are the workflow enactment service of the WfMC model. The Workflow Script and the rest of the components of this architecture are process definition tools in the WfMC model.

The delegation manager is the component in charge of the specification and implementation of delegation of object resources. Its interface consists of the following six operations:

- Create (Name:DelegateSet): create a named empty set of object resources used afterwards to transfer objects
- Delete(Name:DelegateSet): delete a set
- Insert(Name:ObjectResource, NameDelegateSet): add an object to a particular set
- Remove(Name:ObjectResource, Name:DelegateSet): remove an object from a set
- Delegate(Ti, Tj, Name:DelegateSet): transfer the ownership of the object resource set from transaction Ti to transaction Tj
- Acquire(Tj, Ti, Name:DelegateSet): accept the transfer of ownership of the object resource set from Ti to Tj

The dependency manager is the component of charge of the definition and implementation of the inter-transaction dependencies. It has as interface the following operations:

- DefineDependencyType(TypeSpecification): define a new type of dependency
- CreateDependency(Ti, Tj, Name:DependencyType, [perpetual]): create a named dependency of the specified type between the transactions Ti and Tj with an optional perpetual trigger
- DeleteDependency(Ti, Tj, Name:DependencyType): delete the specified dependency between Ti and Tj
- EnableDependency(Ti, Tj, Name:DependencyType): enable the specified dependency between Ti and Tj
- DisableDependency(Ti, Tj, Name:DependencyType): disable the specified dependency between Ti and Tj

Each scheduler can be used as a global scheduler acting as commit co-ordinator for its controlled steps. In this case, the scheduler interacts with the steps on their abort, prepare and commit intentions.

This project was the starting point of our research. The idea of using transactions to build a workflow management system was found interesting. Having two managers one for the delegations and one for the dependencies was not found to be necessary as dependencies can be considered as particular types of delegations.

2.2- Build time environment

We are now going to discuss about the process definition tools and the API1 of the WfMC model. Composing applications out of existing applications is not a new idea, script languages have been used in order to do that for quite a long time. Varieties of languages have been developed for specifying different aspects of software systems. Most notables, the Architecture Description Languages (ADLs) specify the software structure, whereas scripting languages specify the behaviour. The subject of describing the composition of reliable distributed applications is a relatively new field of computing science research. Here the transactional workflow community represents the state of the art. In the following sections, we review a representative set of languages as well as the associated tools provided to help creating specifications in these languages. We will start with general purpose scripting language then workflow languages. Finally, we will present some ADLs as we used their modularity for our own language.

2.2.1 Building environment based on general purpose scripting languages

Lots of scripting languages have been developed to aid the building of applications out of existing applications. Following the success of Unix shell scripting languages (sh, csh, ksh...), several general purpose scripting languages such as Tcl or Perl have been developed to glue applications. They typically provide enough programmability (variables, control flow, procedures) to let users build complex scripts that assemble existing programs into a new application. The reader will find a discussion of scripting languages versus system programming languages in [53].

2.2.1.1 Tool Common Language (TCL)

This language [52] is fully interpreted and allows dynamic modification of the script. It is however based on strings, and provides very little structure. It's easy to connect strings together but it can become really hard to manage large complex scripts. Tcl is an action-oriented language rather than an object-oriented language as there is one command for each action that can be taken on an object and the command takes the object as an argument.

It is a glue language as a Tcl application can include many different packages, each of them providing an interesting set of Tcl commands. It can run external shell programs using the command exec and as a result is used as a job control language. It is also quite good as a communication mechanism allowing different applications to work together. For instance, any Tk application (i.e. windowing application written using the Tk package) can send a Tcl script to any other Tk application to be executed there. It provide constructs for major control structures: "**if then else**", "**switch**", "**for**" (with "**break**" and "**continue**"), "**foreach**", procedures, as well as "**eval**". "**Eval**" is a general-purpose building block to create and execute Tcl script. It adds a level of parsing. It is possible to create your own control structures by using the command "**uplevel**". These control structures are defined as Tcl procedures.

Tcl was written with the thought that it should be easy to extend. As a result, a lot of extension packages are available (such as **GroupKit** is a package that makes it easier to develop GroupWare application to support run-time distance-separated collaborative work between two or more people). We will briefly describe two of them that are more relevant to our study: expect, and Tcl-DP

Expect is a Tcl program that can talk to interactive programs. It knows what output can be expected for a program as well as what the correct answers should be. It is usually used to control automatic programs such as telnet, ftp, fsck, and rlogin. It also allows the user to take control and interact directly with the program whenever needed.

Tcl-DP was developed by the University of Berkeley and stands for Tcl Distributed Programming. It's built on top of the Tcl's built-in socket command for its low-level networking. Creating a Client/Server application becomes really simple with this package. A dummy example would be:

```
set id 0
proc GetId {} {
global id;
ncr id;
return $id;
}
```

```
MakeRPCServer 4000
```

This code creates a server listening on port 4000. This dummy server just provides a method "GetId" that adds one to a global variable (id) each time that it is called and returned its value to the client. If needed, two security checks can be provided as optional arguments, the first one to add a check on the login process (checkHost) and the second one on the commands that can be executed cmdCheck (specified as a procedure). The command becomes MakeRPCServer 4000 checkHost checkCmd.

The RPC client is even simpler:

```
set server [makeRPCclient host.ncl.ac.uk 4000]
RPC $server GetId
```

The first line opens a connection to the RPC server previously created (assuming that it was created on host.ncl.ac.uk) and save an identifier to later have a reference to that connection. Then the RPC command on the second line just forwards the script given as argument to the server.

Tcl also provides some exception support (catch, error), object replication as well as an asynchronous RPC mechanism aimed at getting results from long-lived programs. Using a Tcl/Tk plug-ins, it is also possible to run Tcl/Tk programs as applets using the construct <embed src=script.tcl width=wsize height=hsize>.

One of the main problems of Tcl was that it was not found to be adaptive enough to model object-oriented problems (Incr Tcl is an object-oriented extension to Tcl).

2.2.1.2 Practical Extraction and Report Language (PERL)

This language [75], [67] was optimised to scan arbitrary text files and extract information from them. It however is generic enough to be considered as a general-purpose language. It is semi-compiled; the script is first parsed, then it is turned into a syntax tree that is optimised in a final step. It's based on some pattern matching and is a good language for many system management tasks. It has the usual control structures "**if elsif else**", "**while**", "**until**", "**for**", "**foreach**", with the loop control "**last**" (break) and "**next**" (continue). It has also procedures and socket support and can execute other programs using "**exec**". It allows object-oriented programming and some support for CORBA is available (idl2perl working with Omniorb and Orbix among others has been developed).

Both of these scripting languages are good at gluing simple programs together, they assume that there already exists a collection of useful components written in other languages and plug

them together. Unix shell scripts are used to assemble filter programs into pipelines, Tcl is mainly intended to arrange collections of user interface controls on the screen, Perl is good at extracting information from textual results and reporting about them or use them to trigger something else. Python [77] and JavaScript [44] are some other general purpose scripting language, the first having a strong model of object-oriented programming and the latter being specifically designed for the Internet. Most of the general-purpose scripting languages are typeless, and as a result the detection of errors is done very late. Similarly most of those languages do have an extension for object oriented programming, which allows encapsulation and interface inheritance that increase even more reusability.

2.2.2 Workflow specific build time environments

In this part, we will focus on the related work in the fields of workflow management. All research projects we are aware of do have their own specific languages. All provide a graphical specification language and many also provide a textual specification language. Similarly to our scripting language, all those languages are higher-level than standard programming languages such as C, C++ and Java. They support the specification of the task structure (control flow) as well as of the information exchange between tasks (data flow). They also usually provide some support for exception handling as well as some support for temporal dependencies.

Several techniques from other fields of computing science have been used as basis to specify task collaboration: for instance event-condition-action have been used to specify the rules upon which a task should be triggered in some rule-based workflows. The METEOR project [13] developed by the University of Georgia is a typical example of this class of languages. Some other projects have chosen to base their work on an extension of Petri nets, which enable them to model the control flow using tokens. Some other projects from the database community use built-in SQL statements.

2.2.2.1 METEOR

In this project [13], they have decided to divide the definition of a workflow between the TSL (Task Specification Language) and the WFSL (WorkFlow Specification Language) that can be both compiled or interpreted. The TSL briefly described in section 4.6 is used to specify the basic tasks and is both a programming language and an embedding language (e.g. it can provide a wrapper for legacy applications). The WFSL specified the dependencies between

tasks and can be used to create complex tasks by composing them out of existing tasks.

The requirements taken into account while designing these languages were the following ones:

- Inter-task dependencies.
- Data management including filters to translate different formats.
- Modularity of the Workflow definition, and separation of conceptual model (WFDL) and of details of tasks, interfaces... (TSL)
- Error management (fault tolerance)
- Dynamic workflows
- Controller co-ordinating the tasks according to the constraints (dependencies...)

Workflow Specification Language

The WFSL is a rule-based language and can be either generated via a GUI or directly coded. The designer has to define the class (task structure type + set of inputs and outputs) of the tasks he wants to use.

The WFSL is divided in several parts:

- Type definitions and variable declarations, similar to the C syntax.
- Task type definitions
- Task class and filter definitions
- WF definition
  - task instantiations
  - rules
- WF instantiation

We will now describe these components one by one:

Task type definition:

This allows the description of the transition and states of a task class.

```
   typeName    {SIMPLE_TRANSACTIONAL    |    SIMPLE_NON_TRANSACTIONAL    |
TRANSACTIONAL_OPEN2PC, COUNPOUND_NON_TRANSACTIONAL }
   {
       {CONTROLLABLE   |   NOT_CONTROLLABLE}   transitionName(initialState,
finalState) [input | output];
   }]
```

The CONTROLLABLE or NOT specifies whether or not the transition can be enabled by

the workflow controller. The input, output specifies whether or not the transition can receive inputs and produce outputs.

Example:

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

In this example a new task type called newType was declared a simple transactional task with three transitions, one of them start letting the task go from the state initial to the state executing, controllable and receiving an input, and the two other non controllable and generating some outputs.

Task classes definition:

In order to define a task class, you have to associate a type of task as well as declare what input/outputs are visible externally. The syntax is given below:

```
typeName className {SIMPLE_TRANSACTIONAL | SIMPLE_NON_TRANSACTIONAL |
TRANSACTIONAL_OPEN2PC, COUNPOUND_NON_TRANSACTIONAL}
  ( input@{stateName} type name, output@{stateName} type name};
```

As an example, we can define a task class of type newType that is transactional and received as input for the initial state input1 and returns as output for the commit state the output output1. Both of input1 and output1 being themselves of type type1. Notice that the output is only visible for the commit state and not for the abort state.

```
newType newClass SIMPLE_TRANSACTIONAL (input@{initial} type1 input1,
output@{commited} type1 output1);
```

Once instantiated, the tasks can be linked between each other via some rules. A rule is divided in two parts: a control part and an optional data transfer part.

Workflow definition:

It's just a task of type COMPOUND_NON_TRANSACTIONAL with some task instantiations and a set of rules.

```
TaskClassName WorkflowClassName COMPOUND_NON_TRANSACTIONAL
(input@{initial} TYPE input1, output@{done, failed} TYPE output1)
{
    task instanciations;
    variables declaration;
    rules;
}
```

The instantiation is identical to the instantiation of the simple tasks:

```
WorkflowName WFinstanceName;
typeName SimpleTaskinstanceName;
```

We can instantiate a simple task called T1 of task type newType by adding in the script:

```
newType T1;
```

Then we have to specify the rules that the workflow has to follow, the syntax to specify a rule is:

```
<left hand side> EVALUATOR <right hand side>
```

ENABLES is a predefined evaluator that enables a transition.

For instance, we can specify a rule that enables the transaction start of the task L2 with as input for input1 of T2 the output output1 of T1 filtered using the filter f1. If we want as preconditions to trigger this rule that the task L1 is in the state done, that the global variable outval4 is greater than 5 and that L1 has terminated with its output1 valid. The validity of output1 is determined using the function success that decides whether the task L1 has succeeded with its output "output1" valid

```
[L1, done] & (success(L1.output1) = TRUE) & (outvalL4 > 5)
     ENABLES [L2, start] % f1(L1.output1) -> L2.input1;
```

Some useful instructions to quantify element in a set have been added to the script language. They can be used for instance for a fork, a join where only a percentage of tasks must succeed before it can proceed.

```
forall i in a..b [condition]
exists i in a..b [condition]
```

These two instructions have their usual meaning.

2.2.2.2. Webflow and the Co-ordination Language Facility (CLF)

Webflow [21] is an environment supporting distributed co-ordination services on the web and is typically used to describe applications such as distributed document workspaces, enterprise workflow and electronic commerce. It is using the CLF middleware environment [4] for distributed co-ordination, which provides a basic set of library tools for building co-ordinators and resource managers. We will now describe the CLF and then briefly describe the tools available for build time.

CLF is a process-oriented extension of object-oriented programming, aiming at providing

support for the co-ordination of heterogeneous, possibly distributed, active objects within larger units implementing work processes. It has two types of objects:

- The co-ordinators requesting performance of actions, whose course has to be negotiated among the multiple participants, ensuring that there are no conflict between the ways of enacting the actions chosen by the participants and declaratively implemented as rules,
- The participants that instantiate an interface specifying the negotiation dialogue invoked at run-time by co-ordinators.

The rules are proactive, e.g. the co-ordinator actively looks for participants that can fulfil the rules by querying them and trying to find agreements among the participants.

A CLF program includes four sections: implementation, signature, interface and rules.

- **The implementation section** links the resource bank names to external object and is implementation specific.
- **The signature section** distinguishes the parameters of the rule tokens between input and output parameters, it has the following syntax:

```
   TokenName1(ParameterList1):          InputParameterList1          ->
outputParameterList1
   TokenName2(ParameterList2):          InputParameterList2          ->
outputParameterList2
```

For instance:

```
   book_seat(client, flightId, seatNb) : client, flightId -> seatNb
```

declares that an inquiry for the token book_seat given a value for client and flightId return a value for seatNb

- **The interface section**: allows programmers to use multiple signatures for a resource bank. The syntax to associate a resource bank to a signature is:

```
TokenName1 = BankName1
TokenName2 = BankName2
```

Each signature must have an associated resource bank. The following statement then does declare that the token book_seat uses the resource bank travel_agent:

```
book_seat = travel_agent
```

- **The rules section:** specifies the behaviour of the co-ordinator and consists of a set of rules consisting of two multi-sets of tokens (separated by @). The two sets are called left- and right-hand side of the rules. The tokens in the left-hand side are removed when the rule is triggered while the right-hand side tokens are inserted in the list of tokens to be fulfilled.

Rules appear on different lines. Two special tokens were introduced: `#b` meaning that there is nothing to insert and `#t` meaning that the co-ordinator should terminate. Both tokens can only be used on the right-hand side of the rule. The syntax to specify one rule is as follows:

$Token_1$ @ $token_2$ @ ... @ $token_i$ <>- $token_{i+1}$ @ ... @$token_n$

For instance:

```
flight_reservation(client, flightId) @ book_seat(client, flightId,
seatNb) <>- printTicket(client, flightId, seatNb)
```

Some build time tools are provided to help with building these applications. The main component is the co-operative *process editor* that allows the definition of processes as flows of activities, assignment of roles, as well as temporal and document scoping. The resulting specifications are called process maps and are translated into a set of CLF rules needed to co-ordinate the distributed execution of the process. The collaboration environment used to support the co-editing of the process map is the BSCW system [7]. This system allows to share documents (process maps) across workspaces and supports versioning.

2.2.2.3 Workflow on Intelligent and Distributed database Environments (WIDE)

This project [12] uses a GUI language to specify workflows, however a Workflow Textual Definition Language (WTDL) is also available and will be described thereafter.

The WTDL can be divided in several parts: The initial part is the definition of the flow (or workflow for us) introduced by the keyword WF. This definition can itself be subdivided into two:

- The flow definition, where you can find the type definitions and variable declarations, databases used and kind of access granted on those databases.
- The flow structure specifying the start, total and partial joins and forks.

Then the component tasks are described. There are two different types of tasks in their model. The super tasks (equivalent to the compound tasks) and the task (simple or replicated simple tasks). A super task is sub-divided into three parts: the task definition where the type definitions and declarations, functions used (code), SQL queries can be found, the task control including the preconditions (wait), conditions (tests SQL, or functions), error handling (On... Do) are specified and the flow structure. There are two sorts of simple tasks: the simple tasks and the replicated simple tasks (called multi tasks) which on top of what a simple task specifies

also has to specify the number of replica and the quorum required to decide of the outcome. The specification of a simple task is divided in 3 parts: task definition, task control and task actions with can be: insert, delete, update, select-one SQL queries or functions. Below, an example illustrating how a workflow is specified using the language is given. The workflow is quite simple, after being started, it runs task1 and task2 and then using the function myFct makes a query to check the result and depending on the result of that query it either runs Accept or Reject. Task1 waits for a SQL query to succeed and then tests whether another query is successful or not, if it was then the whole workflow is ended. Task2 is a multitask (replicated): three instances are started, out of which two are needed as quorum. Task2 deletes some records from the database. Task2 also sends a notification after 60 days if it has not yet completed by then so that somebody can have a look at what is happening.

```
WF myWF {
    struct myStruct {...};
    int myVar;
    uses myDB(Key, field1, field2);
    grant select, insert, delete on myDB;
    int myFct()
    {
          /* code including SQL code */
    }
    start {
          task1;
          task2;
          if (myVar == myFct()) {
                Accept;
          } else Reject;
    }
}
task task1
{
    wait exists(select ...);
    get record;
    on exists (select * from myDB where cond1) do endWF;
}
multitask task2
{
    NumberOfInstance=3;
    Quorum = 2;
    int myVar2 = myFct();
    delete from myDB where (myDB.field1=myVar2);
    on Elapsed(60 days) do NOTIFY("Deadline!");
}
task Reject
{
    update myDB set ... where ...;
    delete from myDB ...;
}
task Accept
{
    insert...;
}
```

Specific ideas developed

- Partial join and fork using the multitask construct and quorum
- Loops available: **while**, **do**... **while**...
- Replication via specialised task
- **exclIf** cond instructions [**exclIf**...] else instructions : exclusive if (mutual exclusion)
- Reactions to errors : **NOTIFY** (send a message to the responsible), **END** (end as done), **cancel** (end as not done), **REFUSE** (the agent supposed to process the task refuses the job), **endWF** (force the end of the whole workflow), SQLaction

The notations used by the GUI are described in figure 2.7 and will be now briefly presented:

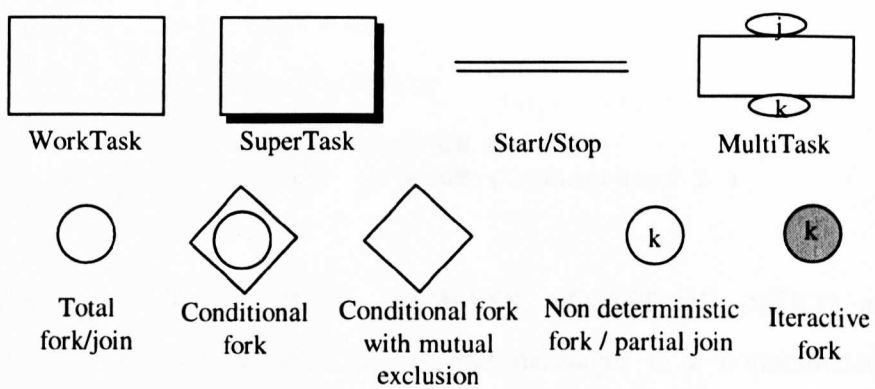


Figure 2.3: Notations for GUI in WIDE

The different tasks are represented by rectangles. For replicated tasks (multi tasks), two ovals with as associated value j as number of replicated tasks to be started and k as quorum are added to the rectangle representing the work task or the super task. The routing tasks are represented by circles and diamond-shaped forms. The non-deterministic fork, iterative fork and partial join tasks do have a value k associated to them. Tasks are connected using unidirectional arrows.

Some tools for editing and compiling WF schema written in WFDL are expected to be provided as well as some administration tools for agent management and monitoring (history report).

2.2.2.4 The CBORD system: a script based on tasks and transactions

This model [46] is based on communication via messages. It has two kinds of tasks, some simple transactional tasks (called transaction) and some compound tasks (called tasks)

A valid event (basic message) is one of the following instructions followed by "to" and the name of an agent:

- get(...)
- enter(...)
- assign(...)
- extract(...)

Transaction modelling

```
Transaction name
    agents : list_of_agents
    agent1 can send :
          list_of_valid_events
    Constraints:
          event1 UNTIL event2 OR event2
          ALWAYS( event1 => NEXT ( constraint ) )
end Transaction
```

Four temporal operators (UNTIL, ALWAYS, SOMETIME, NEXT) are provided to describe the dependencies between events and messages in a communication process. A constraint is either an event1 UNTIL event2 (e.g. you have to wait that the event2 happens before event1). An ALWAYS (something) which means that it's looping. SOMETIME conveys the obligation to honour the events in future states and NEXT is the equivalent of a trigger in active databases.

Task modelling

```
Task: name_of_task
    Constraints:
          task1 SUCCEED task2 SUCCEED task3
          transaction1 BEFORE transaction2
    Goal = THRESHOLD(100%, task1)
    Exit = OR(THRESHOLD(...), CANCEL(task1))
    Alternative = XOR(..., (THRESHOLD(..., ...) BEFORE ...))
    Commitment = ALWAYS(task2)
end Task
```

A task is specified using a similar structure as transactions, i.e. name, communicating

agents, messages exchanged between tasks and tasks constraints. Some extra information has to be provided:

- A goal (what makes the task succeed),
- An exit (what makes the task fail or abort),
- Some alternatives (what other tasks can be executed instead of the task),
- Authorisations (what constraints specified in the transactions can be overwritten in case of conflict),
- Commitment (what the task is obliged to do regarding the workflow)

The following operators are available: AND(..., ...), OR(..., ...), XOR(..., ...), XOR(..., skip) for non vital tasks

The following task dependencies are available:

- (T1 SUCCEED T2) when T1 is successful, T2 starts
- (T1 BEFORE T2) T1 must occurs before T2
- (T1 OVERLAP T2) both tasks can occur simultaneously
- CANCEL(T1)
- SOMETIME(T1)T1 will be executed in the future
- ALWAYS(T1) T1 will be repeatedly executed

Numerical constraint can also be specified:

- THRESHOLD(number, T1) if the goal of T1 can be measured numerically, T1 will need to fulfil its goal over this number.

2.2.3 Commercial Workflows

All the commercial workflow management tools have a process definition tool via GUI available. Only half of them have a textual process definition tool. Notes (Lotus) and FloWare (Plexis) do not have a script language, InConcert [28] (Xerox) has one, but just for administrative functions. ActionWorkflow [37] (Action Technology) and Staffware have a scripting language. Hundreds of products are nowadays claiming to support workflow, ObjectFlow (DEC), SAP Business Workflow (SAP AG), WorkFlo [16] (FileNet), WorkManager (HP), FlowMark [27] (IBM)... Usually commercial workflows provide some support for dynamism, some weak testing or analysis tools, and have no support for fault tolerance, FlowMark excepted.

2.2.4 Architecture Description languages (ADL)

Current ADLs focus on the structure of the components of a software system and their inter-relationships [36]. ADLs were proposed as an answer to the need for formal modelling notations, analysis and development tools that operate on architectural specifications to support architecture-based development. ADL-based specifications model the application as a set of components communicating through connectors. Typically, an application is composed out of a group of other components, where a component provides services to other components through ports. The interaction between ports can be done using various methods, for instance buffered message passing. An ADL can be either an in-line configuration language or an explicit configuration language that models both components and connectors separately from configurations. Then they can be also divided between the implementation independent languages and the implementation constraining languages (i.e. those that do and those that do not assume a particular relationship between an architectural description and an implementation).

Some of the well known ADL languages are Aesop [18], ArTek [72], C2 [38], Darwin [35], Lileanna [73], MetaH [23], Olan [6], Polylith [58], Rapide [33], SADL [42], UniCon [69] and Wright [1]. We will now present two of those languages: Darwin and Olan. A classification and comparison framework for ADLs can be found in [39].

2.2.4.1 Darwin

In Darwin [35], a component is defined as the basic element from which systems are constructed. Complex components are constructed by composing them from more elementary components. The overall architecture of a software system is then specified as a hierarchical composition of primitive components that have a behaviour specification.

Darwin sees components in term of the services provided to other components and services required from other components. Each service is further elaborated with an interaction mechanism that implements the service (for instance, outputs are done via ports, command accept entry calls and trace services are implemented with events). The textual and graphical specifications of a component interface in Darwin will typically be:

```
component myComponent {
    provide myOutput<type_of_service>;
    require myInput<type_of_service>;
}
```

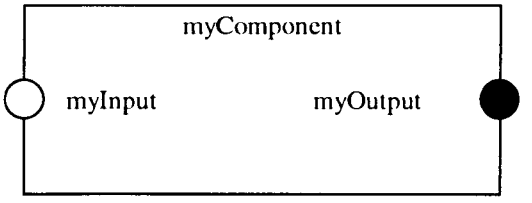


Figure 2.4: A component in Darwin

In this example, the component called myComponent provides one service called myOutput and requires another one called myInput. The type of a service is specified in angle brackets. An example of type would be <port, int> for a service accepting messages of type int or <stream char> for a service implemented by a stream as communication mechanism, with as communicating datatype char. Had it required two services, the specification would be done by separating them using a comma.

```
require myInput1<type_of_service>, myInput2<type_of_service>;
```

It has to be noticed that a component does not need to know the global name of the services or where they can be found in the distributed environment, the names are local to the component type specification.

In order to create a composite component out of existing component, the "**bind**" and "**inst**" constructs are provided. The "**inst**" construct is used to declare the instances of components that it consists of, while the "**bind**" construct associate required services to provided services of compatible types. It has to be noticed that the language imposes as restriction that a particular requirement can only be bound to a single provision.

```
component myCompositeComponent {
        provide myOutput;
        require myInput;
        inst F: myComponent;
        inst G: myComponent;
        bind
                F.myInput -- myInput;
                G.myInput - F.myOutput;
                myOutput - G.myOutput;
}
```

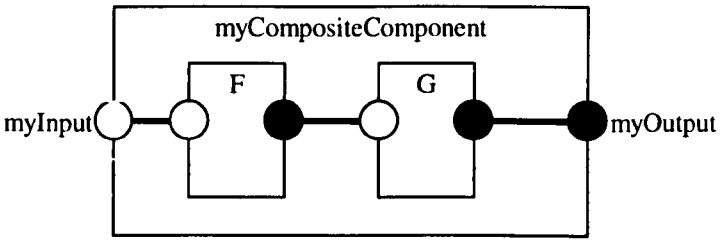


Figure 2.5: Composite component in Darwin.

A component can also take some parameters, (e.g. component myComponent(int i, string s)). A component type can also be defined as a partial evaluation of another one, (e.g.

myComponentHello = myComponent (, "hello"))

Portal declarations define a set of component portals that can be bound internally to be encapsulated sub-component portals or externally to the portal of peer components. There are five categories of portals:

- Portal declarations declaring a component portal,
- Provide declarations specifying portals that are being provided by the defining component to other encapsulating components,
- Export declarations declare portal that are being provided by the defining component to an external trader/name server,
- Require declarations for portals being provided by other encapsulating or external components
- Import declarations to introduce portals provided by an external trader/name server.

A support for dynamic reconfiguration is also provided using the **dyn** construct. These dynamic changes have to be known a priori. Bindings made to dyn components cause a new anonymous instance of the component type to be instanced each time the component in invoked by a bound portal. In the example below, invoking the service myDynInput creates a new myDynComponent instance and passes it a single integer parameter. The bindings are made with the type rather than the instances. Darwin only support unidirectional communication with these components as the services provided by such components can only be accessed by passing service references in messages to form bindings dynamically. The Darwin program can not capture these bindings, because the dynamic instances are anonymous.

```
component myCompositeComponent {
     provide myOutput<port smsg>;
     require myDynInput<dyn int>;
     inst C: myComponent;
     bind
        C.myOutput -- myOutput;
        myDynComponent.myOutput - C.myInput;
        myDynInput - dyn myDynComponent;
}
```

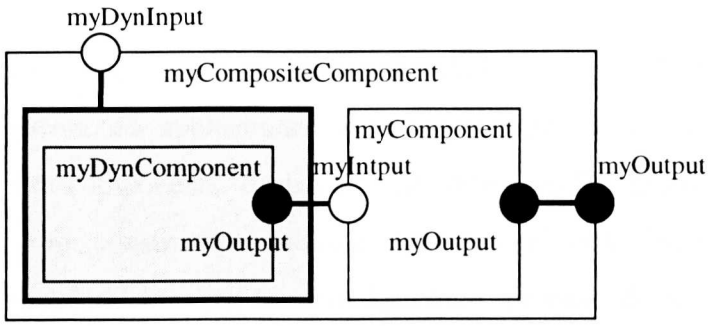


Figure 2.6: Dynamic reconfiguration in Darwin

Darwin also provides what is called a lazy instantiation in which the component providing a service is not instantiated until a user of that service attempts to access the service. The combination of lazy instantiation with recursion allows the description of potentially unbounded structures and can be useful for alternative actions. Such a component is introduced using the **dyn** construct

```
myLazyComponent: dyn myComponent;
```

Several control flow commands have been provided:

- **forall** `k = 0 to n-1` where k is a loop variable and n the number of iterations,
- **when** `condition` instruction. This adds a guard in front of an instruction that is only executed when the condition is true.

Arrays can also be used, the instruction below introduce (but does not instance) an array of n filters.

```
array F[n]:filter;
```

Afterwards component types can be instanced by the instruction:

```
inst F[k];
```

The user may also want to run each instance on a different machine for replication purposes for instance. Adding a tag @k+1 to the previous instruction does this. Tags are introduced by the construct @ and are a mechanism to attach non-structural information (e.g. constraints, resources specifications...) to a Darwin specification.

External definitions are introduced using the construct **spec** *external_language_id* `{code}` to allow externally written definitions (for instance IDL, LTS definitions). Generic types can also be used as some kind of templates and are introduced between angle brackets e.g. `<T>`. The asset construct allows integrity checks during elaboration, in case of failure, an error is produced and the elaboration is aborted.

2.2.4.2 Olan

Olan [6] is a language and a run time support intended to facilitate the design, configuration and evolution of distributed applications made of distributed applications made up of heterogeneous software components. It claims to provide a single unified description of those applications, adequate for construction, management and evolution. The overall description is implementation independent so that the configuration process does not depend on the programming process.

Like in Darwin, applications are viewed as a hierarchy of components linked by some connectors. Each level of the hierarchy is a separate description derived from a component class encapsulating components in the next level. The leaves are primitive components deriving from a primitive component class encapsulating real pieces of software such as a C++ object or a C module. Components are described by their interfaces, which contains services, notifications and attributes. The services can be either provided or required and correspond to the Darwin services. Notifications are events that are broadcast and can trigger on the receiving components a piece of code sequence called reaction. Notifications can be ignored and reactions are not necessarily triggered on reception of a notification. Attributes are typed variables whose values can be imported from the implementation or set outside the component. Connectors are the units mediating the interactions between components. They establish the rules driving the component interactions such as the conformity rules (parameter type checking, homogeneity of connections...), the protocol used as well as the behaviour specification, and the constraints (Quality of service...). A connector description defines its kind (interconnection, mapping to an implementation), the allowed sources and destinations of communication, as well as the specification of the expected behaviour, constraints and protocol. Currently these connectors are built-in in the OLAN language.

We now describe how a component is specified:

```
component class myComponent {
interface
      require myOperation(in operation);
      provide myService(in operation);
      ...
implementation
      C = inst myComponent;
      D = inst myOtherComponent;
// Mapping using connector
      myService => to C.myService;
      C.myOperation => myOperation;
// Interaction
      C.Init => D.Init;
```

```
        C.myOperation => D.myService;
        D.myOperation => C.myService;
};
```

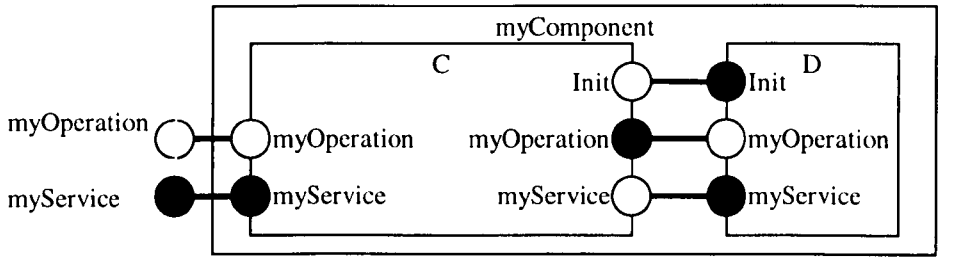


Figure 2.7: A component in OLAN

The lazy instantiation of Darwin has an equivalent introduced by the construct **dyn inst**. A construct named **collection** was also introduced to gather multiple components of the same class

```
myComponentSet = collection[1..n] of myComponent;
```

Two services, called create and delete, have been provided to use collections as way to dynamically create instances of a component. In addition a special connector can operate as a creator of a new component of a collection before accessing the specified service. This is illustrated below.

```
C.Init
        => myCollectionSet.Init
        using createInCollection;
```

Attributes are used to distinguish the elements of a collection which otherwise would be anonymous as are the arrays in Darwin. In order to specify a connection with a specific element, the following connector is used:

```
C.myRemoteOperation(operation, remoteApplId)
        => myComponentCollection.myService(operation)
        where myCollectionSet.ApplId = remoteApplId
        using methodCal;
```

In this case C sends its operation to the element whose applId attributes fits the identity of the target application remoteApplId.

2.2.5 Discussion

This section was about how you can model a process as a workflow. Different approaches have been tested, including Petri nets, SQL, transactions... Workflow systems provide a set of specification tools (GUI and script languages) allowing the specification of processes at a high level. The GUI usually represent the processes as a directed graph with the nodes representing the processes and the arrows representing both the data flow and dependencies between them. The script languages can be quite different and there is no standardisation despite some efforts.

Usually at least two types of tasks are provided: compound tasks (workflow) and basic tasks (steps, units of work). Sometimes some new types of tasks are added, such as replicated tasks, alternative tasks... These tasks have dependencies between them, such as data flow dependencies (object delegations) or temporal dependencies (notifications). Several approaches have been chosen to specify them. The main difference being them are whether you let the user create some complex dependencies including basic or complex computations (Task i terminated in state commit and value of its output is worth four) or whether you restrict the dependencies to a boolean tree of tasks reaching certain outcomes (task j reached state success or task k reached state failed).

In the comparison matrix below, we sum up how the different build time environments presented deal with fault tolerance, dynamism, locality of modification, composition, specification of the temporal structure of the applications as well as which tools are provided to help building your application.

Language	Model	Fault tolerance Exceptions	Dynamism	Locality of modifications	Composition	Temporal.	Tools
Tcl	Generic programming facilities	Using error and catch constructs	Interpreted so can be modified at run time	Script	None	None	Interpreter
Perl	Generic programming facilities	None	None	Script	None	None	Pre-compiler / interpreter
Darwin	Components communicating via connectors	None	If known a priori, via dyn construct, arrays	Components	Composite construct	None	Text/Graphical editor, compiler, simulator, parser, code generator
Olan	Components communicating via connectors	None	If known a priori, via dyn inst construct, collections	Components	Composite construct	None	Visual programming environment, compiler, admin tool
Meteor	Simple and compound tasks linked by rules	Using error states, associated rules specifying action	If known a priori, via Foreach/exists constructs, arrays,	Compound tasks	Compound task	ECA Rules	Text/graphical editor, interpreter, compiler, simulator
CLF	Proactive co-ordinators with participants	None	Addition/removal of coordinators	Script	None	Prolog-like rules	Process graphical editor, versioning, translator process to rules
WIDE	Work task , super tasks ; SQL-based	Replication using quorum (multi tasks), on exception do...	None	Super tasks	Super tasks	Join/fork, serialisation	Text/graphical editor
CBORD	Transactions and tasks, message-based	Alternative actions	None	Transaction	Task constructs	Transaction and task dependencies	Text editor
Newcastle	Basic,compound tasks linked by dependencies	Alternative inputs and outputs	Addition/removal of tasks/dependencies	Tasks	Compound tasks	Notification and dataflow dependencies	Text/graphical editor, simulator, consistency check

Figure 2.8: Comparison of the built time features associated to the languages considered

The models adopted by the different projects are quite different and impact on the features supported by the languages. Generic scripting languages are not really adapted to support dynamism in the specification, as they have to be stopped if changes are requested. There is no special provision for run-time dynamism to specify the changes in the relationship between activities. Interpreted scripts such as Tcl can however be changed at run-time, Perl scripts

being semi-compiled can't... No special composition mechanisms are provided, nor workflow specific tools.

Script languages using ECA rules usually list the rules at the compound task level and as a result are not that modular. CLF does not address fault tolerance and has a flat structure in the definition that imposes modifications at script level. WIDE and CBORD do not address dynamic modification and but have some support of fault tolerance. ADLs are not adapted to the specification of the temporal structure. The issue of fault-tolerance was not really addressed in the two languages chosen. ADLs are interesting because of their modularity that allows locality of modifications. They also provide some support for dynamic reconfiguration when modifications are known a priori, and have some interesting tools available. Our language on the other hand tries to use the best parts of the previous languages. It takes an object-oriented approach by following the ADL approach of modelling the tasks as components that can be gather in compound tasks. The dependencies are kept at the most relevant level therefore providing locality of modification. The fault tolerance is provided by alternative input and output sets. This provides a natural way to provide user-level fault tolerance. Our model however lacks some of the ADL features that allow the specification of the software structure of the workflow. It may also gain from the addition of more control structures to ease the specification. The task implementation is also described in a distinct specification in the ADLs, Meteor, CBORD and in our language.

The duality of Fault-Tolerant structures between a model incorporating objects and actions as the entities for program construction and another model based on communicating processes and conversations has been established in [71]. Our System Structure is following the first model as the area targeted (e.commerce, office automation) are typically following the first model..

2.3- Run time environment for workflows management systems

A workflow system is aimed at co-ordinating tasks. Workflow Management Systems do provide an execution environment where the instances of the workflow are run, the steps are controlled in that environment and activities are mapped to real resources. The components in charge of the co-ordination, usually called the schedulers, are usually based on Event Condition Actions interpreters or on finite-state automata. Then some monitoring tools are also available to trace what happened and what is happening. When the systems are dealing

with human participants (i.e. some tasks are performed by humans), they also provide some work lists (inboxes for human participants to let them know which activities were assigned to them).

We will now describe some run time environments for workflow management systems and see how they provide reliability. First we will describe some transactional workflow systems starting with what could be regarded as the ancestors of the transactional workflow systems, the Saga and ConTract models. We will then present the ORBWork project from the University of Georgia, notable for the tools for the analysis and design of flexible transactions. We will then describe the IBM solutions Exotica/FMQM and RainMan.

2.3.1 Sagas

Work on Saga [17] represents an early attempt to develop a model of long running transaction. A Saga consists of a set of ACID sub-transactions with a predefined order of execution. Each of those sub-transactions T_i does have its compensating sub-transaction T_i^{-1}. A Saga completes successfully if all its components have committed. Otherwise committed sub-transactions are undone by executing their compensating sub-transactions. It also allows backward/forward recovery when system or application save points are available. In this case, the transactions started after the save point are aborted or compensated and the execution is restarted from the saved point. Pure forward recovery can also be supported if save points are automatically taken at the beginning of each transaction. This allows the execution of long-lived transactions. Sagas relax the isolation property of the traditional ACID transactions as well as increase inter-tasks concurrency. Some extensions have been made such as the nested sagas allowing the nesting of sagas.

The main problem of Saga is that it can only model applications composed as a serie of sequential tasks that also need to be transactional. As a result it could only be used to represent a small subset of workflow applications.

2.3.2 ConTract

The ConTract project [74] is aiming at providing a way of grouping transactions together into a multi-transactional activity. It sits on top of a Database Management System, which acts as a resource manager. A ConTract consists in a sets of steps (predefined actions with ACID properties) and a script describing how to execute these activities. Control flow between steps

can be modelled by using the usual elements: sequence, branch (IF THEN ELSE), loop and some parallel (PAR_FOREACH) constructors. Steps can be grouped as an ACID transaction using the construct TRANSACTIONS ... END_TRANSACTIONS. Dependencies can be specified based on the outcome of a step for instance if T1 aborts then T2 should begin will be described by the construct DEPENDENCY(T1 abort -> begin T2). Some synchronisation invariants (before starting and after completion the step) and conflict resolution rules can also be specified. Fault tolerance is provided by forward recovery using compensating actions using as arguments those used for the execution of the step they are supposed to compensate (semantical undo). The compensating actions are declared with the construct COMPENSATIONS C1: compensating_action(...)... END_COMPENSATIONS. Each step has to have its compensating action specified. In case of failure, the state of the ConTract is restored and the execution can continue. Contract provide both relaxed isolation and atomicity (so that a ConTract can be interrupted and re-instantiated). At run-time nested transactions are used to structure the system's work during the execution of the ConTract. The execution of a step is divided into several sub-transactions that can include for instance the execution of the code, the evaluation of the pre and post execution invariants... APRICOTS [68] is a prototype implementation of the Contract project.

Contract is for its time a good example of how to model workflows. It however has no support for dynamism and is quite restricted by only allowing two outputs depending on the validity of the post execution invariants: success output or failure output that triggers the execution of the associated compensating actions.

2.3.3 ORBWork

ORBWork [13] is a CORBA-based enactment system for the METEOR2 Workflow Management System developed at the University of Georgia, and now a commercial product of Infocosm. It is fully distributed and supports scalability, multi-database access as well as some fault tolerance in the form of an error detection and recovery framework using transactional concepts.

METEOR2 consists of a designer and two workflow enactment systems, ORBWork (CORBA-based) and WEBWork (Web-based). The designer is a GUI used to specify the workflow, the data objects manipulated, as well as the component tasks. It assumes nothing about the run-time. The specified design is stored in Workflow Intermediate Language for

subsequent code generation. The specification is kept in the workflow model repository. The designer has two different modes: the process modeller and the workflow builder, the latter allowing the user to refined the specification created by the first one by knowing the design of the run-time system. There are three components: the map designer, the data designer and the task designer which respectively allow to express the dependencies between tasks, data object manipulated and their flow, and finally the details of the individual tasks.

At run-time a code generator associated to the enactment system is used to create the workflow application, including the task managers, their scheduling components, and some recovery mechanism. The run-time system consists of the various task managers and associated tasks, the user interfaces, the distributed recovery mechanism and scheduler as well as the monitoring components. The task managers are responsible for the controller and the scheduler, while the tasks are just the executable. Different task models have been provided for the tasks (transactional, non-transactional, two-phase commit...), each of them having an associated type of task controller supporting different features (recovery...) and specified via an IDL. The task managers are automatically generated by the code generator from the MIL specification and are aware of their successors. It is itself started by its predecessors via the Activate method. When the pre-conditions (specified as an AND-OR tree) associated to the task it's controlling are fulfilled and all the input data are available, this task is started. Once completed, a post activation part is used to decide what to do and which (if any) successors to activate. The task managers are responsible for the consistency of the data that they are using. They saved the state of the data objects used by calling the save method or using the persistent object services of CORBA. The system gets a pre and post image of the data object, which allow audit.

The recovery system is based on a hierarchical error model and includes mechanisms for persistence, monitoring and recovery. The errors have been categorised in three main types: task errors, task manager errors and workflow errors. For the task errors, ORBWork allows the users to define errors and specify their handlers. If no handler is provided, the error results in erroneous conditions in the task manager. At the task manager level, the unhandled errors as well as the errors resulting from abnormal behaviour of the execution of the task manager (preparation of the inputs failing...) are considered. If the error can not be treated (by retrying to run the task for instance or running an alternative task), it becomes a workflow error. Another type of workflow error is the failure of enforcing the inter-task dependencies. It can

be due to communication failure. The system tries to deal with the error by for instance moving/replicating a faulty task manager on another node. If it can not be solved the error is reported to a human via a workflow monitor. Local Recovery managers, polling the critical CORBA components on their node are used to detect potential errors, while a Global Recovery Manager is used to check the Local Recovery Manager. The components to be monitored register (respectively deregister) when they need to be monitor (respectively when they stop needed this service). On detection of a failed component, this component is restarted using the factory associated to the recovery manager.

2.3.4 Exotica or FlowMark on Message Queue Manager

FlowMark [27] is using a layered client/server architecture, compliant with the WfMC standards. The built and run-time clients are linked to a FlowMark server itself client of a centralised database (ObjectStore) where both build and run-time information are kept. At built-time the built-time client interacts directly with the database and the FlowMark server remains passive. The run time architecture is depicted on figure 2.9. OSS and DB acting as the storage server represent the ObjectStore database on the figure. The navigation server (the FMS component) is a client of this database since most steps involve getting information in and out of the database. FlowMark Servers can also be connected among themselves. The rest of the components are connected to these servers. Usually the application and user interface (RTC) are kept on the same host to keep accesses as local as possible

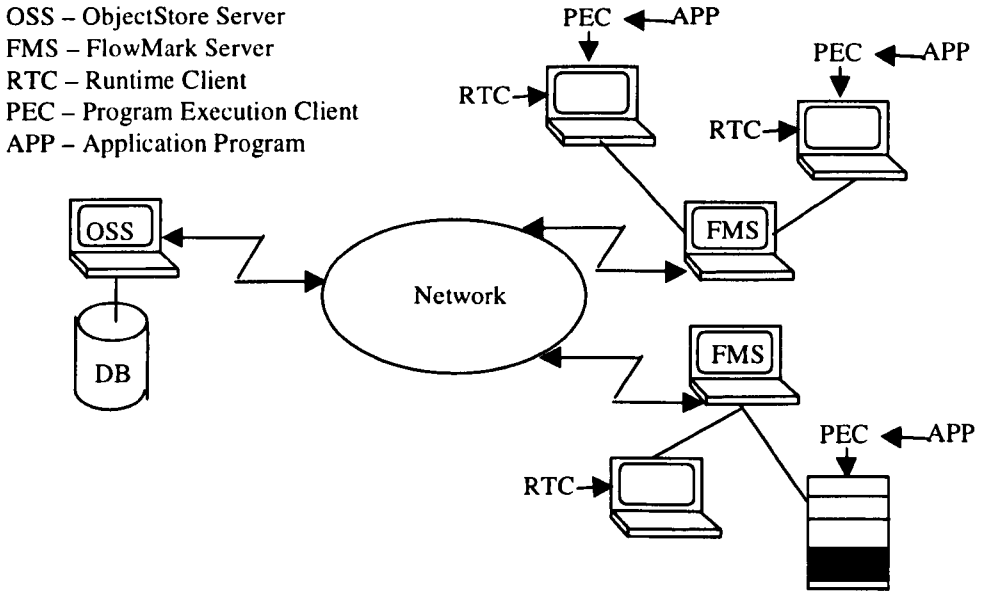


Figure 2.9: Run time architecture of FlowMark

FlowMark allows forward recovery and plans have been made to also support backward recovery using Spheres of Joint Compensations [32] in the future.

Exotica [40] is a distributed workflow system based on FlowMark. There is no dynamic modification, as changes to a schema (specification) do not affect the instances already started. Each activity has a start condition (a boolean expression) used to know when the activity can start and an exit condition determines when the activity was successfully completed. Control connectors and data connectors connect activities. The start condition is evaluated when all control connectors have their origin activity terminated and can be as a result evaluated to true or false. The data connectors link input and output data containers (one of each per activity). Clients are not persistent and there is no provision for crash recovery at the client level. IBM also defined an API standard for message passing called Message Queue Interface (MQI) [26]. MQSeries [41] is an IBM set of products supporting MQI.

Distribution in Exotica is carried out using message-oriented middleware based on MQSeries. The messages exchanged are persistent, which eliminates the need for the centralised database. This allows a set of autonomous nodes to co-operate to carry out the execution of a process. Exotica supports the mapping of Sagas and flexible transactions into FlowMark process schema with the restriction that it can not make changes to resource manager. In practice that excludes interesting models such as nested transactions or split transactions [57].

In FlowMark, the specification is done via a GUI by creating a process diagram showing the activities and their sequence, or it can also be done using the FlowMark Definition Language (FDL) [25]. The FDL is quite complex and could be more modular as all the dependencies within a compound task are listed as part of the compound task and not delegated to the task concerned. Figure 2.10 depicts such a process diagram. The dotted arrows represent control connectors (flow of control between two activities) while the plain ones represent data connectors (flow of data between two activities). The green circular wheels represent some program activities, while the one in the square represents a block, which is a set of activities that can be repeated until an exit condition is met.

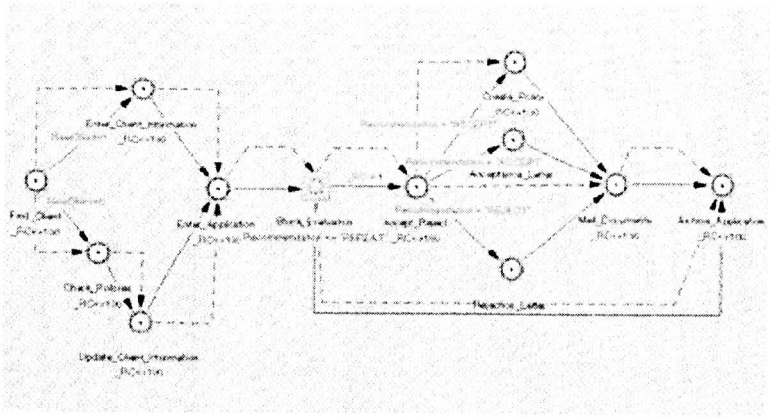


Figure 2.10: Process diagram in FlowMark

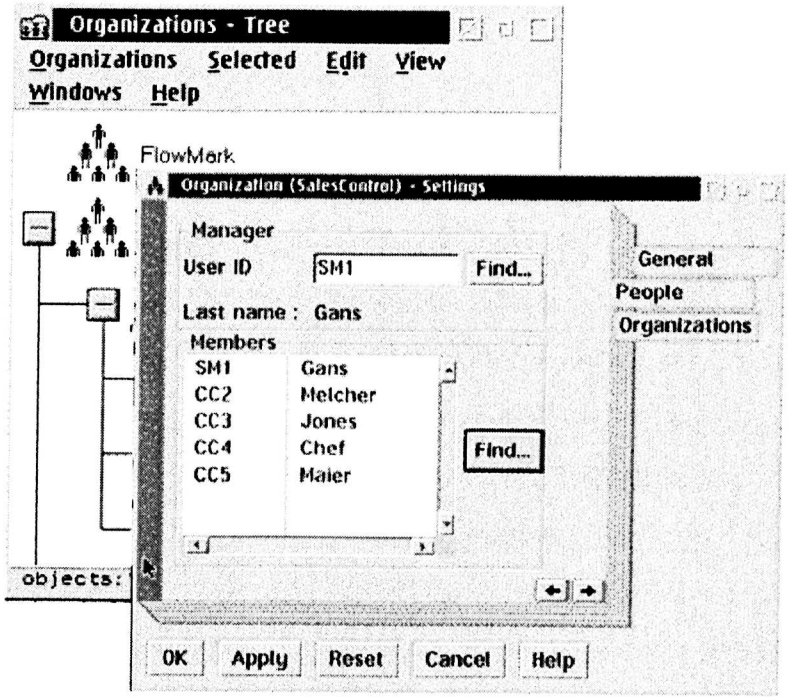


Figure 2.11: Specification of an organisation

An editor to model your organisation is also provided (figure 2.11). It also allows the definition of staff (figure 2.12) and the assignment of roles to them. In FlowMark, a role is a function or ability that a person or a group of persons have. In figure 2.11, information on an organisation is being edited. The option shown is the list of persons, member of this organisation (sale control). Organisations are hierarchically organised as a tree. A similar editor allows the administrator to assign persons to a role such as sales person, secretary, etc. There is a many-to-many relationship between roles and persons.

