Data Exchange in a Component Based Workflow Environment

Paul Fahey

A dissertation submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

September 1999

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____
Paul Fahey
September 1999

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____
Paul Fahey
September 1999

Summary

There is a need to revise existing (successful) software subsystems. One can't afford to develop a bespoke system, as done previously, so therefore 'shrink wrapped solution sets' have to be realised. The problem is how to integrate heterogeneous components to support specific, sometimes unique, enterprise business processes.

One approach is the use of Workflow Engine to co-ordinate and enact distributed components based on explicitly represented business processes. An important element of any Workflow Engine is the exchange of input parameters and output results between components. Traditionally components invoke each other and pass data directly into each other. However, in Workflow Engines component invocation is performed by the engine and not performed directly between components. Performance would deteriorate if all data flow needed to pass through the Workflow Engine itself. A more efficient approach is to provide a shared component data exchange, responsible for the storage, management and retrieval of data, which is passed between components.

Acknowledgements

I would like to thank my supervisor Vinny Wade, for the advice and help that he has given me during the dissertation. I would also like to thank Brian Cullen , John Fuller, Sinead Muldowney, Andrew Nolan, and Cliff Redmond for the help and very useful support during the dissertation.

Also a thanks is due to the classmates of the MSc Networks and Distributed Systems for all their help during a tough year.

Contents

List of Illustrated Materials and Tables

Figures:

Tables:

1. Introduction

1.1 Introduction

There is a trend in software engineering towards distributed componentisation of software elements and the use of workflow to co-ordinate distributed component execution. A Workflow Management System (WFMS) as defined by the Workflow Management Coalition (WfMC), is a system that defines, creates, and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke applications (or components) [WfMC]. Componentisation allows developers to take advantage of software reuse, and enables organisations to build on existing applications therefore avoiding the need to develop systems from scratch.

Within the WFMS there are various components that tackle activities that need to be completed as described above. These components need access to some sort of repository to be able to use input parameters and then store the output parameters. Another view of this issue is that each process, and therefore each component needs to be able to invoke data to either test conditions on the state of each task, or to use data to complete the activity. The results of the process, the output data, must be stored somewhere, as they may need to be revisited at some stage in the future. It would be unwise to discard the data produced from the process. The workflow engine is responsible for the scheduling of the processes that must be completed in the WFMS.

The difficulty in this storage of data is obviously how and where to store the data, and also how the components invoke the data contained in it. The goal is to have an efficient data storage system that will allow the WFMS to execute more efficiently. This entails taking the data storage function away from the workflow engine itself. In removing this functionality from the workflow engine one is augmenting its capacity as a co-ordination tool, and therefore improving its efficiency, and speed in completing activities. To enable this data to be stored and invoked effectively, a Data

Manager must be implemented. A data manager of course could allow additional functionality to be added [on the data manager side], which would be difficult to build into the workflow engine [Alon97a].

A business process can be separated into a number of predefined activities. Each activity is viewed as separate to the other activities in the business process, The business process, is added to a work list. The work list gives the state of all business process', in the WFMS. A business process is executed by a WFMS as each activity is completed. The business process is completed when all its activities have been completed, only then is it signed off, and removed from the work list. An example that is regularly used is the process of an expense form that passes through the different departments of an organisation, so that it can be authorised by the different departments. The business process is the complete authorisation of the expense form. An example of an activity can be considered the authorisation by one department.

There is a distinction to be made with regards to the data that is used in a WFMS. There are control data and production data. Control data relates to controlling the flow of a process through a WFMS, whether an activity should be initiated, or whether it should wait until the completion of another activity. Control data is considered persistent data and therefore a method of retaining this data is important. Production data relates to the input parameters and the output data of an activity or business process, and can therefore be classed also as persistent data. This is where the data manger in this project is needed, to store the production data. Building this separately to the workflow engine eases the load on the workflow engine. Also, invocation of the data by components, or different applications would be better served by a data manager rather than the workflow engine itself, as the engine should deal more with the scheduling and co-ordinating of the activities in a business process. Most WFMS have a central repository that contains control data, i.e. application data employed to evaluate the transition conditions governing the control flow, and this suggests that the past work done on WFMS neglected the production data flow aspects and focused on implementing control flow.

What is being investigated here is the role of database technology and research in the area of WFMS in relation to the exchange of production data. It is hoped that it will be possible to identify the role that database technology can take in improving the WFMS, and to identify the database technology that would be best suited to the Serene Workflow Engine, the WFMS under development in the Computer Science department of Trinity College Dublin. The data manager must be a shared component data server (SCDS), as there will be different components, invoking the data contained in the SCDS, and storing output data that may be used by other components for other activities. It is seen that the SCDS should be as intelligent as possible, therefore easing the load on the workflow engine, and on the components.

1.2 Objectives

The objective of this thesis is: to investigate the issues and propose a solution(s), to support information flow between components in an engine based workflow environment.