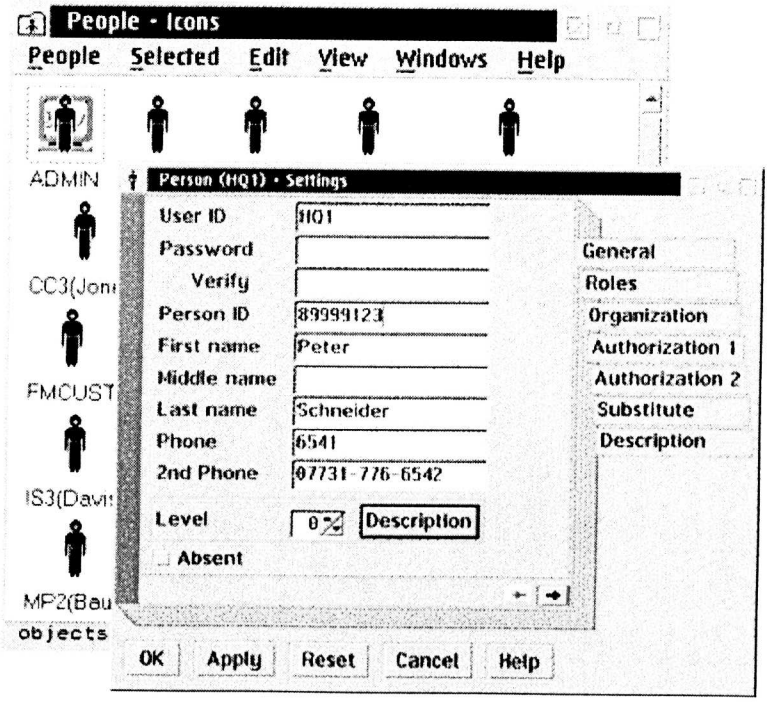


Figure 2.12: Specification of a person

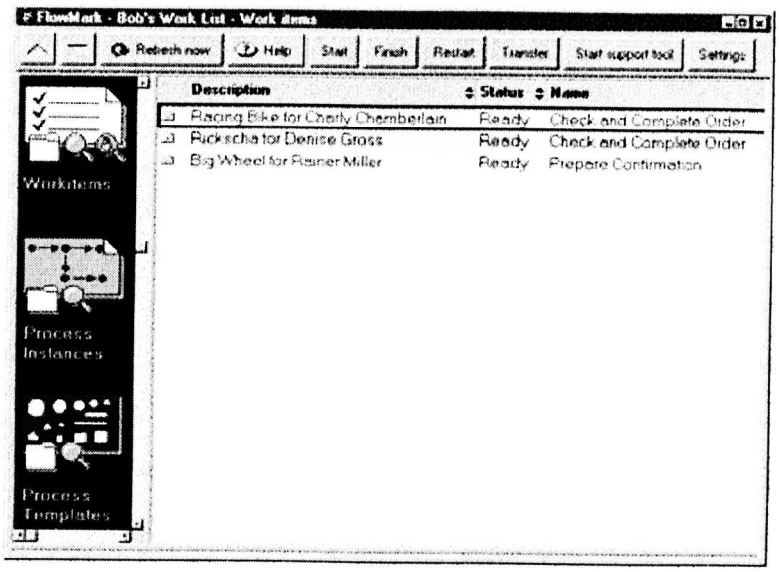


Figure 2.13: Worklists in FlowMark

In figure 2.12, the substitute option allows the administrator to specify another person who can substitute the person whose record is edited.

It can distribute tasks by people, roles, levels or organisations. The association of roles, levels and organisation to people is resolved dynamically. Imagine that a task has to be done by a certain type of person, the task does appear in their worklists as depicted in figure 2.13 and as soon as one person from that group accept it, it disappears from the others worklists.

Worklists can be consulted from Lotus Note. FlowMark is also responsible to invoke the right application to be used to carry out a task.

Using the GUI, the administrator of the workflow can simulate its application, as well as use the audit trail to debug its specification. When simulating the application, he has to take action on behalf of the program or persons responsible. The specification has to be debugged manually. He can also monitor the progress of its application. He can also check the status of the tasks, including who (if any) is dealing with it. In the example depicted in figure 2.14, the activity "prepare and deliver is active" and somebody has just accepted to take care of it. This is shown on the picture by having one of the group members with a different colour from the rest of the group.

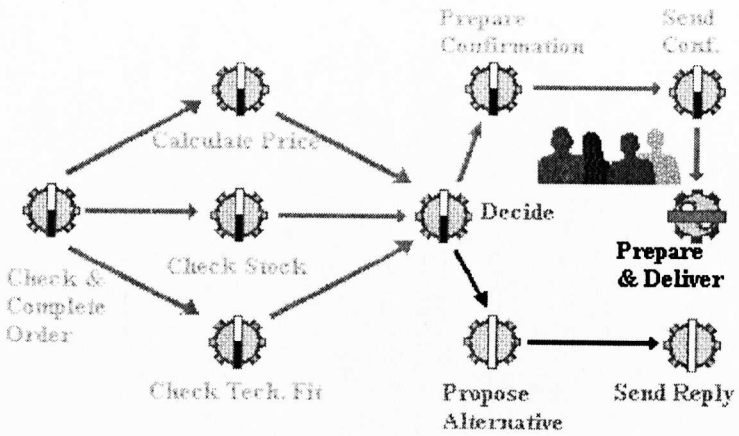


Figure 2.14: Monitoring of a FlowMark workflow

2.3.5 RainMan

This IBM project [63] aims at supporting decentralised workflow execution, as well as interoperability and dynamic modification. RainMan is a distributed workflow system for the internet implemented in Java. It is based on the RainMaker generic framework that defines a core of abstract interfaces for workflow components. RainMaker has four main abstractions: the workflow instances (sources, service requestors), activities (service requested), the performers (human, applications... in charge of executing the activities), and finally tasks that are the units of work managed by the performers and implements the activities. Tasks are sent to performers independently of their implementation for interoperability reasons.

The RainMan system itself is a collection of lightweight services implemented using Java RMI (Remote Method Invocation). The services implemented are a builder tool, a directory

service, a repository service, a work list service, a work list client and an administrator tool. The builder tool allows users to specify a workflow as a directed, acyclic graph, and then to monitor it. Performers are assigned to the activities specified by querying the RainMan directory service. These specifications are stored and retrieved from a repository service of the system. The builder is also a graph interpreter that generates the sources. As a result, the specification (the graph) can be modified at run-time allowing dynamic reconfiguration. A specification language based on directed acyclic graphs is also provided. The work list client is provided for an easy access of a human work list (implemented as a persistent FIFO queue), the client can connect to the (distributed) work list service to view a work list and select some task to do locally and disconnected. The client just has to reconnect to let the work list to return the activity results. The directory service is both a naming service and a trading service and contains information on the different performers.

The application level fault tolerance is addressed with forward recovery using compensation activities. Performers are expected to provide support for compensation for the activities that they handle.

2.3.6 TOWE, Transaction-Oriented Workflow Environment

This project [54] provides facilities for the construction and coding of long-lived concurrent, nested, multi-threaded activities. The idea behind this work is to provide a software development environment for workflow management system. The environment is based on flexible transaction models (usually data-centred) extended to support process-centred activities, by unifying the notions of class and process. The programming of these applications is based on library modules representing abstractions of system aspects and functionality of long-lived activities in a concurrent object-oriented environment. These libraries are built from a small set of fundamental concepts that are extensions and refinements of open nested transaction constructs. A prototype of TOWE has been developed on top of two database management systems: Oracle and the object-oriented prototype OBST. The overall architecture of TOWE is depicted in figure 2.15.

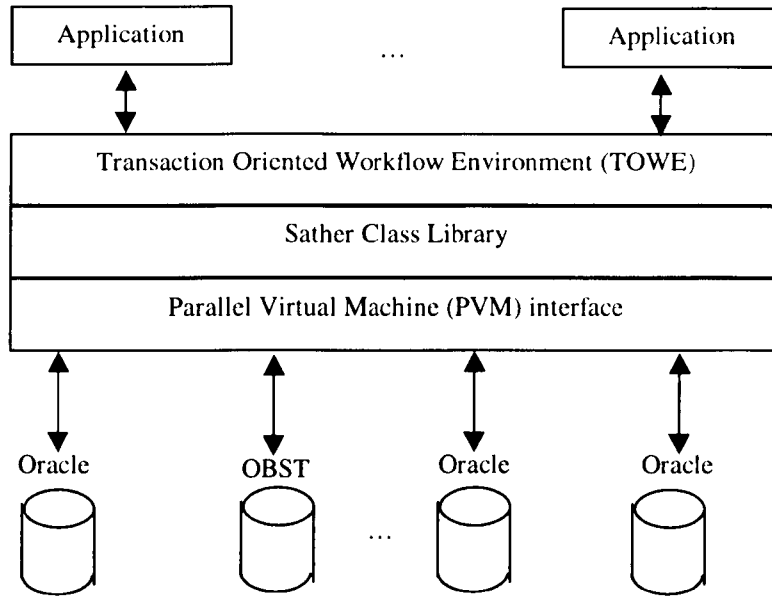


Figure 2.15: The TOWE system architecture

A workflow in TOWE is a long-lived activity co-ordinating the execution of multiple process-oriented tasks with transactional properties, which are related by data and control flow dependencies. A long-lived activity in TOWE is divided into some work units nested to multiple levels, and executing sequentially or concurrently. Leaf level work units of the activity tree are called actions while intermediate nodes are referred as intermediate activities or compound actions. The actions are atomic unit of work mapped to flat ACID transactions. They can be vital or non-vital, a vital action aborting forcing its parent to abort. A scheduler process manages the flows of control and data between work units. Four types of scheduler processes are provided: serial (begin on commit dependencies), parallel, serial-alternative and parallel alternative scheduler. The latest two types are processes attempting actions sequentially or in parallel until one of the alternative succeeds. The actions of a serial scheduler may have data object (the target action is awaiting for an entire data object) dependencies while the actions of a parallel scheduler can have value (the target action is awaiting for a value to be sent from the source action) or commit (the target actions can only commit once the source action has committed) dependencies. Schedulers can also have conditional actions as well as replicated ones.

The programming language is the object-oriented language Sather [51] interfaced with PVM (Parallel Virtual Machine) [19]. PVM brings some support for distributed programming and message passing. TOWE provides a library of classes that can be extended and specialised and provides some support for the specification of temporal, value and data object

dependencies among activities, exception handling mechanisms, commit dependencies among actions, compensating actions, contingency actions, as well as ordinary and express message (not queued). This is shown in figure 2.16. The system distinguishes five groups of classes: distributed system and communication support, transaction primitive, atomic transaction, scheduling and application program classes. The relationships between these classes are described on the figure below.

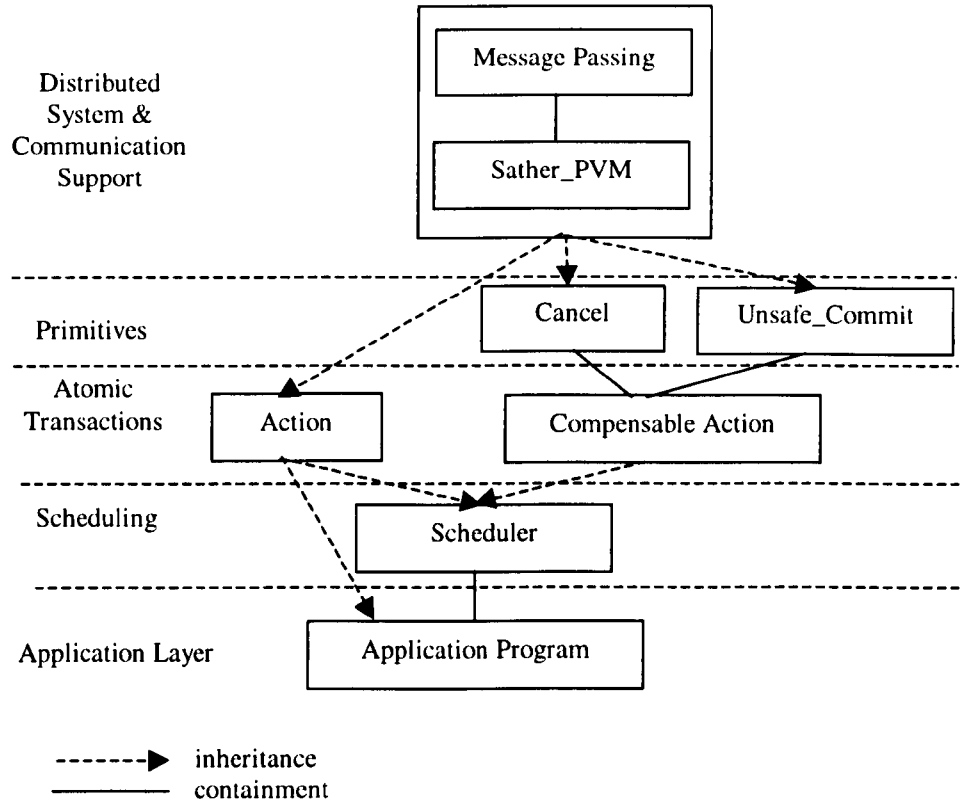


Figure 2.16: Library classes of the TOWE

After instantiation, the process object (instance inheriting from a scheduling class) executes its work routine (piece of code between the keywords "work is" and the keyword "end") which describes the sequence of actions it runs during its lifetime. The work statement describes the behavioural part of a process object, which provides the means to create other processes, actions and objects at remote sites; and to request asynchronous execution of their features and to communicate with them. Calling an action would typically be done using the following code:

```
action.name("name of action").arg(List of arguments)
```

Proxies are used at run-time to perform remote calls. Mutually exclusive access to shared data object is provided. Due to the usage of shared objects, intermediate results do become

visible leading to the introduction of the unsafe commit notion. Unsafe commitment of an action requires logging enough information to undo it before actually committing and releasing the locks held. Several versions of the data items are kept in a stack in case of cancellation of an action involved in modifying the item.

Scripts are used to specify the workflow application. A Web GUI is being developed using CGI scripts and forms.

2.4- Discussion

Workflow Management Systems have been implemented using a lot of different infrastructures such as Lotus Notes [61], TP-monitors [83], the web [10], CORBA [70], DCE [66], Customised Transaction Management [20], however a certain number of features are common to these systems.

System	Model	Fault tolerance	Dynamism	Interoperability
Sagas	Serialised transactions with associated compensating actions	Save points & compensations	None	Homogeneous
ConTract	Group of transactions	Compensations, Steps seen as transactions	None	Homogeneous
ORBWork	CORBA workflow system	Recovery managers using persistent storage & application level	Some	CORBA Web
Exotica	Message based workflow management	Persistent messages, atomicity of changes not guaranteed	None	Proprietary
RainMan	Sources co-ordinating performers executing the process	Persistent worklists, long-run conversations being considered.	Dynamic updates of workflow graph	Written in Java, heterogeneous environment
TOWE	Transactional Workflow system	Basic units of work are ACID. Open-nested transactions.	None	Homogeneous
Newcastle	Transactional CORBA workflow system	Both system and application level (alternative), persistent storage	Full dynamic updates of specification	CORBA, heterogeneous environment

Figure 2.17: Comparison of the features supported by the different systems considered

The usual weakness of workflow systems are the limited support for heterogeneous and distributed environments, the lack of interoperability, and of support for reliability, as well as the lack of dynamic reconfiguration. Our system intends to propose a solution to these problems. It can deal transactional and non-transactional tasks contrary to TOWE that has ACID basic tasks. It also uses persistent storage. It provides support for changes at run-time, which a lot of systems don't provide. Our alternative inputs and outputs also bring some more

flexibility for exception handling, by allowing an homogeneous specification of the handlers for faults.

Chapter 3

Architecture

In this chapter, we will present the architecture of our workflow system [59], [60]. First the requirements for our workflow system will be listed as well as the approaches adopted to deal with them. Then the overall software architecture will be described before ending up by describing the components of the architecture as well as the execution environment [82].

3.1- Requirements

The workflow management system that is described in this thesis addresses the requirements that were listed in the introduction. The main requirements are modularity, scalability, interoperability, dependability and dynamic reconfiguration. In the following sections, we will describe the requirements as well as the approaches adopted to fulfil them.

3.1.1 Modularity

The specification of a workflow application should be modular. Indeed, it should be possible to decrease the complexity of the workflow application being described by using several modules dealing with simpler part of the application. The system deals with modularity by providing the notion of compound task. A compound task consists of a set of workflow tasks gathered together. The reasons to gather tasks together can be multiple: you may want to show a set of tasks as a single task thereby hiding the details of these tasks (for instance in a student registration workflow, the task adding a new student to the system is likely to include a task creating of a login account. The task creation of a login account can be itself sub-divided into creation of the new user system identity, adding the new user to a group of users, sending the user a welcoming message...). Another reason is that you may just want to make a task

fault tolerant (for instance grouping a task and its alternative(s) in a single task to hide the details of fault tolerance handling). A compound task can be composed out of other compound tasks hence allowing arbitrary nesting. This provides a way to add some levels of abstraction in the specification of your application. The tasks included in the same compound task will be referred thereafter as *peer tasks*, and the compound task embedding them as their *parent task*.

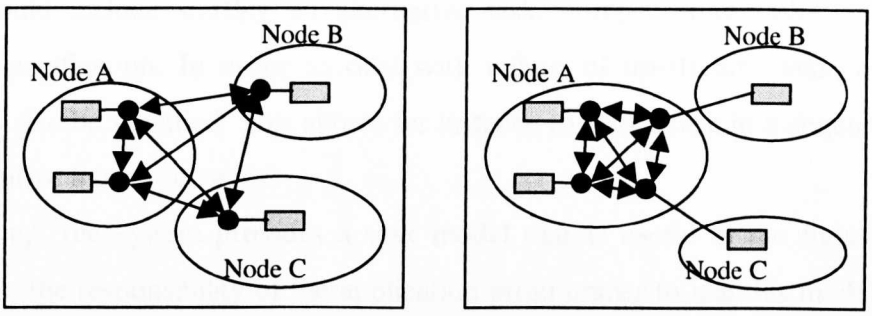
3.1.2 Scalability

As business processes becomes more and more automated, it is likely that workflow applications will increase both in size, complexity, and will span across administration boundaries. As a result the workflow management system has to be able to cope with an increasingly large degree of distribution, as the resources required by a workflow could be located at arbitrary places.

In order to cope with scalability, the Workflow System has first to avoid reliance on any centralised service as it is likely to add performance bottlenecks to the workflow application and add a central point of failure. Our system addresses scalability issues by only sending the minimal number of messages needed to support the application proper execution. It does that by only sending some messages of notification of events to the task controllers that have registered their interest in the event. We have opted for a model where each unit of work (task) has attached to it a manager (referred to as task controller in the rest of the thesis) that is taking care of the co-ordination of the task. In particular we have avoided to use a global (centralised) co-ordinator that decides how to schedule the steps of the workflow given the history of event. Each controller is responsible for starting its associated task once a set of preconditions (input dependencies) is satisfied and for delivering the outputs of the task to other tasks as specified in the workflow specification.

Furthermore to cope with potential problems of performance and to allow more flexibility, it would be a nice feature to be able to control the location of the task controllers of an application on request to optimise the communications between task controllers by a trade off task controller located on same node as associated task versus grouping of task controllers that are highly dependant of each other. In figure 3.1, we present different policies used to place the task controller. Tasks are represented by a grey rectangle, and are linked to their controllers represented by black circles. These tasks are distributed across three nodes. A centralised co-ordination approach would be to gather all of the controllers on a single node while a

distributed approach would be to have them on the node where their associated tasks are running. Ideally we would like to have a workflow management system allowing the users to be able to choose their location policy depending on what they are aiming at. It is likely that a real application will not be fully distributed nor fully centralised but just partly distributed. Our system aims at providing different levels of distribution of the control of the tasks to be able to cope with the different policies that users may want.



Distributed co-ordination Centralised co-ordination

Figure 3.1: Distributed and centralised co-ordinations

3.1.3 Interoperability

The workflow management system has been structured as a set of CORBA services running on top of a CORBA compliant ORB. CORBA described in section 3.2.1 brings the support for interoperability needed.

3.1.4 Dependability

The fault model that our system assumes is the possibility of crash failures of nodes, as well as network partitions. Failing nodes can eventually recover, and network partitions eventually heal.

The system provides both system and application level support for dependability. The system level support has been achieved by using persistent storage for recording inter-task dependencies and transactions for the delivery of task outputs to their destinations. As a result the destination tasks receive references to their input objects despite presence of failure such as temporary network partitions or temporary node crashes. Finite number of retries of affected transactions achieves this. This is the system level fault tolerance measure used to ensure

forward progress of applications.

Support for application level fault tolerance is provided by the task model that provides alternative input sets and output sets as well as multiple sources for inputs of all tasks and outputs of compound tasks (see section 3.3). These facilities provide powerful underlying application level exception handling capabilities to cope with the errors not handled by the underlying system such as for instance a task that despite a finite number of retries does not start. A task can terminate in a "normal state" or in one of "exceptional state". Exception handling could include starting an alternative task, compensating tasks..., as part of the workflow specification. In order to deal with failure of up-stream tasks, alternative input sources can also be specified. This allows for instance to start a task in a degenerate state using these alternative input sources.

To sum up, the system provides a task model that is useful in the fight against faults ; however it is the responsibility of the application programmer to use this model to incorporate application-level fault tolerance. Alternative input sets are seen as a way to let the programmer carry on despite failure of up stream tasks. Output sets as a way to deal with unhandled exceptions, by only allowing one output as a "normal" state, while the other correspond to "exceptional" states. By adding extra compensating tasks to try to cope with these exceptions, the programmer can add 'forward recovery' in the application. Chapters 4 and 5 describe these aspects in detail.

3.1.5 Dynamic reconfiguration

In an environment where application requirements are likely to change during run-time, a workflow management system needs some capability for reconfiguring the structure of the application. Mechanisms are needed to allow forward progress of the workflow despite changes to the environment calling for a modification of the specification. For instance, imagine that a service of document translation that was used by the workflow application disappears while the workflow is executing, and that it is possible to create an equivalent service by using two translation services in sequence. Then you would like to be able to replace the task that no longer exists by say a compound task composed out of these two services in sequence. Of course, all these modifications should be done without having to abort the workflow application. As a result, there is a need of being able to modify the composition of the workflow application at run-time. In our system, addition and removal of tasks composing

the workflow and dependencies between the composing tasks are carried out by using transactions. This ensures that the modifications of the specifications are done atomically. Changes of the instantiation criteria associated to a task are also supported. Details of how the toolkit allows you to perform dynamic reconfigurations of your application can be found in chapter 6.

3.2- Software structure

The system has been designed to support specification, execution, monitoring and control of workflow applications. The implementation consists of a set of CORBA services and applications. These services and applications are grouped to form the five main software components of the workflow system. The software structure of the workflow management system is depicted in figure 3.2.

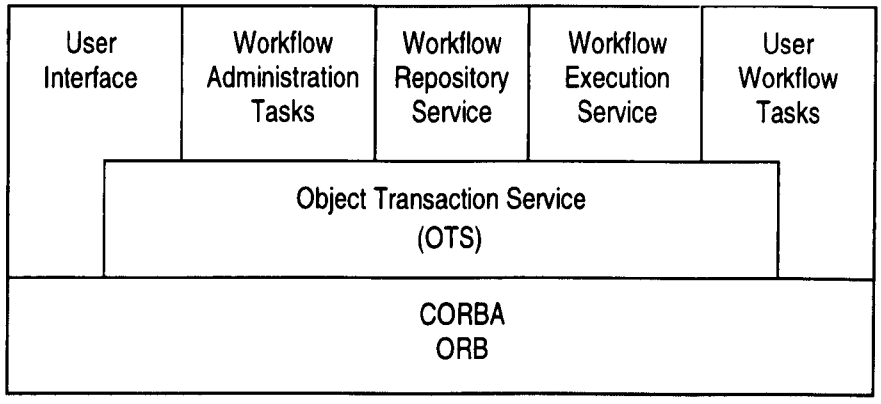


Figure 3.2: Software structure of the toolkit

To allow interoperability with future system and applications the workflow system can make use of existing CORBA service such as the transaction, the security or the naming services. At present only the naming and object transaction service have been integrated into the workflow system. The Object Transaction Service has been provided by the Arjuna distributed transaction system [55], which has been adapted to be OTS compliant and capable of running on a given ORB.

The relationships between the software components of the workflow system are depicted in figure 3.3. In the rest of this section we will describe, in more detail, the components of the workflow system.

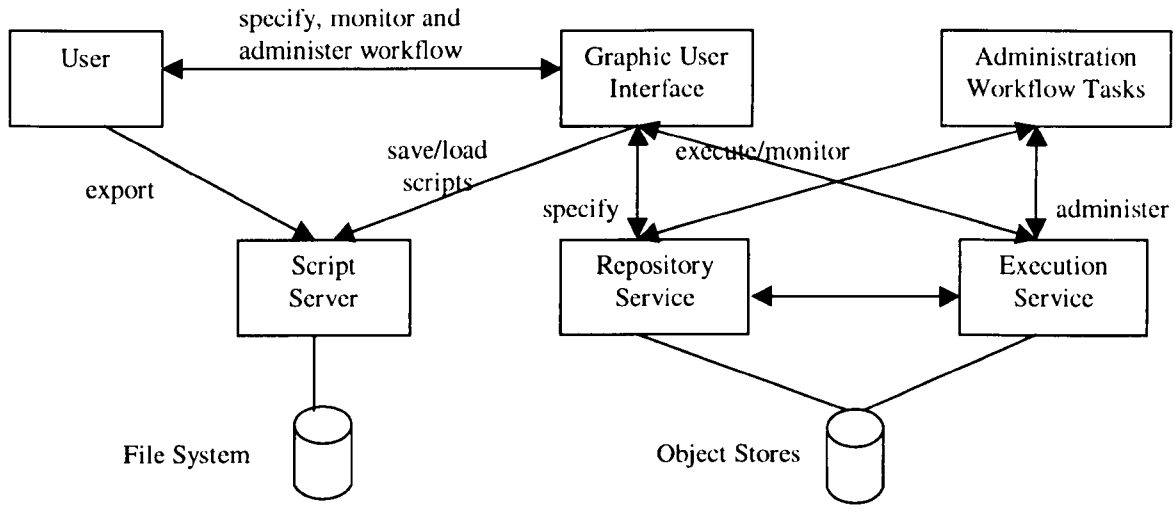


Figure 3.3: Relationship between the different components

3.2.1 Common Object Request Broker Architecture (CORBA)

CORBA is the core of the Object Management Architecture (OMA) [47] proposed by the Object Management Group (OMG). The OMA was designed for distributed object systems, and is a set of services organised around a software bus called an Object Request Broker (ORB). The architecture and specification of this ORB can be found in the CORBA specification [48]. CORBA is structured to allow integration of a wide variety of object systems. The ORB is responsible for providing some virtual homogeneity regardless of the programming language, operating systems, tools and networks used to realise and support components specified using the OMG Interface Definition Language (IDL) which is described below.

The reference model (the OMA) depicted in the figure below, consists of four components:

- *Object Request Broker* responsible for transparent sending and receiving requests and responses by objects in a distributed environment. This is the basis for the creation of distributed applications and for inter-operability between such applications in an homogeneous or heterogeneous environment.
- *Object Services* [49], a set of fundamental services providing basic features for using and implementing objects. These services are supposed to be modular so that clients can use as many or as few as needed. These include the Object Transaction Service (OTS), the Trading Service, the Naming Service, the Life Cycle Service, the Event Service...

- *Common Facilities* [50]: a set of facilities that can be shared by applications. These services are not as fundamental as the object services. The Workflow Management and the Business Object Management Facilities are relevant to our work and are under discussion [45][30].
- *Application Objects*: the uppermost layer of the Reference model, in fact some products developed by vendors... compliant to the OMG specifications.

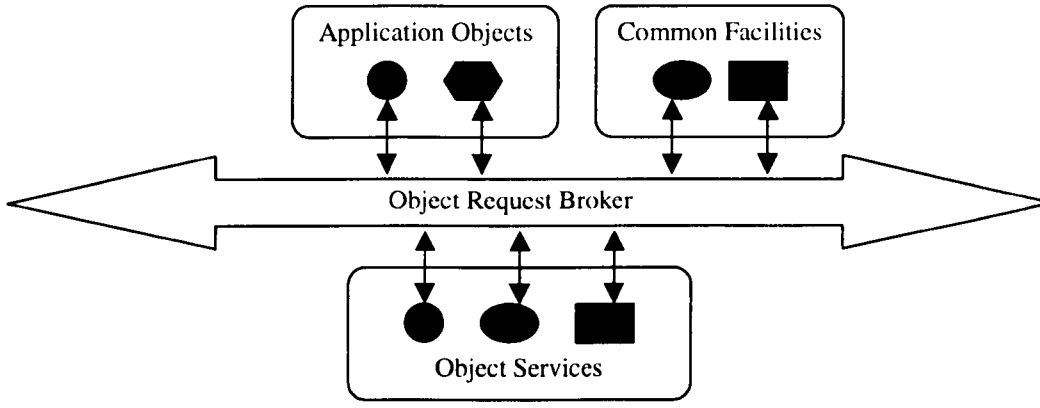


Figure 3.4: Object Management Architecture

The structure of an ORB is depicted in figure 3.5. The client (the object invoking an operation) sends a request to an object implementation (code and data implementing the object). The ORB is responsible for the mechanisms needed to find the correct object implementation for the request, to prepare it to receive the request and to communicate the data making up the request. The interface that the client sees is totally independent of the location of the object, of the programming language in which the object was implemented... To make the request the client can either use the dynamic invocation interface (the same interface independent of the target's object interface) or an OMG IDL stub (dependent on the interface of the interface of the target object). The client can also interact directly with the ORB for some functions.

The Object Implementation receives the requests as an up-call via the OMG IDL generated skeleton or via a dynamic skeleton. Skeletons are specific to the interface and object adapter. During the processing of the request, the object implementation may request some service from the object adapter or the ORB. Object adapters are the primary ways that an object implementation can access services provided by the ORB such as the generation and interpretation of object references, method invocations, object and implementation activation and deactivation... It is also possible at other times. When the request is completed, control

and output values are returned to the client.

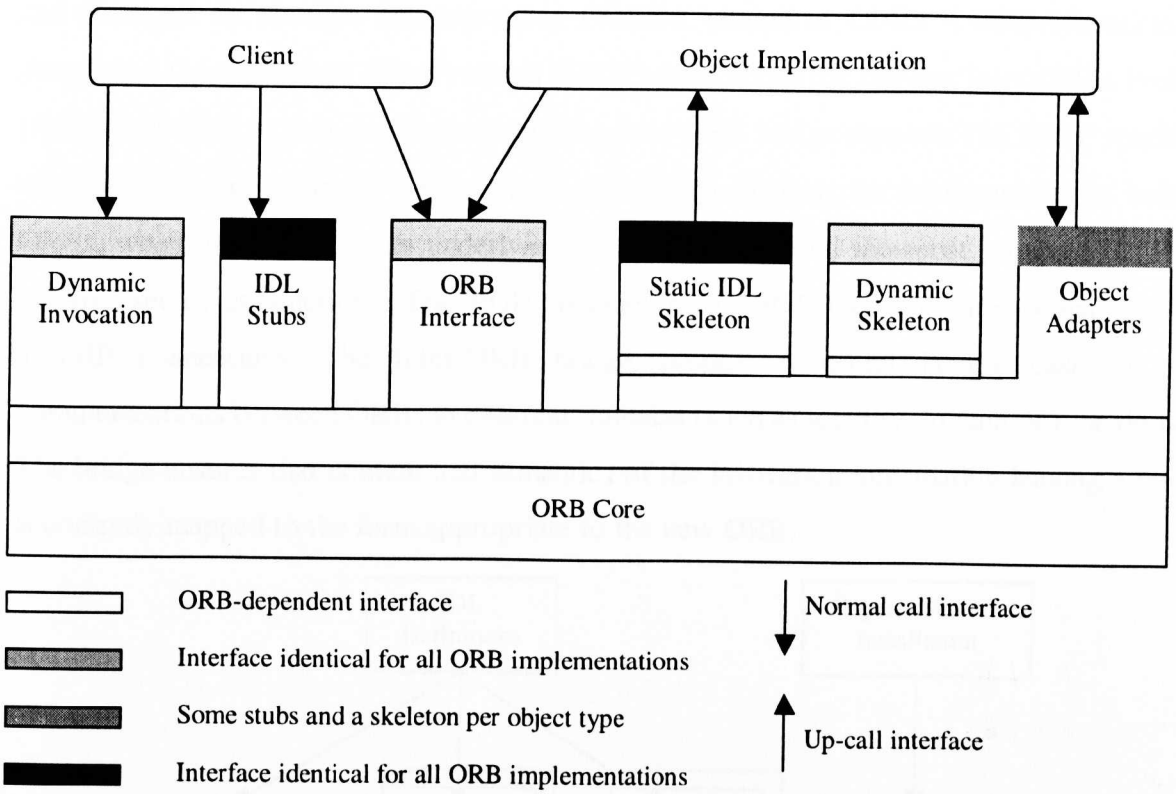


Figure 3.5: Structure of an Object Request Broker

The OMG provides an Interface Definition Language (IDL) to enable the description of the interface to a CORBA object. The interface to objects can be defined either by using the OMG IDL or by being added to an interface repository service that provides persistent objects to represent the IDL information under a run-time form. IDL is a neutral language (programming language independent) using the same lexical rules as C++ to which some new keywords have been added to support distribution concepts. IDL specifications can then be translated to several programming languages using the OMG standard language mappings. The IDL interface of an object describe the interface that the object implementation provides, or in fact which operations client objects can invoke. This provides the information that clients need to be able to use operations on a CORBA object independently of its implementation. Figure 3.6 shows how interfaces and implementation information are made available to clients and object implementations. The definition is used to generate the client stubs and object implementation skeletons.

The implementation repository contains information allowing the ORB to locate and activate implementations of objects.

ORB interoperability specifies a flexible approach to support networks of objects located and managed by multiple heterogeneous CORBA compliant ORBs. CORBA specification describes a General inter-ORBs Protocol (*GIOP*) from which the Internet Inter-ORBs Protocol (*IIOP*) is derived. It also provides support for inter-ORB bridge support. The GIOP specifies a standard transfer syntax as well as a set of message formats for communications between ORBs, which only requires an underlying connection-oriented transport protocol fulfilling a minimal set of assumptions. The IIOP specifies how GIOP messages are exchanged using TCP/IP connections. The Inter-ORB bridge support is useful in the case of direct communications between ORBs in different domains (such as security domain or type domain). The bridge ensures that content and semantics of the invocation information leaving a domain is correctly mapped to the form appropriate to the new ORB.

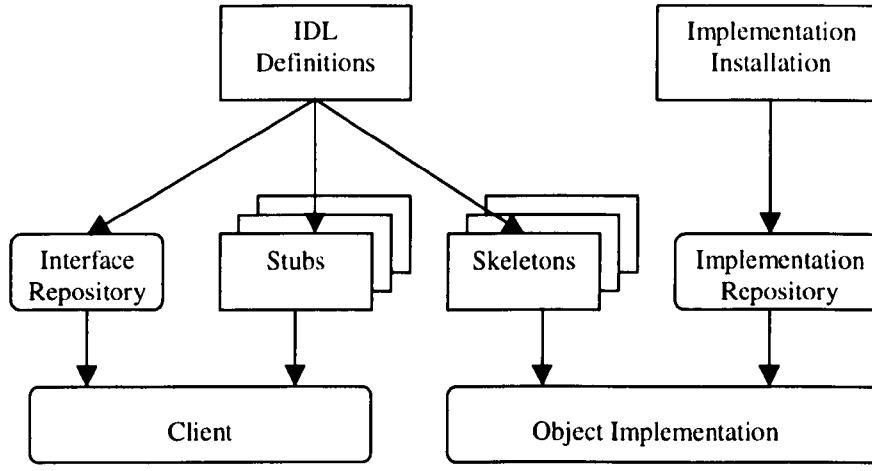


Figure 3.6: From IDL specification to implementation and use.

3.2.2 Object Transaction Service (OTS)

The OTS is one of the services defined by the OMG and specified in the OMA. The architecture of the OTS is depicted in figure 3.7.

The Transaction Service provides transaction management services and transaction propagation protocol through well-defined interfaces. The ORB provides communication support for transparently invoking operations on objects and receiving the results. The transaction originator is the client creating the transaction and invoking operations. A recoverable object is a transactional object defined as an object whose data is affected by the commitment or the rollback of a transaction. Such an object must participate in the Transaction Service protocols. This is achieved by registering an object called *resource* with the transaction

service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction. A recoverable server consists of one application object or more and one resource object or more registered with the Transaction Service. It issues requests to these registered resource objects to drive the commit protocol. Several objects can take part in a transaction. They shared a transaction context, which defines the scope of the transaction. To simplify coding, most applications use the current pseudo object, which provides access to an implicit per-thread transaction context. The recoverable server can also register a resource called subtransactionAwareResource to keep track of the completion of sub-transactions.

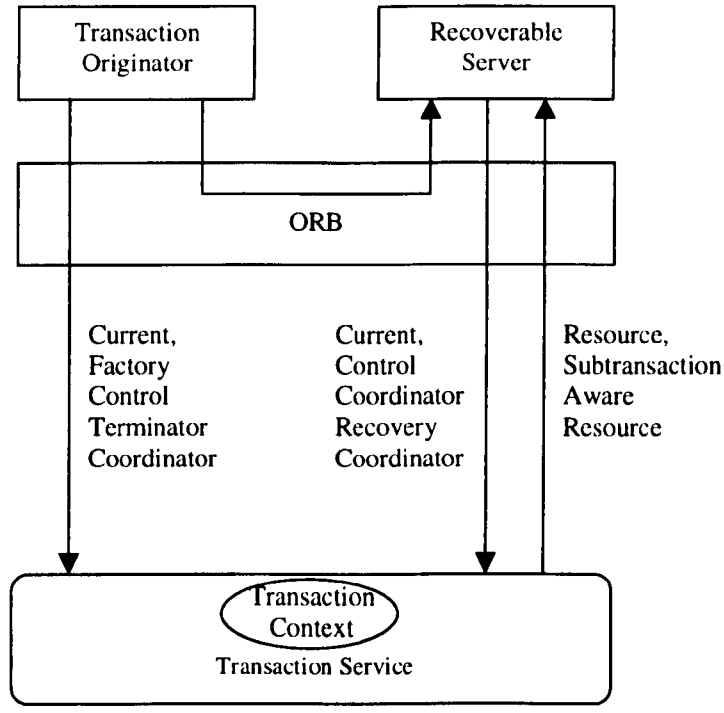


Figure 3.7: OMG OTS architecture

A transaction is executed as depicted in figure 3.8 and can be divided in five steps:

1. A client invokes a transaction service via the factory or the current interface to start a transaction. The transaction service creates a transaction context and a globally unique transaction identifier.

2. The client invokes the server including the transaction propagation context as parameter (implicit or explicit) which contains the transaction unique identifier as well as the transaction service object reference.

3. On reception of an object request the server registers its resource objects with the

transaction service via the co-ordinator interface.

4. The client terminates the transaction (commit or rollback) by using the terminator interface of the transaction service.

5. The transaction service propagates the completion decision by using two-phase commit requests to the registered resources via resource interfaces.

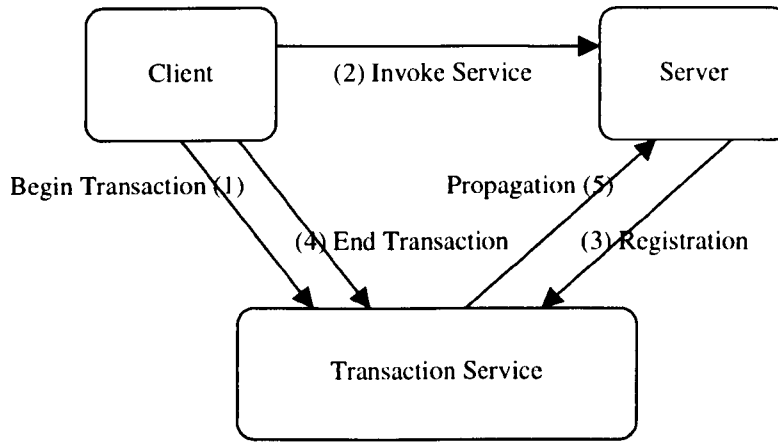


Figure 3.8: OMG OTS execution flows

Interoperability between heterogeneous transaction services as well as co-operation between multiple transaction services is also provided by the OTS specification (feature known as interposition).

3.2.3 Graphical User Interface

This component allows users to interact with the rest of the workflow system. It allows users to graphically specify workflow applications, start them and monitor their behaviour. The Graphical User Interface (GUI) also enables users to load existing workflow scripts that could have been written off-line. This specification can then be checked for consistency. Simulations of the workflow application specified can also be carried out to try to find out potential problems. Once exported to the repository service, the workflow application can be instantiated. The execution of this instance of the workflow application is then controlled by the execution service and can be monitored using the GUI. During the execution dynamic modifications can be initiated from the GUI.

3.2.4 Workflow Repository Service

This component is responsible for maintaining the specifications of workflow applications in

a form that can be used by other workflow applications. This way of storing workflow application is equivalent to a script specification. This service provides operations to create, modify, delete and inspect specifications. A workflow specification kept in the repository service is called workflow schema.

3.2.5 Workflow Execution Service

This component is responsible for co-ordinating the execution the basic units of work (referred as basic tasks in the rest of the thesis) that form the workflow application being run. This service has to be reliable to be able to cope with temporary network failures, node crashes... In order to do that, it records inter-task dependencies in shared persistent atomic objects.

3.2.6 Workflow Administration Tasks

These tasks are designed to manage other workflow applications. Provision has to be made to provide some tasks starting workflow applications, terminating tasks that no longer need to be run as their outcomes are no longer needed, or dynamically modify the specification of workflow applications. Dynamic modifications include the addition, removal of tasks or dependencies, modification of the code mapped to a basic task...

3.2.7 User Workflow Tasks

These tasks are applications that have been built by users. They are built as a set of basic tasks linked between each other by dependencies. These tasks can be implemented in any language. A wrapper is then added to map them to the basic tasks of our system.

3.2.8 Script Servers

They are used to store the textual (ASCII) specification of workflow applications. The main reason for their existence is to give more flexibility to the users by allowing them to store some uncompleted specifications enabling the storage of workflow specifications being built.

They allow users to store or retrieve directly textual specifications that can then be loaded into the specification service if correct. It allows off-line specification of workflow applications.

3.3- Task model

In this section, the structure of a workflow task as well as the different types of tasks will be described. As previously stated in section 1.1, workflows can usually be divided into smaller units of work (called tasks) carried out by participants. Participants have to collaborate to reach a common aim, the achievement of the global process. This collaboration is usually carried out by exchanging data and by ensuring that the dependencies between units of work are respected.

3.3.1 Structure of a task

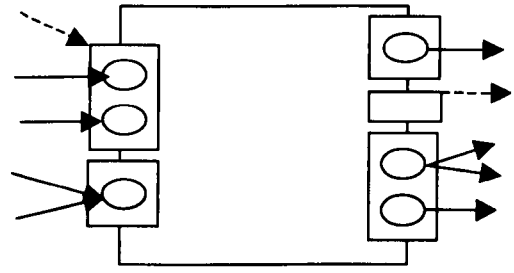


Figure 3.9: Representation of a task

A task is represented in the system by its interface. The only visible parts of a task are its inputs and its outputs. The internals of a task are hidden except when this task is itself composed out of other workflow tasks. In this case, the dependencies among these tasks as well as the mapping between the inputs and outputs of the embedding task and its components are described. The structure of a task is depicted in figure 3.9. At run time, a task will typically gets some inputs and then terminates producing some outputs. To add some flexibility, alternative sets of both inputs and outputs can be specified.

The inputs and output sets are represented by rectangle boxes, the input and output objects by ovals, the data flow dependencies by arrows and the notification dependencies by dotted arrows. The direction in which the arrow leads shows whether it is a dependency with this task as source or as destination. The overall specification can be represented as an acyclic graph. In the rest of the thesis the tasks that are having dependencies on a task will be referred to as *down-stream tasks*, while the tasks on which this task depends will be referred as *up-stream tasks*. The domain of a task specification or in other words, what is described in the specification of a task is depicted in figure 3.10. The grey box delimits the part of workflow specification associated to the task.

It has to be noticed that our system has as particular feature that the task is only aware of the dependencies it has on up stream tasks. It has no knowledge whatsoever on which down-stream tasks are using it as source of dependency. This make is possible to address the issue of locality of modification: if a user wants to modify a task dependency this is done locally at the level of the concerned task.

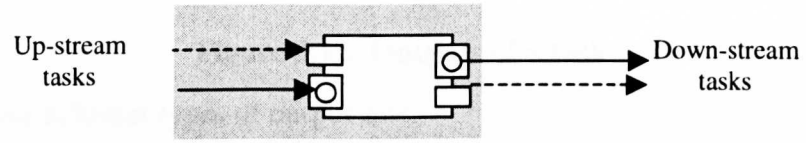


Figure 3.10: Domain of a task specification

Inputs

A workflow task has the possibility to have one or more input sets (represented in figure 3.11 by the boxes in light grey).

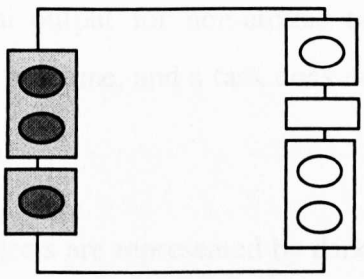


Figure 3.11: Inputs of a task

These sets are alternative and have some input objects associated to them. In figure 3.11, these input objects are represented by dark grey ovals. The first input set has two associated input objects and the second input set has just one input object. Each of these input objects has a list of references on other input/output objects that can be used as alternative input sources. In order to start, a task needs to have the totality of the input objects of one of its input set available. An input object becomes available when one of its input alternatives is available. In the event that several input sets become available, the first one listed is chosen.

Outputs

A workflow task has the possibility to have one or more output sets (represented in figure 3.12 by the boxes in light grey). These sets also referred as output states or outcomes are alternative and have some output objects associated to them.

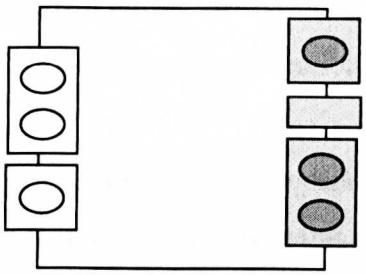


Figure 3.12: Outputs of a task

There are four different types of output sets:

- *Final outcome*: the normal output for a task.
- *Abort outcome*: a final outcome for atomic tasks, it only allows output objects of type error corresponding to some error codes.
- *Repeat outcome*: a special output for loop tasks allowing the outputs to be fed as inputs for the next iteration of the loop.
- *Mark outcome*: a special output for non-atomic tasks allowing publishing partial results. This is an intermediate outcome, and a task does not terminate whenever it reaches such an outcome.

In figure 3.12, these output objects are represented by dark grey ovals. The first output set has one associated output object, the second one none and the third output set has two output objects. Once started, a task has to end up in one of its output sets and the output objects associated to the chose output set become available to all tasks having some data flow dependencies involving them as source. In the event that several outcomes of a compound task become available, the first one listed is chosen.

Dependencies

There are two types of dependencies considered: data-flow dependencies and temporal dependencies. Data flow dependencies are dependencies where an input/output object reference on an object from task A (called *source object*) is given as alternative to an input/output object of another task B (called *destination object*). The meaning of a data flow dependency is that the destination object is allowed to use the source object as alternative. In other words, the destination object becomes available as soon as the source object is itself available.

There are seven types of possible couples of source-destination objects depicted in figure 3.13. In this figure, we have chosen as naming convention to call the object source S and the object destination D.

- (a) The obvious data flow dependency is a dependency between two peer tasks A and B (Peer tasks were defined in section 3.1.1). S is in this case an output object of task A, and D is an input object of task B. This shows that task B needs to use some of the results of task A.
- (b) Another form of data-flow dependency between two peer tasks A and B is depicted in figure 3.13 b. S is in this case an input object of task A, while D is an input object of task B. This can be used when a task (B) needs to use the same input object as the input object from another task (A).
- (c) This time task B is task A's parent, the object source is an input object of task A, and the D is an output object of task B. It has to be noticed that task B has to be a compound task in this case. This is used to transmit some results back to the parent task.
- (d) This type of dependency is similar to the previous one, but this time the object source is an output object of task A. It has to be noticed that task B has to be a compound task in this case. This is used to transmit some results back to the parent task.
- (e) A data flow dependency can also be between task B and its parent task A (as defined in section 3.1.1). In this case B is a task embedded in the compound task A, S is an input object of task A and D is an input object of task B. This is used to transmit some object references received by the parent task.
- (f) In this case A and B are again referring to the same task. In this case, it has to be a compound task, which has D as one of its output object and C as one of its input object. This can be used for instance to return an input object as output object in the event of an abnormal execution of the task.
- (g) A data flow dependency can also involve only one task (tasks A and B refers to the same task). In this case, the task needs to have a repeat outcome that can be used as source object. D is in this case an input object of the task. This is used to simulate loops.

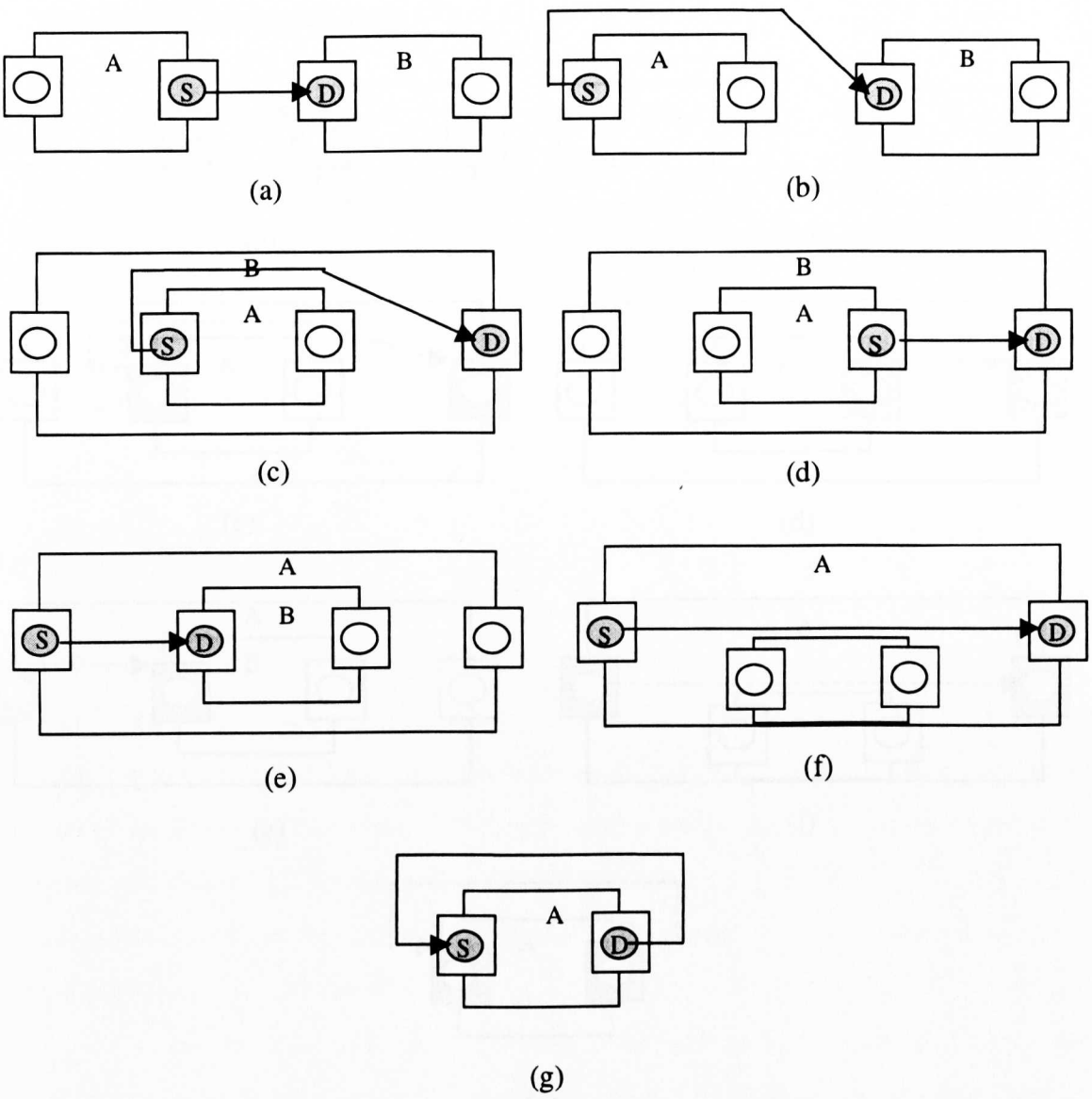


Figure 3.13: Types of data-flow dependencies

There are also seven different types of notification dependencies, depicted in figure 3.14. Similar conventions are used as the one used for data dependencies. A is the task with the source set and B the task with the destination set. The sets were named respectively S and D.

- (a) The notification dependency with as source an output set belonging to task A that is a peer of task B. D is a destination input set. This could be used for instance to enforce that task B will only start after completion of task A in a certain state.
- (b) A notification dependency can also use as source an input set from task A and as destination an input set from peer task B. This could be used to specify that B can only start

after A starts in a certain state.

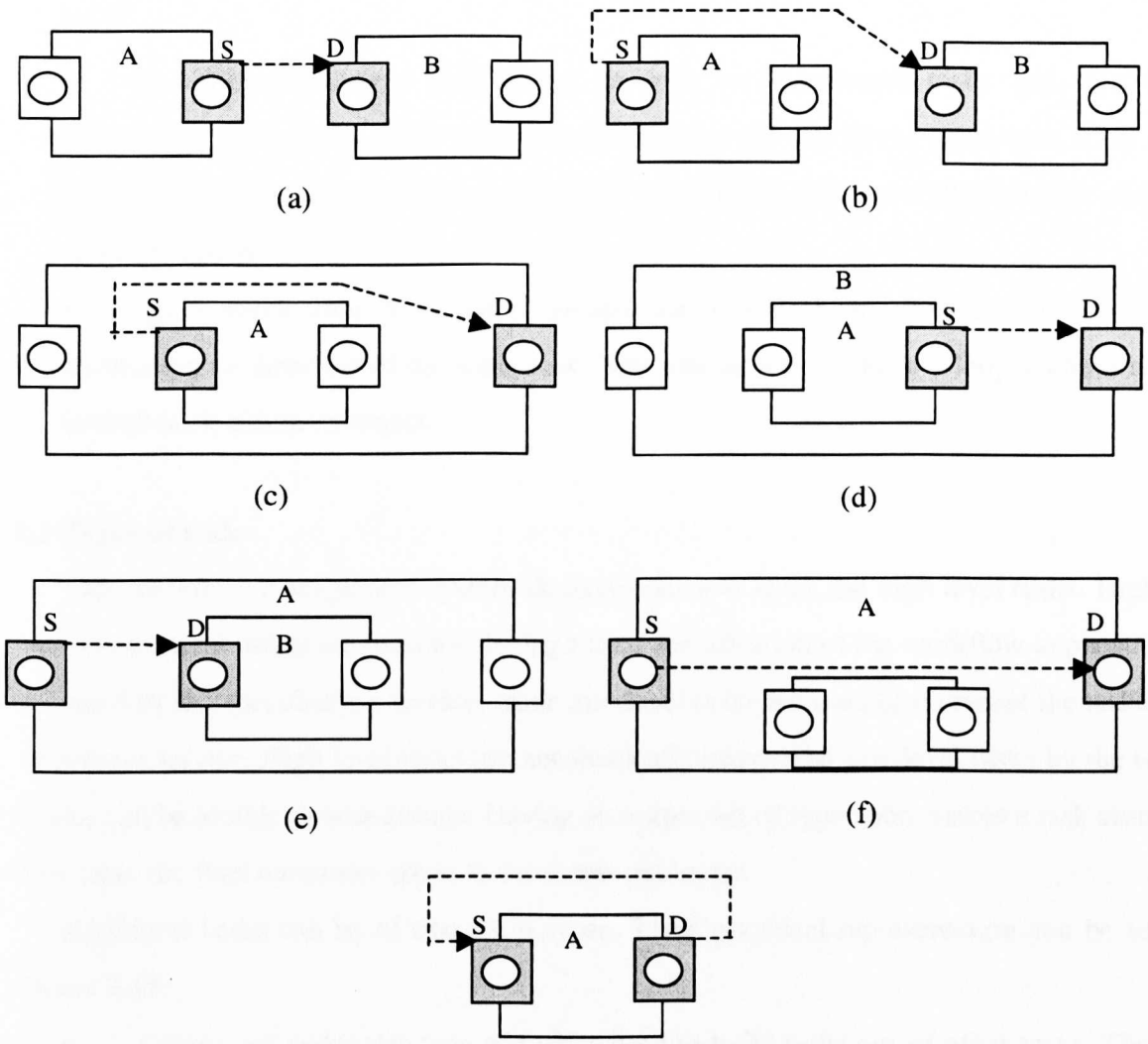


Figure 3.14: Types of notification dependencies

- (c) A notification dependency can use as source an input set of task A and as destination an output set of its parent task B. This could be used for instance to state that if task A starts with an "abnormal" input set, then its parent task B should reach a certain outcome.
- (d) A notification dependency can use as source an output set of task A and as destination an output set of its parent task B. This could be used for instance to state that if task A aborts then its parent task B should reach a certain outcome.
- (e) A notification dependency can use as source an input set of a compound task A and as destination object an input set of one of the tasks embedded in A. This can be used

to enforce that a task is only started if its parent task A was started with the particular input set S,

- (f) This notification dependency is only for compound tasks and describes a dependency between an input set (S) and an output set (D) of the same task. This can be useful to specify that a certain output set D can only be reached if the input set chosen to start the task was S.
- (g) A notification dependency can also use as source an output set of a task and as destination an input set of the same task. This can be used to have a loop without needing to feed back a dummy object.

3.3.2 Types of task

This workflow management system distinguishes low level and high level tasks. High level tasks (user level tasks) are used for the high level specification of the workflow application and are used by the specification service, while low-level tasks are used to represent the tasks in the execution service. High level tasks are automatically mapped to low-level tasks by the toolkit. Tasks can be atomic or non-atomic. Having an output set of type abort makes a task atomic. In this case, the final outcomes are in fact commit outcomes.

High level tasks can be of two main types. Their graphical representation can be seen on figure 3.15:

- *Compound tasks*: this type of task is used to build tasks out of other tasks. There are several possible reasons for using compound tasks. For instance, a compound task can be used as a way to hide the fault tolerance associated to a task. Indeed, compound task could be used as a black box to specify a task and an alternative or a task and a compensating task. It can also be used as a way to specify different levels of details. For instance, using an e.commerce example, a company director can model a business process as two tasks, the first one providing a service and the second one billing for this service. The department or person in charge of completing this task can then further describe the tasks that he has identified. For instance the finance department will describe how the billing of the service is done by decomposing the billing task as a set of tasks responsible for simpler activities. This allows having a good support for modularity.
- *Basic tasks*: this type of task is the basic unit of work of the system. It can no longer be divided in a set of workflow tasks. This type of tasks has associated to it a code that

performs the actions that the workflow task represents.

These two main types can be specialised as:

- *Loop tasks*: this type of tasks allows to introduce repetition in the specification, it is equivalent to a while with as condition that the output state feeding back the inputs is reached after completion of the body of the task.
- *Genesis tasks*: this is just a late instantiation of the task

There are three low-level types of task:

- *Basic tasks*: these low-level tasks correspond to the high level basic tasks, and is how the execution service represents basic tasks. In the rest of the thesis, both low and high level basic tasks will be referred to as basic tasks.
- *Compound tasks*: these tasks correspond to the high level compound tasks and is how the execution service represents compound tasks. In the rest of the thesis, both low and high level compound tasks will be referred to as compound tasks.
- *Genesis tasks*: these tasks are the main way to introduce dynamism in the system. They are in fact a particular kind of tasks that hold a task structure represented by a task schema (workflow script). A genesis task is a task with lazy instantiation. By lazy instantiation, we mean that they are only instantiated on demand and at the last moment. As a result they are really useful for large workflow applications as they provide an efficient way to manage these workflows by only instantiating the constituent tasks of the workflow application really needed. They are used to specify at low-level the loop tasks. A loop task is in fact modelled by a genesis task, which has a task structure that includes itself as component task (cf. chapter 6, section 7.2). High level tasks that have been requested to be only lazily instantiated are also represented at the low level by a genesis task, with them as task structure.

Basic Tasks and Compound tasks are represented on figure 3.15 a) and b) as rectangles with respectively single line and double line borders. Task specialised as Genesis will be rounded (figure 3.15 c for a Basic task specialised as Genesis), while loop task will have a feedback arrow (figure 3.15 d for a compound task specialised as Loop).

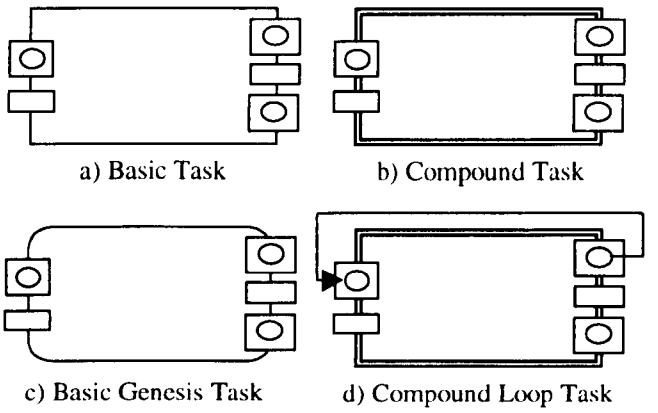


Figure 3.15: Graphical representation of Workflow Tasks.

3.4- Run time environment

Once the workflow application specification has been stored in the repository service, it is possible to instantiate the corresponding workflow schema. This is carried out by the execution service that is responsible for the proper execution of the following steps:

- Creation of the task controllers: each task of the schema has an associated task controller who is responsible for maintaining and managing the dependencies of this task. The placement of the task controller dependent on the type of control you want to implement (cf. section 3.1.2) and on some constraints set by the user.
- Creation of the tasks: each task of the schema will have an object of type task created to represent it in the run-time environment. It is at this level that the code is mapped to the task.
- Assignation of the task to its task controller: task controllers need to know which task they are responsible for, hence a reference to a previously created task is passed to each of the task controller.
- Setting up the inter-task dependencies: Each task controller registers its interest in the events that are important for its execution. It does it by contacting all task controllers associated with tasks involved as source in one of its dependency. For instance, imagine that a task (E) has a temporal dependency on an outcome of a task B. In this case, the task controller associated with task E registers its interest in the outcome reached by task B with the task controller associated to B. This allows the task controller managing task E to receive a notification of the outcome reached by B whenever it occurs. This notification is

sent by the task controller associated to task B, using a transaction. It has to be noticed that a task controller only registers with task controllers associated to its up stream tasks. The task controller maintains a persistent atomic object (called TaskControl) that is responsible for recording the inter-task dependencies involving this task as destination. Task controllers are represented in the system as processes containing an object of type TaskControl.

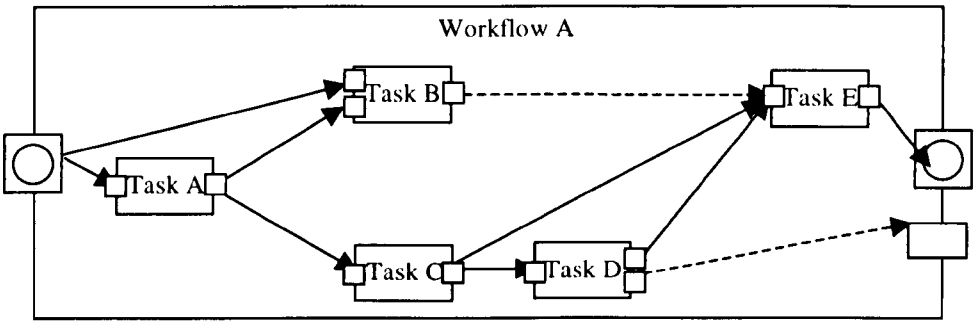


Figure 3.16: Specification of a workflow application

At the end of this instantiation the workflow schema depicted in figure 3.16 has been translated into a set of interacting task controllers managing the tasks. The event notification requests sent during the phase setting-up the inter-task dependencies are shown in figure 3.17.

In the workflow application pictured in figure 3.16, the dependencies between the workflow application and its component tasks are described. It has to be noticed that the task representations were simplified by not showing the input and output objects associated to these tasks. The input object of the workflow is using as input object, by both tasks A and B. Task B also has a data flow dependency on task A. Task D is an alternative task for task C. The output of task C or alternatively of task D if task C fails, is used by task E. If both tasks C and D failed then the workflow fails else it succeed with as output the output of task E.

The run-time representation (figure 3.17) shows the interactions between the different task controllers (one per task). Issuing the event notification requests shown on this figure has set up the dependencies shown in the previous picture.

Once instantiated, the workflow application has to be started. This is carried out by an administration task responsible for starting workflow applications by providing the initial input(s) to the task controller associated to the workflow application instances.

Once the initial inputs become available, the workflow application starts executing. Typically a workflow application is composed out of simpler tasks themselves potentially compound tasks, and its inputs becoming available trigger the starting of several of its

component tasks. The way this is done is that the task controllers of the tasks having dependencies on the reception of the workflow application are notified of the fact that their have become available and those having all their dependencies fulfilled start the execution of their associated task. For instance, in the previous example, when the workflow inputs become available, the task controllers associated to task A and B receive a notification of the availability of the inputs of the workflow applications. All the dependencies kept by the task controller of task A being fulfilled, it can start the execution of task A. On completion, task A reaches an output triggering the notification to the task controller of task B that can then be started.

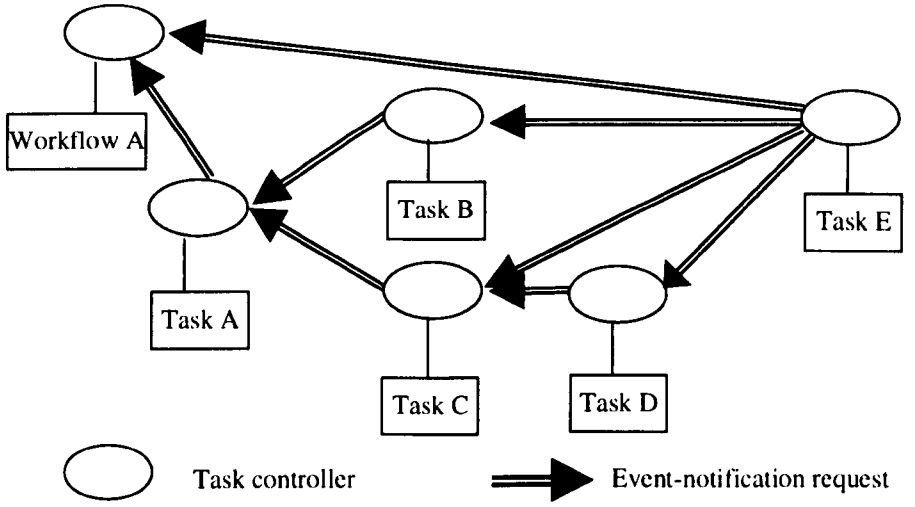


Figure 3.17: Run-time representation of a workflow application

The high level task state diagram is described in figure 3.18: As long as the workflow application has not been started the task is been built and in the state *building*, then it enters a state *set up* where the inter-task dependencies related to the task are set up. Once this has been done, the task enters the state *waiting* till it gets one of its input sets fulfilled (all object references available). At that point, it goes in the state *executing*. After completion of the task, it enters the state *completed*. It has to be noticed that the state of a high level task is given by the state of its associated task controller when it is no longer in its building state. In other words, the task controller state diagram is depicted on figure 3.18; if you remove the building state form the task diagram.

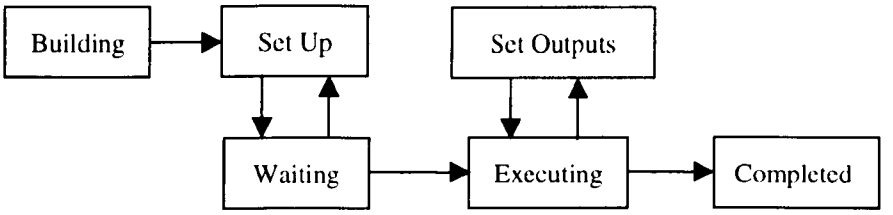


Figure 3.18: Task state diagram

As stated earlier, there is a need to be able to change the specifications at run-time due to some potential changes of the run time environment. Dynamic reconfiguration has been provided to cope with these changes. As long as the task controller is still in its waiting state, it is possible to modify the interface, implementation criteria as well as its associated dependencies of the task it is managing. This is achieved by bringing it back in its set-up state, where the changes can be done. Once the execution has started, it is still possible to modify the outputs of a compound task as well as to add some component tasks or delete some other component tasks as long as their task controllers haven't reached their completed state. It is also possible to modify the dependencies involving the outputs of the compound task. This is achieved by setting the state of the state of the task controller associated to this compound task to *set outputs* while it is being modified. To sum up, the changes that can be made are listed below:

1. The implementation bound to a basic task can be changed as long as it is in a wait state
2. Tasks can be added or removed from workflow instances
3. The constituent tasks of a compound task can be changed
4. Input alternatives can be added or removed from a task, and their priority can be changed as long as the tasks is not in its state executing or completed
5. Output alternative can be added or removed from a task, and their priority can also be changed as long as the task has not completed

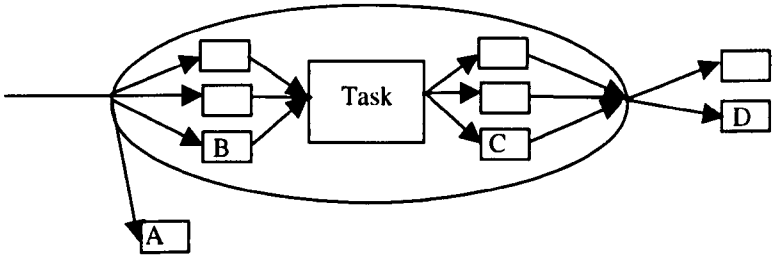


Figure 3.19: Event notifications

The system uses atomic actions to propagate co-ordination information to ensure that tasks are run according to the specification. The system also has a flexible way to ensure that the task controller sends event notifications as an atomic action: the task body as well as some (or all) of its associated notifications can be guaranteed to be performed atomically. This is achieved by enclosing both the execution of the task and its event notifications in an atomic action. Users can specify which tasks do not need atomic notifications by labelling them as non-vital. Usually monitoring tasks are non-vital tasks. Figure 3.19 shows how atomic and non-atomic notifications are propagated. There are four sets of event notifications labelled on the figure as A, B, C and D. Event notifications of type A and B correspond to some event notifications of inputs available while event notifications C and D are for outputs becoming available. Event notifications B and C are to be delivered atomically with the execution of the task and as a result are embedded in a transaction. A and D are just event notifications that are not vital and will be referred thereafter as eventual event notifications.

To sum up, we have presented in this chapter the architecture of our system. What is now needed is a way to specify the workflow applications that are to be executed in the workflow system. In the next chapter, a language will be introduced to serve this purpose.

Chapter 4

Language

In this chapter, the reader will find the specification of our textual language as well as the graphical notation that has been adopted to represent workflow applications within the GUI of our toolkit to be presented in chapter 6. This is also presented in [60].

4.1- Overview

The aim of the workflow language is allow the specification of workflow applications. As was seen in the previous chapters, a workflow application in our system consists of a set of workflow tasks linked by dependencies. A workflow task uses some inputs to perform some work and generates some outputs. In chapter 3, the architecture of the system was described as well as the task model chosen to support the requirements of the system. The notations presented in this chapter allow the specification of the high level tasks of the system.

A workflow specification in our system consists of a list of object classes, a list of task classes and a workflow application. As a result, the language used to describe a workflow is divided into three parts:

- Object class definitions part to specify the type of input/output objects used,
- Task Class definitions part to specify the task interfaces used,
- Task instances part to specify the workflow tasks used.

Let's imagine that we have a task that adds two numbers together op1 and op2 of type integer, and returns the object res of type integer. It would be specified using the textual notation as:

```
// object class definitions
objectclass integer;
// task class definitions
taskclass Operation {
```

```
    inputs { input main { op1 of class integer; op2 of class integer } };
    outputs { outcome done { res of class integer }}
}
// Task instances. It has to be noticed that the addition workflow
application is a compound task as it is composed out of other tasks. The
details of these component tasks has been removed here for simplicity
compoundtask addition of taskclass addition {
    inputs { input main { inputobject op2 {}; inputobject op2 {} } };
    ...
    outputs { outcome done { outputobject res {...} } }
}
```

The three parts of the description can be easily identified on this basic example.

And using the graphical notation, it can be represented by picture 4.1.

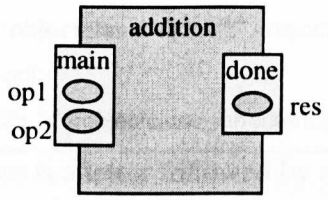


Figure 4.1: Basic example

In the next sections, the extracts of the grammar relevant will be given each time that a component of the specification is discussed.

On the extract relevant to the specification of a workflow application, it has to be noticed that the list of object classes is optional as we could have a workflow application with no data flow dependencies at all, but not the list of task classes as a workflow application has by nature an interface that has to be defined. It also has to be noticed that there is an order in the specification: first the object classes then the task classes and finally the workflow application.

```
<specification>        ::= <objectclass_definitions>  ";"  <taskclass_definitions>  ";"
<workflow_application>
                       |  <taskclass_definitions> ";" <workflow_application>
```

In the next sections, these different components of the language will be described in details. For each component, the relevant extract of the grammar for the language will be given as well as some examples of how to specify such a component.

4.2- Object Classes

4.2.1. Overview

The goal of the object classes is to introduce some type checking in the specification. Each object resource in the system has an associated object class. Workflow applications need to specify the type (object class) of object resources that they want to use. An object class is fully

defined by its name. This system supports multiple inheritance to allow more flexible specifications.

4.2.2 Grammar

The declaration of an object class is introduced by the keyword **objectclass.** The character ':' denotes multiple inheritance. Object classes inherited are separated using a coma

<objectclass_definitions>	::= <objectclass_definition>
	| <objectclass_definition> ";" <objectclass_definitions>
<objectclass_definition>	::= "**objectclass**" <idf>
	| "**objectclass**" <idf> ":" <objectclass_inheritances>
<objectclass_inheritances>	::= <idf>
	| <idf> "," <objectclass_inheritances>

A valid name for an object class is a letter followed by alphanumeric characters (called *idf* in the grammar). The object classes inherited also have to be defined.

4.2.3 Examples

Let's imagine that our application uses some objects of type person and employee, the latter inheriting from the former. In this case, the declaration of the object classes will just be:

```
objectclass person;
objectclass employee : person;
```

Shall employee also inherited from another object class such as bankAccount, the declaration would become:

```
objectclass person;
objectclass bankAccount;
objectclass employee : person, bankAccount;
```

4.3- Task Classes

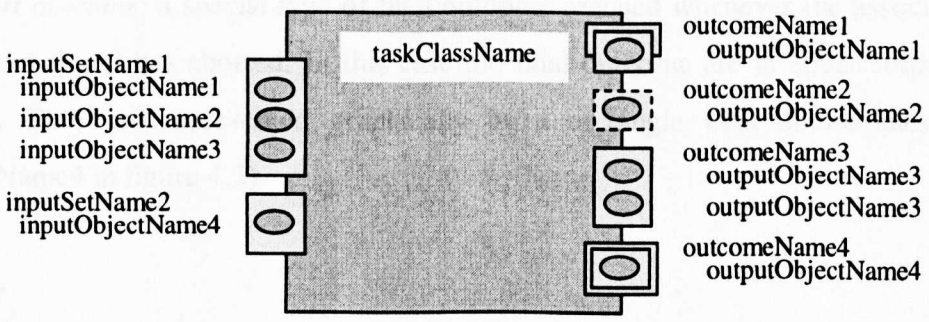


Figure 4.2: Graphical notation for a task class

4.3.1 Overview

The goal of a task class is to allow specifying a type of interface for a workflow task. A task class can be used for creating many tasks with the same interface. It is fully defined by its name, inputs and outputs.

On figure 4.2, inputs have names starting with "input" and the names of the outputs start with "output".

The inputs of a task class consist of one or more named input sets. The input sets act as recipient for alternative starting conditions for the task. An input set consists of a set of (input) objects, each of them associated to a particular object class.

Symmetrically, the outputs of a task class consist of one or more named sets of (output) objects. These sets are also called outcomes. There are four different types of output sets as specified in chapter 3:

- *Mark outcome*: it is an outcome that provides outputs while the task is running. However using mark outcomes breaks the isolation property of the atomic actions hence the tasks with such outcome are not transactional. This type of outcome is usually used to publish partial results (output objects) of a compound task before its completion. The graphical notation for mark outcome is a rectangle with dotted lines. In figure 4.2, outcomeName2 appears as a mark outcome.
- *Repeat outcome*: it is an outcome whose output objects are being used as input objects by the same task. This type of outcome is used to implement loops. The graphical notation for repeat outcomes is depicted in figure 4.2 (outcomeName1).
- *Final outcome*: it is an outcome that is only reached whenever a task does complete. They are represented by rectangles with plain borderlines as depicted in figure 4.2 for outcomeName3.
- *Abort outcome*: a special type of final outcome reached whenever the associated task is transactional and has aborted. In this case the final outcome are in effect some commit outcomes. They are represented graphically by a rectangle with double-lines borders (outcomeName4 in figure 4.2).

4.3.2 Grammar

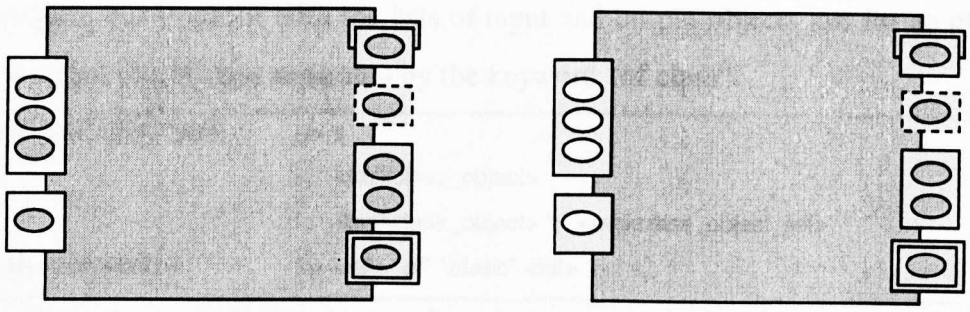
The declaration of a task class is introduced by the keyword "**taskclass**" followed by the name of the task class. Then the specification is divided between the inputs and the outputs

specifications.

```
<taskclass_definitions>     ::= <taskclass_definition>
                            |   <taskclass_definition> ";" <taskclass_definitions>
<taskclass_definition>      ::= "taskclass" <idf>
                                "{" <taskclass_inputs ";" <taskclass_outputs> "}"
```

The specification of the inputs of a task class is introduced by the construct "**inputs**", and then the input sets are introduced by the construct "**input**" followed by the name of the input set. The specification of the input sets corresponds to the specification of the white boxes of figure 4.3a. Then the input objects, represented by white ovals in figure 4.3b, are described using taskclass_object_list.

```
<taskclass_inputs>          ::= "inputs" "{" <taskclass_input_list> "}"
<taskclass_input_list>      ::= <taskclass_input>
                            |   <taskclass_input> ";" <taskclass_input_list>
<taskclass_input>           ::= "input" <idf> "{" <taskclass_object_list> "}"
```

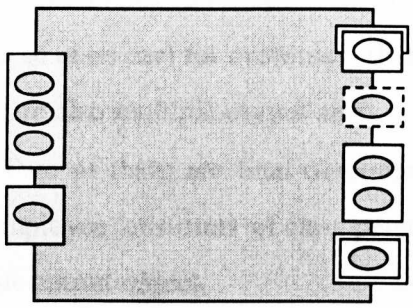


a) input sets of a TaskClass b) input objects of a TaskClass

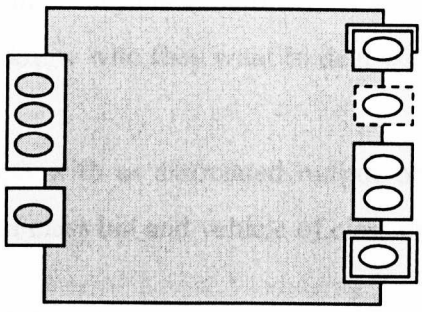
Figure 4.3: Inputs of a task class

The specification of the outputs of a task class is similarly introduced by the construct "**outputs**", then the output sets are introduced one by one using, to specify their type, one of the following keywords: "**mark**", "**repeat**", "**outcome**" or "**outcome abort**", followed by the name of the output set. The specification of the output sets corresponds to the specification of the white boxes of figure 4.4a. Then the output objects, represented by white ovals in figure 4.4b, are described using taskclass_object_list.

```
<taskclass_outputs>          ::= "outputs" "{" <taskclass_output_list> "}"
<taskclass_output_list>      ::= <taskclass_output>
                             |   <taskclass_output> ";" <taskclass_output_list>
<taskclass_output>           ::= <taskclass_outcome_type> <idf> "{" <taskclass_object_list> "}"
<taskclass_outcome_type>     ::= "mark" | "repeat" | "outcome" | "outcome" "abort"
```



c) output sets of a TaskClass



d) output objects of a TaskClass

Figure 4.4: Outputs of a taskclass

The object lists used for both the lists of input and output objects just list an object used as a couple name, object class separated by the keyword "**of class**".

```
<taskclass_object_list>      ::= ε
                             |   <taskclass_object>
                             |   <taskclass_object> ";" <taskclass_object_list>
<taskclass_object>           ::= <idf> "of" "class" <idf>
```

The object classes used here have to have been present in the object classes definitions. The names for the task class as well as its input and output sets follow the same rules as for the object class names. A task class also has to have at least one input and one output set. Each input or output set is associated to zero or more objects,

4.3.3 Examples

The following script extract describes how a company could automate the selling process presented afterwards. Usually a customer identifies a car he would like to buy and negotiate with the person in charge of selling this car. There are two possible final outcomes: either the deal is made and a contract is signed after having checked the customer references, or the deal is off. The company also needs to know as early as possible that the deal is done to withdraw the car from the offer and know the price of the sale. The negotiation process can also iterate with a new vehicle.

To model this application, we need an interface named for instance BuyACar. It has two alternative input sets:

- A set named main with three associated input objects (customer of class person, staff of class employee and vehicle of class car) for a customer who wants to deal with a specific person in the company selling the car,
- A set named alternate with only two input objects (customer of class person and vehicle of class car) for customers that do not know who they want to deal with.

There are also multiple output sets:

- Two of them are final outcomes: success with as associated output objects staff of type employee, customer of class person, bill of class bill and vehicle of class car, and failure without output object.
- The intermediate result is modelled with a mark outcome with as associated output objects vehicle of class car, and cost of type integer.
- There is a retry outcome to model the loop with three associated output objects: staff of class employee, newvehicle of class car and customer of class person.

The textual specification is listed below:

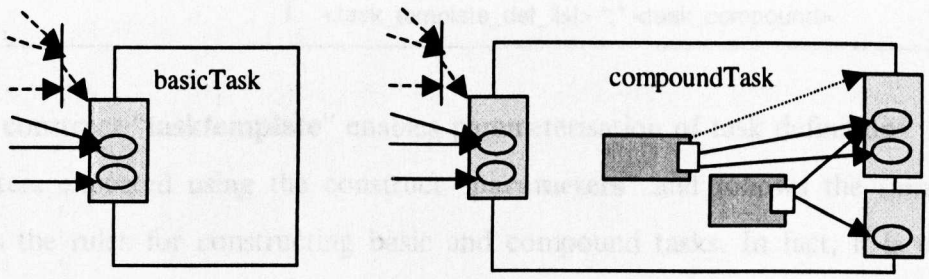
```
taskclass BuyACar
{
   inputs
   {
      input main
      {
         customer of class person;
         staff of class employee;
         vehicle of class car
      };
      input alternate
      {
         customer of class person;
         vehicle of class car
      }
   };
   outputs
   {
      repeat renegociate
      {
         customer of class person;
         staff of class employee;
         newvehicle of class car
      };
      mark deal
      {
         vehicle of class car;
```

```
            cost of class integer
        };
        outcome success
        {
            customer of class person;
            staff of class employee;
            vehicle of class car;
            bill of class bill
        };
        outcome failure
        {
        }
    }
}
```

4.4- Task instances

4.4.1 Overview

The overall aim is to specify the tasks and their dependencies that make up a workflow application. Each task has to belong to a task class.



a) For a basic task b) For a compound task

Figure 4.5: Specification of responsibilities for a task

The information from the associated task class is then used to find out the inputs of the task and to specify the mapping between these inputs and the rest of the application. The information needing to be specified is different for a basic task and a compound task. On figure 4.5a, the information that a basic task needs to specify is shown. The dotted arrows represent the notification dependencies associated to the input sets of the task while the plain arrows represent the data dependencies that input objects have (e.g. a list of alternative objects that they can be mapped to). Similarly, figure 4.5b shows what compound tasks have to specify. On top of what a basic task has to specify, they also need to specify their constituent tasks (dark grey rectangles) as well as the mapping of their outputs with them in terms of both notification and data-flow dependencies. Notice that a compound task is only responsible for the dependencies having it as target. In particular, each component task is responsible for its own

dependencies. This allows locality of modification as well as modularity.

Information on the details of the implementation of a task also need to be provided, these information can include the priority of the task, the resources needed, the type of node on which the task is to be run, etc. This information is provided has a set of couples of keyword-value that are used afterwards at creation time of the task. With the current system, this set of couples is sent to a task factory to give it some details on the task instance to be created.

We are now going to present in detail how our language allows the specification of task instances.

4.4.2 Grammar

A workflow application can be simply a compound task or a compound task with a set of task templates associated to it.

```
<workflow_application>      ::= <task_compound>
                            |   <task_template_def_list> ";" <task_compound>
```

The construct "**tasktemplate**" enables parameterisation of task definitions. It specifies the parameters expected using the construct "**parameters**" and follows the same specification rules as the rules for constructing basic and compound tasks. In fact, it is using the same headers and bodies as they are using.

```
<task_template_def_list>      ::= <task_template_def>
                              |   <task_template_def> ";" <task_template_def_list>
<task_template_def>           ::= "tasktemplate" <task_basic_header>
                                  "{" <task_template_parameters> ";" <task_basic_body> "}"
                              |   "tasktemplate" <task_compound_header>
                                  "{" <task_template_parameters> ";" <task_compound_body> "}"
<task_template_parameters> ::= "parameters" "{" <task_template_parameter_list> "}"
<task_template_parameter_list ::=          <idf>
                              |   <idl> ";" <task_template_parameter_list>
```

The name of a task template follows the same rules as those for the names of an object class or of a task class.

Tasks are then divided into three categories. Two of them: basic and compound tasks being the building stones for the third one, the instantiation of a task template. The difference between these two types of tasks is that a basic task is a basic unit of work for the process that

we are modelling, while a compound task gathers other tasks that may be themselves basic or compound. As a result, basic tasks will be seen as indivisible. The third type of task is the task template that is just an instance of a task template specification; itself parameterised version of the two previous task types.

```
<task_list>          ::= <task>
                     | <task> ";" <task_list>
<task>               ::= <task_basic>
                     |   <task_compound>
                     |   <task_template>
```

The keywords "**task**" and "**compoundtask**" respectively introduce basic and compound tasks. Both of them have their associated task class introduced by the keywords "**of taskclass**".

The first part of the body of the compound task definition is identical to the specification of a basic task. First it defines some implementation information. Both types of tasks carry on with the description of the mapping of the inputs. A compound task then requires its component tasks to be specifying before specifying the mapping of its outputs.

```
<task_basic>              ::= <task_basic_header>"{" <task_basic_body> "}"
<task_basic_headers>      ::= "task" <idf> "of" "taskclass" <idf>
<task_basic_body>         ::= <task_inputs>
                          |   <task_implementation> ";" <task_inputs>
<task_compound>           ::= <task_compound_header> "{" <task_compound_body> "}"
<task_compound_header>    ::= "compoundtask" <idf> "of" "taskclass" <idf>
<task_compound_body>      ::= <task_implementation> ";" <tasks_inputs> ";" <task_list> ";"
<task_outputs>
                          |   <tasks_inputs> ";" <task_list> ";" <task_outputs>
```

Once against the same rules are used for the specification of the names.

A task template is the instantiation of one of the task template definitions. The only thing needed there is the mapping of the arguments to the parameters requested by the definition. The name of the task template that it is instantiating is introduced by the keywords "**of tasktemplate**". Arguments are given between brackets and separated by comas.

```
<task_template>           ::= <idf> "of" "tasktemplate" <idf> "(" <task_template_argument_list> ")"
<task_template_argument_list> ::=      <idf>
                          |   <idf> "," <task_template_argument_list>
```

The task template specification instantiated has to have been specified earlier on.

Implementation information is introduced by the construct "**implementation**" as a list of couple "keyword **is** value". The implementation criteria are used to specify the task controller factory to be used, the placement of a task at run time, as well as information on its implementation. It can also be used to specify the priority and deadlines associated to a task. The GUI also used them to store the task co-ordinates.

```
<task_implementation>          ::= "implementation" "{" <task_implementation_criteria> "}"
<task_implementation_criteria> ::=          <task_implementation_criterium>
                                 | <task_implementation_criterium> ";" <task_implementation_criteria>
<task_implementation_criterium> ::= "\"" <idf_crit> "\"" "is" "\"" <idf_crit> "\""
```

Both keywords and values are enclosed between double quotes and can have whatever value is needed, with as restriction that the double quote is an invalid symbol that can not be used.

Tasks input mappings are introduced by the keyword "**inputs**". Then each input set is listed in turn and includes both some notification dependencies introduced by the keywords "**notification from**" and some mapping information for their associated input objects introduced by the construct "**inputobject**" "**from**".

```
<task_inputs>                  ::= "inputs {" <task_input_list> "}"
<task_input_list>              ::= <task_input>
                                 |   <task_input> ";" <task_input_list>
<task_input>                   ::= "input " <idf> "{" <task_input_dependency_list> "}"
<task_input_dependency_list>   ::=          ε
                                 |   <task_input_dependency>
                                 |   <task_input_dependency> ";" <task_input_dependency_list>
<task_input_dependency>        ::= "notification" "from" "{" <task_notification_list> "}"
                                 | "inputobject" <idf> "from" "{" <task_delegation_list> "}"
```

It has to be noticed that at least one input set has to be declared. They potentially have no dependencies attached to them as some dependencies could be created during run-time. Similarly, input objects can have no delegation dependencies on them for the same reasons.

The outputs are similarly specified with as only difference the use of the construct "**output**" where "**input**" was previously found.

```
<task_outputs>                  ::= "outputs" "{" <task_output_list> "}"
<task_output_list>              ::= <task_output>
                                |   <task_output> ";" <task_output_list>
<task_output>                   ::= "output " <idf> "{" <task_output_dependency_list> "}"
<task_output_dependency_list>   ::=          ε
                                |   <task_output_dependency>
                                |   <task_output_dependency> ";" <task_output_dependency_list>
<task_output_dependency>        ::= "notification" "from {" <task_notification_list> "}"
                                | "outputobject" <idf> "from" "{" <task_delegation_list> "}"
```

It has to be noticed that at least one output set has to be declared. They potentially have no dependencies attached to them as some dependencies could be created during run-time. Similarly, output objects can have no delegation dependencies on them for the same reasons. The input and output names used here have to exist in the task class associated to the task being instantiated. There is one and only one such description per input and output sets, as well as per input and output objects defined in the task class. To be valid, all the inputs and outputs defined in the task class and only them have to be found in these declarations.

The notification dependencies are specified as a list of notifications. The notifications are of the form "**task** myTask **if input** myInputSet" and "**task** myTask **if output** myOutputSet" which should respectively be read as this task requests a notification that task myTask has started with as input set myInputSet and that task myTask has reached the output set myOutputSet

```
<task_notification_list>        ::= <task_notification>
                                |   <task_notification> ";" <task_notification_list>
<task_notification>             ::= "task" <idf> "if" "input " <idf>
                                |   "task" <idf> "if" "output" <idf>
```

The sets referenced in a task specification have to exist; e.g. both the task and the set with the names requested have to exist within the specification. A discussion on the different valid types of notifications for inputs can be found in chapter 3, section 3.1 with a graphical description on figure 3.14.

The delegation dependencies are specified as a list of delegations. The delegations are of the form "myObject **of task** myTask **if input** myInputSet" and "myObject **of task** myTask **if output** myOutputSet". The first form should be read as this task requests a delegation of the object myObject of task myTask if this task has started with as input set myInputSet. The

second one should be read as this task requests a delegation of the object myObject of task myTask if this task has reached the output set myOutputSet.

<task_delegation_list>	::= ε
	| <task_delegation>
	| <task_delegation> ";" <task_delegation_list>
<task_delegation>	::= <idf> " **of**" "**task**" <idf> "**if**" "**input** " <idf>
	| <idf> "**of**" "**task**" <idf> "**if**" "**output**" <idf>

The objects referenced in a delegation dependency have to exist, e.g. both the task, the set and its associated object with the names requested have to exist within the specification. A discussion on the different types of notifications valid can be found in chapter 3, section3.1 with a graphical description on figure 3.13.

4.4.3 Examples

Let's use as example two tasks with the same task class BuyACar defined previously. The first task will be a basic task, and the second a compound task.

```
task buyMyCar of taskclass BuyACar
{
   implementation
   {
      "TaskImpl" is "buyMyCar.exe";
   };
   inputs
   {
      input main
      {
         inputobject customer from
         {
            john of task getCustomer if output success;
            boss of task getCorporateCustomer if output success;
            customer of task buyMyCar if output renegociate
         };
         inputobject staff from
         {
            carl of task getCustomerAdviser if output success;
            staff of task buyMyCar if output renegociate
         };
         inputobject vehicle from
         {
            myCar of task getCar if output success;
            newvehicle of task buyMyCar if output renegociate
         }
      };
      input alternate
      {
         notification from
         {
            task getCustomerAdviser if output failure
         };
```

```
            inputobject customer from
            {
               john of task getCustomer if output success;
               boss of task getCorporateCustomer if output success
            };
            inputobject vehicle from
            {
               myCar of task getCar if output success
            }
         }
      }
   }
```

This example assumes the existence of three tasks, namely getCustomer, getCorporateCustomer and getCustomerAdviser. The input set named alternate is used whenever the task getCustomerAdviser reaches its output failure, and allows the task to start in a degenerated state (no staff associated to the deal). Internally the task needs to be able to handle both of these starting conditions. In this example, several other features can be seen: the use of the couple **TaskImpl**, buyMyCar.exe as implementation criterium allows the underlying system to chose the right code to execute. The feedback from the output objects of the repeat outcome is also apparent. The alternative mapping of objects is also shown in this example: the customer can come from three different sources: from the output objects named john and boss from respectively tasks getCustomer and getCorporateCustomer if they reach the outcome success or from itself if it loops.

If now the same task is instantiated using a compound task, the beginning of the task description will be identical to the previous task, except of course for the implementation information about the code. Then the component tasks have to be specified as well as the mapping of the mapping of the outputs of the compound task. The components that can be mapped to an outcome are the input sets of the task as well as any of the input or output sets of the component tasks. For the output objects, the same applies but with the associated objects this time. Let us assume that the task is in fact the result of the composition of two basic tasks that are respectively called dealMade and contractSigned. The script describing this task is listed below.

```
compoundtask buyMyCar of taskclass BuyACar
{
   inputs
   {
      // identical to previous example, removed
   };
   task dealMade of taskclass makeADeal
   {
```

```
   // specification removed
};
task contractSigned of taskclass contractSignature
{
   // specification removed
};
outputs
{
   repeat renegociate
   {
      outputobject staff from
      {
         staff of task dealMade if output newchoice
      };
      outputobject vehicle from
      {
         vehicle of task dealMade if output newchoice
      };
      outputobject customer from
      {
         customer of task buyMyCar if output main;
         customer of task buyMyCar if output alternate
      }
   };
   mark deal
   {
      outputobject cost from
      {
         cost of task dealMade if output success
      };
      outputobject vehicle from
      {
         vehicle of task dealMade if output success
      }
   };
   outcome success
   {
      outputobject staff from
      {
         staff of task contractSigned if output success
      };
      outputobject vehicle from
      {
         vehicle of task contractSigned if output success
      };
      outputobject bill from
      {
         bill of task contractSigned if output success
      };
      outputobject customer from
      {
         customer of task contractSigned if output success
      }
   };
   outcome failure
   {
      notification from
      {
```

```
                task dealMade if output failure;
                task contractSigned if output failure
            }
        }
    }
}
```

In this example, the structure of the specification of a compound task is shown. In particular, its component tasks are embedded in its own specification. The use of notification dependencies to reach the final outcome failure can also to be noticed. The task fails if either the task dealMade or the task contractSigned reach themselves their outcome labelled failure. Had the designer wanted to reach the outcome failure only in the event of both tasks reaching their outcome failure, then the specification of the mapping of the outcome failure would have been:

```
outcome failure
{
    notification from
    {
        task dealMade if output failure
    };
    notification from
    {
        task contractSigned if output failure
    }
}
```

4.5- Extended transaction models and workflows

The structuring mechanisms available within 'standard' transaction systems are for sequential and concurrent composition of (sub-) transactions within a top-level transaction. These mechanisms are sufficient if the overall application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system [22]; they are less well suited for structuring "long-lived" applications of the type considered in this thesis. Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources (e.g., locks) for a long time; further, if such a transaction aborts, much valuable work already performed could be undone. If an application is composed as a collection of transactions, then during run time, the entire activity representing the application in execution is frequently required to possess some or all of the ACID properties of the individual transactions. Much of the research on structuring transactional applications has been influenced by the ideas of spheres of control [14].

In chapter 2, we discussed two such extended transaction models: Saga and Contract. Our

workflow language provides a very flexible way of constructing extended transaction models. For instance, Sagas (presented in chapter 2, section 2.3.1) can be easily modelled as a workflow. Let's imagine a saga T composed out of transactions T_1, T_2, ... T_n with their corresponding compensating transactions C_1, ... C_n. The guarantees provided by the system are that either $T_1T_2...T_n$ is executed or $T_1...T_iC_i...C_1$ is executed for some i between 0 and n. For n =3 an equivalent workflow application is shown on figure 4.6.

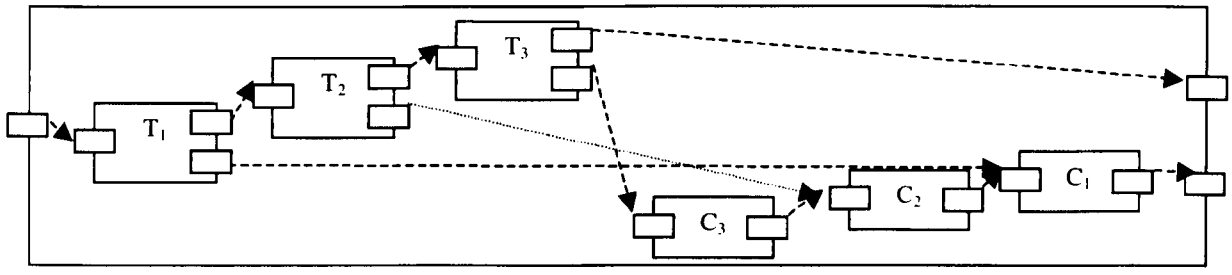


Figure 4.6: Saga modelled as a workflow

In figure 4.6, the workflow has two outputs: commit if the saga was successfully executed and compensated otherwise. Each sub transaction can either commit or abort. This is represented by the two output sets (grey boxes), the upper one being commit. The compensating transactions are (as usual for sagas) supposed to always execute successfully. The dependencies (dotted/dashed lines) are OR-ed (e.g. only one activated dependency is needed to trigger the execution of a task).

Use of workflows for implementing extended transaction models is also discussed in [2]. Our alternative output sets provide a way to specify in which state the activity modelled ended up. Alternative inputs sets and sources for input objects also allow adding some fault tolerance. Compensating tasks, alternative tasks... can be used to model these extended transaction models.

4.6- Comparison with METEOR

In chapter 2, we reviewed a number of languages. In this section, we are going to compare our language with METEOR (introduced in chapter 2, section 2.2.2.1.) as it comes closest to our language in term of functionalities provided and is the best known workflow language in the workflow community. They have two languages: the WorkFlow Specification Language

(WFSL) and the Task Specification Language (TSL).

As stated in chapter 2, the WFSL is divided in several parts:

1. Type definitions and variable declarations, similar to the C syntax.
2. Task type definitions
3. Task class and filter definitions
4. WF definition
   - task instantiations
   - rules
5. WF instantiation

Part 1 corresponds to the declaration of our object classes, however METEOR allows you to define some new types similarly to what can be done in C, while our language only allows you to declare the valid object classes known by the underlying system.

Part 2 and 3 correspond to our task class definitions, the major differences being that they only allow one input set and they specify the internal states and transitions between these states. Our mark outcomes would correspond to their non-initial and non-terminal states, however we have made no provision whatsoever to try to capture the internal transitions between states. Both languages have typed input and output objects. METEOR also introduces the notion of filter, which converts objects of a type to another type. We did not feel that there was a need for a specific entity to filter data and are confident that simple tasks are a good way to handle data filtering.

Part 4 corresponds to our task instances specification. Both languages support simple and compound tasks. The way they deal with dependencies however is quite different. While in the WFSL, they have a rule section localised in their compound tasks; we keep the dependencies with the task that is the destination of the dependency. We do believe that it is a better solution as it provides better modularity and allows a better management of dynamic reconfiguration as described latter in this section. In particular, our tasks have no knowledge whatsoever on which downstream tasks are using them as source of dependency (cf. section 3.3.1). METEOR allows complex rules with some "control dependencies" (preconditions) that can involve some computations.

The following rule for instance would be read as: if task L1 reaches state done and the function success applied to task L1 and its output (object) output1 is evaluated to TRUE and

the variable outvalL4 is greater than 5 then task L2 should enter state start using output2 from task L1 as input for input1.

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

With our language, we would have a special output set for L1 which fulfils the listed conditions, and a dependency between output1 of task L1 and input1 of input set start of task L2.

As far as part 5 is concerned, we do not allow specification of the initial objects to be used to start the workflow in the script. A CORBA interface to start the specification from the outside world has however been provided.

As far as application fault tolerance is concerned, METEOR creates an extra state per controllable transition. This error state called "transition name"_err is reached in case of failure. They have also proposed a single default common error state. On the other hand, we consider that we have a normal execution output state and that all the others are exception handling output sets. The binding is made at the task implementation level and is not seen at the Workflow specification level.

As far as dynamic reconfiguration is concerned, METEOR tackles the issue by using two different techniques. The first one is to use arrays of tasks whose size is set up at run-time and whose dependencies are specified based on the index of the task in the array. This allows them to create an arbitrary number of tasks at run-time. The other technique used is to use a task of class controlClass that is allowed to rewrite the specification at run-time. This task needs to know about the whole current workflow application, as the specification is fully re-interpreted. We have two ways to handle dynamic reconfiguration: the first one is to use genesis task (lazy instantiation) which allows you to instantiate the genesis task only if it is needed. The second way to handle dynamic reconfiguration is for a task to interact directly with the task controller which does not require to know about the whole workflow applications and does not have the overload of needing to be re-interpreted.

As far as specifying the internals of a simple task is concerned, we did not create yet another programming language for it, as we did not see the need for one, we only specify a task factory to be used and some implementation criteria. The toolkit has an option to create some Java code skeletons that deal with all the interaction with the task controller. METEOR on the other hand created the Task Specification Language. The main aim of the TSL is to avoid rewriting some legacy applications. The TSL enables the developer to specify the interfaces

and the specific reactions to this entity's behaviour (error handlers...), it also enables to perform specific actions to be performed such as SQL queries... It also includes some statements to let the WF manager knows the current state of the task (TASK_EXECUTING(), TASK_ABORTED(), TASK_COMMITTED(object), TASK_DONE()...). All these statements can be seen as a set of macros that can be embedded in the host language such as c, C++. In the case of legacy applications, the TSL program consists in:

- a call to a macro indicating that the task is about to execute
- a call to an interface that submits or calls the legacy application
- a call to a macro when the application complete its execution

The TSL deals with task level failure recovery and error handing and error handling specific to the interface or processing entity used.

To sum up, we do believe that our language is simpler than METEOR WFSL yet powerful enough to specify the applications that WFSL can specify. It can also be seen that our language provides better fault tolerance and dynamic reconfiguration features thanks to our multiple input sets and local dependency specifications.

In this chapter, our language was presented as well as its associated graphical notation for the GUI. In the next chapter, we will validate our design by showing how it can be used to specify a series of workflow applications.

Chapter 5

Examples

In this chapter, the reader will find some examples illustrating the main features of our language as well as its suitability for specifying dependable workflow applications. Other examples are presented in [60]. The first example is a process-ordering example showing how a complete (yet simple) workflow application can be specified. The next example is a travel reservation application illustrating how to add application level fault tolerance to workflow applications. The third example is a network fault management example provided by Northern Telecommunication that shows how to model loops as well as dynamic addition of tasks.

The screen dumps used as support in this chapter were generated using the toolkit presented in the next chapter. For readability, only extracts of the scripts are given in this chapter. However the reader can refer to appendix A for the complete specifications.

5.1- Example I: Customer order processing

This example will illustrate how task classes are specified and how an application can be composed out of other tasks and compound tasks.

The workflow application considered here is doing some customer order processing. A customer order some items such as some softwares, which triggers two activities that can be executed in parallel: an activity checking whether the item ordered is in stock or not and an activity checking the credential of the customer. Once both of these activities have been carried out, two activities can then start: an activity capturing the payment, and an activity dispatching the item. All these activities can reach two outputs success or failed, with the exception of the capture of the payment that is always successful. We also want that the dispatch activity be atomic.

The workflow application will fail if one of the tasks it is composed of fails. It will succeed

if the item is delivered and after that the payment is captured. We do not claim that this is a realistic process, in reality the company may want to wait for the payment to be capture before dispatching the item for instance, and it is likely that some of these tasks will themselves be some compound tasks. A screen dump of the process is presented in figure 5.1. It has to be noted that we have been using a compact representation of the task components on (i.e. neither the input nor output sets nor their associated objects are shown) this figure. The User Manual in appendix 7 has several screen dumps from non-compact representations.

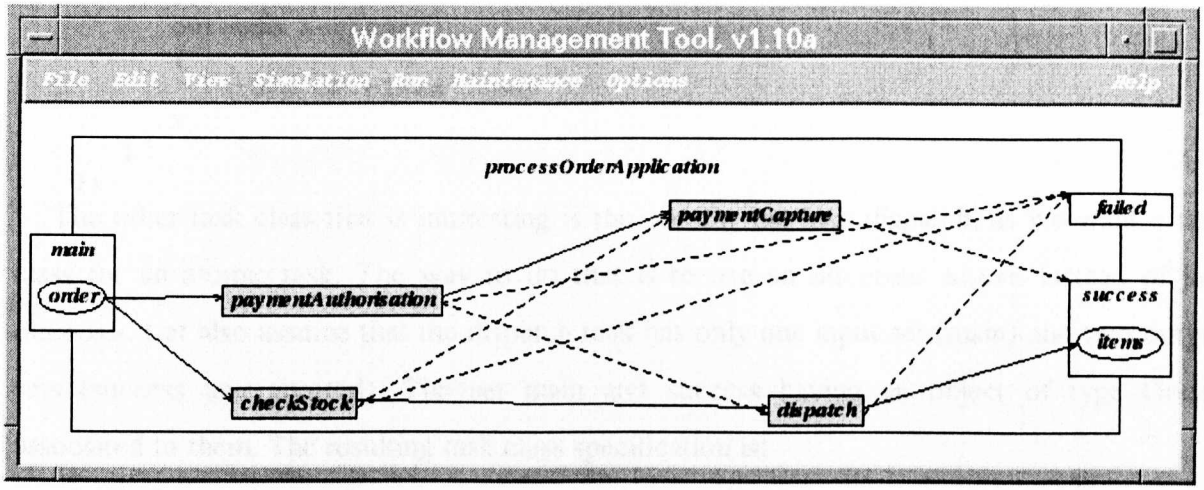


Figure 5.1: Overall process ordering application

In order to specify an application, the first thing to do is to identify the object classes. In this example, we will be dealing with three types of object: objects of type Order, objects of type Goods and objects of type Bill.

As a result, we declare these three object classes in the script:

```
objectclass Bill;
objectclass Goods;
objectclass Order;
```

Now, we need to specify the task classes that we are going to use. Let us for instance specify the task class needed for the workflow application. A taskclass is used to specify the interface of the task, namely its inputs and outputs, and is independent of the implementation of this application. The workflow application takes an object of type Order as input and returns an object of type Goods if the process is successful, nothing if it fails. These two outputs (success and failed) are just final outcomes. The resulting taskclass named ProcessOrder is fully specified using the following code:

```
taskclass ProcessOrder
```

```
{
   inputs
   {
      input main
      {
         order of class Order
      }
   };
   outputs
   {
      outcome failed
      {
      };
      outcome success
      {
         items of class Goods
      }
   }
};
```

The other task class that is interesting is the one for the task dispatch, as we want a task class for an atomic task. The way to do that is to use an `outcome abort` instead of just `outcome`. Let also assume that the dispatch task has only one input set (main) and two output sets (success and aborted). The set main and success having an object of type Goods associated to them. The resulting task class specification is:

```
taskclass Dispatch
{
   inputs
   {
      input main
      {
         items of class Goods
      }
   };
   outputs
   {
      outcome abort aborted
      {
      };
      outcome success
      {
         items of class Goods
      }
   }
};
```

Let's now instantiate the workflow application. We have to declare a compound task with as interface the one describe in the task class ProcessOrder.

```
compoundtask processOrderApplication of taskclass ProcessOrder
```

Then we need to specify implementation criteria allowing to specify some placement information and other information needed at run-time. The two criteria starting with GUI are

automatically generated by the GUI to keep the co-ordinates (GUI_X, GUI_Y) of the task on the GUI. "**Node**" is used to specify that you want the task controller running on the host kellah.

```
implementation
{
   "GUI_X" is "235";
   "GUI_Y" is "100";
   "Node" is "kellah"
};
```

The next stage is to map the inputs. As this particular task is the entire workflow application being modelled, its inputs are not mapped (as they are coming from «outside»). The mapping will be done at run-time with some object provided by an administration task. It has to be noticed that the inputs specified here are those declared in the associated task class.

```
inputs
{
   input main
   {
      inputObject order from
      {
      }
   }
};
```

Then we have to specify the component tasks, for simplicity, we have just considered the specification of one of them, dispatch. Similarly to what we did for processOrder, we first specify the task class and then the implementation criteria. The criteria with keys "**TaskCtrlFactory**" and "**TaskImpl**" are used by the workflow engine to choose a task control factory for the controller of this task as well as a task factory to map the task to an instantiation. Notice the use of the construct task instead of compoundtask as this is a primitive task (i.e. its implementation details can not described as a (sub) workflow). Having done that, we have to specify the dependencies. In this example, we only have one input set (main) with one associated object items. There are two dependencies specified:

- A temporal dependency (notification from) stating that before starting this task with the input set main, the task paymentAuthorisation has to have reached its success output.
- A data-flow dependency (inputobject from) stating that the input items should be mapped to the object items of the output set success of task checkStock.

The resulting specification for this task is:

```
task dispatch of taskclass Dispatch
{
   implementation
   {
      "GUI_X" is "477";
      "GUI_Y" is "184";
      "TaskCtrlFactory" is "Order";
      "TaskImpl" is "Dispatch";
      "Node" is "kellah"
   };
   inputs
   {
      input main
      {
         notification from
         {
            task paymentAuthorisation if output success
         };
         inputObject items from
         {
            items of task checkStock if output success
         }
      }
   }
};
```

Having described the task components as well as the dependencies involving them as targets, we now need to specify the mapping of the outputs of the workflow application. The outputs are described similarly to the inputs. In this case, the application was failing (outcome failed) if any its constituents failed. This is specified by the first notification from. Any of the three alternatives triggers the decision of reaching this output. The second output success can only be reached if its object is available (object items from dispatch if it reaches its success output) and the task paymentCapture reached its output success.

```
outputs
{
   outcome failed
   {
      notification from
      {
         task checkStock if output failed;
         task dispatch if output aborted;
         task paymentAuthorisation if output failed
      }
   };
   outcome success
   {
      notification from
      {
         task paymentCapture if output done
      };
      outputObject items from
      {
```

```
                items of task dispatch if output success
            }
        }
    }
}
```

Given this specification, each task can be viewed as an interface and a set of incoming and outgoing dependencies.

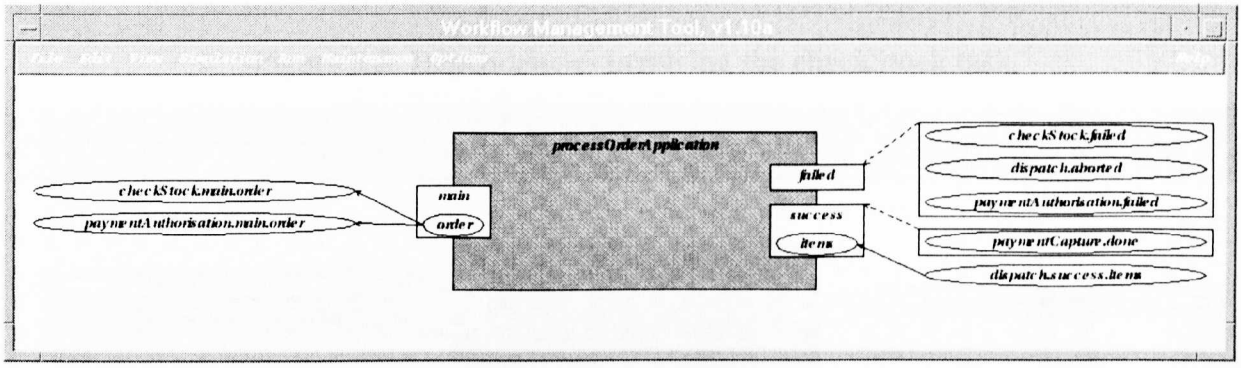


Figure 5.2: Dependencies involving the processOrderApplication compound task

In figure 5.2, the processOrder application is shown with its internal dependencies (e.g. dependencies involving its children and itself). All the dependencies on its input sets are outgoing, as at specification time, we don't know what can be fed to them. Similarly all the output sets have incoming dependencies on them (we don't know yet what will be using the result of the workflow. This is specific to this task as it represents the entire workflow application and its inputs are coming from the "outside" and similarly the «outside» uses its outputs.

Figures 5.3, 5.4, 5.5 and 5.6, show the resulting dependencies on the component tasks of the workflow application.

In order to specify a more detailed description of one of these components, you just need to replace it by a compound task with the same interface (task class) and to specify internally the components of the new task as well as their inter-dependencies. It has to be noticed that the only place where modifications need to be done is within the specification of the task being modified. This locality of the modification is an important feature of our model.

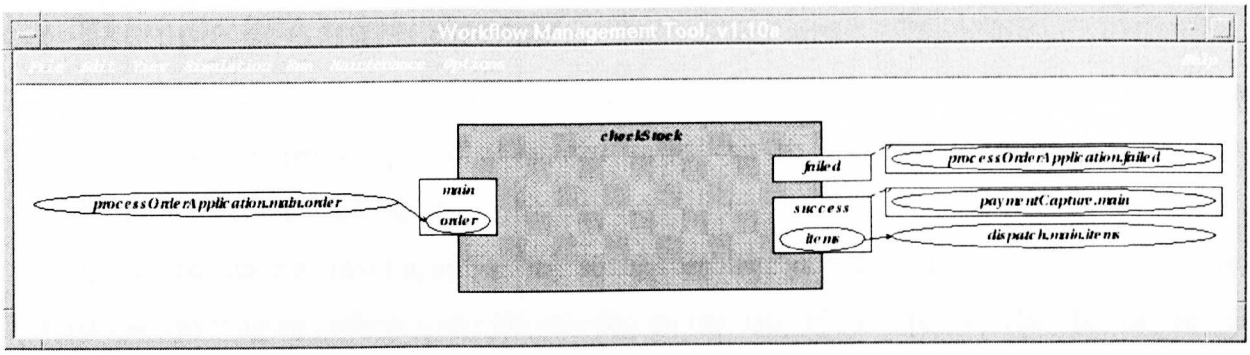


Figure 5.3: Dependencies involving the checkStock task

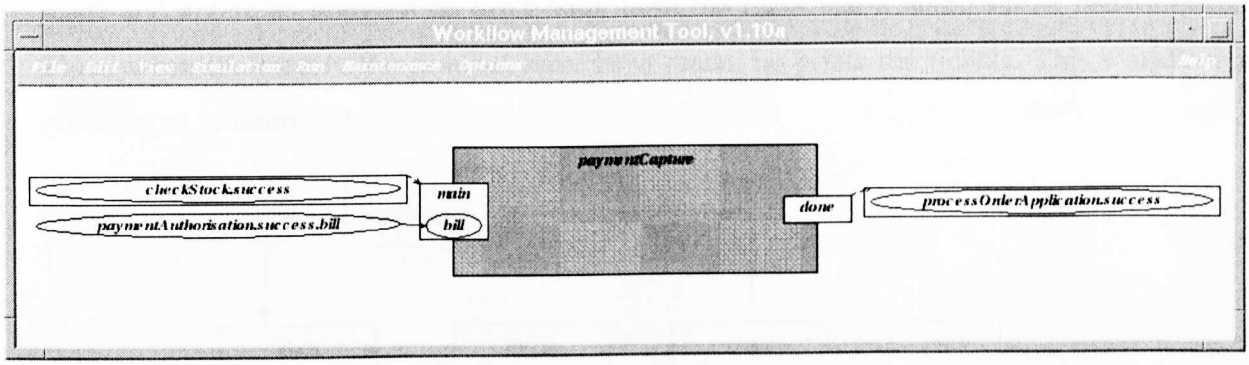


Figure 5.4: Dependencies involving the paymentCapture task

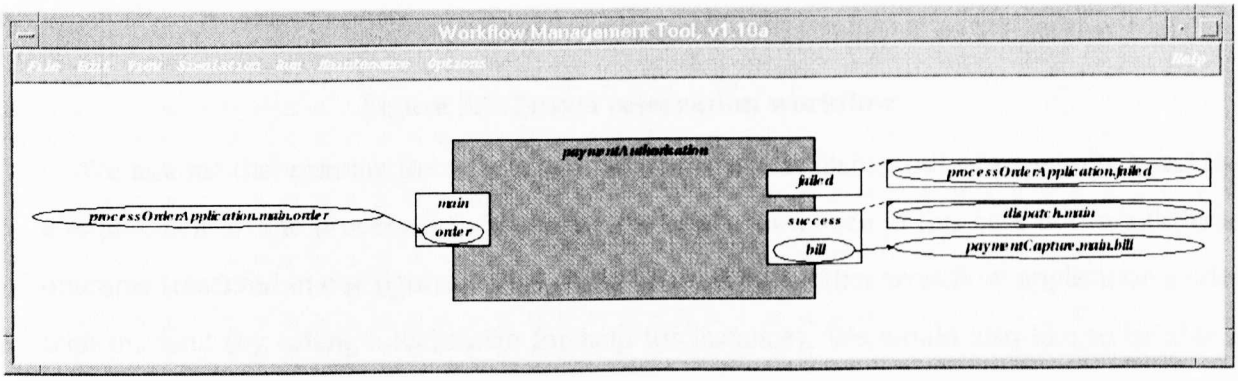


Figure 5.5: Dependencies involving the paymentAuthorisation task

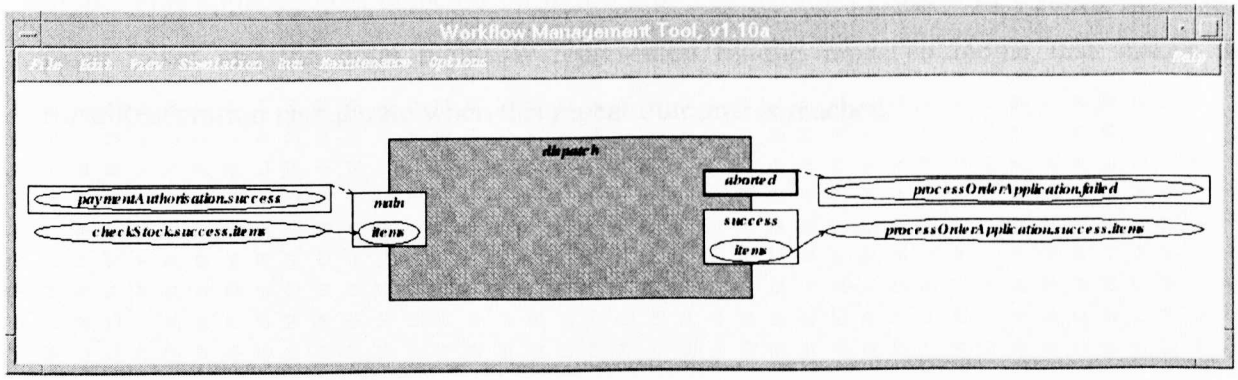


Figure 5.6: Dependencies involving the dispatch task

5.2- Example II: A travel agency

This example will show how to use a compound task to hide some details of an activity as well as how compensating and alternative tasks can be specified to provide some extra application level fault-tolerance.

Let us consider a travel agent selling some combined travel reservations flight plus hotel. First the travel agent gathers some information on the date of the trip, etc. Then he tries to find a combination of a flight and a hotel for its customer (he may have to cancel the reservations made and iterate its search if he can't book both the flight and a hotel) till he finds a match. Then, when the travel arrangements have been made, he prints the tickets. This workflow is represented in figure 5.7.

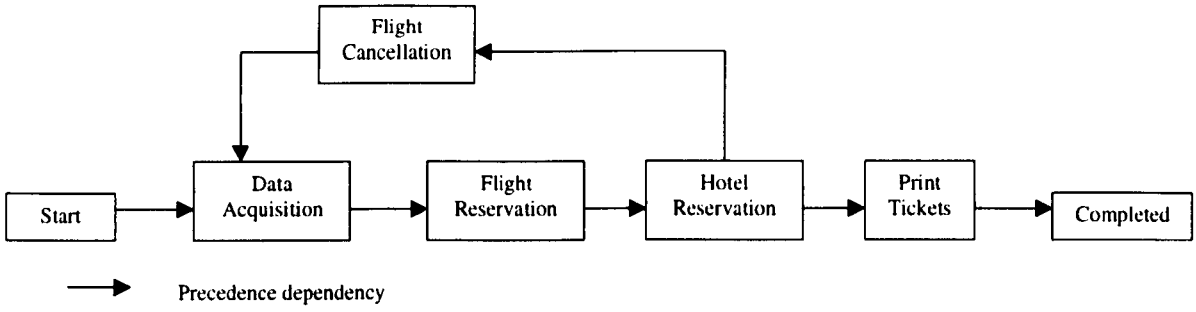


Figure 5.7:Travel reservation workflow

We assume that printing the tickets can fail (toner not available, out of paper, etc.) and that a requirement of the process is to terminate the application even in this case but in a different outcome (reserved in our figures). This could be used by another workflow application to deal with the fault (by calling a technician for help for instance). We would also like to be able to know as soon as the reservations were carried out the total bill to speed up the payment of the order. This application is depicted on figure 5.8. The iteration process of trying to reserve a plane ticket and the hotel nights is represented by the repeat outcome that makes the travelReservation task iterate when this repeat outcome is reached.

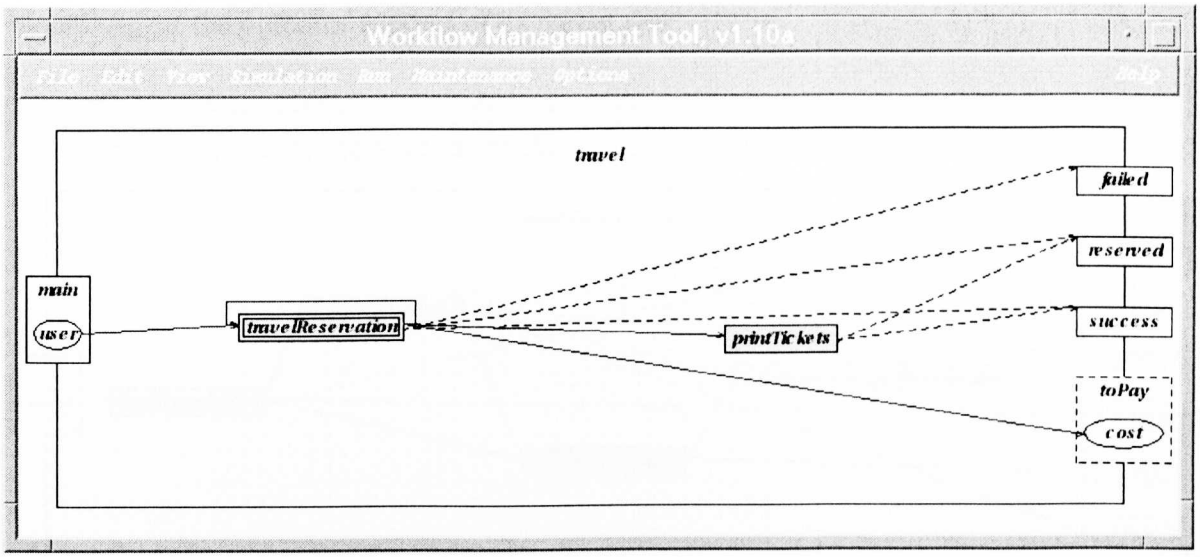


Figure 5.8: Overview of the travel task

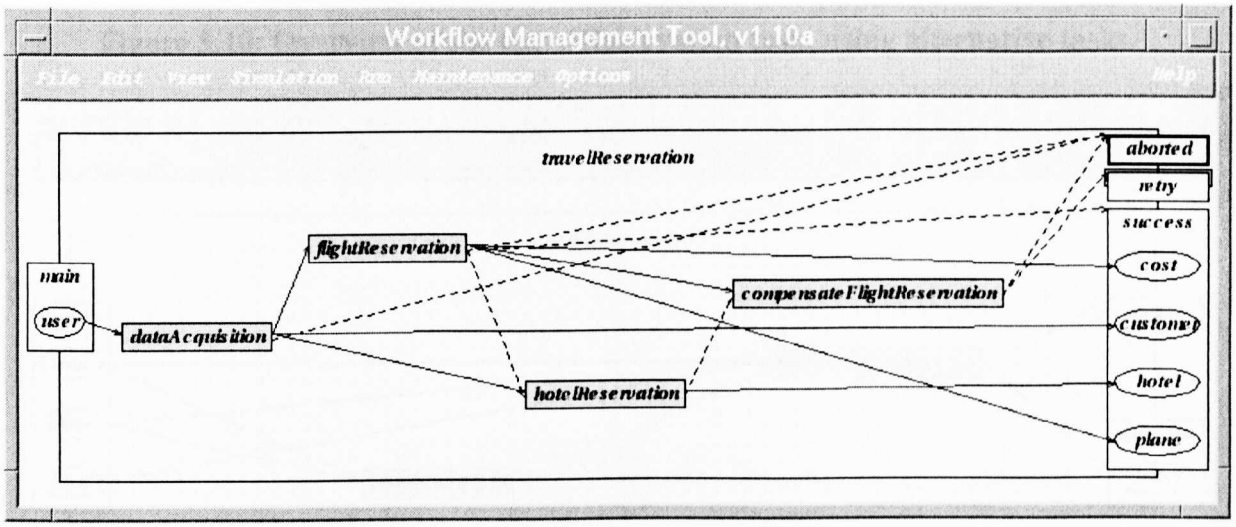


Figure 5.9: Overview of the travelReservation task

Finding the hotel and flight consists in gathering the data such as date, destination... from the customer, then trying to find a flight and a hotel if a flight was available. If a flight was reserved but a hotel could not be booked, a special compensating task must be run to undo the flight reservation. A graphic version of these requirements is shown in figure 5.9. Notice that the compensating task is provided as a normal task part of the workflow. Typically we will have two types of compensating tasks: one doing some forward error recovery and one doing some backward error recovery. In this example, as the task compensateFlightReservation undoes the effects of the task flightReservation, it is a backward error recovery task. Instead of having a compensate task for the flightReservation, we could also have had a forward recovery task for hotelReservation that would for instance try to book an hostel and whose success

would trigger the outcome success of its parent task.

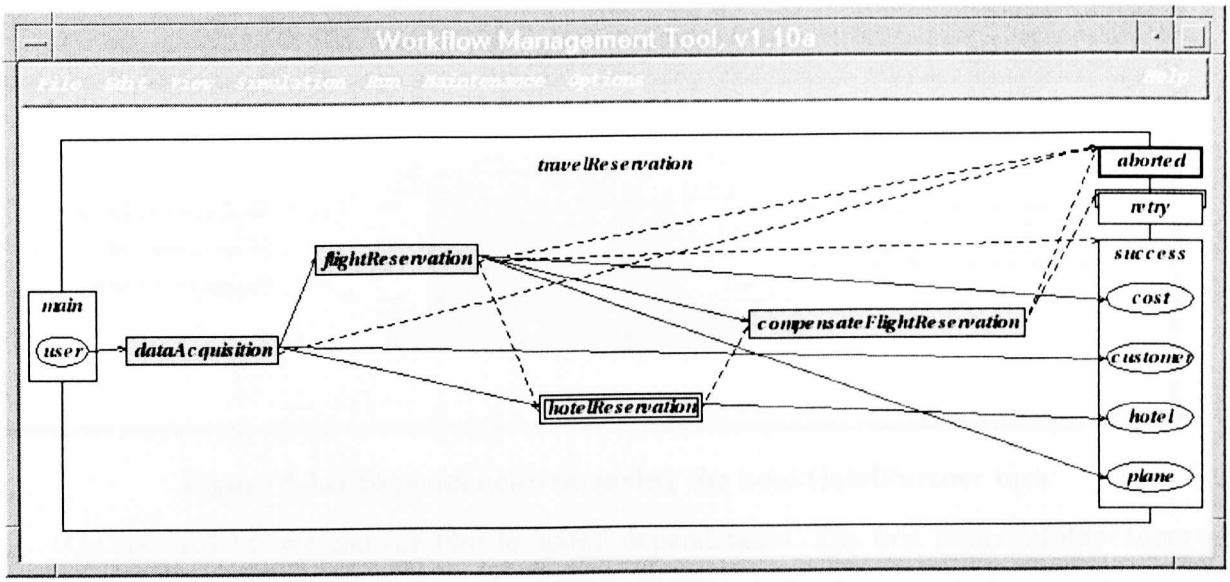


Figure 5.10: Overview of the travelReservation task, using alternative tasks.

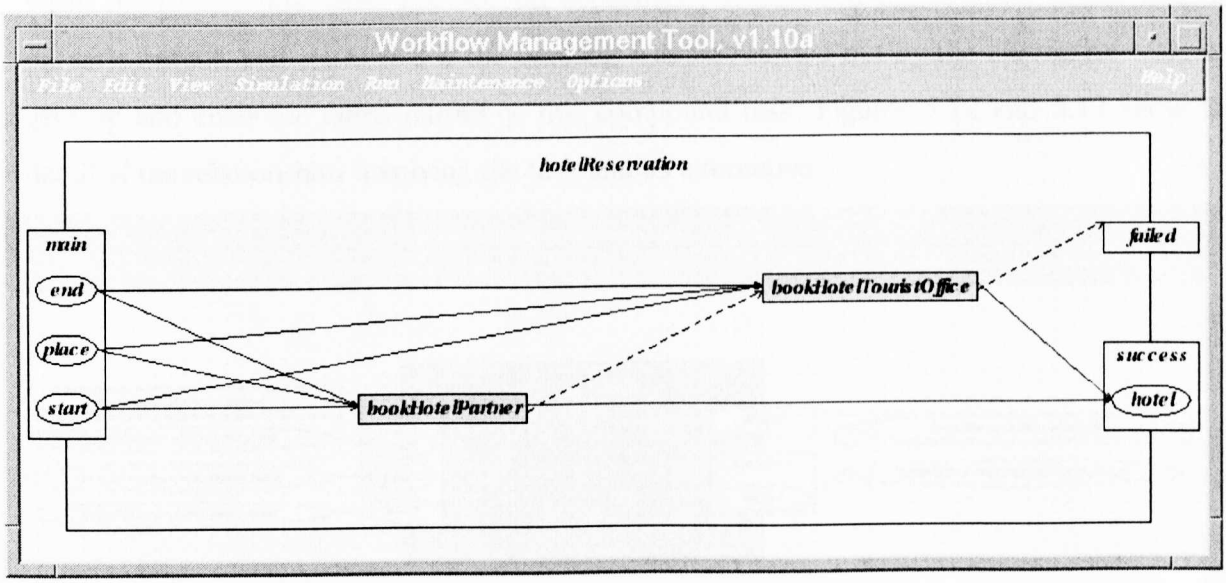


Figure 5.11: Details of the reliable hotelReservation task.

Let's now imagine that this travel agency has some special offers from a particular chain of hotels. As a result the agent has to check first whether he can book a room in that hotel and if it is not possible contact the tourist information centre database to book another hotel. In this case, we just have to replace the hotelReservation task by a compound task with the same interface consisting of the old basic task (renamed as bookHotelTouristOffice) as alternative of a new task trying to book a room with the hotel partner of the agency (named bookHotelPartner). The modified task travelReservation appears in figure 5.10 while the details of the new task are shown in figure 5.11.

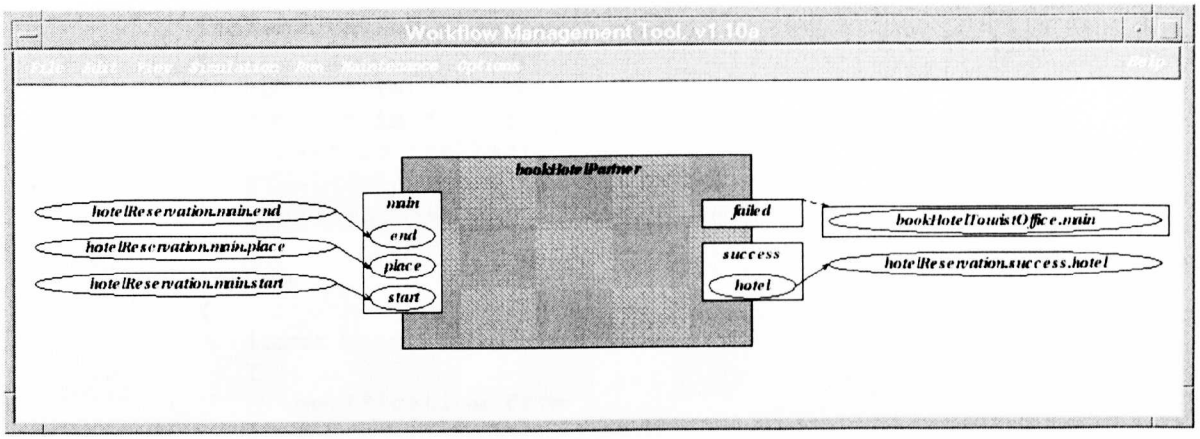


Figure 5.12: Dependencies involving the bookHotelPartner task

On figure 5.11, we can see two temporal dependencies. The first temporal dependency is between bookHotelPartner and bookHotelTouristOffice indicating that the latter task is started when the former task fails. The second dependency is between the alternative task and the parent's output, and states that if the booking was not carried out by the alternative task, we give up and enter the failed output of this compound task. Figures 5.12 and 5.13 show the detail of the relationships involving the task and its alternative.

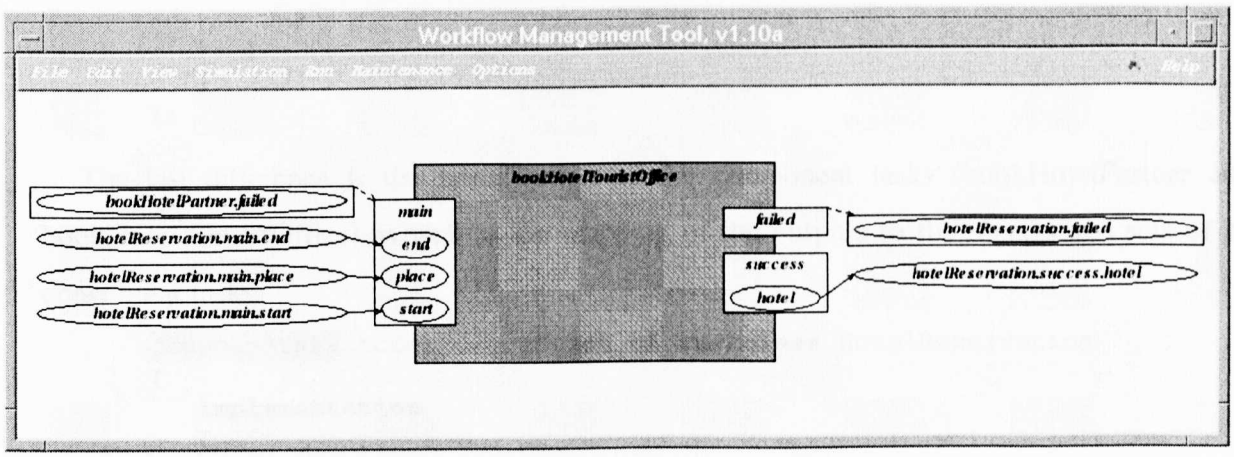


Figure 5.13: Dependencies involving the bookHotelTouristOffice task

The differences between the specifications with and without the alternative hotelReservation tasks are now discussed. First of all, there are no differences between the two specifications outside the specification of the task affected. The code relative to the specification of task hotelReservation is given below, with the differences underlined by a grey background. There are three differences. The first one is that the task is declared as a compound task and no longer as a basic task. The second difference is that it loses the implementation criteria related to the run-time mapping of the task instance.

```
task hotelReservation of taskclass HotelReservation
{
   implementation
   {
      "GUI_X" is "382";
      "GUI_Y" is "192";
      "Host" is "kellah";
      "TaskCtrlFactory" is "Travel";
      "TaskImpl" is "hotelReservation"
   };
   inputs
   {
      input main
      {
         notification from
         {
            task flightReservation if output success
         };
         inputObject end from
         {
            end of task dataAcquisition if output success
         };
         inputObject place from
         {
            place of task dataAcquisition if output success
         };
         inputObject start from
         {
            start of task dataAcquisition if output success
         }
      }
   }
};
```

The last difference is the specification of the component tasks (bookHotelPartner and bookHotelTouristOffice) as well as the mapping of the outputs to the objects and sets of its constituent tasks.

```
compoundtask hotelReservation of taskclass HotelReservation
{
   implementation
   {
      "GUI_X" is "382";
      "GUI_Y" is "192";
      "Host" is "kellah"
   };
   inputs
   {
      input main
      {
         notification from
         {
            task flightReservation if output success
         };
         inputObject end from
         {
            end of task dataAcquisition if output success
```

```
        };
        inputObject place from
        {
           place of task dataAcquisition if output success
        };
        inputObject start from
        {
           start of task dataAcquisition if output success
        }
     }
};
task bookHotelPartner of taskclass HotelReservation
{
   implementation
   {
      "GUI_X" is "414";
      "GUI_Y" is "208";
      "Host" is "www.hilton.com";
      "TaskCtrlFactory" is "Travel";
      "TaskImpl" is "hotelReservation"
   };
   inputs
   {
      input main
      {
         inputObject end from
         {
            end of task hotelReservation if input main
         };
         inputObject place from
         {
            place of task hotelReservation if input main
         };
         inputObject start from
         {
            start of task hotelReservation if input main
         }
      }
   }
};
task bookHotelTouristOffice of taskclass HotelReservation
{
   implementation
   {
      "GUI_X" is "584";
      "GUI_Y" is "120";
      "Host" is "www.travel-reservation.com";
      "TaskCtrlFactory" is "Travel";
      "TaskImpl" is "hotelReservation"
   };
   inputs
   {
      input main
      {
         notification from
         {
            task bookHotelPartner if output failed
         };
```

```
                inputObject end from
                {
                    end of task hotelReservation if input main
                };
                inputObject place from
                {
                    place of task hotelReservation if input main
                };
                inputObject start from
                {
                    start of task hotelReservation if input main
                }
            }
        }
    };
    outputs
    {
        outcome failed
        {
            notification from
            {
                task bookHotelTouristOffice if output failed
            }
        };
        outcome success
        {
            outputObject hotel from
            {
                hotel of task bookHotelPartner if output success;
                hotel of task bookHotelTouristOffice if output success
            }
        }
    }
};
```

Notice that the mapping of the output object hotel associated to the compound task hotelReservation can be done to two objects: either to the output object hotel associated to the outcome success of bookHotelPartner or to the corresponding object of the task bookHotelTouristOffice. Order of the alternative is significant, as the first in the list will be given preference over the second one and so on. Similarly the order of the input and output sets are important, in the event of several of them being able to be triggered, the one chosen is the highest in the list. In this particular case, this feature is not that important as anyhow the two objects will never be available at the same time.

5.3- Example III: Network fault management

In this example, we consider the modelling of a process dealing with network faults and resulting re-negotiation of the services provided if needed. A network is subject to some

possible faults, which trigger alarms. The Alarm Correlation Bridge (AC Bridge) receives such alarms (for instance unspared ATM, etc.) and forwards them to the Service Impact Analysis Agent (SIA). In turn this agent finds out the activities impacted by the alarms as well as their costs (loss of revenue, penalties, etc.). These activities are sent to a Service Impact Resolution Agent (SIR) that proposes several possible solutions to address the problem. This is done via a Service Level Agreement (SLA) negotiation that will for instance negotiate to decrease the quality of a video, reschedule a service, re-route a service, abandon as service, etc. The system then takes corrective actions. We assume that on reception of an alarm the AC bridge starts a workflow dealing with the treatment of the fault.

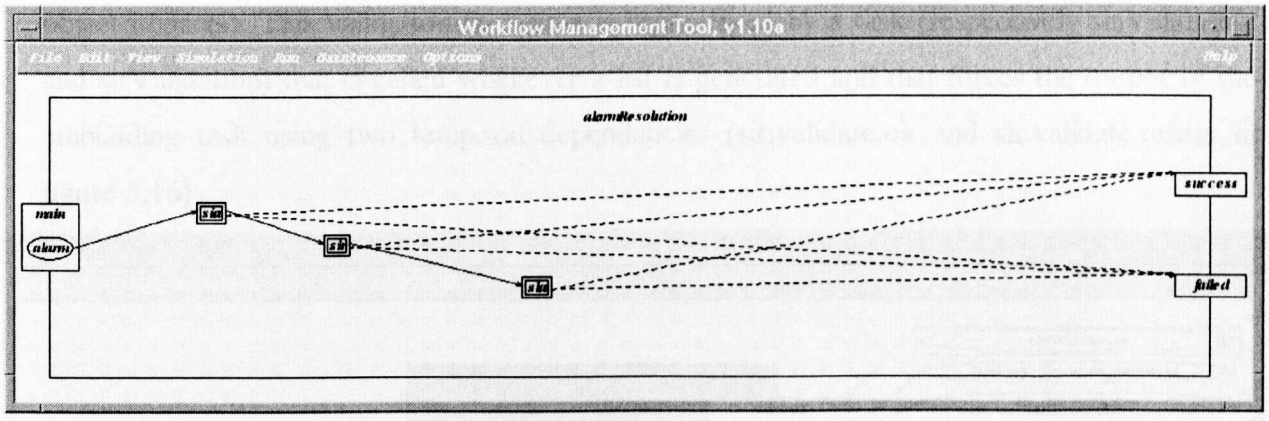


Figure 5.14: Overview of the alarmResolution task

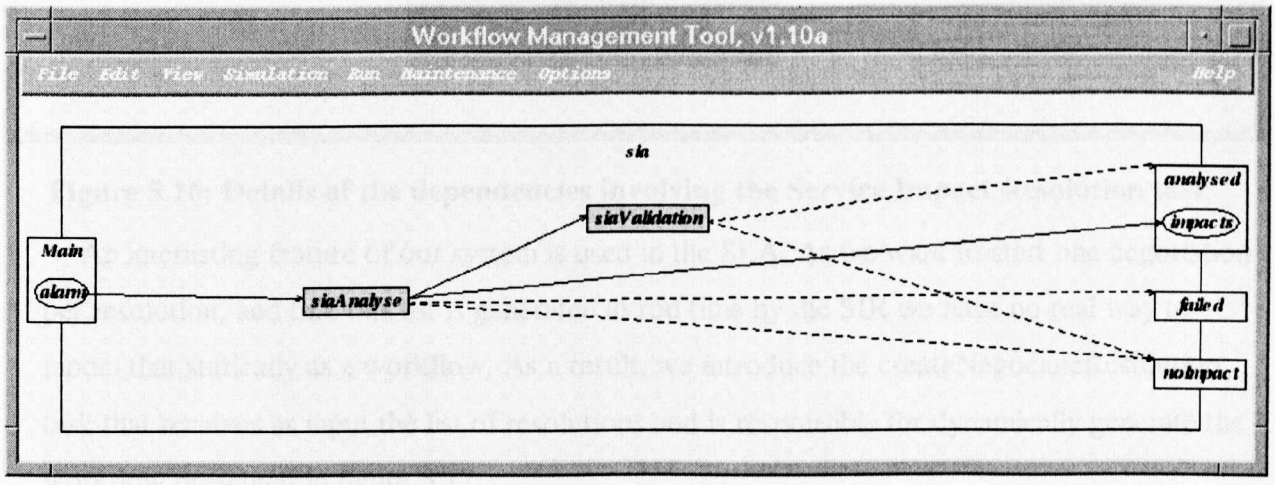


Figure 5.15: Overview of the Service Impact Analysis task

The process can be seen on figure 5.14: on reception of the alarm, it is passed to the SIA that can either fails, decides that there is no follow up and trigger the output success of the workflow, or generate a list of impacted services. This list is then passed to the SIR that has the same choices, but generates a list of possible resolutions. This list is itself used by the SLA

that deals with it and on completion trigger the completion of the workflow.

The overview and details of the SIA (and SIR) are now presented. The overview of SIA is depicted on figure 5.15 and the details of SIR are shown on figure 5.16. Both of these two processes are identical except for the type of data they are supposed to be dealing with. Initially the siaAnalysis (respectively sirAnalysis task) creates the list of impacted services (respectively resolutions) if there were some. If this list is not empty, it is then presented to a human responsible for validating or invalidating the software decision. It has in particular no way to change the list he is presented with. Changing that would just require to change the task class so that it return the potentially modified list and use that list as impacted list (output object impacts). This validation by a user is represented by a task (respectively siaValidation and sirValidation) that is called whenever a list is generated and that forces the output of the embedding task using two temporal dependencies (sir.validate.ok and sir.validate.refuse in figure 5.16)

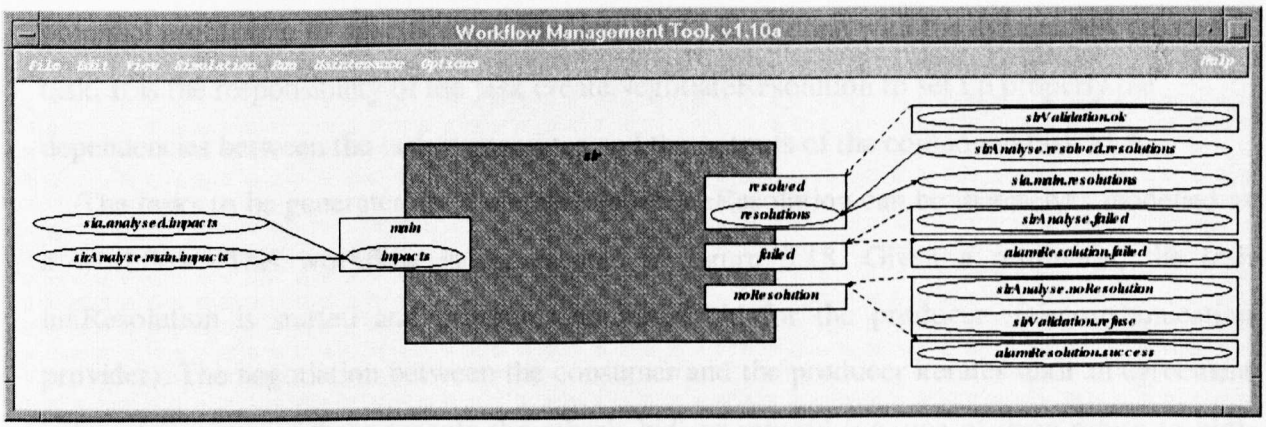


Figure 5.16: Details of the dependencies involving the Service Impact Resolution task

An interesting feature of our system is used in the SLA. As we want to start one negotiation per resolution, and that this list is generated at run time by the SIR we have no real way to model that statically as a workflow. As a result, we introduce the createNegociateResolution task that receives as input the list of resolutions and is responsible for dynamically generate the workflow presented in figure 5.17.

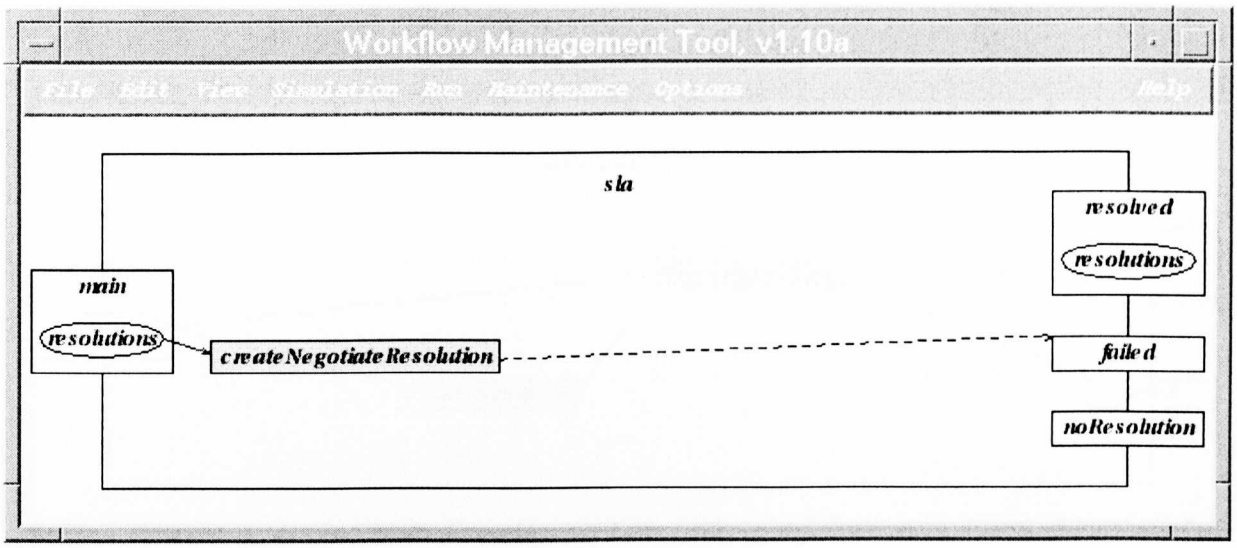


Figure 5.17: Overview of the Service Level Agreement task

It has to be noticed that the mapping of the SLA outputs is not fully specified. As a result some warnings will be generated when the specification is checked to let the user know of a potential problem in its specification. The mapping will be done with the dynamically created task. It is the responsibility of the task createNegotiateResolution to set up properly the dependencies between the task it generates and the outputs of the compound task SLA.

The tasks to be generated by the createNegociateResolution can be themselves modelled as a workflow. This workflow is represented in figure 5.18. Given a resolution, the task initResolution is started and generates an initial bid for the producer (telecommunication provider). The negotiation between the consumer and the producer iterates until an agreement is found (i.e. one of them accepts the other's bid) or refused (i.e. one of them refuse to bid). This iteration is modelled by a repeat output that is feeding back the modified objects after the round. The details of the negotiation process are shown in figure 5.19.

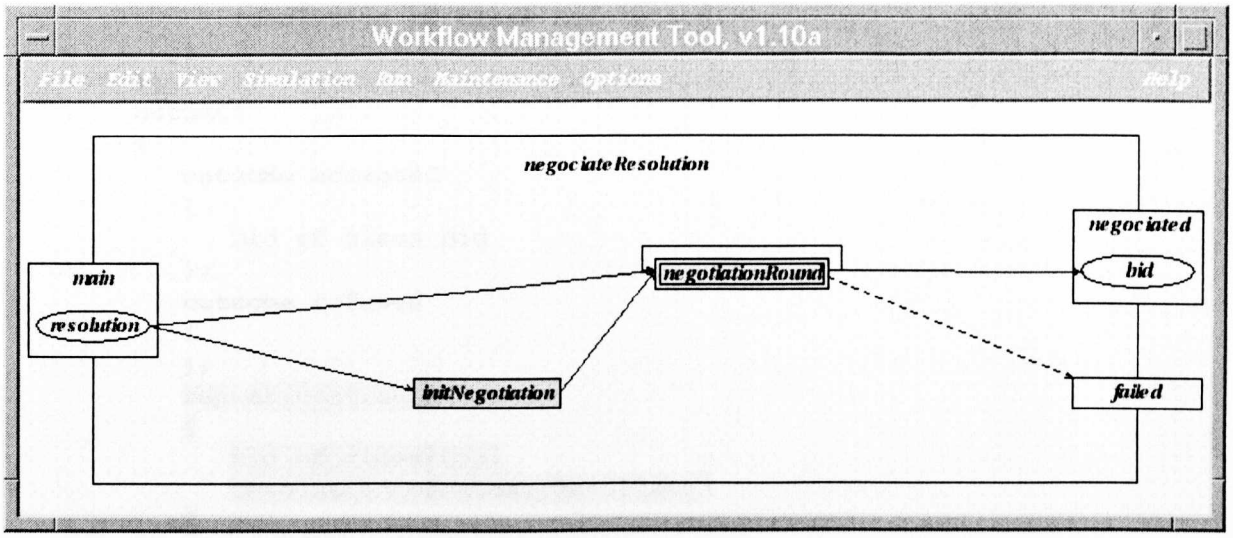


Figure 5.18: Overview of the Negotiation Resolution task

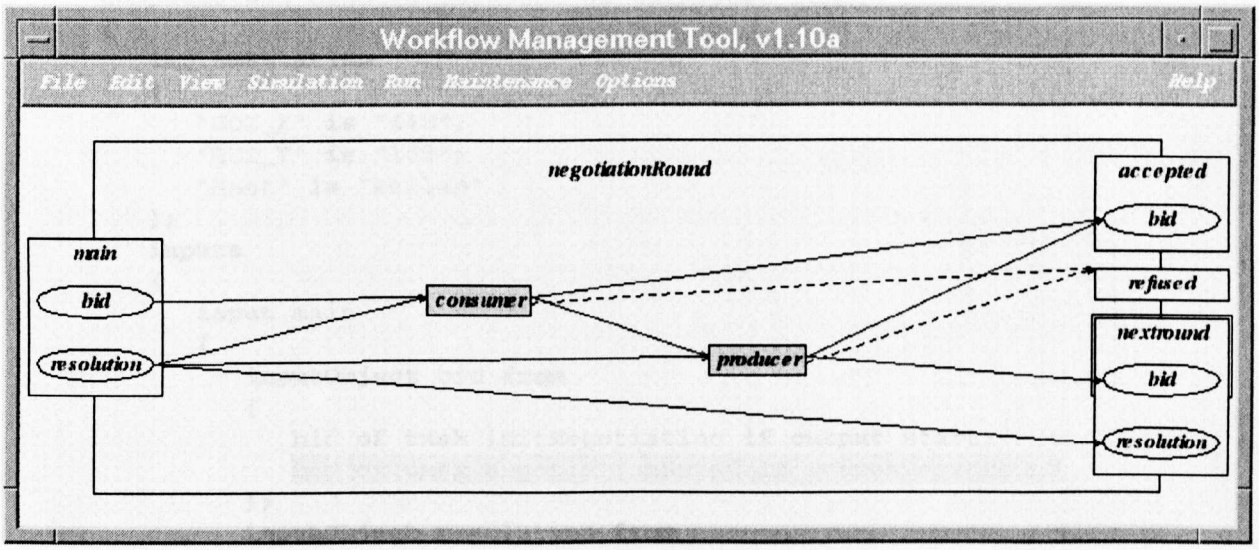


Figure 5.19: Overview of a round of negotiation of the SLA

In figure 5.19, you can also notice that the resolution object of the input set is directly used by the output set nextround. This is due to the fact that the resolution is not modified during the negotiation round. This ability of our system to forward inputs to its outputs can be used to implement some routing tasks. The repeat output sends back both bid and resolution. The code associated with that is now presented, the lines related to the loop are underlined by a grey background:

```
taskclass SLAbid
{
   inputs
   {
      input main
      {
         bid of class Bid;
```

```
            resolution of class Resolution
        }
    };
    outputs
    {
        outcome accepted
        {
            bid of class Bid
        };
        outcome refused
        {
        };
        repeat nextround
        {
            bid of class Bid;
            resolution of class Resolution
        }
    }
};
compoundtask negotiationRound of taskclass SLAbid
{
    implementation
    {
        "GUI_X" is "448";
        "GUI_Y" is "105";
        "Host" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject bid from
            {
                bid of task initNegotiation if output start;
                bid of task negotiationRound if output nextround
            };
            inputObject resolution from
            {
                resolution of task negociateResolution if input main;
                resolution of task negotiationRound if output nextround
            }
        }
    };
    task consumer of taskclass SLAbidRound
    {
        //body suppressed
    };
    task producer of taskclass SLAbidRound
    {
        // body suppressed
    };
    outputs
    {
        outcome accepted
        {
            outputObject bid from
            {
                bid of task consumer if output accepted;
```

```
                bid of task producer if output accepted
            }
        };
        outcome refused
        {
            notification from
            {
                task consumer if output refused;
                task producer if output refused
            }
        };
        repeat nextround
        {
            outputObject bid from
            {
                bid of task producer if output nextround
            };
            outputObject resolution from
            {
                resolution of task negotiationRound if input main
            }
        }
    }
};
```

As can be seen from this example, in order to use a loop, an output of type repeat outcome has to be declared in the task class used. Then its associated objects can be fed as input objects, it itself can be used as source of a temporal dependency having as target an input set of the same task. The mapping of this output is similar to the other types of outputs.

In this chapter, the main features of our language were presented, using examples. This is not an extensive presentation of the possibilities of our language. For instance, input time-out tasks can also be used. This is particularly useful to model deadlines. A deadline will be modelled as an input time-out task that when it completes provides some alternative inputs to the task concerned with the deadline. In this case, alternative input sets are also useful as they allow starting the task in a degenerated state (e.g. without all its inputs) and still being able to carry on.

In the next chapter the toolkit implemented to support the system will be presented. The screen dumps used as support in this chapter were generated using the toolkit. Using the toolkit also allows users to ignore the textual language and use a graphical notation to specify their application.

Chapter 6

Toolkit

The main aim of the Workflow Management Toolkit (WfMT) is to provide high level easy to use facilities to users to enable them to compose workflow applications, and then execute and monitor them. In this chapter, the reader will find a detailed description of the toolkit. After giving an overview of the toolkit as a whole, the notion of class of users for the toolkit will be first introduced, and then the workflow script servers will be described including the protocol used to communicate with it. Then we will present in turn how the toolkit allows you to specify, simulate, execute and monitor a workflow application. Further information is available in the OpenFlow documentation [81] and in the Toolkit User Manual in appendix B.

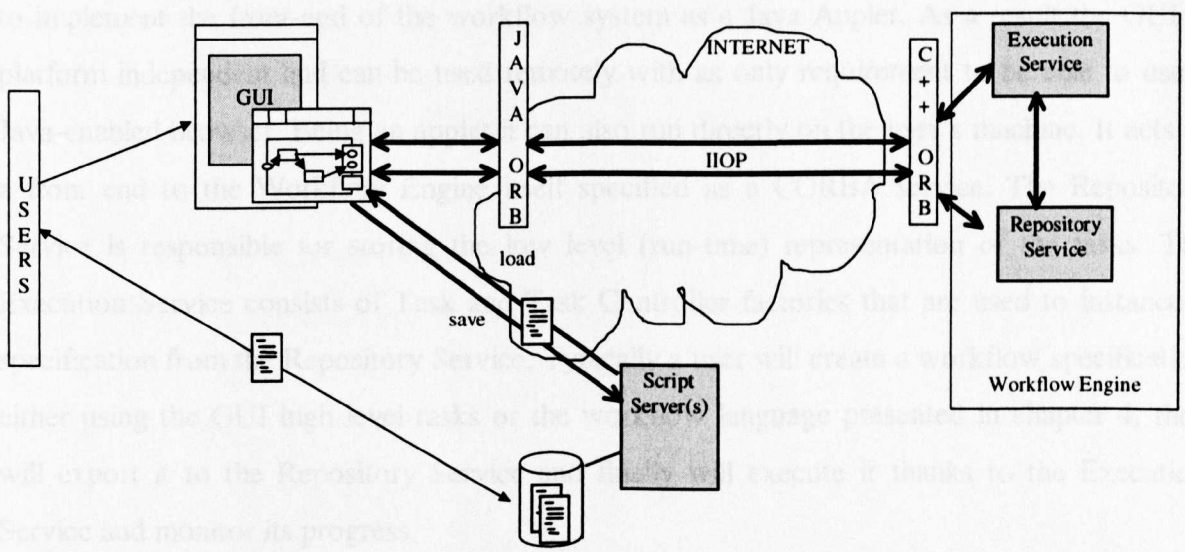


Figure 6.1: Graphical representation of the workflow system

6.1- Overview

Components

Keeping in mind the software structure of the system as well as the relationship between its different components described in chapter 3, and depicted in figures 3.2 and 3.3, a graphical representation of the system with the communication between its components is given in figure 6.1.

The WfMT is composed of three main components: the Graphic user Interface (WfGui), a script server (WfSS) and a workflow engine (WfE). Typically users will only interact directly with the WfGui.

In order to provide greater flexibility by allowing incorrect/incomplete script specifications, workflow scripts servers were added to store specifications being created. Users wishing to use the textual language (described in chapter 4) to specify a workflow application can also directly export it to a workflow script server. Then they can load it into the toolkit using the WfGui. Later on, they can also get it back as a script, modify it and export it back to a workflow script server.

A GUI is provided in order to facilitate the specification, execution and monitoring of the workflows. As one of the requirements was to be able to use the workflow management system from an heterogeneous set of machines and possibly from remote hosts, it was decided to implement the front-end of the workflow system as a Java Applet. As a result the GUI is platform independent and can be used remotely with as only requirement to be able to use a Java-enabled browser. Being an applet it can also run directly on the user's machine. It acts as a front end to the Workflow Engine itself specified as a CORBA service. The Repository Service is responsible for storing the low level (run-time) representation of the tasks. The Execution Service consists of Task and Task Controller factories that are used to instance a specification from the Repository Service. Typically a user will create a workflow specification either using the GUI high level tasks or the workflow language presented in chapter 4, then will export it to the Repository Service and finally will execute it thanks to the Execution Service and monitor its progress.

The main features that the toolkit provides are:

1. Creation of new workflows either by loading a script from a WfSS or by using the graphical notation presented in the previous chapter.
2. Extending, modifying existing specifications

3. Checking workflow applications for loops, and other errors
4. Simulating workflow applications
5. Instantiating and monitoring workflow applications

6.2- Classes of Users

In order to use the Toolkit, you need to login to the GUI by providing a user name and a password. The Toolkit checks these data against a list of registered users and if it finds a match, it also affect to the user a class of connection and an initial name context from the Name Service where all user related servers and workflow objects are kept.

Three different classes of users have been created:

- *Maintainers*: this is the equivalent of the UNIX supervisors. They have all the options available to them. They can create new users on-line as well as change their class of connection or path used for the name server using the form depicted in figure 6.3.
- *Designers*: they can create their own specifications, but can not modify directly the specifications stored in the repository service.
- *Users*: they can only monitor what is happening in the workflow. In particular they can not create or modify specifications.

It has to be noticed that maintainers get a super-set of the features given to designers and themselves getting a super set of the features given to users.

6.3- Workflow model using the WfGui

The WfGui uses the three main objects of our system:

- Object Class
- Task Class
- Task

In the next paragraphs, the models of these objects will be introduced.

6.3.1 ObjectClass model

It is represented as an object with as data a String to keep its name and a list of the TaskClassObjects of this class.

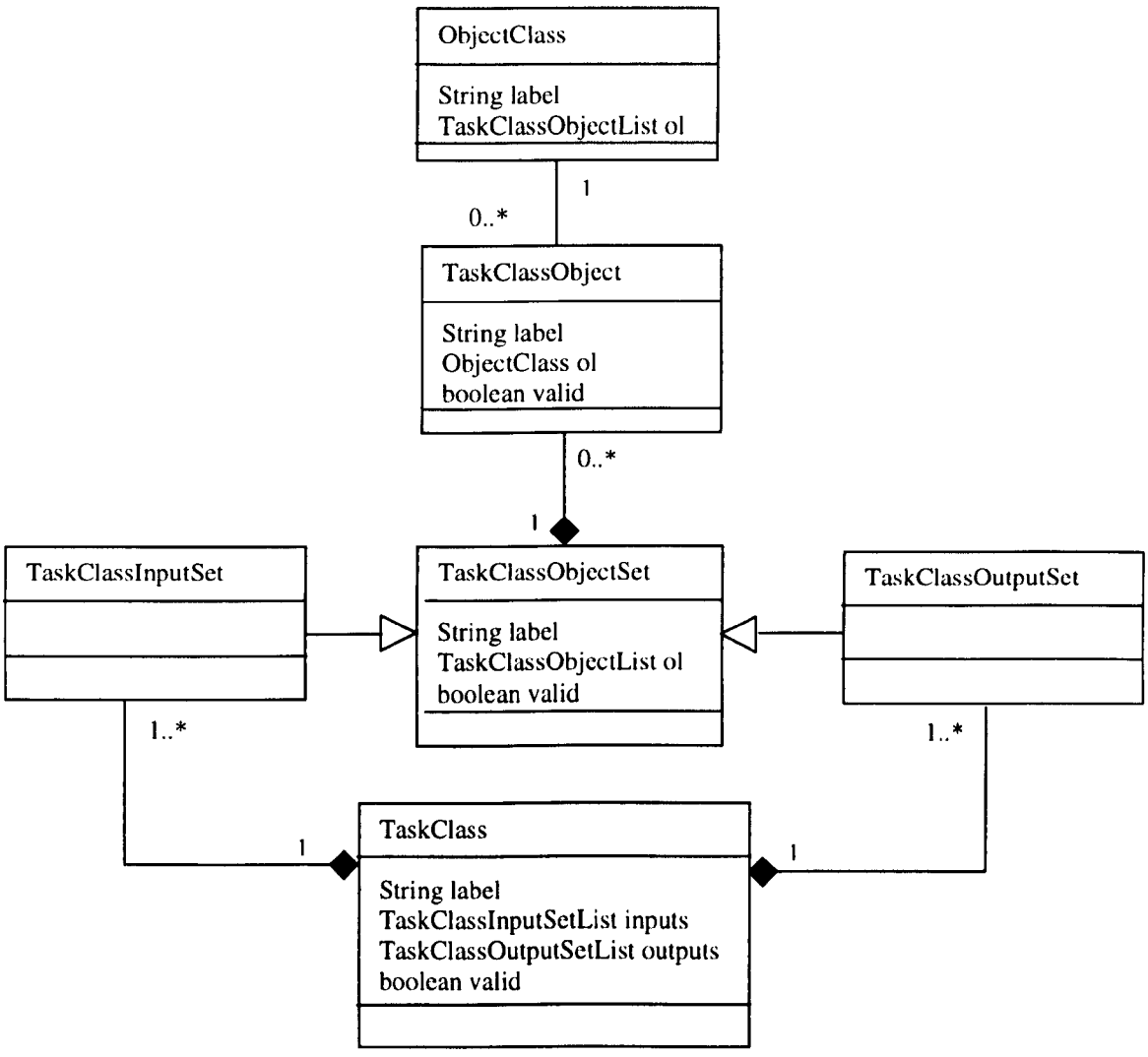


Figure 6.2: UML class diagram (excluding task components)

6.3.2 TaskClass model

A TaskClass is represented by an object with as data a String to keep its name and a list of Task of this class. It also includes a list of TaskClassInputSets and of TaskClassOutputSets. TaskClassInputSets (respectively TaskClassOutputSets) are represented by objects with as data a String to keep their name as well as a list of TaskClassObjects. A TaskClassObject itself is represented by objects with as data a String to keep its name as well as a reference to its ObjectClass. Each of these objects also has a boolean stating whether it is valid or not. The UML class diagram for the TaskClass model is given in figure 6.2. It has to be noticed that the methods associated to these classes do not appear on the diagram.

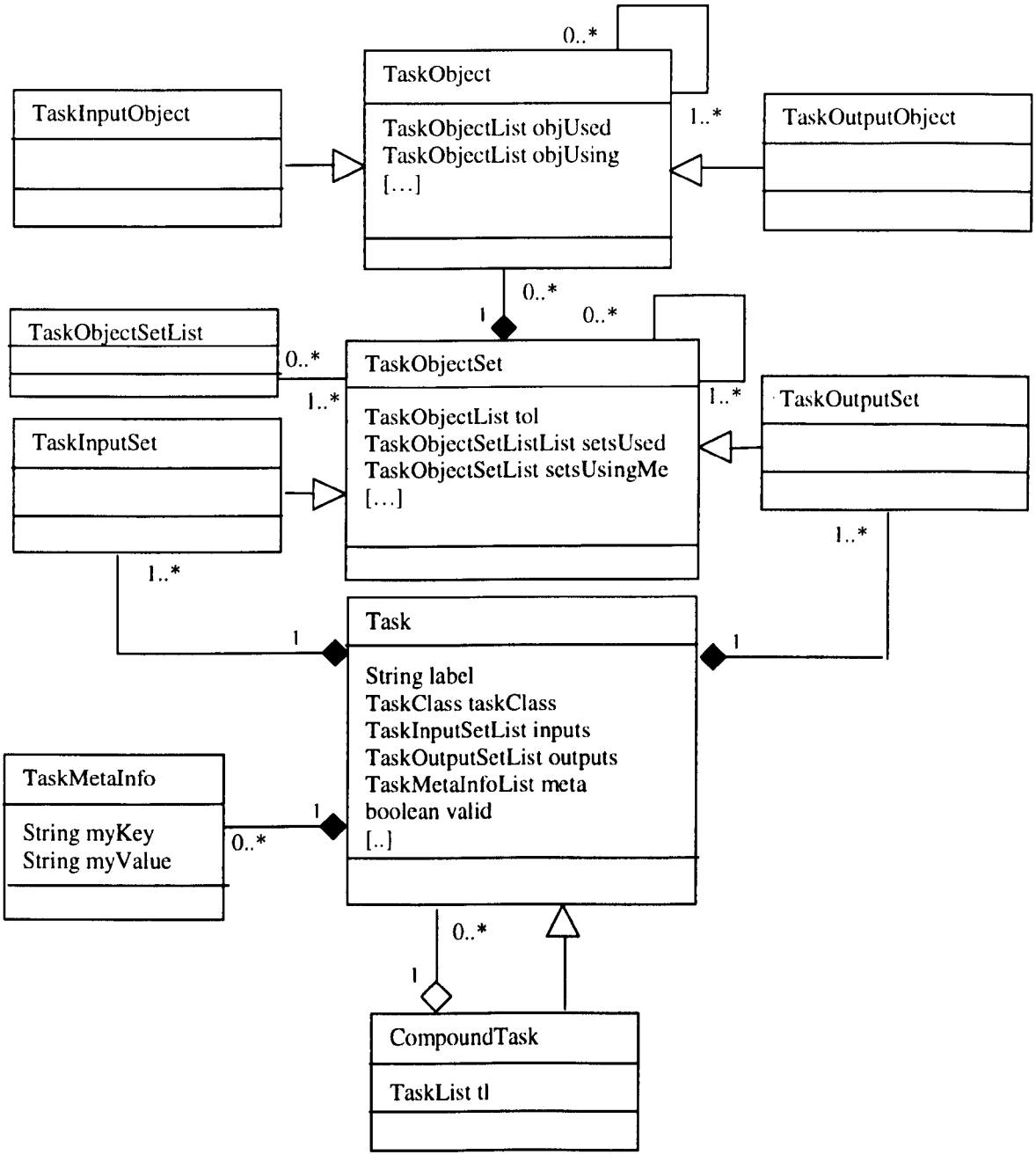


Figure 6.3: UML-like class diagram (excluding TaskClass components)

6.3.3 Basic task and compound task models

A basic Task is modelled by an object of class Task with as data a String for the name of the task, a reference to an object of class TaskClass (its associated TaskClass), a list to objects of class TaskMetaInfo that are used at instantiation time by the Task factory. An object of class TaskMetaInfo represents an instantiation criterium used to choose the best binding at run time for a task. It is modelled by a couple key-value. In the current implementation, the value is of

type String. .

A Task object also contains a list of TaskInputSets as well as a list of TaskOutputSets. An object of task TaskInputSet (respectively TaskOutputSet) contains a list of TaskInputObjects objects (respectively TaskOutputObjects objects), as well as a list of lists of TaskObjectSets (either TaskInputSets or TaskOutputSets) which represents the alternative sets of temporal dependencies on that TaskObjectSet. It also includes a list of the TaskObjectSets that are using it as source of dependencies. It also contains two strings, a Task object and a TaskClassObjectSet object. The two first Strings are used while loading a workflow script from the WfSS and if the TaskObjectSet is invalid, while the last two are used when the TaskObjectSet has been registered or if it was created using the WfGui (with a valid TaskObjectSet).

A TaskInputObject (respectively TaskOutputObject) contains two lists of TaskObjects (either TaskInputObject or TaskOutputObject). One list contains the data delegations associated to this TaskObject and the other contains a list of other TaskObjects using this TaskObject as source of data delegations. Similarly to TaskObjectSets, TaskObjects also keep their names in two ways:

- As three strings and an integer, to store the names of their parent Task, parent TaskObjectSet, their own name and their type (TaskInputObject/TaskOutputObject) while a script is being loaded,
- As a Task, a TaskObjectSet and a TaskClassObject when it becomes valid.

A compoundTask also keep a list of Tasks to represent its components.

This is represented in figure 6.3.

6.4- Workflow File System (WfSS)

This service is similar to FTP in that it is used to transmit some script specifications from a normal file system (such as UNIX) to the workflow toolkit and reverse. The script server was implemented as a Java application and can be run on all platforms supporting Java. It is a multi-threaded server supported multiple clients connecting simultaneously.

6.4.1 Connecting to a workflow script server

As a user, the first thing that you probably want to do is change the default script server to be used. For security reasons, the Toolkit can only use servers located on a couple of machines

of the domain ncl.ac.uk. This is due to the fact that in order to bypass the security managers of the browsers such as Netscape, we have used a proxy for the WFSS. This proxy implemented as a CGI has been coded to only forward requests if they are aimed at this subset of machines, due to security restrictions imposed by the webmaster where it was run.

Once a valid WfSS has been declared, you can load or save your specification as a script using the script server. The aim of the script servers is to enable the user to save its work as a workflow script wherever wanted.

6.4.2 Protocol

We will now describe the protocol used to communicate with the workflow script server. The protocol can be divided into three phases, first a hand shaking, then some work being performed and then closing the session. The protocol is depicted on figure 6.4.

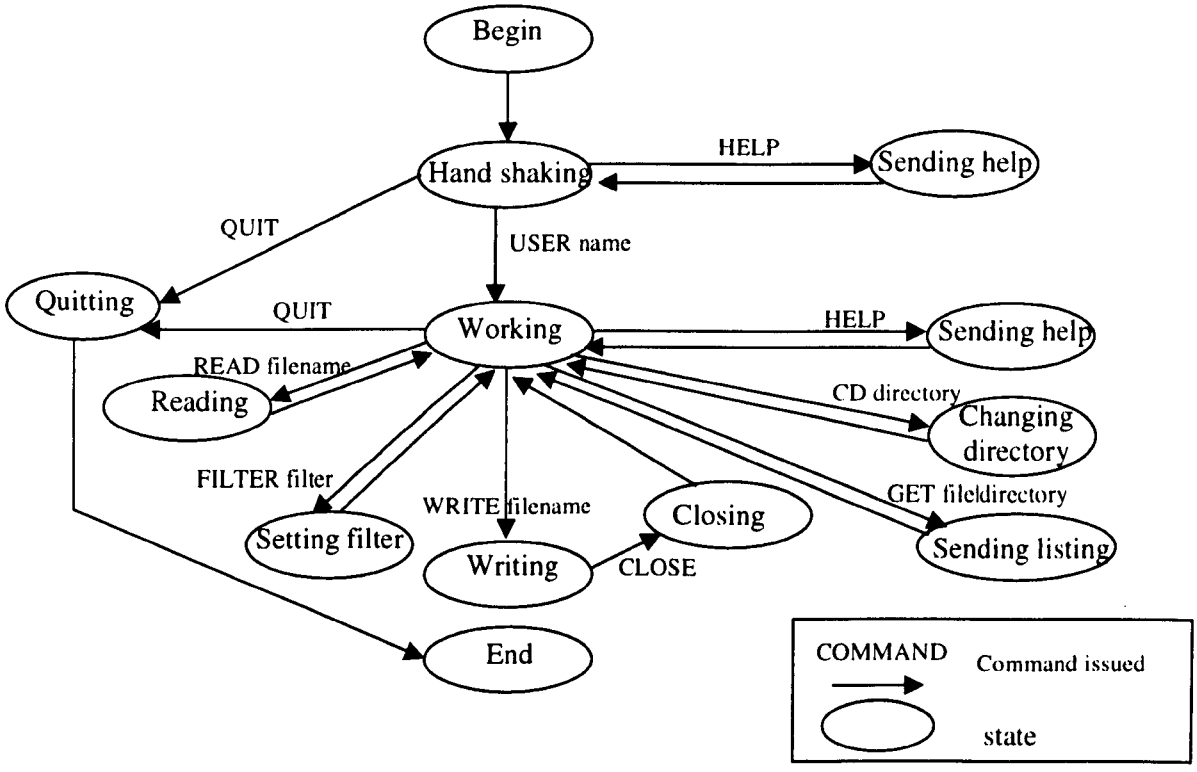


Figure 6.4: State transitions on server side of the WfSS

When the server is contacted, its reaction is to start a new thread to deal with it. This new thread exchanges messages with its associated client. There are eleven types of messages: connected, error, get file, get directory, open succeeded, open failed, close, quit, comment, success and data. Each type of message is associated with an operation code that is send back before its content.

Once created, the thread starts sending a message of type comment to the client. The aim of this message is to let the client know the version of the server. At this stage, the client can issue three commands:

- *QUIT* to close the session. The success of the command is acknowledged by a message of type quit,
- *HELP* to get a list of command supported,
- *USER name* to give its name to the server.

On reception of a message "user name", the server sends back a message of type connected back to the client and that's the end of the first phase.

Once the protocol is in its second phase, the client can get some work done. The server supports eight commands:

- *HELP* to list the commands available and sending them back as messages of type comment,
- *QUIT* to close the session. The success of the command is acknowledged by a message of type quit,
- *GET [file|directory]* to send back the list of files or directories in the current directory respectively as messages of type get file and get directory,
- *CD directory* either to get the path of the current directory or to move in the directory tree. The path of the current directory following the keyword CWD is send as comment if the new directory is the same as the old one, otherwise the new path following the keyword PWD is send as a message of type success,
- *WRITE filename* to start writing in the file filename whatever is sent by the client till reception of a CLOSE command. The CLOSE command is acknowledged by a message of type close. If it fails, a message of type error is sent,
- *READ filename* to get the content of the file named filename from the current directory. The server first acknowledges the success of opening the file for reading by sending a message of type open succeeded. It then proceeds with sending the content as a set of messages of type data. If it fails, a message of type error is sent.
- *CLOSE* this command is only accepted when a file is being written. It closes the file being accessed,
- *FILTER filter* to set a filter on the list of files to be sent back using GET file. This result in a message of type comment if the filter was set or of type error otherwise.

If a command is not recognised, a message of type error is sent back to the client.

CGI program

As browsers were raising some security exceptions and in order to be able to cope with the old browsers not supporting signed applet, a CGI (Common Gateway Interface) program was added to let the toolkit interact with the workflow script servers. It is only used for servers located on different machines from the one where the applet was loaded.

When users want to communicate with the workflow server the WfGui first checks whether the WfSS to be used is running on the same host as the applet. If it's the case then the communication is direct; otherwise the request is posted to the CGI program with some extra information letting it know where to forward that request. On reception of this query, the CGI program sends it as if it was the source and receives an answer that it gives back to the WfGui. The way it was implemented is that the CGI program first gets the host and port of the server to be contacted. It then tests whether it is allowed to send to this host (Departmental security restriction to some hosts part of the ncl.ac.uk domain) and if it is the case forward the rest of the query to this port and host. On reception of the answer, it is sent back to the WfGui as answer to the initial query.

6.4.3 Loading a script

Using the READ operation previously described, it is possible for the WfGui to retrieve a script from a WfSS. The specification will then be pre-processed letting you know whether you have some errors in your script and their location. If the errors were bad enough for the interpreter to fail to load the specification, they are listed and the user can try to fix then directly. This verification of the syntax is weak as it was thought that designers might want to load a non-correct specification and then use the tools provided with the GUI to correct it.

Loading a script can typically be divided into three different stages: the interpretation stage, the referencing stage and the registration stage. These stages will be now described one after another, starting with the interpretation stage.

While being interpreted, the textual specification is converted into a graphical representation. The missing ObjectClasses are created as ObjectClasses specific to the user and are afterwards available to the user. The TaskClasses are also created directly while reading them. Input object and output objects are registered with their ObjectClass to make it easier to

map the invalid ones later on. The interesting problem is to load the task. First the associated TaskClass is read. If it was an unknown TaskClass, then an error is generated and the interpreter stops trying to load the specification. Otherwise the task is generated with its associated TaskInputSets, TaskInputObjects, TaskOutputSets and TaskOutputObjects. The task is also registered with the TaskClass so that the Toolkit prevents the deletion of referenced TaskClasses later on. The specification of the task is then carried out with the generation of the data and temporal dependencies associated to the TaskInputSets and TaskOutputSets as well as on their respective associated TaskObjects. It has to be noticed that an error is generated if the script tries to specify the dependency on a TaskObjectSet or and TaskObject not part of the associated TaskClass. This error triggers the end of the interpretation of the specification. Each data and temporal dependencies is translated as a reference on dummy TaskInputSets, TaskInputObjects, TaskOutputSets and TaskOutputObjects as appropriate. There is no check at this stage of the validity of these dependencies.

Once the interpretation stage has been successfully completed, the referencing stage starts. During this stage, the dummy TaskObjectSets and TaskObjects are compared to the valid ones within the context of the task for which they are used as source of the dependency. When a match is found the dummy object is replaced with a reference on the real object. The valid objects for each possible type of target objects of a dependency will now be described. We assume that the target object belongs to task A:

- TaskInputSet: The TaskInputSets of the parent task of A, as well as both TaskInputSets and TaskOutputSets of the peer tasks of A (The definitions of peer and parent tasks can be found in section 3.1.1) as well as A's TaskOutputSets of type repeat.
- TaskInputObject: The TaskInputSets of the parent task of A as well as both TaskInputSets and TaskOutputSets of the peer tasks of A (The definitions of peer and parent tasks can be found in section 3.1.1) as well as the TaskOutputObjects associated to A's TaskOutputSets of type repeat.
- TaskOutputSet: The TaskInputSets of A, as well as the TaskInputSets and TaskOutputSets of the tasks composing A. This is only for a compoundTask, TaskOutputSets of a basic Task are not target of dependencies.
- TaskOutputObject: The TaskInputObjects of A, as well as the TaskInputObjects and TaskOutputObjects of the tasks composing A. This is only for a compoundTask,

TaskOutputSets of a basic Task are not target of dependencies.

After completion of the referencing stage, the registration stage starts. This stage consists in going through these dependencies and register the interest of the object involved as target, with their sources. This is useful afterwards when removing tasks.

6.5- Composing a specification using the WfGui

There are three different ways that can be used to specify a workflow using our toolkit. The first one is to write directly a script using the textual language and to put it in the workflow script repository managed by one of our file servers. The second way to specify a workflow is to use the GUI and its graphical notations. The third way is to import it from the Specification Service.

6.5.1 Overview

Once a new specification has been loaded into the WfGui, you can then modify it. The graphical environment allows you to deal with the three main objects of our system:

- Object Class
- Task Class
- Task

6.5.2 Object classes

When the WfGui applet is started, it first contact a naming server to get the Specification Service and then recover from a list of ObjectClasses known by the system.

Adding an ObjectClass

At the time being, it was chosen not to allow users of class designer to modify the list of object classes known by the system. Only users of class maintainer can modify this list. Using a form it is possible to specify the name of a new object. The specification server is contacted and asked to add this new Object Class to its list of valid ObjectClasses. The WfGui also updates its local copy of this list.

In the current version of the Toolkit, it is not possible to remove ObjectClasses once they have been added. This choice was made as some users may be using some of these

ObjectClasses without the Specification Server knowledge.

Mapping of an invalid ObjectClass

As the interpreter of workflow script is weak, it is still possible for designers to introduce some invalid ObjectClasses into the WfGui. As a result it is possible to map invalid ObjectClasses to valid ones using a form. When an invalid ObjectClass is mapped to a valid ObjectClass, all TaskClassObjects (and as a result TaskObjects) using the invalid ObjectClass get updated with the new chosen ObjectClass. Once this is completed, the invalid ObjectClass is no longer referenced and is removed from the system.

6.5.3 Task classes

It is possible to add, delete or edit TaskClasses using the WfGui or merge two TaskClasses together.

Adding a TaskClass

The addition of a TaskClass is done via a form where you specify the name of the new TaskClass, as well as the alternative inputs and outputs needed. They are provided as TaskClassInputSets, TaskClassOutputSets and associated TaskClassInputObjects and TaskClassOutputObjects. TaskClassInputSets and TaskClassOutputSets are fully defined by their name while their associated objects are fully defined by their name and class.

When specifying a task class the following errors can occur:

- Invalid name or name already used for another TaskClass.
- No TaskInputSet or no TaskOutputSet
- Outcomes of type mark and abort both present

Deleting a TaskClass

It is possible to delete TaskClass that are not referenced by any Task. To delete a TaskClass in use, you need to remove or re-map the tasks referring to it.

Editing a TaskClass

Editing a TaskClass consists in taking a copy of an existing TaskClass, and uses it as starting point to create a new TaskClass. This copy of the TaskClass is used to let users undo their modifications if they wish to do so.

Mapping a task class

You can also merge two task classes together by mapping or deleting the TaskClassInputSets and Objects (respectively TaskClassOutputSets and Objects) of the task class being mapped to the task class it is being mapped to.

Each task registered with this mapped TaskClass get notified of the changes and updates its specification by creating some new TaskInputSets, TaskInputObjects, TaskOutputSets and TaskOutputObjects that are substituted to the old ones.

6.5.4 Tasks

Tasks can be added, deleted, edited using the WfGui. A Task's TaskClass can also be mapped to another TaskClass. The graphical notation is the same as the one introduced in figure 3.15.

In our model, a task is fully defined by its name, its TaskClass and its dependencies on the other tasks involving its inputs as target. In the case of a compound task (sub-workflow), you also have to specify the dependencies that its own outputs have on the constituent tasks. The temporal dependencies are shown on the picture above as dotted lines while the data flow dependencies are shown as plain lines.

In order to help the designer specifying its applications, commands have been provided to navigate in the tasks and visualise them.

Adding a task

In order to add a task to the specification, users just have to a creation form where they can specify the task name, choose the TaskClass associated to the task being created as well as its type (compound task or basic task). Users can also add some instantiation criteria.

Once this is done, you can start adding some dependencies and delegations on up-stream tasks involving the TaskInputObjects and TaskInputSets.

If you are creating a compound task, you can also add some dependencies and delegations between inputs and outputs of your task. You can also change the priority of the delegations and dependency sets. Decreasing the priority of the item selected does this. It has to be noticed that an item with the lowest priority seeing its priority decrease will actually gets the highest priority. This allows increasing the priority of an element.

Deleting a task

You just need to click on the icon of the task that they want to delete, This task get removed as well as all dependencies where in which involved.

Editing a task

It is a melting pot of the two previous features. You select the task to be edited by clicking its icon and then you edit it using the same form as for the creation but with as default the values of the task edited. When editing the task, a copy of the task is used to let users undo their modifications if they wish to do so.

Mapping the Task Class of a task

Same technique as the one used for merging two task classes, but this time only the task class of the task concerned is changed; the other tasks sharing the old task class are not affected.

Navigating into a Workflow application Specification.

A navigation system is also available that let the users zoom in and out of your specification. Zooming in a compound task let you see its component tasks, while zooming in a simple task display its task class as well as all the dependencies it is involved in. The zoom-out is the reverse action: zooming out let you see the embedding workflow. When you are lost in the depth of your specification, you can always come back at the highest level by requesting the overview. The interface is of type drag & drop, which means that users can click on task icons and move them around the desk to get a better visualisation of the workflow application.

Clicking twice on the icon on a task gives you the task's view of the world (e.g. mappings to its inputs and outputs), while zooming into a compound task give you a graphical view of the body of the workflow (e.g. its component tasks as well as the links between them and with it). This view can be compact (component task are represented by a rectangle) or full (component tasks are fully described and the dependencies clearly show which sets or objects are concerned).

6.6- Simulation

Using the simulation tool, you can start a simulation, do a step-by-step execution, stop or reset the simulation. There you also have two options: the first option is a random simulation where the computer randomly chooses an output state (outcome) when a basic task is executing. The second option lets the user decides on the outcome of the task execution.

Simulation

A colour scheme let users see which dependencies and tasks are being triggered as well as the state of the tasks. This provides a quick way to check that the workflow is executing as forecasted.

The default colour scheme chosen to represent the states of a Task (cf. section 3.4 and figure 3.17) is:

- Waiting (green): the task has some dependencies on it, but may be executed later on
- Set-up (yellow): the task is being modified
- Active (orange): the task is executing
- Completed (red): the task has been executed
- Discarded (grey): the task has been discarded, as some dependencies could not be fulfilled. This state has been added to what is presented in section 3.4, as it allows forecasting the future more easily.

The relevant sub-set (waiting, set-up, completed and discarded) of this colour scheme is used for the TaskObjects.

The tasks can be in several states with the temporal and data flow dependencies depicted on figure 6.5.

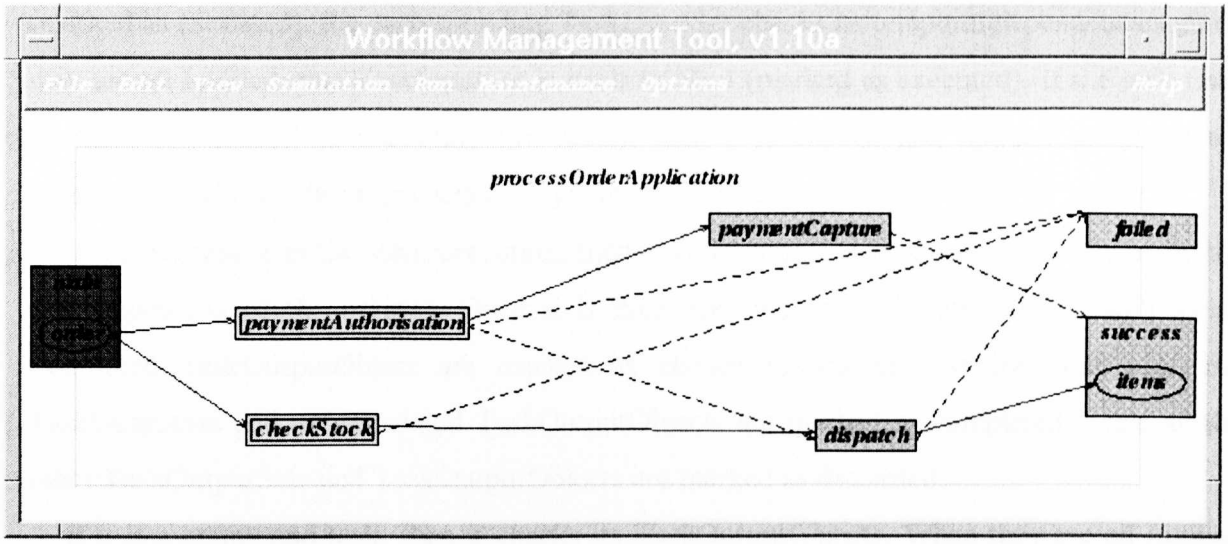


Figure 6.5: Simulation of the execution of a workflow application

In the example depicted in figure 6.5, the user has zoomed into a compound task while it was executing (the rectangle showing the compound Task boundary is orange). This task had a single TaskInputSet with one associated TaskInputObject. Both of them appear in red meaning that they have been used. The two tasks PaymentAuthorisation and CheckStock appear in orange, meaning that they are executing. This is consistent with the specification as they both only have a data flow dependency and it is on the TaskInputObject of their parent task. As they haven't completed yet, they were coloured in orange. The other tasks appear in green as they are still waiting for their inputs. Dependencies that have been used appear in red in the detailed view of a task.

Implementation

Each task and each dependency have a run-time status indicating their current situation. When the simulation is started, the workflow application gets one of its input sources fulfilled. The way to decide which one depends on the simulation options. If it is automatic, a random value is generated by an object of class Toolkit.Gui.Randomize (using the formula Math.abs(generator.nextInt()) % number), where generator is an object of class java.util.Random, and number is the number of input set minus one). The result gives the TaskInputSet that is supposed to be activated.

Afterwards, the system goes one step at the time. Each task still waiting checks whether one of its input sets requirements has been fulfilled. Each TaskInputSet checks in turn which dependencies have been fulfilled. If all its TaskInputObjects have been marked as fulfilled

(marked as executed), the corresponding TaskInputSet checks its temporal dependencies and if they are all fulfilled, this TaskInputSet is itself fulfilled (marked as executed). If the task finds an input set marked as executing (available), then it is chosen and marked as completed (chosen), while the other input sets (if any) are discarded.

If a basic task is in the state executing, then a TaskOutputSet is chosen randomly or by the user depending of the options. Once it is done the chosen TaskOutputSet as well as its associated TaskOutputObject are marked as chosen (executing). At the next step, the TaskOutputSet and its associated TaskOutputObjects are marked as completed, while all the other TaskOutputSets and TaskOutputObjects are marked as discarded.

If it is a compound task, then it checks its TaskOutputObjects. When they are all fulfilled and there are no temporal dependencies on their associated TaskOutputSet, this TaskOutputSet is set as fulfilled. When the task next steps, it will find out about the fulfilled dependencies and set the TaskOutputSet and associated TaskOutputObjects as chosen (completed) while setting their alternatives to discarded.

6.7- Execution

Before being able to start executing a specification, it has to be sent to the Repository Service. The pre-requisite for storage in the repository service being that the specification needs to be correct, some checking tools of the different components of the workflow application have been provided. They will be first presented before describing what needs to be done to transfer a WfGui specification into the Repository Service and run it.

6.7.1 Checking the specification

When the specification is over, you can check that it was correctly written. This process has been divided into three sub processes: Checking the ObjectClasses, the TaskClasses and the Tasks.

Checking the ObjectClasses

This makes sure that all ObjectClasses are known by the system. A list of the ObjectClasses used in the specification appears and let the designers see which ones are not recognised.

If some new ObjectClasses were found, designers have the opportunity to map them to existing ObjectClasses, while maintainers can also add them as System ObjectClasses. The way

to do it has already been presented in section 6.5.2.

Checking the TaskClasses

A task will be tagged as incorrect if:

- Once or more of its TaskClassObject has an invalid ObjectClass.
- It doesn't have at least one TaskClassInputSet and one TaskClassOutputSet
- There is both a mark and an abort TaskClassOutputSet.
- A repeat outcome is used by a task different from its own task

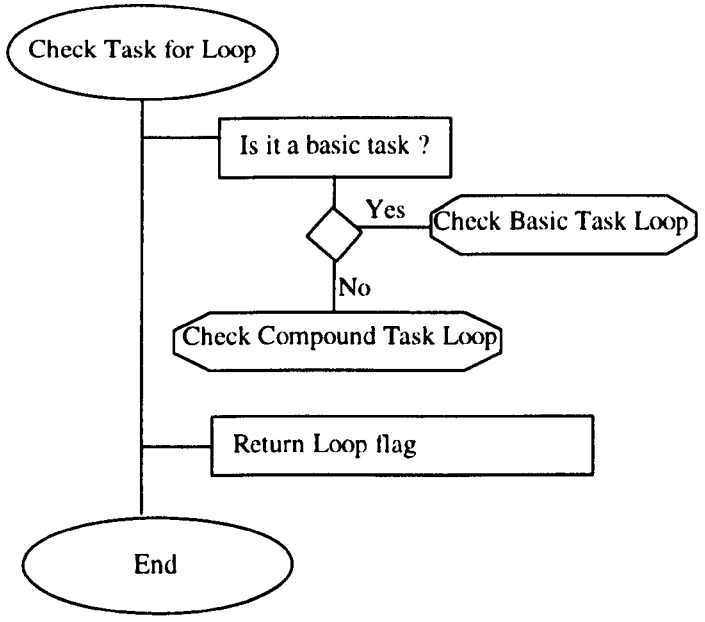


Figure 6.6: Design for "Check task for Loop" process

Checking the Tasks

A task is tagged incorrect if:

- Its TaskClass is incorrect,
- Some of the dependencies with this task as target are on non-existent objects.
- Some data-flow dependencies with this task as target are involving two TaskObjects of different classes.
- Some of its components share their names with it.
- It is a compound task including an unwelcome loop (unwelcome loops are discovered by creating a dependency graph of the component tasks)

While the four first checks are trivial, the check for unwelcome loop is interesting and will be described in the next paragraph.

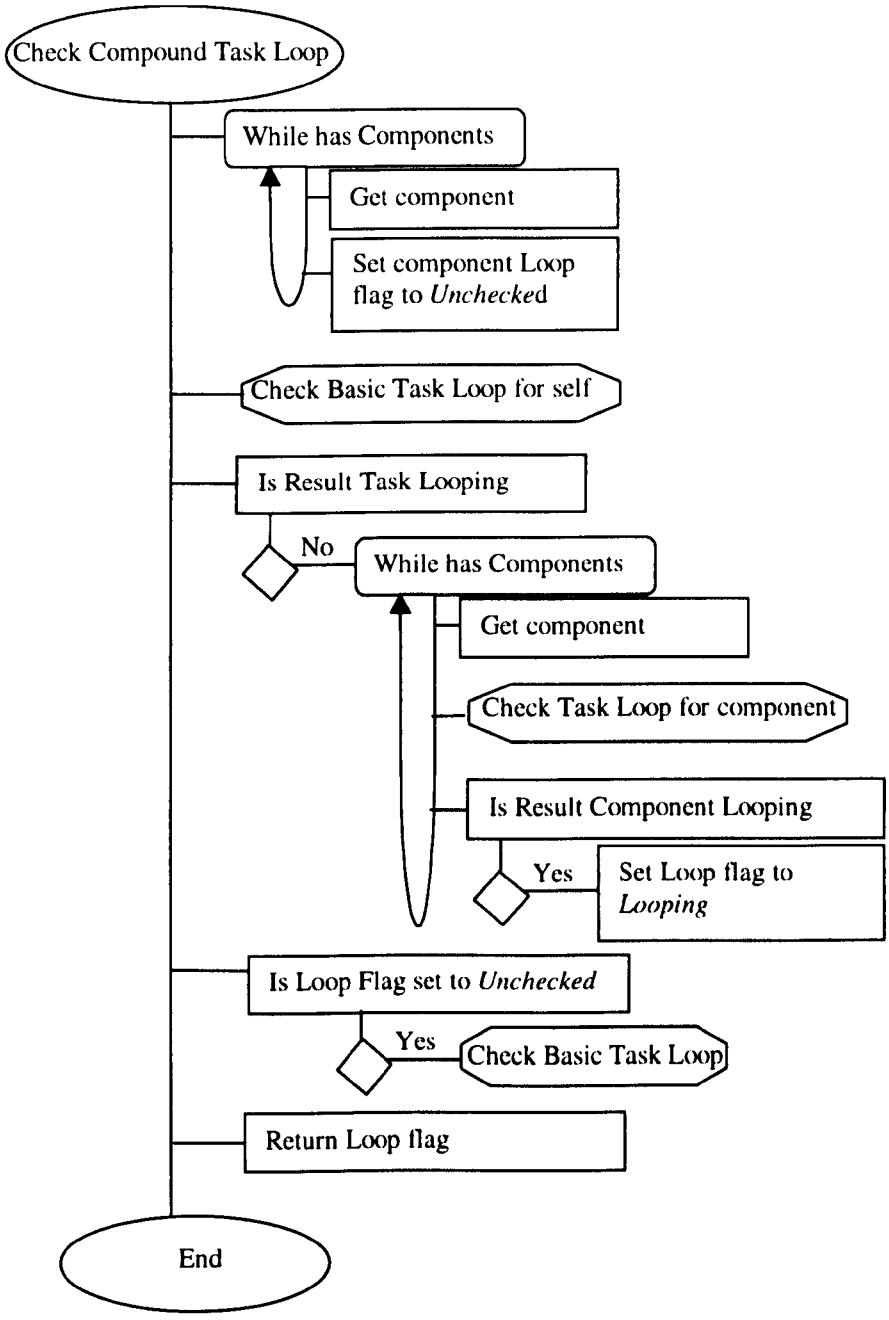


Figure 6.7: Design for "Check compound task for Loop" process

Checking for unwelcome loops

The first thing to realise is that finding unwelcome loops in a workflow application (e.g. loops that are not introduced by repeat outcomes) is equivalent to finding unwelcome loops between the component tasks of a compound task. Indeed, there are no dependencies between

tasks embedded in a compound task and the peer tasks of their parent. As a result, loops can only involve peer tasks.

In order to find out the dependencies, we have proposed the following design: We start the check for loop by a Check task for Loop for the "root" workflow task. The design notation has been used to describe this operation in figure 6.6, 6.7 and 6.8.

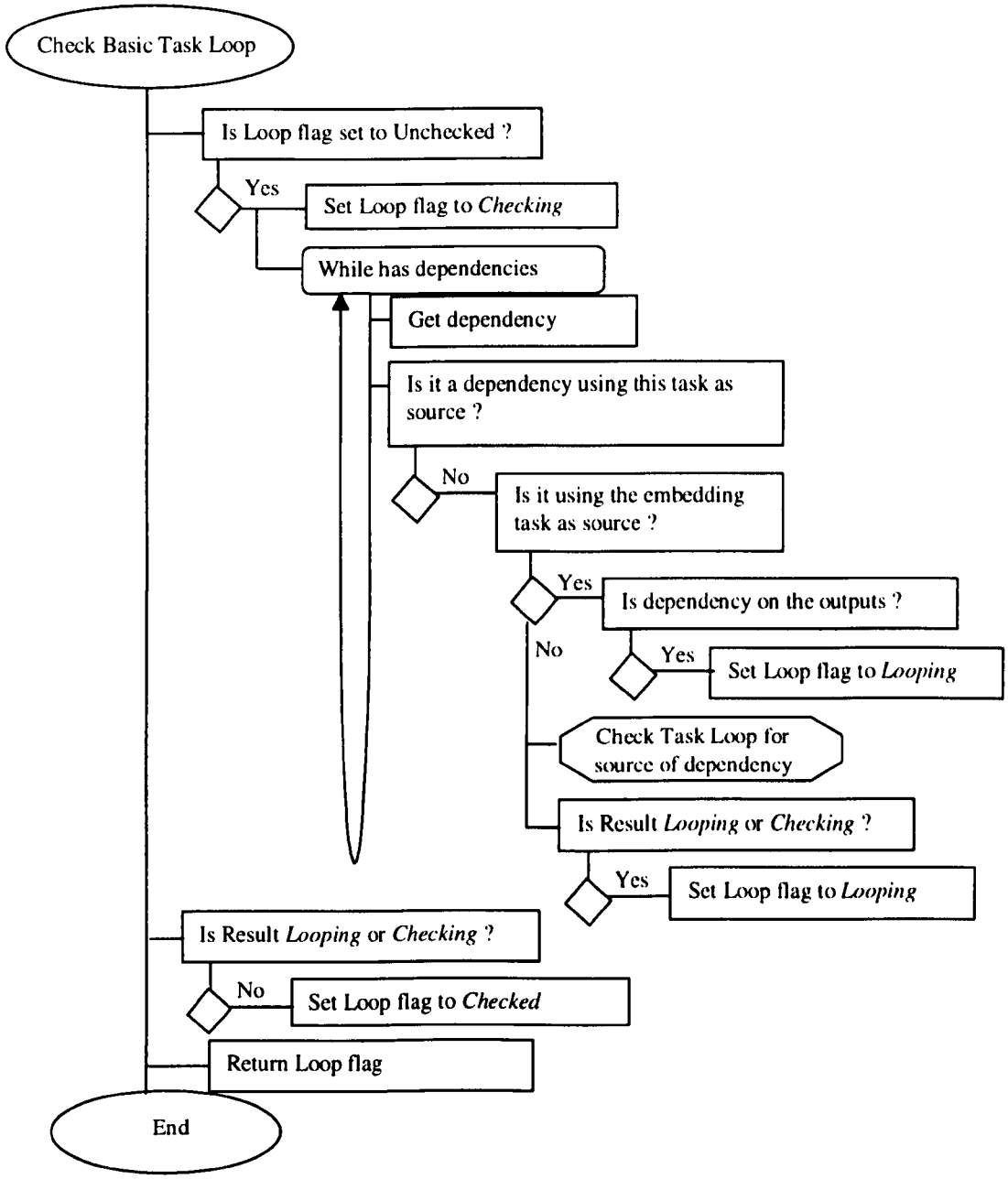


Figure 6.8: Design for "Check basic task for Loop" process

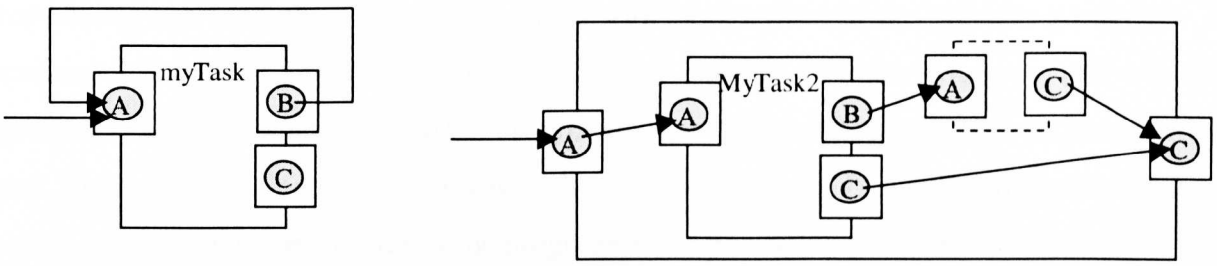
Figure 6.6 just describe the test on the type of task (basic or compound task) the task to be checked is and then do the proper treatment.

The idea behind this approach is that if there is no loops, you can always find a task with as only dependencies involving as target some dependencies with the compound task. If it doesn't exist, it's obvious to prove that there is a loop, as you always can choose an outcome that has a target dependency on a peer task. After having set up a sequence with n+1 elements where n is the number of children, then there is at least a duplicated task as there are only n tasks). Having chosen this task without outgoing dependencies on peer tasks, you can safely remove it from the set of children without changing the result of the check. Indeed this task will never be checked again as it has no outgoing dependency on its peers and having one will be the only way to find out that it is involved in a loop. You then iterate on the remaining n-1 tasks. Checking one by one the tasks allows to be sure that the result returned is correct (none of them is involved in a loop or at least one is involved in a loop).

The algorithm terminates as each task is checked only once. There is a maximum of n-1 + n-2 + ... +1 ((n*n-1)/2) requests of checks in the worst case (no loops and each task has some dependencies involving all its possible up stream tasks. The i^{th} task depends on the outcomes of the i-1 previous tasks.)

6.7.2 Storing in the Repository Service

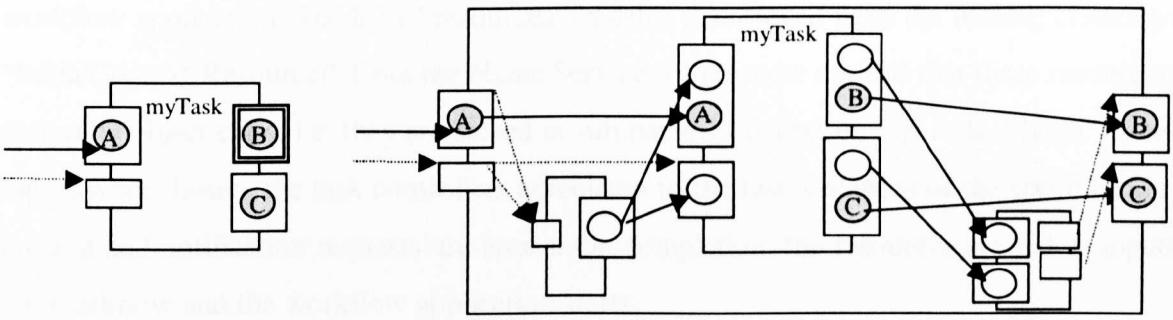
Once the specification has passed the consistency tests, it can be exported to the repository service. The main difference between the specification model of the WfGui and the repository service are the outcomes of type mark that are not yet supported as well as those of type abort and repeat that are not directly understood. As a result, the repeat loops have to be converted into a different representation at run time using genesis tasks. On figure 6.9a, the high level representation of a repeat task is given. The representation of the same task at run-time is given on figure 6.9b. It is modelled as a compound task embedding a task similar to myTask and a genesis task. The main difference between myTask and myTask2 is that the input alternatives that were coming from the repeat outcome disappear. The dependencies that were using a repeat outcome as source are now dependencies on a genesis task with the same inputs as myTask had. This genesis task uses as associated task the compound task that gathers myTask2 and the genesis task.



a) High level representation b) Low level representation

Figure 6.9: Run time representation of loop tasks

Similarly the tasks with an abort outcome (e.g. atomic with its transactional meaning) have to be translated before being sent to the low level



a) High level representation b) Low level representation

Figure 6.10: Run-time representation of abort outcomes

In figure 6.10 a), we can see the high level representation of a task with an abort outcome. Assuming that it is not embedded in another task with an abort outcome, it is translated as the task represented on figure 6.10 b). The first (left) additional task could be called "*generation task*" as it is a task whose only purpose is to create an object of class "Transactional Context" which is then added as parameter for the input sets of the task as well as the output sets of the task. The additional task on the right that could be named "*commitment task*" has two inputs sets abort and commit each with one associated object of class "Transactional Context", and have two outcomes aborted and committed. The references to the Transactional Context object from the abort outcomes are OR-ed on the abort input set of the additional task, while the Transactional Context objects from the other output sets are mapped to the commit input sets. The outputs sets of the additional task are then used as temporal dependencies on the embedding task output sets, the abort output set being used by the output sets of the embedding task that are linked to former abort output sets from myTask.

If myTask is itself embedded in another task with abort output sets, then the outermost

embedding task with abort outcomes is dealt with as described in the previous paragraph. The transactional context object is then fed to all its children without having to add any extra *generation task* or *commitment task*. This provides a way to model nested transactions. For basic task, the implementation must comply with this interface (e.g. the Transactional Context object on top of the expected object) and it is the programmer responsibility to ensure that it does.

6.7.3 Starting an application

Once the low-level specification has been created (e.g. the specification is stored in the Repository Service), it is possible to start an application. The way it was done is to let the user select the initial input set as well as its associated resources to be used as initial inputs for its workflow application. The list of resources available is retrieved from the naming contexts and "InitialContext/Resource/" from the Name Service. It has to be noticed that these resources are stored by object class, i.e. they are stored in sub naming context on a per class basis. Once the inputs were chosen, the task controllers associated to the task instances of the specification are created and notification requests are issues. On completion, the resources are fed as inputs to the workflow and the workflow application starts.

6.7.4 Dynamic modifications

The low-level dynamic reconfiguration is achieved by changing the status of the task controller in charge of the task to set up. Once the task controller is in this status modifications can be carried out on the task it is dealing with. The following modifications (listed in chapter3, section 4) can be carried out on a single task, one at the time in a sequential order :

1. The implementation bound to a basic task can be changed as long as it is in a wait state
2. Tasks can be added or removed from workflow instances
3. The constituent tasks of a compound task can be changed
4. Input alternatives can be added or removed from a task, and their priority can be changed as long as the tasks is not in its state executing or completed
5. Output alternative can be added or removed from a task, and their priority can also be changed as long as the task has not completed

It has to be noticed that there is no provision for making atomic a set of modifications involving different tasks or just a single task. Using the toolkit, you can carry out the following modifications, providing that the task controller of the task involved is still in its wait status:

1-modification of a single task, including changes of every details of the specification, including:

a- modification of the incoming dependencies and object delegations involving this task

b- changes of the meta information associated to this task and in particular the task factory to be used for simple tasks.

c- mapping of the task class of the task to another task class (outgoing dependencies are then automatically transferred to the mapped tasks, or deleted shall the outcomes or their associated objects be removed.

d- change of the type of the task (For instance, a simple task can be transformed into a compound task)

2- setting manually the status of the associated task controller to set up. In this case, you will have to set it back to waiting manually.

Limitations:

If you want to carry out a set of modifications involving a set of tasks atomically, you will have to make sure manually that they have all been frozen (e.g. setting the task controllers status to set up). To do that, you will have to change each task controller of the tasks involved to set up. Notice that setting the status of the task controller of a common ancestor (if possible) will achieve the same result. In a typical example of dynamic reconfiguration involving a big number of tasks, it is likely that you would identify some sets of tasks that can be "frozen" by freezing their parent and would freeze their parent instead of freezing them one by one.

6.8- Monitoring

Once started, a workflow application can be monitored. This is made possible thanks to the task controllers that provide information on the state of the tasks that they are managing.

The monitoring process itself can use pull or push technology. Choosing the pull technology obliges the user to take some manual snapshots to see the current state of the workflow

application. In this case the Task Controllers associated to the specified tasks are requested to provide to the WfGui the status of the task they are controlling as well as which inputs and outputs were chosen. If the push technology is chosen, the WfGui registers its interest in everything happening in the application and update the view as soon as it is notified of changes.

The WfGui displays the application being monitored similarly to simulated applications.

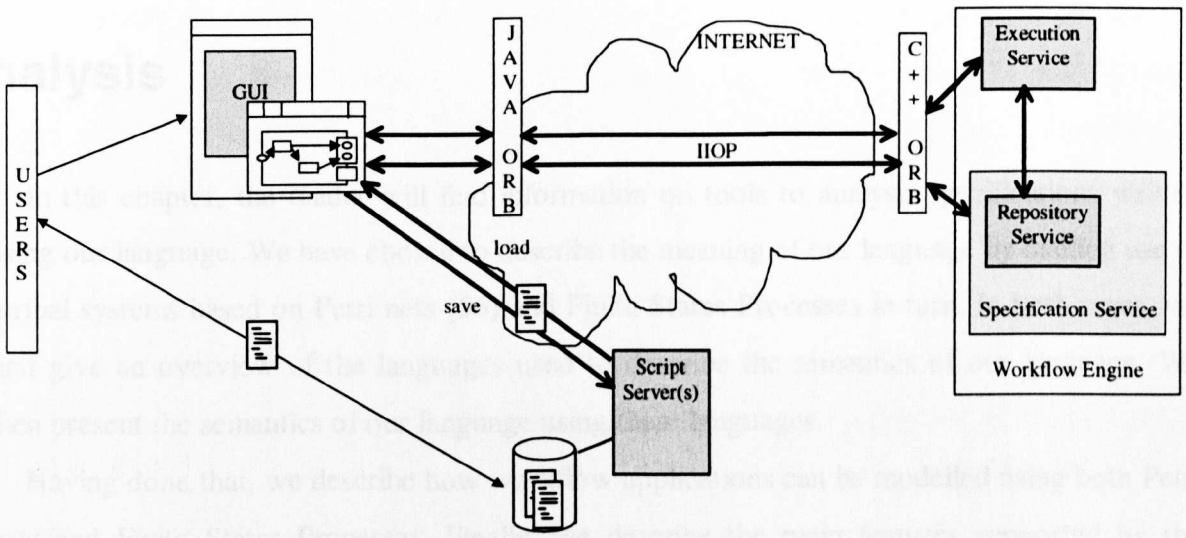


Figure 6.11: Graphical representation of the workflow system with Specification Service

As stated earlier on, specifications stored in the Repository Service lose their high level names. This makes it difficult for other persons to monitor the workflow. As a result, a Specification Service that will keep a high level view of the tasks an in particular keep the names associated is being built on top of the Repository Service as depicted on figure 6.11. It is basically a copy of part of the GUI implementation with a CORBA interface in front of it. This will make it possible to have several observers monitoring the progress of the same workflow.

In this chapter the toolkit was presented. It was shown that the specification needed to be checked before starting to execute it. Some external tools can be used to check specifications. In the next chapter some analysis that can be done on our language are presented as well as some external toolkits that could be used to provide further consistency checks of our workflow specifications.

Chapter 7

Analysis

In this chapter, the reader will find information on tools to analysis specifications written using our language. We have chosen to describe the meaning of our language by making use of formal systems based on Petri nets [56] and Finite States Processes in turn. In both cases, we first give an overview of the languages used to describe the semantics of our language. We then present the semantics of our language using these languages.

Having done that, we describe how workflow applications can be modelled using both Petri nets and Finite States Processes. Finally, we describe the main features supported by the toolkits associated to these two notations, as well as their relevance to our system (e.g. checking for absence of deadlocks).

7.1- Analysis using Petri-nets

The benefit of providing a Petri net semantics for our language is that we can translate one of our specifications into a Petri net. This provides a way to use existing Petri net tools on the workflow specifications to try to find potential problems in the workflow specifications. In this section, the reader will find explanation on how our specification can be translated into a Basic Petri Net Programming Notation (also known as $B(PN)^2$) specification and afterwards using a toolkit such as the Programming Environment based on Petri Net (PEP)[8], what it can be used for.

7.1.1 Overview of $B(PN)^2$

$B(PN)^2$ [9] is a Petri net based programming notation that has been designed to have a clearly expressed and compositional Petri net semantics, allowing the application of Petri net proof methods to complement other techniques. It aims at providing some flexibility to

smoothly integrate a variety of process interaction techniques. There are five types of commands: iterations using "**do** ... **od**", sequential and concurrent compositions, atomic execution of an expression or block. Expressions can be of several types, variable identifiers, **true**, **false**, **Z** (integer type), operation between two expressions or on just one, expression between parentheses, channels (FIFO buffers) or stacks identifiers. The operations supported are the usual arithmetic operations. The syntax is describes below:

- Variables are introduced by the construct **var**. You need to state the valid values of a variable. For instance `var v : {0 , 1} init 1;` declares a variable v that can only takes as value 0 and 1 and initialises it to 1, while `var j : Z;` declare a variable j of type integer. The value of the variable v before and after execution of a command is respectively known as **'v** and **v'**.
- Non deterministic choice using the construct "[]",
- <expr> denotes the atomic execution of the expression expr
- blocks using **begin** scope **end**, where scope can be a command precede or not by a list of declarations of constants or variables,
- Sequential composition using the construct ";",
- Concurrent composition using "||",
- Iteration using "**do** `command` **enter** `alternatives` **od**" or just "**do** `alternatives` **od**", where the alternatives can be "`command ;` **repeat;**" or "`command ;` **exit;**" or a non deterministic choice of two alternatives". The exit alternative corresponds to an exit of the loop, while the repeat alternative just restarts the iteration. Both of these two types of alternatives are preceded by a command acting as a kind of guard. This command needs to be fulfilled before an alternative can be used

For instance, the extract of code below should be read as follows. First the variable v is set to two, then we enter the loop. If the variable v is positive, then we decrement it by one and leave the loop do; otherwise if it is less than four, it is incremented by one and we iterate. The choice between these two alternatives is non-deterministic.

```
do <v' =2> enter
<v > 1>; <v' = 'v -1>; exit;
[]
<v < 4>; <v' = 'v +1>; repeat;
od
```

Procedures are supported in the extended notation that we are using. They are introduced by the construct **proc**, followed by the name of the procedure and the list of parameters

between brackets. Keywords **ref** and **const** indicates reference and constant parameters. The valid values for the arguments have also to be specified. The body of the procedure is a block (embedded between the **begin** and the **end** keywords).

7.1.2 Modelling a workflow application

The modelling of a workflow application can be divided into several sub-problems: how to model the mapping of the inputs of a task, and the mapping of the outputs of a basic and a compound task. Once this has been solved, modelling a workflow task using $B(PN)^2$ becomes simpler.

Modelling the mapping of the inputs of the tasks

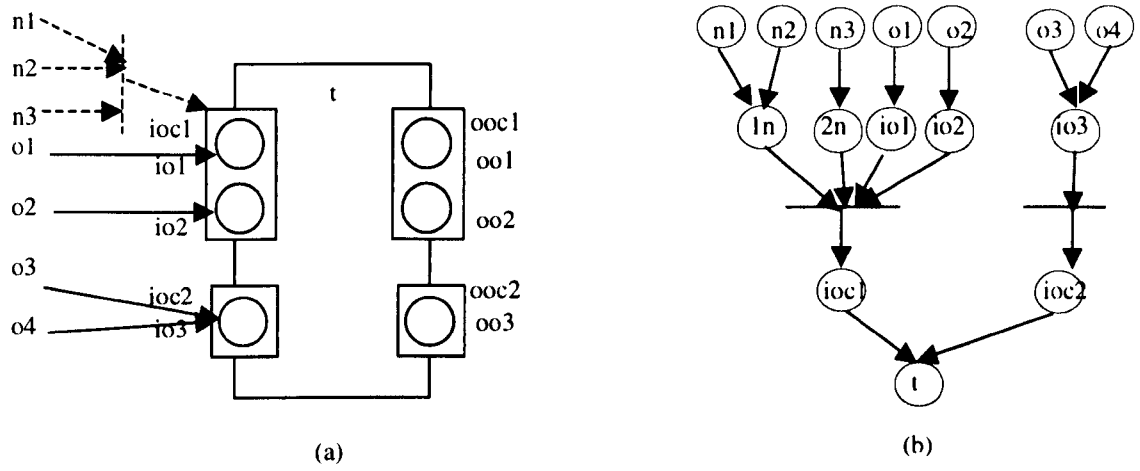


Figure 7.1: Modelling the task inputs

There are several things that need to be modelled. First we need to be able to model a task, its inputs and outputs. The easy way to do that is to create a variable per component based on the task class. A component in this case will be the state of a task, its task input and output sets as well as their associated object inputs and outputs. There is also a need to model the notification dependencies associated to a set. Creating one dummy object per alternative set of notifications does this. This is represented on figure 7.1. An arc leading from one oval to another directly has to be interpreted as an OR-component for the triggering of that task, while an arrow leading to a vertical bar has to be interpreted as an AND-component for the triggering of the ovals receiving an arrow leaving that bar. An OR is simply modelled by the choice construct while an AND is modelled by the sequential composition.

To have no naming problems, the underscore can be used to separate levels of abstractions. For example, the following naming scheme could be used. A workflow named wf will be

renamed wf___, its input set ios1 will be renamed wf_ios1__, its associated input object io1 will be named wf_ios1_io1_ and the virtual input object created will be named wf_ios1_notif1_, wf_ios1_notif2_ and so on. The component tasks of wf will have the prefix wf___ as prefix for their name. This scheme ensures uniqueness of the names while retaining enough information for an easy translation back to the initial names of the workflow specification. Usage of the scoping feature could also be used to simplify our naming scheme as compound tasks also provide some kind of scoping.

Modelling the mapping of the outputs of a basic task

Our main problem was initially to find a way to choose a random output once the task had been started.

The solution adopted is to use the expression `<output' = 1 v output' = 2>`. The choice of the value of `output` is non-deterministic and the chosen value can be used to decide on an output. The resulting model is depicted on figure 7.2.

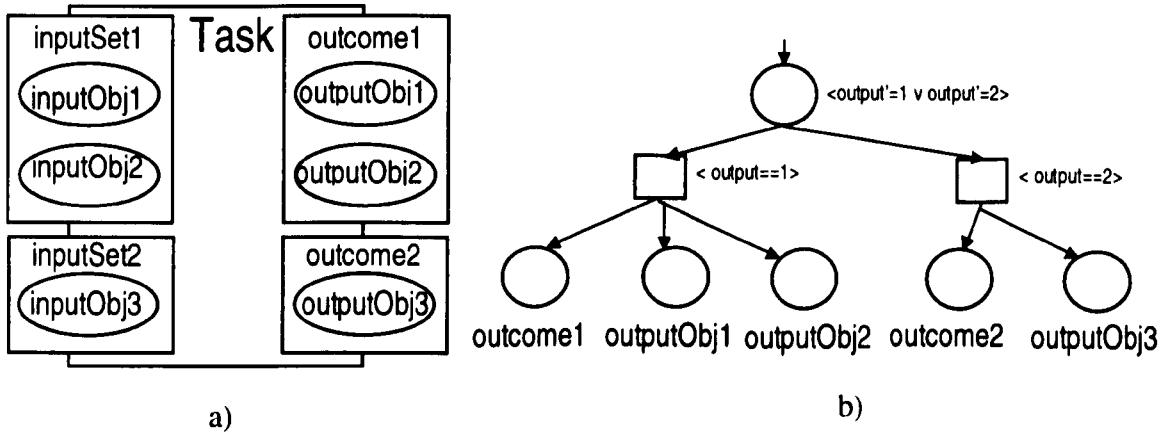


Figure 7.2: Modelling of the reaching of an output for a basic task.

Modelling the mapping of the outputs of a compound task

This is exactly the same model as the one used for the inputs.

Modelling the workflow application

In order to keep track of the state of a task, it is modelled as having a value that can only have three different values: 0 if the task is waiting to be started, 1 if it is executing and 2 if it

has completed. The inputs (sets and objects) as well as the outputs will be represented as having a value of 0 if they are waiting to be used or one if they have been used.

Assuming that we have a workflow application with one input set labelled main with an input object associated startObject and two output sets done and failed with no associated output object, the workflow script would be:

```
objectclass myClass;
taskclass myTaskClass
{
   inputs {
      input main {
         startObject of class myClass
      }
   };
   outputs {
      outcome done {};
      outcome failed {}
   }
}
compoundtask wf of taskclass myTaskClass
{
   ... // body suppressed for clarity
}
```

The equivalent B(PN)^2 script would be:

```
begin

. . .
proc TASK_WF (ref t: {0, 1, 2}, ref inputSet1 : {0, 1}, ref inputObject1
: {0,1}, ref outputSet1 : {0, 1}, ref outputSet2: {0, 1})
begin
<t' = 1>
. . .
end;

var wf___ : {0, 1, 2} init 0;
var wf_main__: {0, 1} init 0;
var wf_main_startObject_ : {0,1} init 1;
var wf_failed__ : {0, 1} init 0;
var wf_done__ : {0, 1} init 0;

do
<'wf_main_startObject_ = 1> ; <wf_main__' = 1> ; exit
od
parallel
do
<'wf_main__ = 1> ; <'wf___ = 0> ; TASK_WF(wf___, wf_main__,
wf_main_startObject_, wf_done__, wf_failed__) ; exit
od
end
```

The specification of a workflow application can be divided in three parts. An initial part

listing procedures that describes the workflow application, its component tasks as well as their inter-dependencies, then a part declaring a set of variables able to take either as value 0, 1, 2 (for the tasks) or 0, 1 (for the sets and objects) and initialising their value. The variables are the state of the workflow application, its component tasks, as well as the states of their inputs and outputs. Then a final part where two processes are run in parallel, the first one waiting for startObject to become available (value 1) to set input set main available (value 1). The second process waits for an input set to become available and if the task hasn't yet started, runs it via a call to the procedure declared in the first part. It has to be noticed that all variables associated to the workflow application are used as arguments as they may be used for the execution of the workflow application. The second process can be read as if the input set (<'wf_main__ = 1>) is available and the task is waiting (<'wf___ = 0>) then start the execution of the task (TASK_WF(...)). Had this workflow application had more than a single input set then we would have had all these inputs separated by "|" in the first term between brackets.

The first process establishes the conditions that have to be fulfilled before that the input set main can become available (startObject available in this case is the only condition)

Let us now expand further the initial part. Let us assume that this workflow application consists of two basic tasks with as dependencies the dependencies seen on figure 7.3. We assume that all input and output objects were named item.

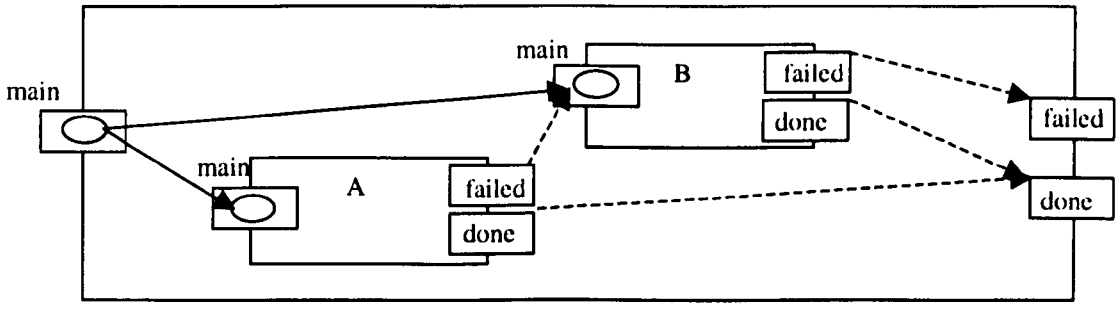


Figure 7.3: Example of workflow application

In the code below, the first procedure corresponds to the random choice of an output set for a basic task. Here there are only two alternatives.

```
  proc TASK_A (ref t: {0, 1, 2}, ref inputSet1 : {0, 1}, ref inputObject1 :
{0,1}, ref outputSet1 : {0, 1}, ref outputSet2: {0, 1})
  begin
  var randomchoice : {1, 2};
  <t' = 1>;
  do <randomchoice' = 1 | randomchoice' = 2> enter
  <'randomchoice = 1> ; <'outputSet1 = 1> ; exit
  []
```

```
   <'randomchoice = 2> ; <'outputSet2 = 1> ; exit
   od
   end;
   proc TASK_WF (ref t: {0, 1, 2}, ref inputSet1 : {0, 1}, ref inputObject1
: {0,1}, ref outputSet1 : {0, 1}, ref outputSet2: {0, 1})
   begin
   var wf____notif1_ : {0, 1} init 0;
   var wf____notif2_ : {0, 1} init 0;
   var wf____notif3_ : {0, 1} init 0;
   var wf___a___ : {0, 1,2} init 0;
   var wf___b___ : {0, 1,2} init 0;
   var wf___a_main__ : {0, 1} init 0;
   var wf___b_main__ : {0, 1} init 0;
   var wf___a_main_item_ : {0, 1} init 0;
   var wf___b_main_item_ : {0, 1} init 0;
   var wf___a_done__ : {0, 1} init 0;
   var wf___b_done__ : {0, 1} init 0;
   var wf___a_failed__ : {0, 1} init 0;
   var wf___b_failed__ : {0, 1} init 0;
   <t' =1>;
   do <'inputObject1 = 1> ; <wf___a_main_item_' =1> ; exit od
   || do <'wf___a_main_item_> ; <wf___a_main__' = 1> ; exit od
   || do <'wf___a_main__ = 1> ; <'wf___a___ = 0> ; TASK_A(wf___a___,
wf___a_main__, wf___a_main_item_, wf___a_done__, wf___a_failed__); exit od

   || do <'wf___a_failed__ = 1> ; <wf____notif1_' = 1> ; exit od
   || do <'inputObject1 = 1> ; <'wf____notif1_ = 1> ; <wf___b_main_item_'
=1>; exit od
   || do <'wf___b_main_item_> ; <wf___b_main__' = 1> ; exit od
   || do <'wf___b_main__ = 1> ; <'wf___b___ = 0> ; TASK_A(wf___b___,
wf___b_main__, wf___b_main_item_, wf___b_done__, wf___b_failed__); exit od
   || do <'wf___b_failed__ = 1>; <wf____notif3_' =1> ; exit od
   || do <'wf___a_done__ = 1> ; <wf____notif2_' = 1> ; exit od
   || do <'wf___b_done__ = 1> ; <wf____notif2_' = 1> ; exit od
   || do <'wf____notif2_ = 1> ; <outputSet1' = 1> ; exit od
   || do <'wf____notif3_ = 1> ; <outputSet2' = 1> ; exit od
   || do <'outputSet1 = 1> ; <t' = 2> ; exit od
   || do <'outputSet2 = 1> ; <t' = 2> ; exit od
   end;
```

In this last procedure, the variables used are first specified, then the state of the task is set to 1 (executing). Afterwards the different dependencies are described, before specifying the mapping of the outputs (last seven lines). Notice the use of some extra variables to model the notification dependencies.

7.1.3 Usefulness for our system

Once the B(PN)^2 specification has been created, the PEP toolkit first expands the procedures and then can be used to generate Petri nets, check them as well as simulate their behaviour. Specifically talking about the PEP toolkit, it has a provision of analysis tools that allow the user to check whether the equivalent low level Petri net is:

- Free choice e.g. if a place is an input to several transitions (potential conflicts), then it's the only input for all of these transitions. Hence either all of these conflicting transitions are simultaneously enabled or none of them are. This allows the choice (conflict resolution) as to which transition is to fire to be made freely; the presence of other tokens in other places is not involved in the choice as to which transition fires.
- A bounded system e.g. if there exists an integer k such that the number of tokens in any place cannot exceed k.
- Safe e.g. the number of tokens in any place cannot exceed one.
- Reachability of markings e.g. whether a marking is reachable from an initial marking. It has to be noticed that this is not available for non-bounded nets.
- Deadlocks: Low level nets that have been found to be safe can then be tested to find out whether they include some deadlocks. A deadlock is defined as a set of places such that every transition, whose outputs to one of the places in the deadlock also inputs from one of these places. An interesting result is that once all the places of a deadlock become unmarked, the entire set of places will always be unmarked. This is useful for identifying some unwanted cycles.

As far as our workflow specifications are concerned, the Petri net of a correct specification is k-bounded, as we have a finite number of tasks having a bounded number of outputs.

As a result, the PEP toolkit can be used to identify the following problems: objects that are never available, tasks that will never be reached, whether or not you have a deadlock in your application... It can also be used to check that our application can be executed in a certain way given some initial conditions. For instance, for a running application, the user might want to know whether it is still possible that a certain set of tasks will be run before completion of the application. This is done using the reachability feature of the toolkit given an initial marking.

7.2- Analysis using Finite State Processes

The Finite State Process (FSP) process algebra notation has been designed for an easy description of component behaviour. In this section, the reader will find explanation on how our specification can be translated into a Labelled Transition System using the Finite State Process process algebra notation [34] (also known as FSP). Then we will explain how the resulting translation of our workflow application can be checked using a tool such as the

Labelled Transaction System Analyser (LTSA) [31].

7.2.1 Overview of FSP

Using FSP, each component of the system is modelled as a finite state machine. As a result the whole system becomes a set of interacting state machines. FSP distinguishes primitive and composite processes. Primitive processes are defined using action prefix, choice and recursion, while composite processes use parallelism, label re-assignation and hiding. The separation of the constructs between primitive and composite processes ensures that only finite system can be generated. A special process named STOP terminating is predefined

The main constructs of FSP are:

- The action prefix "->", used by FSP to specify a process. It specifies an initial action to carry out as well as a process describing the behaviour of rest of the process.
- The choice construct "|" can be used to describe alternative behaviours. For instance, the following process `Q = (a -> P | b ->R).` is initially either starting action a or action b. In the event of a being chosen, the Q will then behave as P, otherwise it will behave like R. The choice is non-deterministic.
- A feature called action sets is also provided to group together the processes that shared the same behaviour after carrying out different initial actions.
- Conditional "`if expr then process [else process].`", where expr is an integer expression such as x > 3
- Guards transitions "`when B a -> P`" are also available For instance, if you want to only have action a available when x > 0, you will use `(Q = (when x>0 a -> P)..`
- The construct "||" or parallel composition between two processes. The resulting LTS allows all interleaving of the actions of the two processes. Actions with the same name are shared and have to be synchronised. The actions can be used to synchronise parallel processes. The specification of a composite process ||COMP composing processes A and B, which sharing the action "synchronise" is now described. A notation convention is to use || to prefix the name of the composite process:

```
A = (a -> synchronise -> A).
B = (b -> synchronise -> B).
||COMP = (A || B).
```

- The **forall** construct is available for process replication.
- Re-labelling is also available. They are applied to processes and change the names of the

actions. The general form of re-labelling is /{new_label/old_label}

- Hiding remove action names from the alphabet of the process, which hides these actions. The syntax for hiding action names is: /{ set of labels to be hidden}. It is also possible to specify the action labels that are not hidden. This is done using @{ set of labels not hidden }.

Parameterisation of processes is also supported. It has to be noted that a default value is required for the parameter. For instance `INPUT(N=0) = inputobject[N].available -> available -> STOP)`. parameterises the process INPUT. Given N =0, inputobject[N].available will generate the action labelled inputobject.0.available.

7.2.2 Modelling a workflow application

Modelling the mapping of the inputs of the tasks

What need to be done here is to specify the transitions between states. But first what is the relationship between input sets and objects: in order for an input set to become available all its associated input objects need to be available. For a task to become available, one of the alternative sets needs to be available. The added dependencies (notifications and delegations) are just extra similar transitions. An input object becomes available when one of the object sources mapped to it becomes available. Notifications can just be seen as a particular type of delegation if you add a dummy input object per set of alternative dependencies. Adding such a dummy object is straightforward and is just an extra transition linking the availability of the dummy input object to the input set concerned.

Modelling the mapping of the outputs of the tasks

As far as the outputs of a basic task are concerned, the availability of an output set implies the availability of all its associated objects. As a result the choice of the outcome to activate is just a set of transitions from task available to one of these outcomes linked between themselves using "or". This is provided by the "|" operator.

The outputs of compound tasks are dealt similarly to its inputs: first the output objects need to become available and only then can the output sets they are associated to become available.

Modelling of the workflow application

FSP is interesting as it provides some support for process labelling by using the construct ":". This allows to make good use of another feature of the language called "**forall**" which allows replication of processes, or instantiation of parameterised parameters using the same template. In order to model the workflow application, we use the run-time version (e.g. without mark). The repeat outcomes being incompatible with finding cycles in the specification have also been removed.

Considering the example given in figure 7.3, the FTS specification would be:

```
// *********************************
// ***   workflow independent    ***
// *********************************

// get input
INPUT(N=0) = (inputobject[N].available -> available -> STOP).
||INPUTS(N=1) = (forall[i:0..N-1] INPUT(i)).
||TASKCLASS_GETINPUTS(N=1) = (if (N > 0) then (INPUTS(N))).

// Basic task, output release.
BTOUTPUT(N=1) = (available -> outputobject[N].available -> STOP).
||BTOUTPUTS(N=1) = ((if (N > 0) then (forall [i:0..N-1] OUTPUT(i))).

// Compound task, output mapping.
CTOUTPUT(N=1) = (outputobject[N].available-> available -> STOP).
||CTOUTPUTS(N=1) = (if (N>0) then (forall [i:0..N-1] CTOUTPUT(i))).
```

This first part is independent of the specification and is always present in the LTS specification. It describes how our input and output objects are related to their parent set. The processes starting with BT are for basic tasks while those starting with CT are for compound tasks. These declarations are parameterised.

```
// *******************************************
// ***   instantiation of the workflow:    ***
// *******************************************

INIT = (inputset[0].inputobject[0].available  -> STOP).
```

This provides the initial input object to the workflow application that can then start.

```
// *******************************************
// ***   specification of the workflow:    ***
// *******************************************

||WF = (wf:INIT  || wf:TASK_INST0).

||TASK0_RESULT      =      (NOTIFICATION1      ||      NOTIFICATION2      ||
outputset[0]:CTOUTPUTS(0) || outputset[1]:CTOUTPUTS(0)).
||TASK0_GETINPUT = (inputset[0]:TASKCLASS_GETINPUTS(1) || TASK_START).
TASK0_START = ( inputset[0].available -> active -> STOP).
```

```
||TASK0_EXEC = ( TASK_INST1 || TASK_INST2).
||TASK_INST0 = (TASK0_GETINPUT || TASK0_EXEC || TASK0_RESULT).

||TASK_INST1 = (task1:TASK1 || DATADEPENDENCY0).
||TASK_INST2 = (task2:TASK2 || DATADEPENDENCY1 || NOTIFICATION0).

// Dataflow Dependencies
DATADEPENDENCY0      =      (inputset[0].inputobject[0].available      ->
task1.inputset[0].inputobject[0].available -> STOP).
DATADEPENDENCY1      =      (inputset[0].inputobject[0].available      ->
task2.inputset[0].inputobject[0].available -> STOP).

NOTIFICATION0          =          (task1.outputset[0].available          ->
task2.inputset[0].available -> STOP).
NOTIFICATION1 = (task2.outputset[0].available -> outputset[0].available -
> STOP).

// Temporal Dependencies (notifications)
NOTIFICATION2 = (task1.outputset[1].available -> outputset[1].available -
> STOP | task2.outputset[1].available)   -> outputset[1].available ->
STOP).

||TASK1_GETINPUT = (TASK1_START).
TASK1_START = ( inputset[0].available -> active -> STOP).
TASK1_EXEC = ( active -> outputset[0].available -> STOP).
||TASK1_RESULT          =          (outputset[0]:BTOUTPUTS(0)          ||
outputset[1]:BTOUTPUTS(0)).
||TASK1 = (TASK1_GETINPUT || TASK1_EXEC ||TASK1_RESULT ).

||TASK2_GETINPUT = (inputset[0]:TASKCLASS_GETINPUTS(1) || TASK2_START).
TASK2_START = ( inputset[0].available -> active -> STOP).
TASK2_EXEC = (active -> outputset[0].available -> STOP).
||TASK2_RESULT          =          (outputset[0]:BTOUTPUTS(0)          ||
outputset[1]:BTOUTPUTS(0)).
||TASK2 = (TASK2_GETINPUT || TASK2_EXEC ||TASK2_RESULT ).
```

Had we had more than one input set then they would be listed in TASKCLASS_GETINPUT as well as in TASK_START as a choice ("|"). Basic tasks with several output sets would list them in TASK_RESULT as well as in TASK_EXEC as choices by using the constructor "|".

It has to be noticed that an alternative to using the choice constructor is to use the range feature of the language. For instance, given a taskclass with I input sets and J output sets, a basic task with that task class could be defined as:

```
TC(I=1, J =1) = ( inputset[0..I-1]:available -> ACTIVE),
ACTIVE = ( outputset[0..J-1]:available -> STOP).
```

This just states that we need one inputset available and that then one outputset non-deterministically chosen is made available. TC has to be started as a parallel process for the task and labelled with the name of the task.

It is also possible to simplify this script by using the re-labelling feature of the language. It has to be noticed that we did not hide the names of the components embedded in a compound task as anyhow the names are unique.

7.2.3 Usefulness for our system

The Labelled Transaction System Analyser is a tool for the verification of concurrent systems. It provides some methods to check safety and properties of a FSP specification. The state interacting machines composing the system can be animated by the tool or used to check that the properties expected are satisfied after compiling the FSP specification. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. It can also perform a breadth first search on the target LTS. If a property violation or deadlock is found, the shortest trace of actions that would lead to the property violation or deadlock is displayed in the output window. It also computes the connected components for the target LTS. Traces are produced for cycles that cause liveness property violations.

As far as our workflow specifications are concerned, it allows testing for cycles, as well as reachability. Deadlocks can be discovered if you instead of finishing the workflow by a STOP, you use another terminating state. For instance, in our example, the workflow terminates either in the done or in the failed outcome of the workflow (outputset[0] and outputset[1]), we should then have

```
FINISH = ({wf:outputset[0], wf:outputset[1]} -> END),
END = (end -> END).
```

Running the workflow now requires running this new FINISH process in parallel. Checking whether the «end» action has been used can now test termination.

In this chapter, it was demonstrated that our language could be easily mapped to some other languages hence allowing usage of the toolkits available for these languages. We have described how our system maps its workflow specifications to Petri nets and FTS specifications that can then be used to test whether different properties are verified. This is mainly useful to test for reachability as well as for potential cycles and deadlocks. The main problem with such toolkits is that they tend to be implemented as closed systems making it difficult to integrate them with other toolkits, such as the workflow toolkit presented in the previous chapter. Ideally, the users of the workflow toolkit should be able to make use of tools such as the LTSA transparently to check relevant properties without having to learn FSP specifications. This is left as a future work item

Chapter 8

Conclusions and future work

This thesis describes the design and implementation of a toolkit allowing users to specify, execute and monitor dependable distributed workflow applications. This work started from the observation that more and more applications are being built by composing them out of other existing applications. Moreover many applications are also likely to be modified dynamically because of the changes of the environment in which they are executing. Underlying mechanisms are therefore needed to support dynamic modifications of the application in a dependable way. As a result, an application building framework is needed to provide users with an easy way to specify, compose, execute and monitor such applications.

In this chapter, we sum up the contributions of this thesis, and list possible directions for future work

Thesis contributions

Most currently available workflow systems possess monolithic structure, so do not provide distributed execution environments. Further, they offer little support for building fault-tolerant applications, nor can they inter-operate, as they make use of proprietary platforms and protocols. Even the reference architecture of the workflow management coalition (WfMC) (presented in section 2.1.1) has a monolithic structure and does not meet all the requirements of distributed workflow execution as it centralises a lot of functions including many service provider domains in a single logical entity (the workflow server). They do not separate the responsibilities between workflow domains and service provider domains as it is the workflow server that decides the task implementation rather than the service provider. Scalability is also an issue as for instance workflow clients must know a priori where work is going to come from and they use a pull model that does not scale when work comes from many servers. A detailed discussion of the drawbacks

of the WfMC model can be found in [63][64]

We have therefore built a transactional workflow system whose architecture is decentralised and open: it has been designed and implemented as a set of CORBA services to run on top of a given ORB. Furthermore, the system has been structured to provide fault tolerance at application level and system level. Support for application level fault tolerance has been provided through flexible task composition facilities that enable an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within an application to deal with a variety of exceptional situations. The language presented in chapter 4 has been specially designed to allow an easy specification of application fault tolerance. Support for system level fault tolerance has been provided by recording inter-task dependencies in (CORBA) transactional shared objects and by using transactions to implement the delivery of task outputs such that destination tasks receive their inputs despite a finite number of intervening machine crashes and temporary network related failures; this also provides a durable audit trail of task interactions. Thus our system naturally provides a fault-tolerant 'job scheduling' environment. Using task factories allows us to let the service provider decide the task implementation as late as possible.

To sum up, the main contributions of the work described in this thesis are:

- A new co-ordination language (scripting language) allowing easy specification of the composition of workflow applications in terms of tasks and their data-flow or temporal dependencies.
- Simple uniform model for specification, execution and monitoring of the workflow applications, allowing a flexible construction of the applications out of other applications.
- Support for dynamic reconfiguration of workflow. The toolkit provides a support for run-time modification of the workflow, with late binding of the task implementation. The dynamic reconfiguration being carried out using transactions at the low level, the system also maintains the workflow integrity.
- Our work is novel in that we do take into account the fault tolerance aspects of the workflow applications. Indeed, the system provides some support for fault tolerance both at the system and at the application level, allowing the construction of dependable distributed applications. The programmer explicitly models the application fault tolerance as part of the workflow specification, with the possibility to use compound task to hide the possible complexity of the failure handling tasks.

- Implementation of a toolkit allowing the specification, execution and monitoring of dependable distributed applications. The toolkit is user friendly as it provides graphical tools that map an easy-to-understand high level specification on to the various CORBA services of the underlying system (task control and task factories...). In addition, it provides a number of consistency checking tools.
- Fully open and interoperable system by using CORBA and Java middleware technologies.

The figure below summarises the differences between the major workflow systems that are around and the Newcastle workflow system.

System	Model	Fault tolerance	Dynamism	Interoperability
Sagas	Serialised transactions with associated compensating actions	Save points & compensations	None	Homogeneous
ConTract	Group of transactions	Compensations, Steps seen as transactions	None	Homogeneous
ORBWork	CORBA workflow system	Recovery managers using persistent storage & application level	Some	CORBA Web
Exotica	Message based workflow management	Persistent messages, atomicity of changes not guaranteed	None	Proprietary
RainMan	Sources co-ordinating performers executing the process	Persistent worklists, long-run conversations being considered.	Dynamic updates of workflow graph	Written in Java, heterogeneous environment
TOWE	Transactional Workflow system	Basic units of work are ACID. Open-nested transactions.	None	Homogeneous
Newcastle	Transactional CORBA workflow system	Both system and application level (alternative), persistent storage	Full dynamic updates of specification	CORBA, heterogeneous environment

The toolkit presented in this thesis is intended to provide a fault-tolerant execution environment for long running distributed applications that represent business processes in fields such as telecommunication, electronic commerce and banking.

We have described the design and implementation of a toolkit supporting the specification, execution and monitoring of such applications. It indeed enables executing workflow applications composed of inter-related tasks, in a dependable manner. The transactional workflow approach chosen to provide the underlying support environment for co-ordinating task execution provides system level fault tolerance, while the language allows the specification of application level fault tolerance in a uniform manner. Further applications can be co-ordinated in a centralised or decentralised manner (the task controllers can be started wherever needed). The system also

meets the requirements of interoperability by using middleware technologies such as CORBA and Java to provide an interoperable, open system. Our task model allows flexible task composition. As far as dynamic reconfiguration is concerned, task specifications can be fully modified till they are started and afterwards, the composition and possible outcomes of compound tasks can still be modified as long as they have not completed. Use of transactions allows any changes to be carried out atomically.

Directions for future work

The implementation of this toolkit is only a first step towards providing a comprehensive framework for building complex dependable workflow applications. It allows users to specify a complex business process as a set of tasks linked by dependencies. Several aspects need to be developed:

- One is to add more pre-defined types of tasks for high level specification. Right now, we have "repeat tasks" that are expanded as low level tasks using genesis tasks. The question is which kind of pre-defined tasks are needed? It may be interesting for instance to consider adding support for replicated tasks, quorum tasks (e.g. starting several identical tasks and reach an outcome when a certain number of these tasks agree on the outcome), alternative tasks (as in the flight and hotel reservation example of chapter 5, section 2). Toolkit support for template tasks (as defined in the language) is also required.
- We should also integrate a model checking toolkit such as LTSA with the Graphic User Interface Specification tools to allow users to check properties of their application before actually running them as right now the toolkit is not providing a full check of the specification.
- If a requested task factory is not available, there is no real handling of the situation as it was assumed that it was part of the responsibility of the programmer to make sure that the factories he plans using are available.
- When a user starts a Workflow application, he is presented with a list of potential objects to choose from based on their class. Right now all objects of that class are registered as possible choices. This is not realistic as there might be occurrences where there are thousands of them. A solution would be to use a Directory Service to only register what is potentially useful.
- Another is to provide better support for the instantiation of a workflow schema based

on an organisation's needs that takes into account variety of criteria, such as placement of task controllers (centralised control versus distributed control), security requirements, resource usage, roles (with associated responsibilities), etc. Currently, we are using some implementation criteria (specified as part of the meta information associated to the tasks) to choose the task/task control factory to be used to create our task and tasks controllers. We need to create a number of task factories each customised for a class of application (e.g. a factory for electronic commerce, another for telecommunication...). A possible improvement of the current model could be a two-stage workflow instantiation process where the factories are chosen depending on a role that they are to fulfil. Task definitions would have two attribute-value items: role name and task type. This information is passed to a first (initial) task factory by invoking its create operation. This factory queries an organisation model held in a database. The database could contain, for every role in that organisation, the name of the role's task factory. A role's task factory is capable of creating all types of task objects that role is responsible for. The initial task factory would then invoke the create operation of the specific task factory associated with the role, passing the task type. The role's task factory would then create the specific task, obtaining all the location specific information from the database.

- There is also considerable room for trying to combine software-architecture-based development environments, and software agents with our approach using transactional workflow management systems. On one hand, agents are software entities that perform operations on behalf of a user or another software entity. As a result, agents could be mapped to our tasks at run time, by creating some agent task factories. Similarly, agents could have some of the services that they are proposing modelled as a workflow. On the other hand, software architecture specifications expressed via Architecture Description Languages (cf. chapter 2 for examples of ADLs) allow the specification of the configuration of the software components and capture the non-behavioural aspects of system structure that our system does not express. We are currently investigating the integration of these three technologies in a research project called C3DS [11].
- In chapter 5, a number of examples were used to illustrate the suitability of our language for specifying workflow applications. As part of the future work, real applications should be developed, executed and monitored using our toolkit, hence validating the use in the real world of the workflow management system. We are planning to do this in an

industry led ESPRIT project, MULTIPLECX [43], on business to business electronic commerce.

Appendixes

Appendix A

Scripts

A.1 Script for the process ordering application described in chapter 5.1.

```
/*
 * This script was generated for root by the Workflow Management Tool v1.20a
 * Copyright (C) 1996, 1997, 1998
 *
 * Department of Computing Science,
 * University of Newcastle upon Tyne,
 * Newcastle upon Tyne,
 * UK.
 */

objectclass Bill;
objectclass Goods;
objectclass Order;
taskclass CheckStock
{
   inputs
   {
      input main
      {
         order of class Order
      }
   };
   outputs
   {
      outcome failed
      {
      };
      outcome success
      {
```

```
            items of class Goods
        }
    }
};
taskclass Dispatch
{
    inputs
    {
        input main
        {
            items of class Goods
        }
    };
    outputs
    {
        outcome abort aborted
        {
        };
        outcome success
        {
            items of class Goods
        }
    }
};
taskclass PaymentAuthorisation
{
    inputs
    {
        input main
        {
            order of class Order
        }
    };
    outputs
    {
        outcome failed
        {
        };
        outcome success
        {
            bill of class Bill
        }
    }
};
taskclass PaymentCapture
{
    inputs
    {
        input main
        {
            bill of class Bill
        }
    };
    outputs
    {
        outcome done
        {
        }
    }
};
taskclass ProcessOrder
{
    inputs
```

```
    {
        input main
        {
            order of class Order
        }
    };
    outputs
    {
        outcome failed
        {
        };
        outcome success
        {
            items of class Goods
        }
    }
};
compoundtask processOrderApplication of taskclass ProcessOrder
{
    implementation
    {
        "GUI_X" is "235";
        "GUI_Y" is "100";
        "Node" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject order from
            {
            }
        }
    };
    task checkStock of taskclass CheckStock
    {
        implementation
        {
            "GUI_X" is "164";
            "GUI_Y" is "180";
            "TaskCtrlFactory" is "Order";
            "TaskImpl" is "CheckStock.wf";
            "Node" is "kellah"
        };
        inputs
        {
            input main
            {
                inputObject order from
                {
                    order of task processOrderApplication if input main
                }
            }
        }
    };
    task dispatch of taskclass Dispatch
    {
        implementation
        {
            "GUI_X" is "477";
            "GUI_Y" is "184";
            "TaskCtrlFactory" is "Order";
            "TaskImpl" is "Dispatch";
```

```
        "Node" is "kellah"
    };
    inputs
    {
        input main
        {
            notification from
            {
                task paymentAuthorisation if output success
            };
            inputObject items from
            {
                items of task checkStock if output success
            }
        }
    }
};
task paymentAuthorisation of taskclass PaymentAuthorisation
{
    implementation
    {
        "GUI_X" is "186";
        "GUI_Y" is "119";
        "TaskCtrlFactory" is "Order";
        "TaskImpl" is "PaymentAuthorisation.wf";
        "Node" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject order from
            {
                order of task processOrderApplication if input main
            }
        }
    }
};
task paymentCapture of taskclass PaymentCapture
{
    implementation
    {
        "GUI_X" is "439";
        "GUI_Y" is "67";
        "TaskCtrlFactory" is "Order";
        "TaskImpl" is "PaymentCapture";
        "Node" is "kellah"
    };
    inputs
    {
        input main
        {
            notification from
                    {
                task checkStock if output success
            };
            inputObject bill from
            {
                bill of task paymentAuthorisation if output success
            }
        }
    }
};
```

```
   outputs
   {
      outcome failed
      {
         notification from
         {
            task checkStock if output failed;
            task dispatch if output aborted;
            task paymentAuthorisation if output failed
         }
      };
      outcome success
      {
         notification from
         {
            task paymentCapture if output done
         };
         outputObject items from
         {
            items of task dispatch if output success
         }
      }
   }
}
```

A.2 Script for the travel agent application described in chapter 5.2.

```
/*
 * This script was generated for root by the Workflow Management Tool v1.20a
 * Copyright (C) 1996, 1997, 1998
 *
 * Department of Computing Science,
 * University of Newcastle upon Tyne,
 * Newcastle upon Tyne,
 * UK.
 */

objectclass CustomerInfo;
objectclass Date;
objectclass Flight;
objectclass Hotel;
objectclass Location;
objectclass User;
objectclass integer;
taskclass CompensateFlightReservation
{
   inputs
   {
      input main
      {
         plane of class Flight
      }
   };
   outputs
   {
      outcome failed
      {
      };
      outcome success
      {
```

```
      }
   }
};
taskclass DataAcquisition
{
   inputs
   {
      input main
      {
         user of class User
      }
   };
   outputs
   {
      outcome failed
      {
      };
      outcome success
      {
         customer of class CustomerInfo;
         end of class Date;
         maxCost of class integer;
         place of class Location;
         start of class Date
      }
   }
};
taskclass FlightReservation
{
   inputs
   {
      input main
      {
         end of class Date;
         maxCost of class integer;
         place of class Location;
         start of class Date
      }
   };
   outputs
   {
      outcome failed
      {
      };
      outcome success
      {
         cost of class integer;
         plane of class Flight
      }
   }
};
taskclass HotelReservation
{
   inputs
   {
      input main
      {
         end of class Date;
         place of class Location;
         start of class Date
      }
   };
   outputs
```

```
    {
        outcome failed
        {
        };
        outcome success
        {
            hotel of class Hotel
        }
    }
};
taskclass PrintTickets
{
    inputs
    {
        input main
        {
            customer of class CustomerInfo;
            hotel of class Hotel;
            plane of class Flight
        }
    };
    outputs
    {
        outcome failed
        {
        };
        outcome success
        {
        }
    }
};
taskclass Travel
{
    inputs
    {
        input main
        {
            user of class User
        }
    };
    outputs
    {
        outcome failed
        {
        };
        outcome reserved
        {
        };
        outcome success
        {
        };
        mark toPay
        {
            cost of class integer
        }
    }
};
taskclass TravelReservation
{
    inputs
    {
        input main
        {
```

```
            user of class User
        }
    };
    outputs
    {
        outcome abort aborted
        {
        };
        repeat retry
        {
        };
        outcome success
        {
            cost of class integer;
            customer of class CustomerInfo;
            hotel of class Hotel;
            plane of class Flight
        }
    }
};
compoundtask travel of taskclass Travel
{
    implementation
    {
        "GUI_X" is "408";
        "GUI_Y" is "167";
        "Host" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject user from
            {
            }
        }
    };
    task printTickets of taskclass PrintTickets
    {
        implementation
        {
            "GUI_X" is "498";
            "GUI_Y" is "158";
            "Host" is "kellah";
            "TaskCtrlFactory" is "Travel";
            "TaskImpl" is "PrintTickets"
        };
        inputs
        {
            input main
            {
                inputObject customer from
                {
                        customer of task travelReservation if output success
                };
                inputObject hotel from
                {
                        hotel of task travelReservation if output success
                };
                inputObject plane from
                {
                        plane of task travelReservation if output success
                }
```

```
        }
    }
};
compoundtask travelReservation of taskclass TravelReservation
{
    implementation
    {
        "GUI_X" is "197";
        "GUI_Y" is "149";
        "Host" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject user from
            {
                user of task travel if input main
            }
        }
    };
    task compensateFlightReservation of taskclass
CompensateFlightReservation
    {
        implementation
        {
            "GUI_X" is "552";
            "GUI_Y" is "139";
            "Host" is "kellah";
            "TaskCtrlFactory" is "Travel";
            "TaskImpl" is "CompensateFlightreservation"
        };
        inputs
        {
            input main
            {
                notification from
                {
                    task hotelReservation if output failed
                };
                inputObject plane from
                {
                    plane of task flightReservation if output success
                }
            }
        }
    };
    task dataAcquisition of taskclass DataAcquisition
    {
        implementation
        {
            "GUI_X" is "116";
            "GUI_Y" is "153";
            "Host" is "kellah";
            "TaskCtrlFactory" is "Travel";
            "TaskImpl" is "DataAcquisition"
        };
        inputs
        {
            input main
            {
                inputObject user from
                {
```

```
                    user of task travelReservation if input main
                }
            }
        }
    };
    task flightReservation of taskclass FlightReservation
    {
        implementation
        {
            "GUI_X" is "240";
            "GUI_Y" is "96";
            "Host" is "www.itn.net";
            "TaskCtrlFactory" is "Travel";
            "TaskImpl" is "FlightReservation"
        };
        inputs
        {
            input main
            {
                inputObject end from
                {
                    end of task dataAcquisition if output success
                };
                inputObject maxCost from
                {
                    maxCost of task dataAcquisition if output success
                };
                inputObject place from
                {
                    place of task dataAcquisition if output success
                };
                inputObject start from
                {
                    start of task dataAcquisition if output success
                }
            }
        }
    };
    task hotelReservation of taskclass HotelReservation
    {
        implementation
        {
            "GUI_X" is "382";
            "GUI_Y" is "192";
            "Host" is "kellah";
            "TaskCtrlFactory" is "Travel";
            "TaskImpl" is "hotelReservation"
        };
        inputs
        {
            input main
            {
                notification from
                {
                    task flightReservation if output success
                };
                inputObject end from
                {
                    end of task dataAcquisition if output success
                };
                inputObject place from
                {
                    place of task dataAcquisition if output success
```

```
                    };
                    inputObject start from
                    {
                          start of task dataAcquisition if output success
                    }
                }
            }
        };
        outputs
        {
            outcome abort aborted
            {
                notification from
                {
                      task dataAcquisition if output failed;
                      task flightReservation if output failed;
                      task compensateFlightReservation if output failed
                }
            };
            repeat retry
            {
                notification from
                {
                      task compensateFlightReservation if output success
                }
            };
            outcome success
            {
                notification from
                {
                      task flightReservation if output success
                };
                outputObject cost from
                {
                      cost of task flightReservation if output success
                };
                outputObject customer from
                {
                      customer of task dataAcquisition if output success
                };
                outputObject hotel from
                {
                      hotel of task hotelReservation if output success
                };
                outputObject plane from
                {
                      plane of task flightReservation if output success
                }
            }
        }
    };
    outputs
    {
        outcome failed
        {
            notification from
            {
                task travelReservation if output aborted
            }
        };
        outcome reserved
        {
            notification from
```

```
            {
               task travelReservation if output success;
               task printTickets if output failed
            }
         };
         outcome success
         {
            notification from
            {
               task travelReservation if output success;
               task printTickets if output success
            }
         };
         mark toPay
         {
            outputObject cost from
            {
               cost of task travelReservation if output success
            }
         }
      }
}
```

New taskHotelReservation with alternative:

```
      compoundtask hotelReservation of taskclass HotelReservation
      {
         implementation
         {
            "GUI_X" is "382";
            "GUI_Y" is "192";
            "Host" is "kellah"
         };
         inputs
         {
            input main
            {
                  notification from
                  {
                        task flightReservation if output success
                  };
                  inputObject end from
                  {
                        end of task dataAcquisition if output success
                  };
                  inputObject place from
                  {
                        place of task dataAcquisition if output success
                  };
                  inputObject start from
                  {
                        start of task dataAcquisition if output success
                  }
            }
         };
         task bookHotelPartner of taskclass HotelReservation
         {
            implementation
            {
                  "GUI_X" is "414";
                  "GUI_Y" is "208";
                  "Host" is "www.hilton.com";
                  "TaskCtrlFactory" is "Travel";
                  "TaskImpl" is "hotelReservation"
```

```
    };
    inputs
    {
          input main
          {
                inputObject end from
                {
                      end of task hotelReservation if input main
                };
                inputObject place from
                {
                      place of task hotelReservation if input main
                };
                inputObject start from
                {
                      start of task hotelReservation if input main
                }
          }
    }
};
task bookHotelTouristOffice of taskclass HotelReservation
{
    implementation
    {
          "GUI_X" is "584";
          "GUI_Y" is "120";
          "Host" is "www.travel-reservation.com";
          "TaskCtrlFactory" is "Travel";
          "TaskImpl" is "hotelReservation"
    };
    inputs
    {
          input main
          {
                notification from
                {
                      task bookHotelPartner if output failed
                };
                inputObject end from
                {
                      end of task hotelReservation if input main
                };
                inputObject place from
                {
                      place of task hotelReservation if input main
                };
                inputObject start from
                {
                      start of task hotelReservation if input main
                }
          }
    }
};
outputs
{
    outcome failed
    {
          notification from
          {
                task bookHotelTouristOffice if output failed
          }
    };
    outcome success
```

```
            {
                  outputObject hotel from
                  {
                        hotel of task bookHotelTouristOffice if output
success;
                        hotel of task bookHotelPartner if output success
                  }
            }
         }
      };
```

A.3 Scripts for the telecommunication application described in chapter 5.3.

```
/*
 * This script was generated for fred by the Workflow Management Tool v1.20a
 * Copyright (C) 1996, 1997, 1998
 *
 * Department of Computing Science,
 * University of Newcastle upon Tyne,
 * Newcastle upon Tyne,
 * UK.
 */

objectclass Alarm;
objectclass ImpactList;
objectclass ResolutionList;
taskclass AlarmResolution
{
   inputs
   {
      input main
      {
         alarm of class Alarm
      }
   };
   outputs
   {
      outcome success
      {
      };
      outcome failed
      {
      }
   }
};
taskclass SIA
{
   inputs
   {
      input main
      {
         alarm of class Alarm
      }
   };
   outputs
   {
      outcome analysed
      {
         impacts of class ImpactList
      };
      outcome failed
      {
```

```
      };
      outcome noImpact
      {
      }
   }
};
taskclass SIAACKN
{
   inputs
   {
      input main
      {
         list of class ImpactList
      }
   };
   outputs
   {
      outcome ok
      {
      };
      outcome refuse
      {
      }
   }
};
taskclass SIR
{
   inputs
   {
      input main
      {
         impacts of class ImpactList
      }
   };
   outputs
   {
      outcome resolved
      {
         resolutions of class ResolutionList
      };
      outcome failed
      {
      };
      outcome noResolution
      {
      }
   }
};
taskclass SIRACKN
{
   inputs
   {
      input main
      {
         list of class ResolutionList
      }
   };
   outputs
   {
      outcome ok
      {
      };
      outcome refuse
```

```
      {
      }
   }
};
taskclass SLA
{
   inputs
   {
      input main
      {
         resolutions of class ResolutionList
      }
   };
   outputs
   {
      outcome resolved
      {
         resolutions of class ResolutionList
      };
      outcome failed
      {
      };
      outcome noResolution
      {
      }
   }
};
taskclass SLAdynCreatNegociation
{
   inputs
   {
      input main
      {
         resolutions of class ResolutionList
      }
   };
   outputs
   {
      outcome completed
      {
      };
      outcome failed
      {
      }
   }
};
compoundtask alarmResolution of taskclass AlarmResolution
{
   implementation
   {
      "GUI_X" is "190";
      "GUI_Y" is "77";
      "Host" is "kellah"
   };
   inputs
   {
      input main
      {
         inputObject alarm from
         {
         }
      }
   };
```

```
compoundtask sia of taskclass SIA
{
   implementation
   {
      "GUI_X" is "168";
      "GUI_Y" is "119";
      "Host" is "kellah"
   };
   inputs
   {
      input main
      {
         inputObject alarm from
         {
               alarm of task alarmResolution if input main
         }
      }
   };
   task siaAnalyse of taskclass SIA
   {
      implementation
      {
         "GUI_X" is "227";
         "GUI_Y" is "137";
         "TaskCtrlFactory" is "Telecom";
         "TaskImpl" is "SIA";
         "Host" is "kellah"
      };
      inputs
      {
         input main
         {
               inputObject alarm from
               {
                     alarm of task sia if input main
               }
         }
      }
   };
   task siaValidation of taskclass SIAACKN
   {
      implementation
      {
         "GUI_X" is "417";
         "GUI_Y" is "109";
         "TaskCtrlFactory" is "Telecom";
         "TaskImpl" is "SIAGUI";
         "Host" is "kellah"
      };
      inputs
      {
         input main
         {
               inputObject list from
               {
                     impacts of task siaAnalyse if output analysed
               }
         }
      }
   };
   outputs
   {
      outcome analysed
```

```
        {
           notification from
           {
                 task siaValidation if output ok
           };
           outputObject impacts from
           {
                 impacts of task siaAnalyse if output analysed
           }
        };
        outcome failed
        {
           notification from
           {
                 task siaAnalyse if output failed
           }
        };
        outcome noImpact
        {
           notification from
           {
                 task siaAnalyse if output noImpact;
                 task siaValidation if output refuse
           }
        }
     }
};
compoundtask sir of taskclass SIR
{
   implementation
   {
      "GUI_X" is "275";
      "GUI_Y" is "147";
      "Host" is "kellah"
   };
   inputs
   {
      input main
      {
         inputObject impacts from
         {
               impacts of task sia if output analysed
         }
      }
   };
   task sirAnalyse of taskclass SIR
   {
      implementation
      {
         "GUI_X" is "227";
         "GUI_Y" is "137";
         "TaskCtrlFactory" is "Telecom";
         "TaskImpl" is "SIR";
         "Host" is "kellah"
      };
      inputs
      {
         input main
         {
               inputObject impacts from
               {
                     impacts of task sir if input main
               }
```

```
            }
        }
    };
    task sirValidation of taskclass SIRACKN
    {
        implementation
        {
            "GUI_X" is "417";
            "GUI_Y" is "109";
            "TaskCtrlFactory" is "Telecom";
            "TaskImpl" is "SIRGUI";
            "Host" is "kellah"
        };
        inputs
        {
            input main
            {
                inputObject list from
                {
                    resolutions of task sirAnalyse if output resolved
                }
            }
        }
    };
    outputs
    {
        outcome resolved
        {
            notification from
            {
                task sirValidation if output ok
            };
            outputObject resolutions from
            {
                resolutions of task sirAnalyse if output resolved
            }
        };
        outcome failed
        {
            notification from
            {
                task sirAnalyse if output failed
            }
        };
        outcome noResolution
        {
            notification from
            {
                task sirAnalyse if output noResolution;
                task sirValidation if output refuse
            }
        }
    }
};
compoundtask sla of taskclass SLA
{
    implementation
    {
        "GUI_X" is "447";
        "GUI_Y" is "183";
        "Host" is "kellah"
    };
    inputs
```

```
    {
       input main
       {
          inputObject resolutions from
          {
                resolutions of task sir if output resolved
          }
       }
    };
    task createNegotiateResolution of taskclass SLAdynCreatNegociation
    {
       implementation
       {
          "GUI_X" is "184";
          "GUI_Y" is "132";
          "TaskCtrlFactory" is "Telecom";
          "TaskImpl" is "SLAdyn";
          "Host" is "kellah"
       };
       inputs
       {
          input main
          {
                inputObject resolutions from
                {
                      resolutions of task sla if input main
                }
          }
       }
    };
    outputs
    {
       outcome resolved
       {
          outputObject resolutions from
          {
          }
       };
       outcome failed
       {
          notification from
          {
                task createNegotiateResolution if output failed
          }
       };
       outcome noResolution
       {
       }
    }
};
outputs
{
    outcome success
    {
       notification from
       {
          task sia if output noImpact;
          task sir if output noResolution;
          task sla if output noResolution;
          task sla if output resolved
       }
    };
    outcome failed
```

```
      {
         notification from
         {
            task sia if output failed;
            task sir if output failed;
            task sla if output failed
         }
      }
   }
}
```

Workflow dynamically created:

```
/*
 * This script was generated for fred by the Workflow Management Tool v1.20a
 * Copyright (C) 1996, 1997, 1998
 *
 * Department of Computing Science,
 * University of Newcastle upon Tyne,
 * Newcastle upon Tyne,
 * UK.
 */

objectclass Bid;
objectclass Resolution;
taskclass SLAbid
{
   inputs
   {
      input main
      {
         bid of class Bid;
         resolution of class Resolution
      }
   };
   outputs
   {
      outcome accepted
      {
         bid of class Bid
      };
      outcome refused
      {
      };
      repeat nextround
      {
         bid of class Bid;
         resolution of class Resolution
      }
   }
};
taskclass SLAbidInit
{
   inputs
   {
      input main
      {
         resolution of class Resolution
      }
   };
   outputs
   {
```

```
      outcome start
      {
         bid of class Bid
      }
   }
};
taskclass SLAbidRound
{
   inputs
   {
      input main
      {
         bid of class Bid;
         resolution of class Resolution
      }
   };
   outputs
   {
      outcome accepted
      {
         bid of class Bid
      };
      outcome refused
      {
      };
      outcome nextround
      {
         bid of class Bid
      }
   }
};
taskclass SLAnegotiation
{
   inputs
   {
      input main
      {
         resolution of class Resolution
      }
   };
   outputs
   {
      outcome negociated
      {
         bid of class Bid
      };
      outcome failed
      {
      }
   }
};
compoundtask negociateResolution of taskclass SLAnegotiation
{
   implementation
   {
      "GUI_X" is "240";
      "GUI_Y" is "106";
      "Host" is "kellah"
   };
   inputs
   {
      input main
      {
```

```
            inputObject resolution from
            {
            }
        }
    };
    task initNegotiation of taskclass SLAbidInit
    {
        implementation
        {
            "GUI_X" is "242";
            "GUI_Y" is "149";
            "TaskCtrlFactory" is "Telecom";
            "TaskImpl" is "SLABidInit";
            "Host" is "kellah"
        };
        inputs
        {
            input main
            {
                inputObject resolution from
                {
                    resolution of task negociateResolution if input main
                }
            }
        }
    };
    compoundtask negotiationRound of taskclass SLAbid
    {
        implementation
        {
            "GUI_X" is "448";
            "GUI_Y" is "105";
            "Host" is "kellah"
        };
        inputs
        {
            input main
            {
                inputObject bid from
                {
                    bid of task initNegotiation if output start;
                    bid of task negotiationRound if output nextround
                };
                inputObject resolution from
                {
                    resolution of task negociateResolution if input main;
                    resolution of task negotiationRound if output nextround
                }
            }
        };
        task consumer of taskclass SLAbidRound
        {
            implementation
            {
                "GUI_X" is "279";
                "GUI_Y" is "117";
                "TaskCtrlFactory" is "Telecom";
                "TaskImpl" is "SLAclient";
                "Host" is "kellah"
            };
            inputs
            {
                input main
```

```
        {
            inputObject bid from
            {
                bid of task negotiationRound if input main
            };
            inputObject resolution from
            {
                resolution of task negotiationRound if input main
            }
        }
    }
};
task producer of taskclass SLAbidRound
{
    implementation
    {
        "GUI_X" is "451";
        "GUI_Y" is "171";
        "TaskCtrlFactory" is "Telecom";
        "TaskImpl" is "SLAprovider";
        "Host" is "kellah"
    };
    inputs
    {
        input main
        {
            inputObject bid from
            {
                bid of task consumer if output nextround
            };
            inputObject resolution from
            {
                resolution of task negotiationRound if input main
            }
        }
    }
};
outputs
{
    outcome accepted
    {
        outputObject bid from
        {
            bid of task consumer if output accepted;
            bid of task producer if output accepted
        }
    };
    outcome refused
    {
        notification from
        {
            task consumer if output refused;
            task producer if output refused
        }
    };
    repeat nextround
    {
        outputObject bid from
        {
            bid of task producer if output nextround
        };
        outputObject resolution from
        {
```

```
                resolution of task negotiationRound if input main
            }
        }
    }
};
outputs
{
    outcome negociated
    {
        outputObject bid from
        {
            bid of task negotiationRound if output accepted
        }
    };
    outcome failed
    {
        notification from
        {
            task negotiationRound if output refused
        }
    }
}
}
```

Appendix B

OpenFlow Toolkit user manual

OPENflow

Integrated Development Environment
User Manual

Version 0.7.0

University of Newcastle
Department of Computing
Newcastle upon Tyne
Tyne and Wear
UK

Requirements

- JDK (Java Development Kit) 1.2 beta 4 or later version
- TCP/IP support
- Web server with CGI-bin access on machine where software is run (applet version)
- 32 Meg RAM (for JDK)

Contact information

Frédéric Ranno
University of Newcastle upon Tyne
Department of Computing
Newcastle upon Tyne
Tyne and Wear
UK

Tel: +44 (0) 191 222 8229
Fax: +44 (0) 191 222 8232
e-mail: Frederic.Ranno@ncl.ac.uk

Table of Contents

Table of Figures

1. Introduction

This manual describes the Integrated Development Environment which is provided with the OPENflow system. The main aim of the OPENflow Integrated Development Environment is to provide high level easy to use facilities to users to enable them to compose workflow applications, and then execute and monitor them. In this document, the reader will find a detailed description of the IDE.

After giving an overview of the IDE as a whole, the reader will find some information on how to set up the IDE in the section "Getting started". Then we will present in turn how the IDE allows you to specify, simulate, execute and monitor a workflow application.

1.1 Overview

The OPENflow Integrated Development Environment is composed of three main components: the Graphic user Interface (WfGUI), a script server (WfFS) and a workflow engine (WfEng). Typically users will only interact directly with the WfGUI.

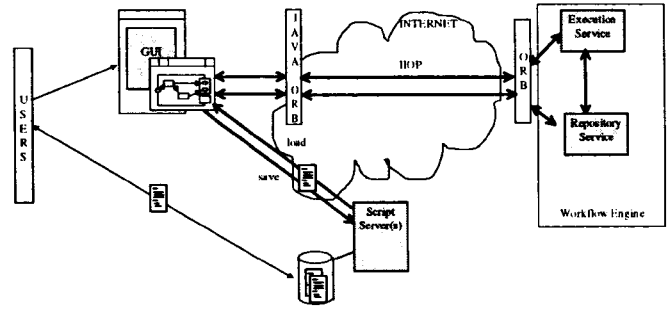


Figure 1.1: Graphical representation of the workflow system

In order to provide greater flexibility by allowing incorrect/incomplete script specifications, workflow file servers were added to store specifications being created. Users wishing to use the textual language to specify a workflow application can also directly export it to a workflow script server. Then they can load it into the toolkit using the WfGUI. Later on, they can also get it back as a script, modify it and export it back to a workflow script server. A full description of the language can be found in *"OPENflow : Textual language"*.

A GUI is provided in order to facilitate the specification, execution and monitoring of the workflows. As one of the requirements was to be able to use the workflow management system from an heterogeneous set of machines and possibly from remote hosts, it was decided to implement the front-end of the workflow system

as a Java application. As a result the GUI is platform independent and has as only requirement to be able to use a Java-enabled machine. It acts as a front end to the Workflow Engine itself specified as a CORBA service.

The Repository Service is responsible for storing the low level (run-time) representation of the tasks. The Execution Service consists of Task and Task Controller factories that are used to instance a specification from the Repository Service. A detailed description of these services can be found in the document entitled *"OPENflow : Workflow Module CORBA Interface Reference Manual"*

Typically a user will create a workflow specification either using the GUI high level tasks or the workflow language, then will export it to the Repository Service and finally will execute it thanks to the Execution Service and monitor its progress.

The main features that the IDE provides are:

- Creation of new workflows either by loading a script from a WfFS or by using the graphical notation used by the GUI.
- Extending, modifying existing specifications
- Checking workflow specifications for errors (loops, etc.)
- Simulating workflow applications
- Instantiating and monitoring workflow applications

2. Getting Started

2.1 Setting up the environment

First of all you will need to change your environment variable CLASSPATH, then you will need to run a naming service as well as several core services.

2.1.1 CLASSPATH, openflow.properties

You need to change your CLASSPATH to include the classes needed to use OPENflow as well as the ORB of your choice.

- In the case of ORBacus, you will need to add $(ORBACUS_HOME)/lib/OB.jar, and $(ORBACUS_HOME)/lib/OBNaming.jar for the ORB.
- If you are using the JDK ORB, the classes are already included

For OPENflow itself, just include $(OPENFLOW_HOME)/lib/OPENflow.jar

You also need to copy the openflow.properties file in your home directory if you are using OPENflow on top of UNIX. This file includes some configuration information for OPENflow.

Some distributions of the software will automatically set it up for you.

2.1.2 Name Service

Assuming that you want to run the Name service on port 8002.

- If you are running "ORBacus", you will need to issue the following command:
  java com.ooc.CosNaming.Server –i –OAport 8002
  You then can add in your openflow.properties file the entry for the IOR
- If you are running the JDK ORB, you will need to issue the following command:
  tnameserv –ORBInitialPort 8002
  (tnameserv is part of the JDK distribution)

Some batch files are provided in most distributions (in the bin directory)

2.1.3 Core Services

There are two sets of core services that are provided: a specification service for the script server and a run-time CORBA set of services needed if you want to use the repository and execution services. If you don't start either of them, the IDE won't work. If you only start one of them, some of the features won't work.

2.1.3.1 Workflow Script Server

If you want to use the script server on port 8081 using as script repository scripts/, you will need to issue the following command .

```
java Toolkit.WfFS.WfFS 8081 scripts/
```

The default Script Server location is localhost port 8081. Shall you wish to use an alternative location, you can specify it using the menu option "Options/Script server"

2.1.3.2 Workflow Engine Services

These services are registered with the Name Service which means that the Name Service must be up and running before you can start these services. All the basic services needed by the Workflow engine have been gathered in a single java program. It includes the various factories needed by the repository and execution services as well as the factories needed by the cgi-form-based example provided with the toolkit.

- If you are running "ORBacus", you will need to issue the following command:
  java com.arjuna.SpecificationServiceImpl.Servers –ORBService NameService
- If you are running the JDK ORB, you will need to issue the following command:
  java com.arjuna.SpecificationServiceImpl.

Check the file Release-Notes.txt or docs/Release-Notes.html for a list of optional extra services to start.

2.2 Login, classes of connections

Once you have started the core services, you can start the application by running the command "java OpenFlowToolkit". The screen represented on figure 2.1 will

then appear and will prompt you for a user name and a password to use the Workflow System.

Three different classes of users have been created:

- *Maintainers*: this is the equivalent of the system Administrators. They have all the features available to them. They can create new users on-line as well as change their class of connection or path used by the name server using the form depicted in figure 3.13.
- *Designers*: they can create their own specifications, but can not modify directly the specifications stored in the repository service.
- *Users*: they can only monitor what is happening in the workflow. In particular they can not create or modify specifications.

The following login have been pre defined: *root* as maintainer, *demo* as designer and *monitor* as user. They all have the same password: *Arjuna*

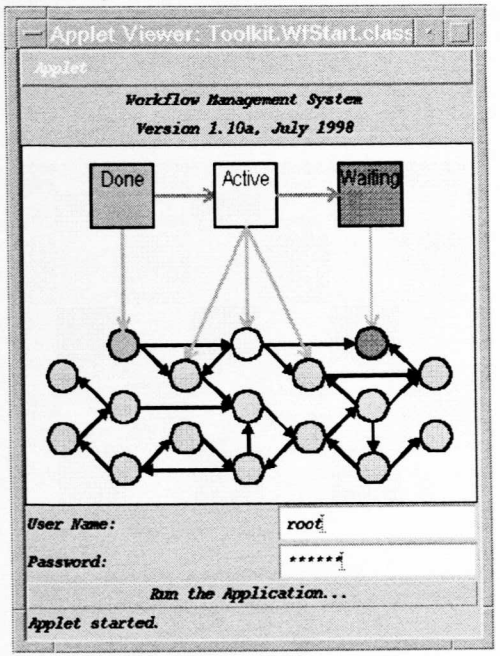


Figure 2.1: Login screen

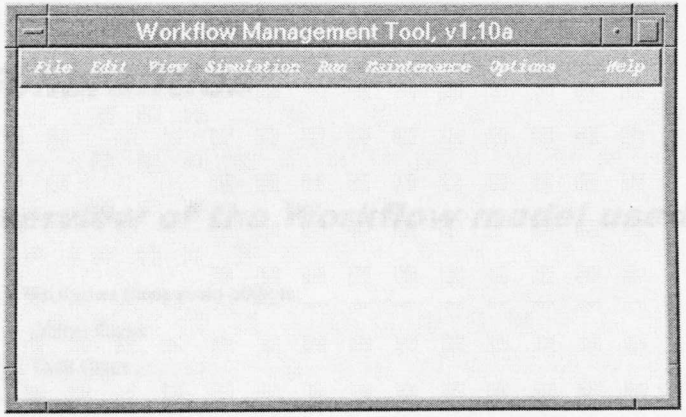


Figure 2.2: Initial screen for maintainers

Once the user has provided its user name and password, the toolkit checks that the user is registered by comparing its user name and password with what is stored in the password file. It also recovers the class of connection and path for the WfEngine of the user and give access to the WfGUI. A user of class maintainer will then see the GUI depicted in figure 2.2, while users of other classes will get access to a subset of the menu.

The commands that can be issued via the WfGUI have been divided into eight main categories: File, Edit, View, Option, Simulation, Run, Maintenance and Help.

- The file menu gathers all the commands related to creating, loading or saving a specification using the WfFS as well as the usual commands exit (destroy the GUI) and close (only hide the GUI).
- The edit menu gathers the commands to create, delete and modify the object classes, task classes and tasks.
- The View menu allows the user to travel in the workflow by zooming in and out of the tasks.
- The options menu allows the user to get on line access to the configuration options such as the location of the workflow file server to use, option to choose what part of the tasks you want to see, the size of the font used...
- The simulation menu allows the user to start, stop, make a step, reset the simulation of a workflow application.
- The run menu allows the user to check the validity of the object classes, task classes, tasks as well as export a specification to the repository service, it also allows the user to start or monitor a task.

- The maintenance menu lets the user add an account if he is a maintainer or change his password otherwise.
- The help menu provides some links to the on-line help. (only if the GUI was started using an applet)

3. Generalities

3.1 Overview of the Workflow model used

The WfGUI uses three main objects:

- Object Class
- Task Class
- Task

In the next paragraphs, the models of these objects will be introduced.

3.1.1 Object Class model

Object classes are used to type the object resources used by the workflow tasks. It allows to type-check the specifications.

3.1.2 Task Class model

A Task Class describes the interface of a task. It specifies the alternative input sets (Task Class Input Sets) and alternative ouput sets (Task Class Output Sets) supported. Task Class Input Sets (respectively Task Class Output Sets) have a set of associated object resources (Task ClassObjects) which could be seen as parameters.

Typically the first input set will be the input set usually used to start the task of this class, while the first output set will be the outcome reached if the task executes without problem. The other output sets are usually seen as outputs corresponding to exception, the latest one being usually reserved for system errors or un-handled application errors.

3.1.3 Task models

In this section, the structure of a workflow task as well as the different types of tasks will be described. As previously stated in section 1.1, workflows can usually be divided into smaller units of work (called tasks) carried out by participants. Participants have to collaborate to reach a common aim, the achievement of the

global process. This collaboration is usually carried out by exchanging data and by ensuring that the dependencies between units of work are respected.

3.1.3.1 Structure of a task

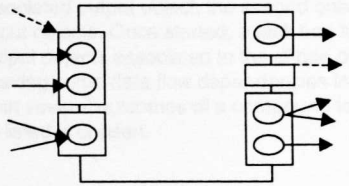


Figure 3.1: Representation of a task

A task is represented in the system by its interface. The only visible parts of a task are its inputs and its outputs. The internals of a task are hidden except when this task is itself composed out of other workflow tasks. In this case, the dependencies among these tasks as well as the mapping between the inputs and outputs of the embedding task and its components are described. The structure of a task is depicted in figure 3.1. At run time, a task will typically gets some inputs and then terminates producing some outputs. To add some flexibility, alternative sets of both inputs and outputs can be specified.

The inputs and output sets are represented by rectangle boxes, the input and output objects by ovals, the data flow dependencies by arrows and the notification dependencies by dotted arrows. The direction in which the arrow leads shows whether it is a dependency with this task as source or as destination. The overall specification can be represented as an acyclic graph. In the rest of this document the tasks that are having dependencies on a task will be referred to as *down-stream tasks*, while the tasks on which this task depends will be referred as *up-stream tasks*. The domain of a task specification or in other words, what is described in the specification of a task is depicted in figure 3.2. The grey box delimits the part of workflow specification associated to the task.

It has to be noticed that our system has as particular feature that the task is only aware of the dependencies it has on up stream tasks. It has no knowledge whatsoever on which down-stream tasks are using it as source of dependency. This make is possible to address the issue of locality of modification: if a user wants to modify a task dependency this is done locally at the level of the concerned task.

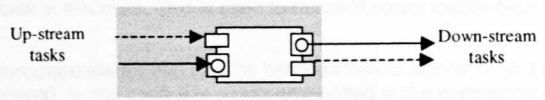


Figure 3.2: Domain of a task specification

Inputs

A workflow task has the possibility to have one or more input sets (represented in figure 3.3 by the boxes in light grey).

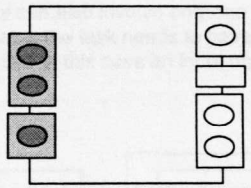


Figure 3.3: Inputs of a task

These sets are alternative and have some input objects associated to them. In figure 3.3, these input objects are represented by dark grey ovals. The first input set has two associated input objects and the second input set has just one input object. Each of these input objects has a list of references on other input/output objects that can be used as alternative input sources. In order to start, a task needs to have the totality of the input objects of one of its input set available. An input object becomes available when one of its input alternatives is available. In the event that several input sets become available, the first one listed is chosen.

Outputs

A workflow task has the possibility to have one or more output sets (represented in figure 3.4 by the boxes in light grey). These sets also referred as output states or outcomes are alternative and have some output objects associated to them.

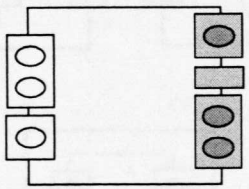


Figure 3.4: Outputs of a task

There are four different types of output sets:

- *Final outcome*: the normal output for a task.
- *Abort outcome*: a final outcome for atomic tasks, it only allows output objects of type error corresponding to some error codes (not yet supported by the underlying system)
- *Repeat outcome*: a special output for loop tasks allowing the outputs to be fed as inputs for the next iteration of the loop.

- *Mark outcome*: a special output for non-atomic tasks allowing publishing partial results. This is an intermediate outcome, and a task does not terminate whenever it reaches such an outcome. (not supported by the underlying system)

In figure 3.4, these output objects are represented by dark grey ovals. The first output set has one associated output object, the second one none and the third output set has two output objects. Once started, a task has to end up in one of its output sets and the output objects associated to the chose output set become available to all tasks having some data flow dependencies involving them as source. In the event that several outcomes of a compound task become available, the first one listed is chosen.

Dependencies

There are two types of dependencies considered: data-flow dependencies and temporal dependencies. Data flow dependencies are dependencies where an input/output object reference on an object from task A (called *source object*) is given as alternative to an input/output object of another task B (called *destination object*). The meaning of a data flow dependency is that the destination object is allowed to use the source object as alternative. In other words, the destination object becomes available as soon as the source object is itself available.

There are seven types of possible couples of source-destination objects depicted in figure 3.5. In this figure, we have chosen as naming convention to call the object source S and the object destination D.

(a) The obvious data flow dependency is a dependency between two peer tasks A and B (Peer tasks are task embedded in the same task). S is in this case an output object of task A, and D is an input object of task B. This shows that task B needs to use some of the results of task A.

(b) Another form of data-flow dependency between two peer tasks A and B is depicted in figure 3.5 b. S is in this case an input object of task A, while D is an input object of task B. This can be used when a task (B) needs to use the same input object as the input object from another task (A).

(c) This time task B is task A's parent, the object source is an input object of task A, and the D is an output object of task B. It has to be noticed that task B has to be a compound task in this case. This is used to transmit some results back to the parent task.

(d) This type of dependency is similar to the previous one, but this time the object source is an output object of task A. It has to be noticed that task B has to be a compound task in this case. This is used to transmit some results back to the parent task.

(e) A data flow dependency can also be between task B and its parent task A (embedding task). In this case B is a task embedded in the compound task A, S is an input object of task A and D is an input object of task B. This is used to transmit some object references received by the parent task.

(f) In this case A and B are again referring to the same task. In this case, it has to be a compound task, which has D as one of its output object and C as one of its input object. This can be used for instance to return an input object as output object in the event of an abnormal execution of the task.

(g) A data flow dependency can also involve only one task (tasks A and B refers to the same task). In this case, the task needs to have a repeat outcome that can be used as source object. D is in this case an input object of the task. This is used to simulate loops.

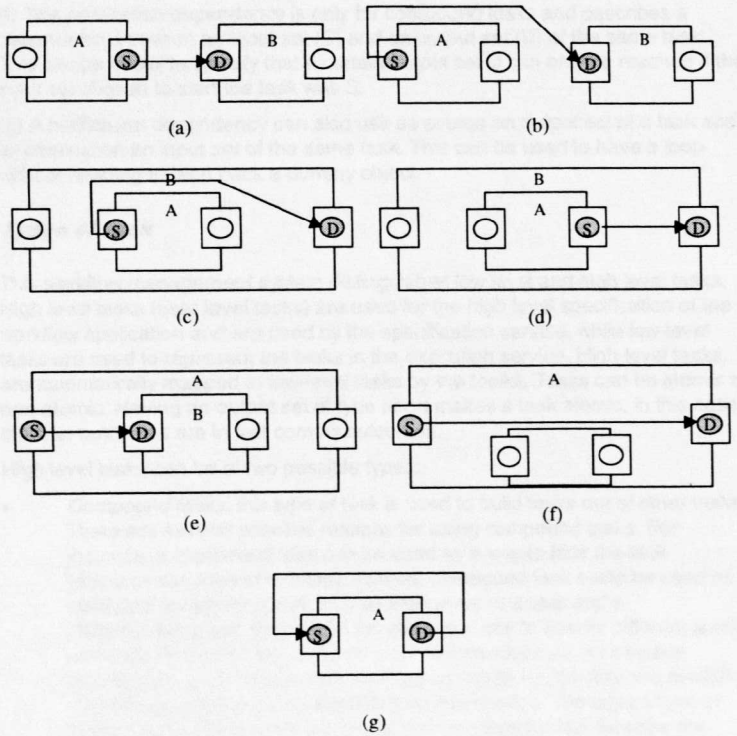


Figure 3.5: Types of data-flow dependencies

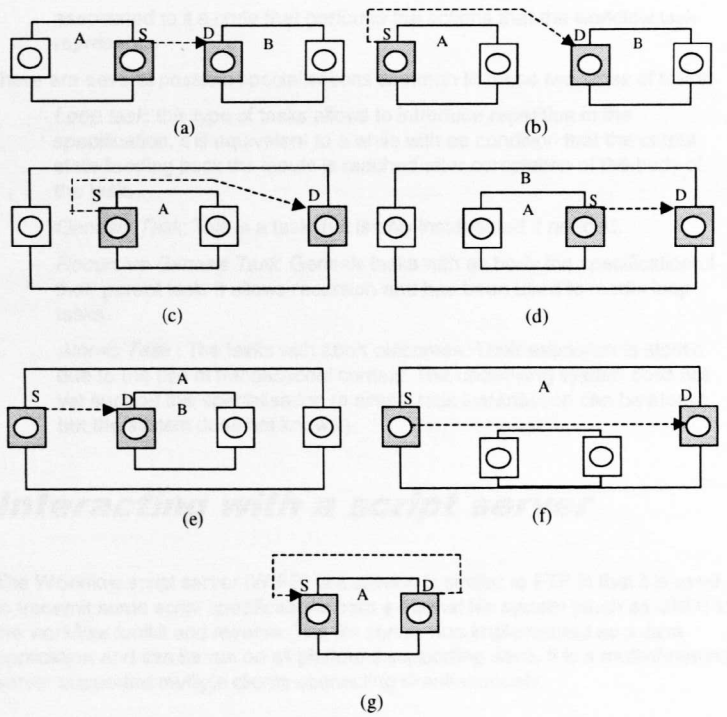


Figure 3.6: Types of notification dependencies

There are also seven different types of notification dependencies, depicted in figure 3.6. Similar conventions are used as the one used for data dependencies. A is the task with the source set and B the task with the destination set. The sets were named respectively S and D.

(a) The notification dependency with as source an output set belonging to task A that is a peer of task B. D is a destination input set. This could be used for instance to enforce that task B will only start after completion of task A in a certain state.

(b) A notification dependency can also use as source an input set from task A and as destination an input set from peer task B. This could be used to specify that B can only start after A starts in a certain state.

(c) A notification dependency can use as source an input set of task A and as destination an output set of its parent task B. This could be used for instance to

state that if task A starts with an "abnormal" input set, then its parent task B should reach a certain outcome.

(d) A notification dependency can use as source an output set of task A and as destination an output set of its parent task B. This could be used for instance to state that if task A aborts then its parent task B should reach a certain outcome.

(e) A notification dependency can use as source an input set of a compound task A and as destination object an input set of one of the tasks embedded in A. This can be used to enforce that a task is only started if its parent task A was started with the particular input set S,

(f) This notification dependency is only for compound tasks and describes a dependency between an input set (S) and an output set (D) of the same task. This can be useful to specify that a certain output set D can only be reached if the input set chosen to start the task was S.

(g) A notification dependency can also use as source an output set of a task and as destination an input set of the same task. This can be used to have a loop without needing to feed back a dummy object.

3.1.3.2 *Types of task*

This workflow management system distinguishes low level and high level tasks. High level tasks (user level tasks) are used for the high level specification of the workflow application and are used by the specification service, while low-level tasks are used to represent the tasks in the execution service. High level tasks are automatically mapped to low-level tasks by the toolkit. Tasks can be atomic or non-atomic. Having an output set of type abort makes a task atomic. In this case, the final outcomes are in fact commit outcomes.

High level tasks can be of two possible types:

- *Compound tasks*: this type of task is used to build tasks out of other tasks. There are several possible reasons for using compound tasks. For instance, a compound task can be used as a way to hide the fault tolerance associated to a task. Indeed, compound task could be used as a black box to specify a task and an alternative or a task and a compensating task. It can also be used as a way to specify different levels of details. For instance, using an e.commerce example, a company director can model a business process as two tasks, the first one providing a service and the second one billing for this service. The department or person in charge of completing this task can then further describe the tasks that he has identified. For instance the finance department will describe how the billing of the service is done by decomposing the billing task as a set of tasks responsible for simpler activities. This allows having a good support for modularity.
- *Basic tasks*: this type of task is the basic unit of work of the system. It can no longer be divided in a set of workflow tasks. This type of tasks has

associated to it a code that performs the actions that the workflow task represents.

There are several possible specialisations common to these two types of tasks:

- *Loop task*: this type of tasks allows to introduce repetition in the specification, it is equivalent to a while with as condition that the output state feeding back the inputs is reached after completion of the body of the task.
- *Genesis Task*: This is a task that is only instantiated if needed.
- *Recursive Genesis Task*: Genesis tasks with as body the specification of their parent task. It allows recursion and has been used to model loop tasks.
- *Atomic Task* : The tasks with abort outcomes. Their execution is atomic due to the use of transactional context. The underlying system does not yet support this specialisation (a simple task instantiation can be atomic but the system does not know it).

3.2 Interacting with a script server

The Workflow script server (WfFS) is a service is similar to FTP in that it is used to transmit some script specifications from a normal file system (such as UNIX) to the workflow toolkit and reverse. The file server was implemented as a Java application and can be run on all platforms supporting Java. It is a multi-threaded server supported multiple clients connecting simultaneously.

3.2.1 Connecting to a workflow file server

As a user, the first thing that you probably want to do is change the default file server to be used. This is achieved by changing the location of the file server using the menu option "Options/File Server". The form depicted in figure 3.7 then appears and you can specify the machine and port where your preferred file server is running. Currently such a server is available at (kellah.ncl.ac.uk, 8081). For security reasons, the Toolkit can only use WfFS located on the localhost and a couple of machines from the domain ncl.ac.uk when used via an applet. This is due to the CGI gateway that has been coded to only forward requests if they are aimed at this subset of machines.

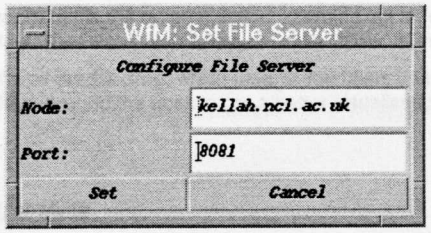


Figure 3.7: Form used to set the location of the workflow file server used.

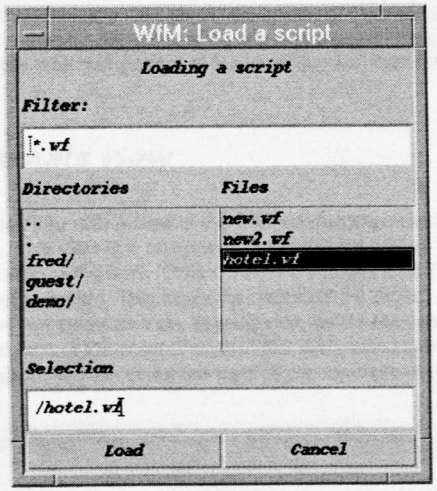


Figure 3.8: Form used to load a script from the WfFS

Once a valid WfFS has been declared, you can load or save your specification as a script using the file server. The aim of the file servers is to enable the user to save its work as a workflow script wherever wanted.

3.2.2 Management via the Toolkit

After selecting the menu option, "File/Load", the form depicted in figure 3.8 appears and the user can choose to load whatever script is needed.

To save a file in the WfFS, the option "File/Save As..." is used. A form similar to the one in figure 3.8 appears and the user can specify under which name the newly created script should be known as well of the location where it should be stored.

3.2.3 Loading a script

It is possible for the WfGUI to retrieve a script from a Workflow script server. The specification will then be pre-processed letting you know whether you have some errors in your script and their location. If the errors were bad enough for the interpreter to fail to load the specification, they are listed and the user can try to fix then directly. This verification of the syntax is weak as it was thought that designers might want to load a non-correct specification and then use the tools provided with the GUI to correct it.

While being interpreted, the textual specification is converted into a graphical representation.

- The missing Object Classes are created as Object Classes specific to the user and are afterwards available to the user.
- The Task Classes are also created directly while reading them. Input object and output objects are registered with their Object Class to make it easier to map the invalid ones later on.
- The interesting problem is to load the task definitions. First the associated Task Class is read. If it was an unknown Task Class, then an error is generated and the interpreter stops trying to load the specification. Otherwise the task is generated with its associated Task Input Sets, Task Input Objects, Task Output Sets and Task Output Objects. The task is also registered with the Task Class so that the Toolkit prevents the deletion of referenced Task Classes later on. The specification of the task is then carried out with the generation of the data and temporal dependencies associated to the Task Input Sets and Task Outputs Sets as well as on their respective associated Task Objects. It has to be noticed that an error is generated if the script tries to specify the dependency on a Task Object Set or and Task Object not part of the associated Task Class. This error triggers the end of the interpretation of the specification. Each data and temporal dependencies is translated as a reference on dummy Task Input Sets, Task Input Objects, Task Output Sets and Task Output Objects as appropriate. There is no check at this stage of the validity of these dependencies. Dangling dependencies can be created and will appear as inconsistencies when the task is checked.

3.3 Navigating into a Workflow application Specification.

A navigation system is also available that let the users zoom in and out of your specification. Zooming in a compound task let you see its component tasks, while zooming in a simple task display its task class as well as all the dependencies it is

involved in. The zoom-out is the reverse action: zooming out let you see the embedding workflow. These commands are all under "View" in the menu bar.

The interface is of type drag & drop, which means that users can click on task icons and move them around the desk to get a better visualisation of the workflow application.

3.4 Task's views

Clicking twice on the icon on a task gives you a view of the interactions of the task with the external world, while zooming into a compound task give you a graphical view of the task component of the workflow and their inter-dependencies.

3.4.1 Task's internals view

Figure 3.9, provides you with a view of the internal decomposition of the workflow BusinessProcess. This view is a *compact view* where the option "Level of details for Compound Tasks" of the form "Options/View Dependencies Options" (figure 3.11) has been set to limited. This hides the details of the tasks' interdependencies has been chosen. In particular, we do not see in this view which input/output sets of the components of this compound task are being used by the dependencies. Normal views are used in the example section (figures 10.2, 10.3, 10.4)

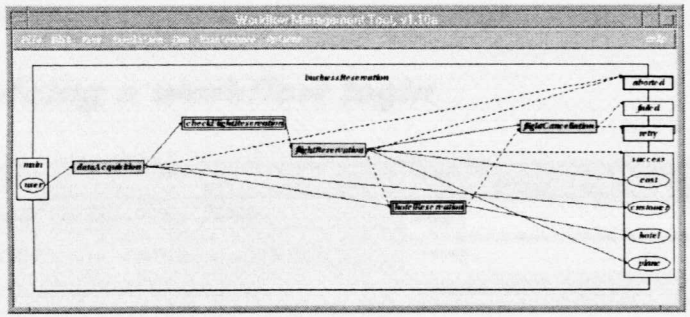


Figure 3.9: Zooming into a compound task

3.4.2 Task's Interaction View

Figures 3.10 and 3.12 gives you a view of a task's interactions. As can be seen on figure 3.10, the graphical representation can be quite complex. As a result, a tool has been provided to configure how much detail users get.

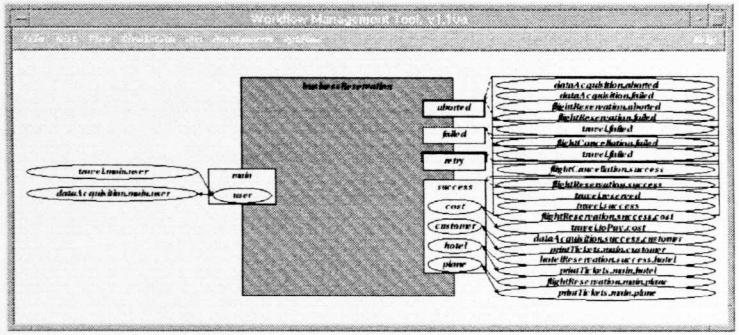


Figure 3.10: Task's view of the world

Here against, you can decide what you want to see and what should be hidden using the options hide/show incoming and outgoing dependencies via the "options/View Dependencies Options" form depicted on figure 3.11.

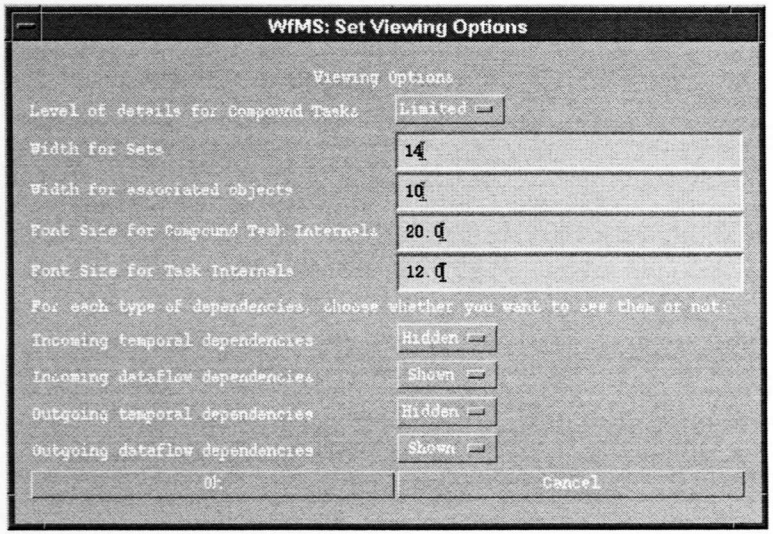


Figure 3.11: Configuring which dependencies are shown by the WfGUI

Having chosen to hide the temporal dependencies, figure 3.11 is simplified and the result is shown on figure 3.12.

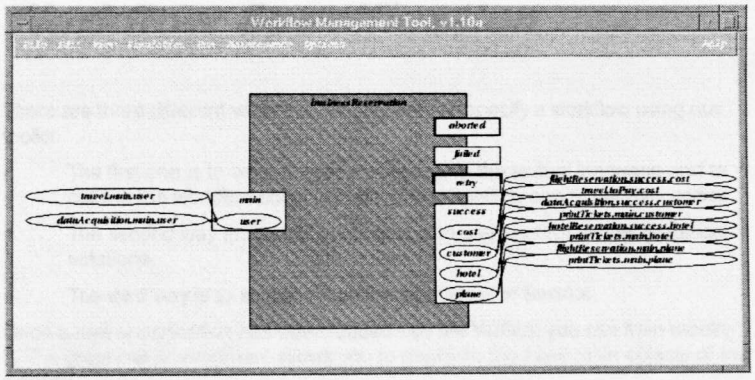


Figure 3.12: Simplified task's view of the world

3.4.3 Compound Genesis Tasks

It has to be noticed that the navigation in compound genesis tasks is similar to the navigation in basic tasks as long as they are not executing. As soon as they are / have been activated, then they are expanded and the navigation is identical to navigation with compound tasks (e.g. the Task's internals view become available).

3.5 Adding a workflow login

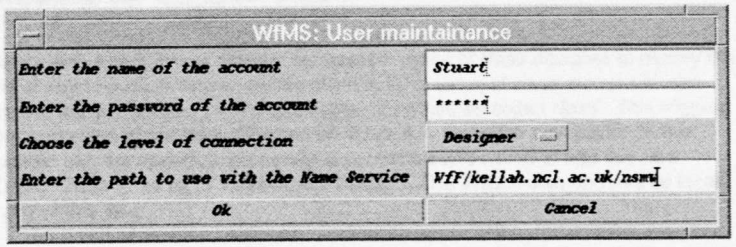


Figure 3.13: Form to manage user accounts.

Users of class Maintainers have the ability to create logins for workflow toolkit users using the form depicted on figure 3.13. In demonstration versions, the path to use should always be set to the name prefix that is used in your

openflow.properties file. The form is created using the menu option "Maintenance/Add an account"

4. Workflow Specification

There are three different ways that can be used to specify a workflow using our toolkit.

- The first one is to write directly a script using the textual language and to put it in the workflow script repository managed by one of our files server.
- The second way to specify a workflow is to use the GUI and its graphical notations.
- The third way is to import it from the Specification Service.

Once a new specification has been loaded into the WfGUI, you can then modify it. The graphical environment allows you to deal with the three main objects of our system:

- Object Class
- Task Class
- Task

4.1 Object classes

When the WfGUI applet is started, it first contact a naming server to get the Specification Service and then recover from a list of Object Classes known by the system.

4.1.1 Adding an Object Class

At the time being, it was chosen not to allow users of class designer to modify the list of object classes known by the system. Only users of class maintainer can modify this list, by using the menu option "Edit/Add an object class". This triggers the apparition of the form depicted on figure 4.1. Once the user clicks on the button "ok", the specification server is contacted and asked to add this new Object Class to its list of valid Object Classes. The WfGUI also updates its local copy of this list.

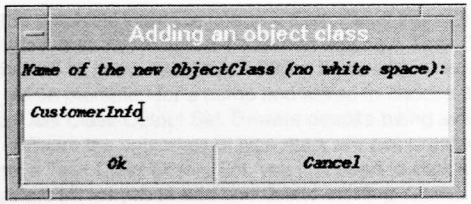


Figure 4.1: Form to add an Object Class

In the current version of the Toolkit, it is not possible to remove Object Classes once they have been added. This choice was made as some users may be using some of these Object Classes without the Specification Server knowledge.

4.1.2 Mapping of an invalid Object Class

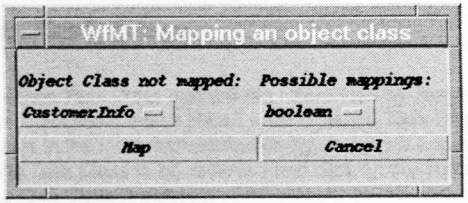


Figure 4.2: Mapping of an invalid Object Class

As the interpreter of workflow script is weak, it is still possible for designers to introduce some invalid Object Classes into the WfGui. It is possible to map invalid Object Classes to valid ones by using the form depicted on figure 4.2. It can be reached by choosing the menu option "Edit/ Map an invalid object class".

When an invalid Object Class is mapped to a valid Object Class, all Task Class Objects (and as a result Task Objects) using the invalid Object Class get the new chosen Object Class. Once this is completed, the invalid Object Class is no longer referenced and is removed from the system.

4.2 Task classes

It is possible to add, delete or edit Task Classes using the WfGUI or merge two Task Classes together.

4.2.1 Adding a Task Class

The addition of a Task Class is done via the form represented in figure 4.3. This form can be called by choosing the menu option "Edit/Add a task class". The user can then choose the name of the new Task Class, as well as create some Task Class Input Sets, Task Class Output Sets and associated Task Class Input Objects and Task Class Output Objects, It's possible to add or remove whatever is needed.

As can be seen on figure 4.3, the labels of the buttons do change accordingly of whether the Task Class Object Sets are selected or not. When it's not selected, you can add and delete Task Class Object Sets. In order to select a Task Class Object Set, you just need to click on its name. The associated list of objects appears and it is possible to add or delete them.

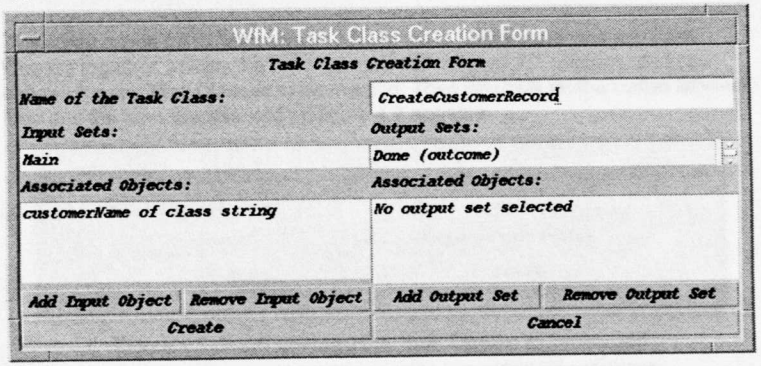


Figure 4.3: Form for Task Class addition

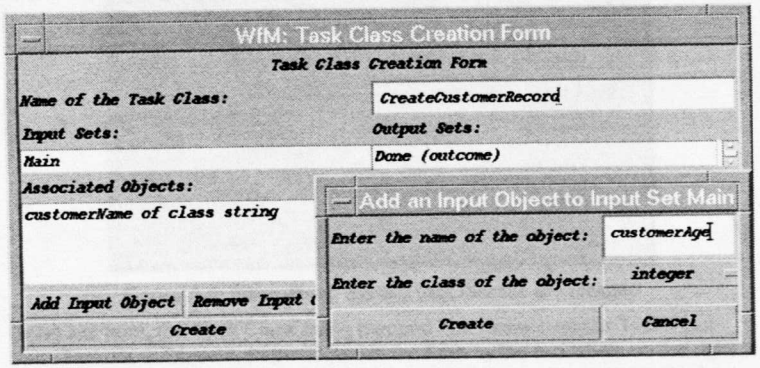


Figure 4.4: Form for addition of a Task Class Object to a Task Class

An example of the graphical interface to add a Task Class Input Object to Task Class Input Set main is shown in figure 4.4. When adding a Task Class Object, you must specify both its name and Object Class. If you wish to add a Task Class Object Set you will be prompted for a name and asked to choose the type of outcome if it is a Task Class Output Set. Beware despite being available in some versions of the software the outcomes of type *mark* are **not created** at the engine level. To deselect a Task Class Object Set, you just need to click a second time on it, and the system allows you to add and delete existing

In order to create your Task Class, the user just need to click on the button labelled "Create" and the system tries to create it. It will generate an error message if one of the following conditions is fulfilled:

- Invalid name or name already used for another Task Class.
- No Task Input Set or no Task Output Set
- Outcomes of type mark and abort both present

4.2.2 Deleting a Task Class

It is possible to delete Task Class that are not referenced by Tasks. By selecting the menu option "Edit/Delete a task class", a list of all the Task Classes that can be deleted appears in the form represented on figure 4.5 will appear. You just need to select the Task Class to be removed and click on the button labeled "Delete" and the Task Class is deleted.

To delete a Task Class in use, you need to remove or re-map the tasks referring to it.

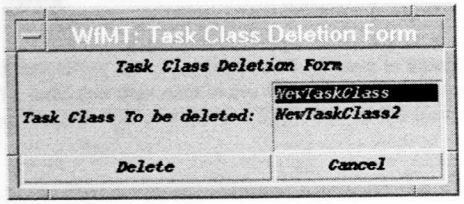


Figure 4.5: Form for Task Class deletion

4.2.3 Editing a Task Class

Editing a Task Class consists in taking a copy of an existing Task Class, and uses it as starting point to create a new Task Class. The selection form of figure 4.6 will appear if you select the menu option "Edit/Edit a task class". By selecting one of the tasks, you will get the form of figure 4.3 appear with as initial structure the structure of the Task Class just chosen. It also gets an extra button to check

the validity of the specification. When editing a Task Class, a copy of the Task Class is used, to let users undo their modifications if they wish to do so.

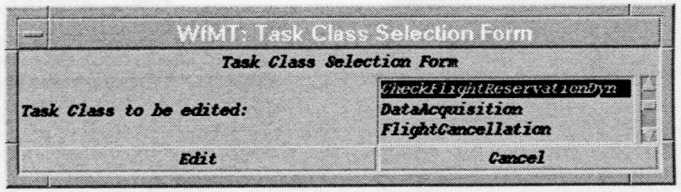


Figure 4.6: Form to choose the Task Class to be edited

4.2.4 Mapping a task class

This menu option ("Edit/Map an invalid task class") allows merging two Task Classes together. Initially the form represented in figure 4.7 appears, and you choose the two Task Classes to be merged. Then you click on the button labelled Map and the form represented on figure 4.8 appears.

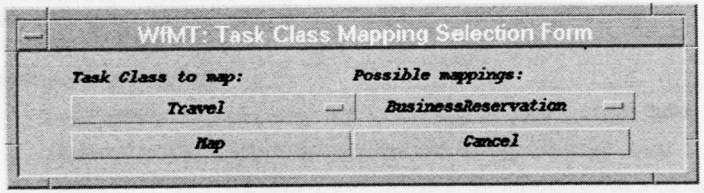


Figure 4.7: Form selecting the Task Classes to be merged

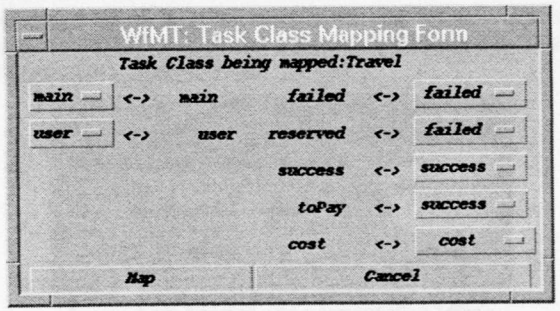


Figure 4.8: Form where the two task classes are merged

Using this form, the Task Class being mapped has respectively its Task Input Sets, Task Input Objects, (represented in the form by the two labels on the right of the arrows) Task Output Sets and Task Output Objects (represented in the form by the five labels on the left of the arrows) mapped to respectively the Task Input

Sets, Task Input Objects, and the Task Output Sets and Task Output Objects of the task it is merging with (they appears as a Choice represented in the picture by a rectangle with a label and another small rectangle included).

Each task registered with this mapped Task Class get notified of the changes and creates some new Task Input Sets, Task Input Objects, Task Output Sets and Task Output Objects that are substituted to the old ones.

4.3 Tasks

Tasks can be added, deleted, edited using the WfGUI. A Task's Task Class can also be mapped to another Task Class.

There are two different types of tasks: basic tasks that can not be sub-divided in smaller workflow tasks and compound tasks that can also be seen as sub-workflows. They are represented on the picture above as rectangles with respectively single line and double line borders. If you wish to have a late instantiation (i.e. that the task is only created if it is needed), then you can specify it as genesis. Genesis tasks appear with rounded edges.

In our model, a task is fully defined by its name, its Task Class and its dependencies on the other tasks. In the case of a compound task (sub-workflow), you also have to specify the dependencies that its own outputs have on the constituent tasks. The temporal dependencies are shown on the picture above as dotted lines while the dataflow dependencies are shown as plain lines.

4.3.1 Adding a task

In order to add a task to the specification, users just have to select the menu option "Edit/Add a task" and then click in the canvas on the location where they would like it to be created. This prompts the apparition of the form represented in figure 4.9.

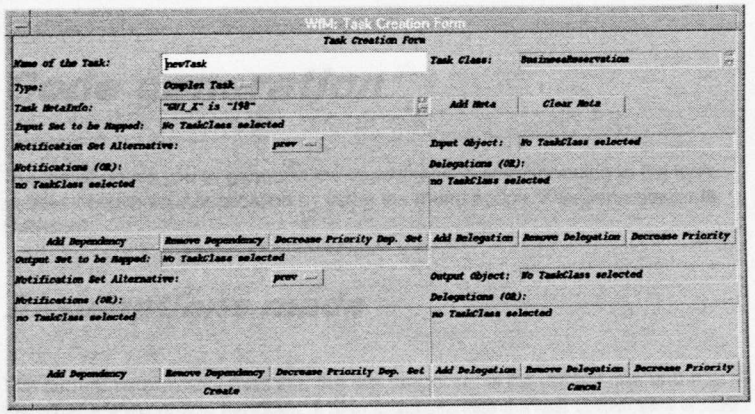


Figure 4.9: Form for the creation of a Task

Using this form, you can set the task name. Then you have to choose the Task Class associated to the task being created as well as its type (compound task or basic task). Users can also add some instantiation criteria using the form represented in figure 4.10.

Meaningful keys includes:

- **TaskFactory** to specify which task factory is to be used to create tasks
- **TaskImpl** to specify which task type the task factory should be using
- **Type** with as value **Genesis** to force a task to be instantiated as a genesis task (i.e. lazy instantiation just when it is really needed)
- **GenesisDef** with as value the name of the Task definition to be used. If it's its parent's name, then it creates a Recursive Genesis task. This has to be used in conjunction with ("Type", "Genesis")

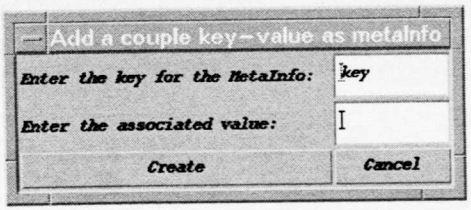


Figure 4.10: Form to add a instantiation criteria

Once this is done, you can start adding some dependencies and delegations on up-stream tasks involving the Task Input Objects and Task Input Sets (using the form depicted on figure 4.11 to add a delegation).

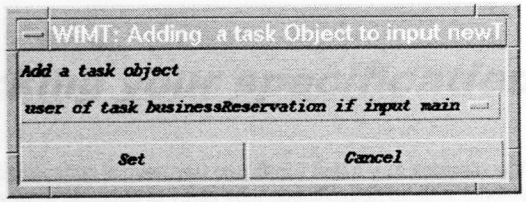


Figure 4.11: Form to add a dataflow dependency

If you are creating a compound task, you can also add some dependencies and delegations between inputs and outputs of your task. You can also change the priority of the delegations and dependency sets. Decreasing the priority of the item selected does this. It has to be noticed that an item with the lowest priority seeing its priority decrease will actually gets the highest priority. This allows increasing the priority of an element.

4.3.2 Deleting a task

It is really easy to delete a task using the WfGUI, Users just have to choose the menu option "Edit/Delete task" and click on the icon of the task that they want to delete, This task get removed as well as all dependencies in which it was involved.

4.3.3 Editing a task

It is a melting pot of the two previous features. Users choose the menu option "Edit/Edit a task" and click on the task they want to edit. A form similar to the one of figure 4.9 appears initialised with the current delegations, dependencies... of that task. When editing the task, a copy of the task is used to let users undo their modifications if they wish to do so.

4.3.4 Mapping the Task Class of a task

It is again re-using forms that have been providing to select a Task Class and to merge two Task Classes (figures 4.6 and 4.8). Once again, to map a task's Task Class, users have to choose the menu option "Edit/Change Task's Task Class" and click on the task concerned, and these forms appear.

5. Code generation

The toolkit allows you to generate the code skeleton corresponding to the task classes used in your application by using the menu option "File/Generate code Skeleton".

5.1 Assumptions made

The current version assumes that the first output set is the default one and that all the other are exception handlers. It also assumes that the last output sets is used in case of system exceptions.

5.2 Parameters

Right now you can only chose where the code should be generated and what the package name should be.

6. Checking your specification

When the specification is over, you can check that it was correctly written. Three main options were provided to verify your specification on-line. Checking the Object Classes, the Task Classes and the Tasks.

6.1 Checking the Object Classes

This makes sure that all the Object Classes were known by the system. A list of the Object Classes used in the specification appears and let the designers see which ones are not recognised. The result can be seen on figure 6.1. It has to be noticed that the incorrect information have a '*' in front of their name. Noticed that CustomerInfo has been added by a maintainer to the system Object Classes and as a result is no longer part of the invalid ones.

If some new Object Classes were found, designers have the opportunity to map them to existing Object Classes, while maintainers can also add them as System Object Classes.

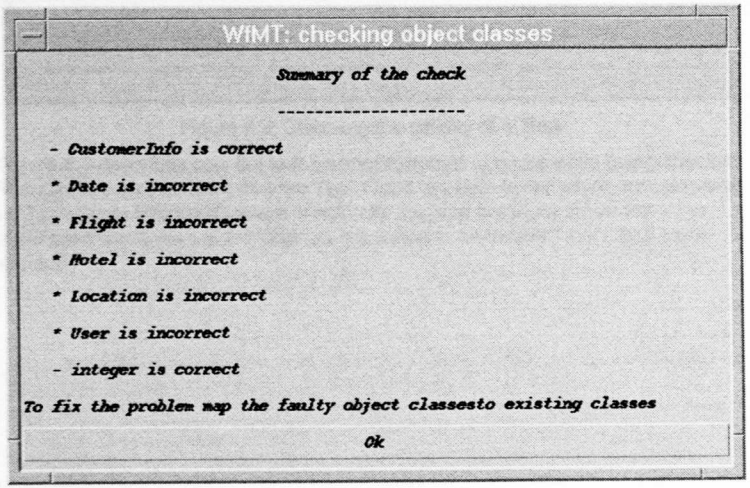


Figure 6.1: Checking the validity of the Object Classes

6.2 Checking the Task Classes

After choosing the menu option "Run/Check task classes", a list of incorrect task classes (similar to the list presented in figure 6.1) will be displayed. Then the edition of the incorrect Task Classes will let you see the incorrect information subsequently displayed in red, while the correct ones will be displayed in green. It has to be noticed that the incorrect information have a '*' in front of their name.

A task class will be tagged as incorrect if:

- Once or more of its Task ClassObject has an invalid Object Class.
- It doesn't have at least one Task Class Input Set and one Task Class Output Set
- There is both a mark and an abort Task Class Output Set.

It is always possible to check the correctness of your task class by pressing the button "Check" on the edition forms. In figure 6.2, the Task Class Input Object "bill of class Bill" is tagged as incorrect, as the Object Class Bill was not defined as a System Object Class.

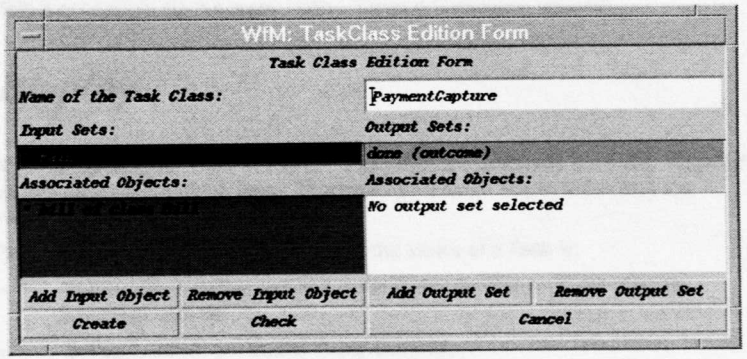


Figure 6.2: Checking the validity of the Task Classes

6.3 Checking the Tasks

Similarly you can check the correctness of your task and a list of incorrect tasks will be generated.

A task being edited in mode check (click on button check while editing to toggle this mode) appears with the incorrect part in red and the correct ones in green. It has to be noticed that an incorrect information within a set of dependencies make the whole set appears in red. The component tagged as incorrect will have a star

('*') in front of its name. A star followed by the word "Loop" means that the compound task concerned contains an unwelcome loop. The check button is also provided on the forms that are used to edit the tasks, and can be used at any time to re-check the correctness of the task being modified.

A task is tagged incorrect if:

- Its Task Class is incorrect,
- Some of the dependencies with this task as target are on non-existent objects.
- Some data-flow dependencies with this task as target are involving two Task Objects of different classes.
- It has the same name as its parent or a peer task
- A repeat outcome is used by a task different from itself
- It is a compound task including an unwelcome loop (unwelcome loops are discovered by creating a dependency graph of the component tasks)

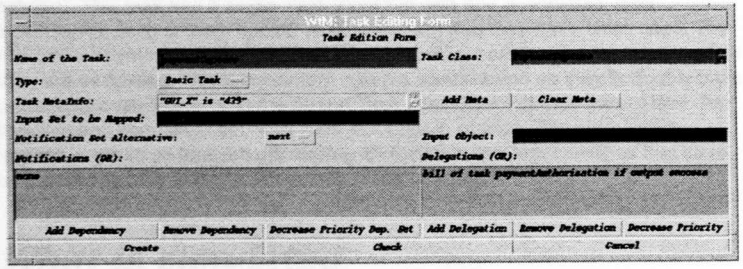


Figure 6.3: Checking the validity of a Task

Figure 6.3 describes how the task paymentCapture appears while being checked. We can see on this figure that the Task Class appears in red which indicates that its Task Class PaymentCapture is not valid and that the input object Bill of the input set main is not valid neither (as it is linked to an invalid Task Class Input Object)

7. Workflow Simulation

Using the simulation features, you can start a simulation, do a step-by-step execution, stop or reset the simulation.

7.1 Options

There you also have two options:

- the first option is a random simulation where the computer randomly chooses an output state (outcome) when a basic task is executing.
- the second option lets the user decides on the outcome of the task execution.

This is accessible via the menu option "Options/Simulation options".

7.2 Display

A colour scheme let users see which dependencies and tasks are being triggered as well as the state of the tasks. This provides a quick way to check that the workflow is executing as forecasted.

The colour scheme chosen to represent the states of a Task is:

- Waiting (green): the task has some dependencies on it, but may be executed later on
- Set-up (yellow): the task is being modified
- Active (orange): the task is executing
- Completed (red): the task has been executed
- Discarded (grey): the task has been discarded, as some dependencies could not be fulfilled.

The relevant sub-set (waiting, set-up, completed and discarded) of this colour scheme is used for the Task Objects.

The tasks can be in several states with the temporal and dataflow dependencies depicted on the figure below.

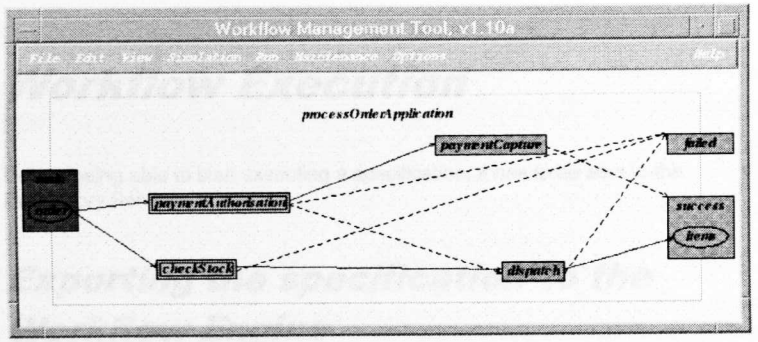


Figure 7.1: Simulation of the execution of a workflow application

In the example depicted in figure 7.1, the user has zoomed into a compound task while it was executing (the rectangle showing the compound Task boundary is orange). This task had a single Task Input Set with one associated Task InputObject. Both of them appear in red meaning that they have been used. The two tasks PaymentAuthorisation and CheckStock appear in orange, meaning that they are executing. This is consistent with the specification as they both only have a dataflow dependency and it is on the Task InputObject of their parent task. As they haven't completed yet, they were coloured in orange. The other tasks appear in green as they are still waiting for their inputs. Dependencies that have been used appear in red in the detailed view of a task.

7.3 What is simulated

Each task and each dependency have a run-time status indicating their current situation. When the simulation is started, the workflow application gets one of its input sources fulfilled. The way to decide which one depends on the simulation options. If it is automatic, a random value is generated. The result indicates the Task Input Set that is activated.

Afterwards, the system goes one step at the time. Each task still waiting checks whether one of its input sets requirements has been fulfilled. Each Task Input Set checks in turn which dependencies have been fulfilled. If all its Task InputObjects have been marked as fulfilled (marked as executed), the corresponding Task Input Set checks its temporal dependencies and if they are all fulfilled, this Task Input Set is itself fulfilled (marked as executed). If the task finds an input set marked as executing (available), then it is chosen and marked as completed (chosen), while the other input sets (if any) are discarded.

If a basic task is in the state executing, then a Task Output Set is chosen randomly or by the user depending of the options. Once it is done the chosen Task Output Set as well as its associated Task Output Object are marked as

chosen (executing). At the next step, the Task Output Set and its associated Task Output Objects are marked as completed, while all the other Task Output Sets and Task Output Objects are marked as discarded.

If it is a compound task, then it checks its Task Output Objects. When they are all fulfilled and there no temporal dependencies on their associated Task Output Set, this Task Output Set is set as fulfilled. When the task next steps, it will find out about the fulfilled dependencies and set the Task Output Set and associated Task Output Objects as chosen (completed) while setting their alternatives to discarded.

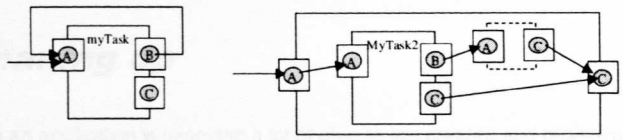
8. Workflow Execution

Before being able to start executing a specification, it has to be sent to the Repository Service.

8.1 Exporting the specification to the Workflow Engine

This is carried out automatically if you start a specification that has not been exported. The pre-requisite for storage in the repository service being that the specification needs to be correct. The specification is as a result checked before being exported.

Once the specification has passed the consistency tests, it is the repository service. The main difference between the specification model of the WfGUI and the repository service are the outcomes of type mark that are not yet supported as well as those of type repeat that are not directly understood. As a result, the repeat loops have to be converted into a different representation at run time using genesis tasks. On figure 8.1a, the high level representation of a repeat task is given. The representation of the same task at run-time is given on figure 8.2b. It is modelled as a compound task embedding a task similar to myTask and a genesis task. The main difference myTask and myTask2 is that the input alternatives that were coming from a repeat outcome disappear. The dependencies that were using a repeat outcome as source are now dependencies on a genesis task with the same inputs as myTask had. This genesis task uses as associated task the compound task that gathers myTask2 and the genesis task.



a) High level representation b) Low level representation

Figure 8.1: Run time representation of loop tasks

8.2 Feeding the initial inputs to a workflow application

Once the low-level specification has been successfully created (e.g. the specification is stored in the Repository Service), it is possible to start the execution of an instance of the workflow application specified.

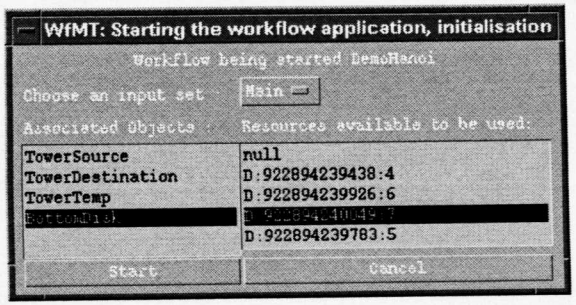


Figure 8.2: Feeding initial inputs to a Workflow Application

The way it was done is to use the menu option "Run/Start Workflow Application". It creates a form similar to the one displayed on figure 8.2. This form lets the user select which input set should be used to start the workflow as well as which initial resources should be used by its associated input objects. Only resources registered with the Name Service as resources of the same object class as the class required can be mapped to any particular input object. By default all the resources are mapped to null.

8.3 Dynamic modifications

It is possible to modify the specification while it is executing. You can only modify tasks (including their incoming dependencies) that are still in a waiting state.

- Editing/adding/deleting a task or a set of incoming dependencies associated to a task is identical to what is done at build-time, provided that this task is in its waiting state.
- If you want to make "atomically" a set of modifications involving different tasks, you have to explicitly change the status of a common "ancestor" to setup so that the part of specification that you want to change stay "frozen" for the duration of the modifications. Once those modifications are completed, you will have to explicitly change the status of the ancestor back to waiting. Two

menu options have been provided to let you explicitly change the state of a task (Edit/Start dynamic modifications on task, Edit/End dynamic modification on task)

8.4 Display

A task can be in five states: "setup", "waiting", "active", "completed" or "discarded". Each of this states has an associated color. The default color scheme can be changed using the menu option "Options/Color scheme". The form on figure 8.3 then appears and let you choose your own colors based on their RGB code (pressing on the button labeled "Show chosen colors" change the color of the background of the line corresponding to a given state to the color that has been specified).

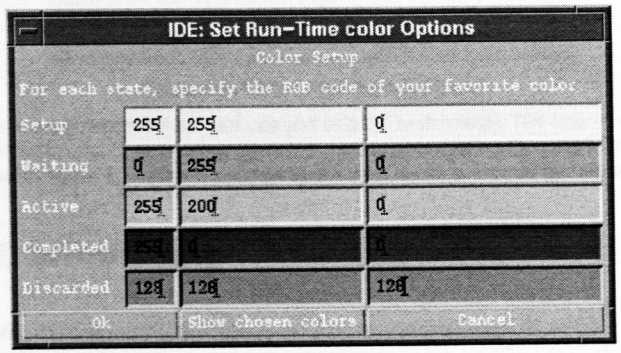


Figure 8.3: Choosing your own color scheme

8.5 Cleaning up

When an application is executed a lot of objects are created and registered with the Name Service. A menu option named "Run/Get rid of execution traces" has been provided to do some garbage collection of what is no longer required. Using this features get rid of the task controllers, task definitions and expansion of the Genesis tasks that have been created during the execution. It also deregisters these objects.

9. Workflow Monitoring

Once started, a workflow application can be monitored. This is made possible thanks to the task controllers that provide information on the state of the tasks that they are managing.

9.1 Options

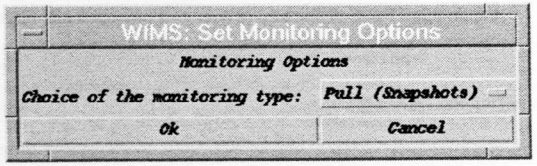


Figure 9.1: Options for the monitoring

The monitoring process itself can use pull or push technology. The user can choose his preferred technology by using "Options/Monitoring Options/". The form depicted in figure 9.1 appears prompting the user for its preferred technology.

9.2 Pull Monitoring

Choosing the pull monitoring obliges the user to use the option "Run/Take Snapshot" when he wants to see the current state of the workflow application.

9.3 Push Monitoring

If the push option is chosen, the changes of state are updated as they occur. The Task Controller contact themselves the GUI. The view is updated as soon as it is notified of changes. This option is only available in some versions of the software.

9.4 Multi-user monitoring

Using the menu option "Run/Monitor workflow application" a list of the workflow applications that can be monitored appears. You then just have to select which application you want to monitor.

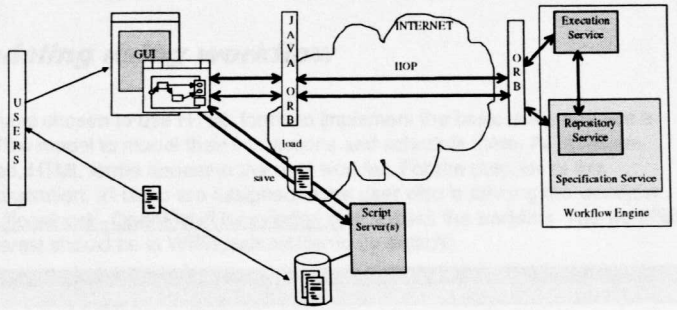


Figure 9.2: Graphical representation of the workflow system with Specification Service

As specifications stored in the Repository Service lose their high level names. This makes it difficult for other persons to monitor the workflow. As a result, a Specification Service that will keep a high level view of the tasks and in particular keep the names associated is being built on top of the Repository Service as depicted on figure 9.2. This make it possible to have several observers monitoring the progress of the same workflow.

9.5 Display

The WfGui displays the application being monitored similarly to simulated applications.

10. Examples

Two examples will be presented to give you an idea of what can be specified using OPENflow.

10.1 Form-based example: order Processing

This is a typical workflow example of an office process. We have used as base for this example the ordering process in use within the Department

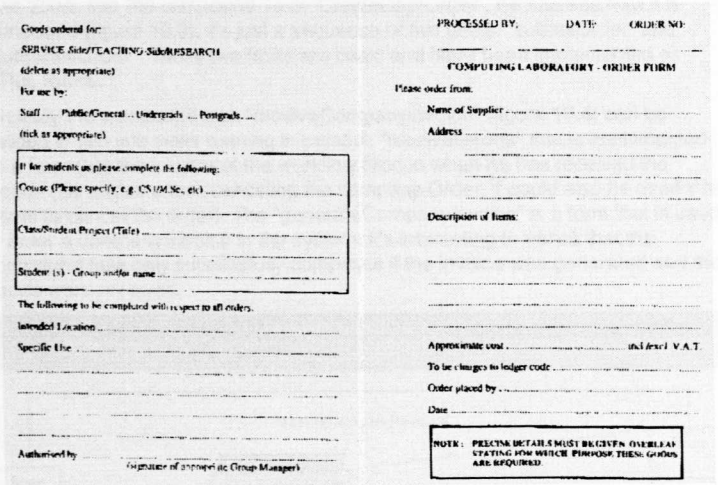


Figure 10.1: Form for Departmental order

10.1.1 Overview of the application

This process is started when somebody needs to order equipment and can be divided in four sequential steps :

- The first step is to fill in the form depicted on figure 10.1. Then this form has to be submitted to the group manager of the order initiator for authorization.

- In the second step, a company order has to be generated and authorized by the group manager before being sent to the Company.
- The next step is to receive the goods and invoice from the company.
- The last step is to send an internal invoice to the university finance office asking them to pay the company that has sent the goods. This internal invoice also has to be authorized.

10.1.2 Modeling using workflow

We have chosen to use HTML forms to implement the basic tasks and use a workflow model to model their interactions and schedule them. As tasks are started, HTML forms appear in the user worklist. For the purpose of this demonstration, all tasks are assigned to the user who is running the workflow. (http://localhost/~OpenFlow/Users/index.pl to access the worklists, The workflist of interest should be in WfF/localhost/demo by default)

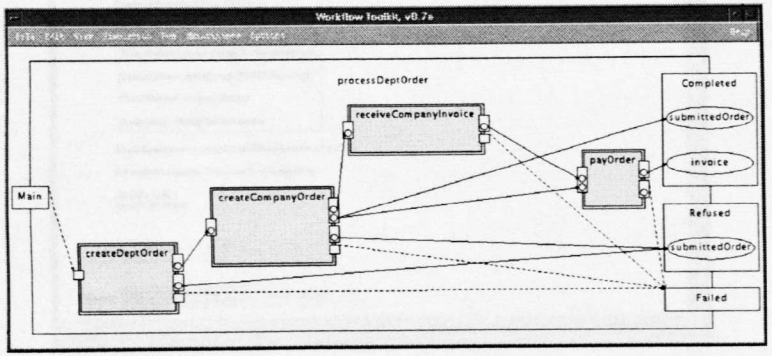


Figure 10.2: Ordering process modeled as a Workflow

The Workflow specification is shown on figure 10.2. The sequential happening of the four steps identified is the normal behavior of the workflow (everything authorized, delivered and paid. The alternative output sets (such as authorization refused) are directly mapped to some output sets of the embedding workflow in this example. It would be possible to introduce some extra tasks dealing with those abnormal outcomes (For instance an alternative company could be chosen if it does not deliver the goods...). In case of system failure, the specification just assume that if the underlying system can not cope with them, then the workflow application will end in the "Failed" output set.

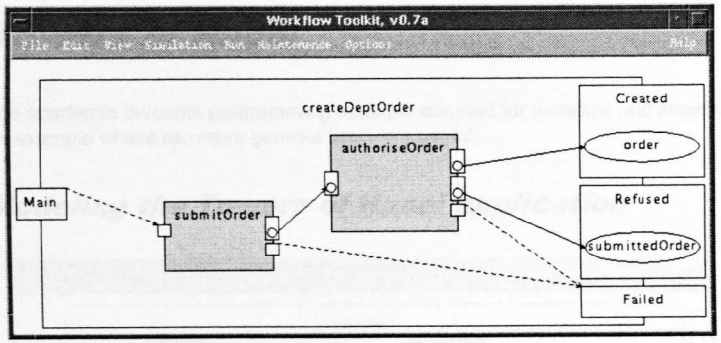


Figure 10.3: Internal details of the compound task "CreateDeptOrder"

If we zoom into the compound Task "CreateDeptOrder", we can see how it is composed (figure 10.3): it's just a sequence of two tasks: "submitOrder" and "AuthoriseOrder". These two tasks are basic and have been implemented as HTML forms.

Similarly the compound task "receiveCompanyInvoice" (figure 10.4) can be divided in two sub tasks running in parallel: "receiveGoods" that is implemented as a form that the initiator of the workflow filled in when he has received the Goods (By added a task canceling the company Order, it could also be used if he wants to cancel the order). The "generateCompanyInvoice" is a form that is used to enter a company invoice in the system. It's interesting to notice that the compound task only successfully completes if the invoice was generated and the goods were received.

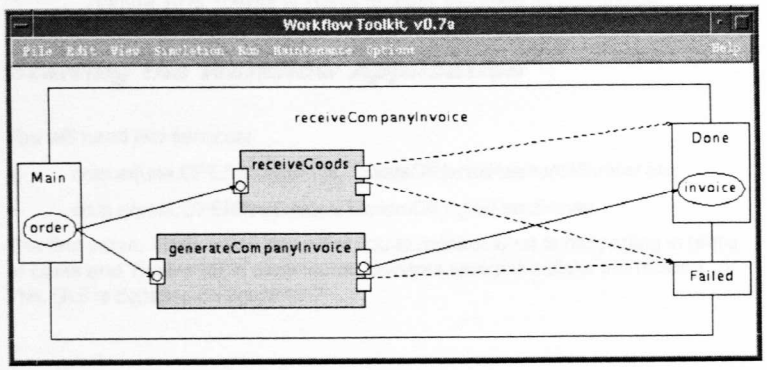


Figure 10.4: Internal details of the compound task "receiveCompanyInvoice"

The forms generated by the workflow applications are standard HTML forms. For instance a screen dump of the authoriseOrder task is presented in figure 10.5.

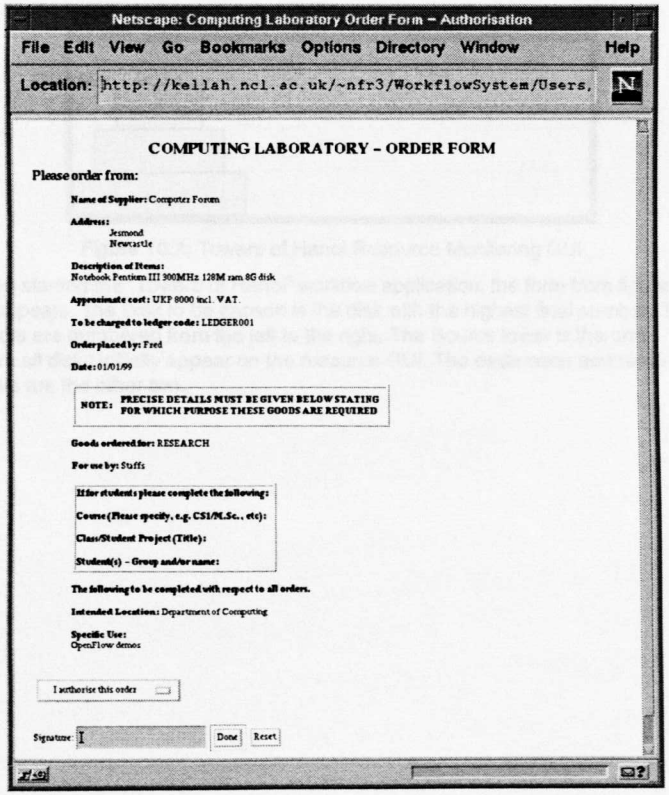


Figure 10.5: Form implementing the basic task "authoriseOrder"

For logging reasons, the reference of the CORBA "Order" object that is created or modified is always returned when you submit the form as well as a dump of what you have authorized. It could eventually be printed/archived.

10.1.3 Run-time requirements

The current implementation of this example adds a requirement :

- Web server with PERL cgi-script support

The task and resource factories for this example are started by default.

10.2 Towers of Hanoi

The academia favourite programming example adapted for workflow and showing an example where recursive genesis tasks are useful.

10.2.1 Modeling the Towers of Hanoi application

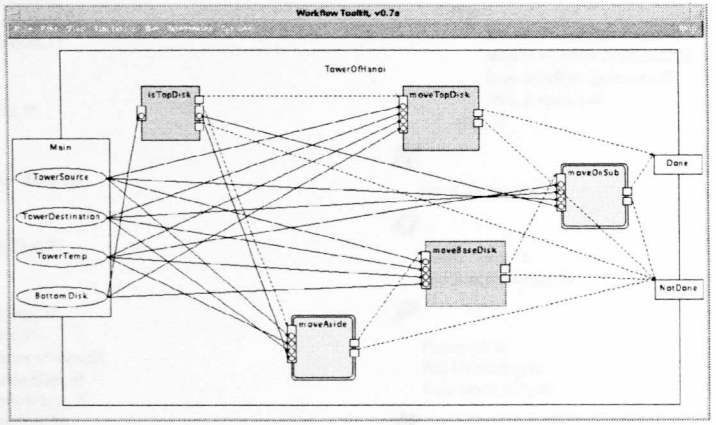


Figure 10.6: Towers of Hanoi process modeled as a Workflow

10.2.2 Starting the Workflow Application

You will need two services:

- com.arjuna.OPENflowDemos.TowersOfHanoiResourceServer and
- com.arjuna.OPENflowDemos.TowersOfHanoiTaskServer.

The first server starts a GUI that allow you to monitor what is happening in terms of Disks and Towers (or in other words monitors where the disks are moving...). This GUI is depicted on figure 10.7

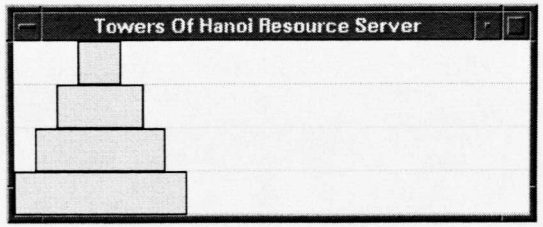


Figure 10.7: Towers of Hanoi Resource Monitoring GUI

When starting the "Towers of Hanoi" workflow application, the form from figure 8.2 appears. The Disk to be chosen is the disk with the highest final number. The Towers are numbered from the left to the right. The Source tower is the one where all disks initially appear on the resource GUI. The destination and temp towers are the other two.

Index of Contents

References

[1] R. Allen and G. Garlan, "Formalizing Architectural Connection", Proceedings of the 16th International Conference on Software Engineering, pp 71-80, Sorrento, Italy, May 1994.

[2] G. Alonso, D. Agrawal, M. Kamath, R. Gunthor, C. Mohan,, "Advanced Transaction Models in Workflow Contexts", 12th International Conference on Data Engineering New Orleans, February 1996.

[3] Amazon.com, http://www.amazon.com/

[4] Jean Marc Andreoli, Steve Freeman and Remo Pareschi, "The Coordination Language Facility: Coordination of Distributed Objects", Theory and Practice of Object Systems, vol 2(2) pp 77-94, 1996.

[5] Barclays, http://www.barclaysquare.co.uk/

[6] Luc Bellisard, Michel Riveil, "Distributed Application Configuration", Proceedings of the 16th IEEE International Conference on Distributed Computing Systems, Hong-Kong, May 1996.

[7] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel, "Basic Support for Cooperative Work on the World Wide Web", International Journal of Human Computer Studies, 1997.

[8] Eike Best, Dipl. Inform. Bernd Grahlmann, "PEP: Programming Environment based on Petri Nets, Documentation and User Guide", version 1.4, Institut für Informatik, University Hildesheim, November 1995.

[9] Eike Best and Richard Pinder Hopkins. "$B(PN)^2$ – a Basic Petri Net Programming Notation", Proceedings of PARLE'93, volume 694 of Lecture Notes in Computing Science, pp 379-390, A. Bode, M. Reeve and G. Wolf editors, Springer-Verlag, 1993.

[10] Gregory Alan Bolcer and Richard N. Taylor, "Endeavors: A Process System Integration Infrastructure", Information and Computer Science, University of California, Irvine

[11] C3DS, "Control and Coordination of Complex Distributed Services, Project

Programme", ESPRIT Long Term Research Project 24962

[12] F. Casati, S. Ceri, B. Pernici, G. Pozzi, "Conceptual Modeling of Workflows", OOER'95, Gold Coast, Australia, December 12-15, 1995.

[13] S. Das, K. Kochut, J. Miller, A. Sheth, D. Worah, "ORBWork: a Reliable Distributed CORBA-based Workflow Enactment System for METEOR2", LSDIS, The University of Georgia.

[14] C. T. Davies, "Data processing spheres of control", IBM Systems Journal, Vol.17, No. 2, 1978, pp. 179-198.

[15] Digital Equipment Corporation, "TeamRoute Programming Guide", AA-PM6FA-TE, DEC, Maynard, MA, June 1992.

[16] FloWare, http://www.plx.com/html/floware_scalable_workflow.html

[17] H. Garcia-Molina and K. Salem, "Sagas", Proceedings. 1987 SIGMOD International Conference on the Management of Data, Pp. 249-259, May 1987.

[18] D. Garlan, R. Allen and J. Ockerbloom, "Exploiting Style in Architectural Design environments", Proceedings of SIGSOFT'94, USA, December 1994

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck and V. Sunderam, "PVM3 User's Guide and Reference Manual", Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

[20] D. Georgakopoulos, M. Hornick and A. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure", Distributed and Parallel Databases, 3(2) pp 119-154, April 1995

[21] Antonietta Grasso, Jean-Luc Meunier, Daniele Pagani and Remo Pareschi, "Distributed Coordination and Workflow on the World Wide Web", Rank Xerox Research Center, Grenoble, Computer Supported Cooperative Work: The Journal of Collaborative Computing Volume 6, pp 175-200, 1997.

[22] J. N. Gray, "The transaction concept: virtues and limitations", Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[23] Honeywell, "MetaH Programmer's Manual Version 1.09", Technical report, Honeywell Technology Center, April 1996.

[24] M. Hsu, "Special Issue on Workflow and Extended Transaction Systems", Bulletin of the Technical Committee on Data Engineering, IEEE, 16(2), June 1993

[25] The IBM Corporation, "IBM FlowMark Modeling Workflow", SH19-8241-02, October

1996.

[26] The IBM Corporation, "Message Queue Interface", Technical Reference, April 1993, Document SC33-0850-01

[27] The IBM Corporation, "IBM FlowMark Managing Your Workflow", SH19-8243-02, October 1996, http://www.software.ibm.com/ad/flowmark/

[28] D. Mc Carthy and S. Sarin, "Workflow and Transactions in InConcert", Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society Vol 16, no 2, June 1993.

[29] Stefan Jablonski, Christoph Bussler, "Workflow Management", Thomson Computer Press.

[30] jFlow, "OMG Business Object Domain Task Force, BODTF-RFP 2 Submission, Workflow Management Facility", bom/98-03-04

[31] J. Kramer, J Magee, "Analysing Dynamic Change in Software Architecture", pp 91-100, Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE Computer Society, May 4-6, 1998

[32] F. Leymann, "Supporting Business Transactions Via Partial Backward Recovery in Workflow Management Systems", GI-Fachtagung datenbanken in Buro Technik und Wissenschaft, BTW'95, Dresden, Germany, Springer Verlag, March 1995.

[33] D.C. Luckman et al, "Specification and Analysis of Software Architecture using Rapide", IEEE Transactions on Software Engineering, pp 336-355, April 1995.

[34] Jeff Magee. "LTSA: Labelled Transition System Analyser", http://www-dse.doc.ic.ac.uk/~jnm/LTSdocumentation/User-manual.html

[35] Jeff Magee, N. Dulay, Susan Einsenbach and Jeff Kramer, "Specifying Distributed Software Architectures", Proceedings of the 5th European Software Engineering Conference, Barcelona, September 1995

[36] Jeff Magee and Jeff Kramer, "Dynamic Structure in Software Architecture", SIGOFT 96, ACM Software Engineering Notes, Vol. 21, No 6, November 1996.

[37] R. Medina-Mora, T. Winograd and R. Flores, "The ActionWorkflow Approach to Workflow Management", Proceedings of the 4th Conference on Computer-Supported Cooperative Work, June 1992

[38] N. Medvidovic and R. N. Taylor, "Reusing off-the-shelf Components to Develop a Family of Applications in the C2 Architectural Style", Proceedings of the International

Workshop on Development and evolution of software architectures for product families, Las Navas del Marques, Avila, Spain, November 1996.

[39] Neno Medvidovic, "A Classification and Comparison Framework for Software Architecture Description Languages", Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, USA, February 1996

[40] C. Mohan, D. Agrawal, G. Alonso et al., "Exotica: a Project on Advanced Transaction Management and Workflow Systems", IBM Almaden, ACM SIGOIS bulletin, August 95, vol. 16 no 1 (http://www.almaden.ibm.com/cs/exotica/exotica_papers.html)

[41] C. Mohan and R. Dievendorff, "Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queuing", Bulletin of the Technical Committee on Data Engineering, Volume 17, no 1 pp 22-28, March 1994, IEEE Computer Society.

[42] M Moriconi, X. Qian and R.A. Riemenschneider, "Correct Architecture Refinement.", IEEE Transactions on Software Engineering, page 356-372, April 1995.

[43] MULTIPLECX, "Multi-party Processes for Large-scale Electronic Commerce Transactions, Project Programme", European 4th Framework Programme IT RTD, ESPRIT, Project no. 26810.

[44] Netscape Incorporation, "JavaScript Guide", http://developer.netscape.com/docs/manuals/communicator/jsguide4/

[45] Nortel & the university of Newcastle, "Workflow Management Facility Specification", OMG, BOM/98-03

[46] Anne H.H Ngu, Toncan Duong, Uma Srinivasan, "Modeling Workflow using Tasks and Transactions", University of South Wales, NFS workshop, USA, April 1996.

[47] OMG, "Object Management Architecture Guide", http://www.omg.org/

[48] OMG, "The Common Object Request Broker: Architecture and Specification", revision 2.0, July 1995, http://www.omg.org/.

[49] OMG, "The Common Object Service Specification", http://www.omg.org/

[50] OMG, "Common Facilities", 1995, http://www.omg.org/

[51] S. Omohundro and C. Lim, "The Sather Language and Libraries", Technical Report TR-92-017, International Computer Science Institute, Berkeley, March 1992.

[52] John K. Ousterhout, "Tcl and the Tk Toolkit", Addison Wesley Professional Computing Series.

[53] John K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", IEEE

Computer magazine, March 1998.

[54] M. Papazoglou, A. Delis and A. Bouguettaya, "Class Library Support for Workflow Environments and Applications", IEEE transactions on computers, volume 46, no 6, June 1997.

[55] G. D. Parington, S. K. Shrivastava, S. M. Wheater and M. Little, "The Design and Implementation of Arjuna", USENIX, Comp. Systems Journal, December 1996.

[56] James L Peterson, "Petri net theory and the modelling of systems", Prentice-Hall, Inc.

[57] C. Pu, G. E. Kaiser and N. Hutchinson, "Split transactions for Open-Ended Activities", Proceedings of the 14th Conference on Very Large Data Bases (VLDB), pp 26-37, Los Angeles, California, 1988

[58] J. M. Purtilo, "The Polylith Software Bus", ACM TOPLAS, Vol. 16 no.1, pp 151-174, Pittsburgh, March 1994.

[59] F. Ranno, S. Wheater, and S. K. Shrivastava, "A System for Specifying and Coordinating the Execution of Reliable Distributed Applications", Proceedings of the International Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany, October 1997

[60] F. Ranno, S. Wheater, and S. K. Shrivastava, "A language for Specifying the Composition of reliable Distributed Applications", The 18th International Conference on Distributed Computing Systems (ICDCS'98), IEEE, Amsterdam, the Netherlands, May 1998.

[61] B. Reinwald and C. Mohan, "Structured Workflow Management with Lotus Notes release 4", Proceedings of the 41th IEEE Computer Society International Conference, pp 451-457, Santa Clara, California, February 1996

[62] Jane Rickard, "E.commerce, big business or fad?", Micromart, pp 116-117, 23rd October 1997, based on a survey by CommerNet and Nielsen Media Research.

[63] Paul Santanu, Edwin Park and Jarir Chaar, "RainMan: a Workflow System for the Internet", IBM Watcom, USENIX Symposium on Internet Technologies & Systems, 1997.

[64] Paul Santanu, Edwin Park, Jarir Chaar, "Extending the WfMC Standard to the Distributed World", Workflow Management Coalition Meeting, London, October 20-22, 1997.

[65] Paul Santanu, Edwin Park, Jarir Chaar, "Essential requirements for a workflow

standard", OOPSLA'97, Business Object Workshop III, Atlanta

[66] Alexander Schill and Christian Mittasch, "Workflow Management Systems on Top of OSF DCE and OMG CORBA", Distributed Systems Engineering Journal, vol 3, pp 251-262, December 1996

[67] Randall Schwartz and Tom Christiansen, "Learning Perl", O'Reilly editions

[68] F. Schwenkreis, "APRICOTS – A Workflow Prgramming Environment", 6th High Performance Transaction System workshop, Asilomar, Pacifi Grove, California, September 1995

[69] M. Shaw, R. DeLine and G. Zelesnik, "Abstractions and Implementations for Architectural Connections", Proceedings of the third International Conference on Configurable Distributed Systems, May 1996.

[70] A. Sheth et al., "Supporting State-wide Immunisation Tracking using Multi-paradigm Workflow Technology", Proceedings of the 22nd International Conference on Very Large Databases, Bombay, India, September 1996

[71] S.K. Shrivastava, L. V. Mancini and B. Randell, "The duality of Fault-Tolerant System Structure", Technical Report Series n. 305, February 1990

[72] Teknowledge, "The ARDEC/Teknowledge Architecture Description Language (ArTek), version 4.0", technical report, Teknowledge Federal Systems, Inc. and US Army Armement Research, Development and Engineering Center, July 1995.

[73] W. Tracz, "Parameterized Programming in LILEANNA", Proceedings of ACM symposium on Applied Computing (SAC'93), February 1993.

[74] H. Wächter and A. Reuter "The ConTract model" in "Transaction Models for advanced Database applications", (editor A. Elmagarmid), Chapter 7, pp. 220-262, Morgan-Kaufman, February 1992.

[75] Larry Wall, Tom Christiansen, Randall Schwartz, "Programming Perl", O'Reilly editions

[76] John Warne, "Flexible transaction framework for dependable systems", ANSA report No 1217, 1995

[77] Aaron Watters & al, "Internet programming with Python"

[78] WfMC, "The Workflow Reference Model" version 1.1, November 1994, WfMC-TC-1103, http://www.wfmc.org/

[79] WfMC, "Process Definition Interchange", WfMC TC-1016, http://www.wfmc.org/

[80] WfMC, "Workflow Process definition Read/Write Interface", WfMC-WG01-1000,

http://www.wfmc.org/

[81] S. Wheater, "OPENflow Workflow Module CORBA Interface Reference Manual", version 0.6.1, Arjuna Solutions, Ltd, Newcastle upon Tyne.

[82] S. Wheater, S.K. Shrivastava and F. Ranno, "A CORBA Compliant Transactional Workflow System for Internet Applications", Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE'98), The Lake District, England, September 15-18, 1998.

[83] D. Wodtke et al., "The Mentor Project: Steps towards Enterprise-wide Workflow management", Proceedings of the 12th IEEE International Conference on Data Engineering, New Orleans, LA, February 1996