This will be carried out as follows:

- Research into how other Workflow Engines exchange component data.
- Design of the integration of a Workflow application data exchange
- Implementation of the design
- Evaluation of the implementation

1.3 Technical Approach

The first phase of this project is to investigate WFMS prototypes that have been developed and how they approach the area of production data exchange. The investigation will include an analysis of their modes of exchanging data, and the mechanisms for storing data. The focus will be on distributed WFMS. An

examination of Component Models is also necessary to identify their methods in addressing persistent storage.

Phase two of the project is to design a data exchange for the Serene WFMS and integrate this with version 2 of the Serene workflow engine which is being developed. Interfaces are to be designed so as to allow the integration of the design of the data exchange with the Serene workflow engine.

Phase three involves the implementation of the data exchange, once an adequate design has been proposed.

Phase four of the project is the evaluation of both the design and the implementation. The evaluation will consist of a comparison with the research carried out in the first phase, an evaluation of the objectives and the achievements of the project, and finally an evaluation of the prototype developed for the project.

2. Workflow Overview and Systems

2.1 Workflow Overview

Workflow is defined as a collection of tasks organised to accomplish some business process (e.g. processing purchase orders over the phone, processing insurance claims) [Geog95]. A business process can be completed automatically by a software system, manually by human intervention, or both of these. It is seen as the automation of a business process.

A business process is broken down into a number of steps or activities, which can be drawn as an annotated directed graph, which defines the process in a step by step fashion. Each step is completed in a structured manner. The business process is represented as a workflow, i.e. computerised models of the business process, which specify all the parameters involved in the completion of the process. Figure 1 is an example of a business process, and the example used is an expense from passing through an authorisation process.

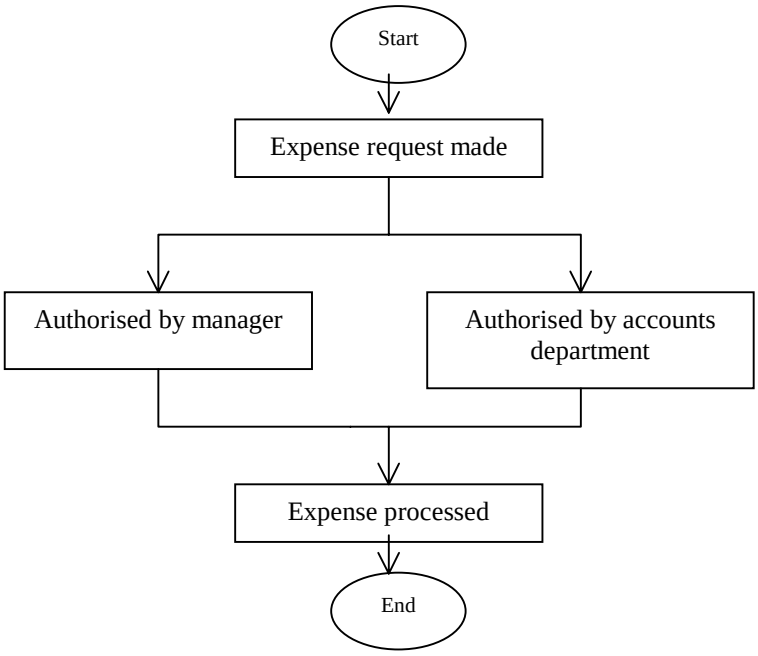

Figure 1: Sample Business Process for an Expense Request

Workflow is seen in many areas as a complement to Business Process Re-Engineering (BPR), which is a term that has crept into the corporate world and is dominating most system development projects today. A WFMS is used to co-ordinate, and streamline the business process. The parameters that are involved in the completion of a process range from defining the individual steps (entering customer information, consulting a database, getting a signature), to establishing the order and conditions in which the steps must be executed including aspects such as data flow between steps, who is responsible for each step, and the applications (databases, editors, spreadsheets) to use with each activity [Alon97b].

Each sub-task, or activity, is passed from one participant to another and it is automatically, or manually, completed once the start condition for the activity has been passed. The WFMS provides the ability to specify, execute, report on, and dynamically control workflow [Geog95]. This is one form of a WFMS. The other occurs by the activity being initiated in a controlled sequence by the WFMS, that is, once an activity is completed the WFMS is notified and it initiates the next activity in the sequence.

2.2 Types of Workflow Systems

Workflow systems fall into two broad categories:

- Forms and messages based workflow systems which perform electronic routing of forms to user's e-mail in-boxes
- Engine based workflow systems, which communicate with humans or components via specialised client software [Wade99]

Workflow systems can be viewed in another fashion, as will be discussed in the rest of this section. There are four main categories of WFMS [Alon97b], although some sources suggest that there are only three. The four categories that are specified are as follows:

- Administrative

- Ad hoc
- Collaborative
- Production

The differences that these categories are based on are: a) repetitiveness and predictability of workflows and business processes; b) how the workflow is initiated and then controlled; and c) requirements for WFMS functionality.

Administrative workflows have steps that are well established, and the set of rules governing the process is known by everybody involved. They are simple repetitive predictable processes that have simple co-ordination rules. An example of this would be the routing of an expense report through an authorisation process, or the registration of a student in university. There is little complexity to the workflow process here, and the WFMS in this category would be classed as non-mission critical [Geog95]. In this category it is the users that are actively prompted to perform their tasks.

Ad Hoc workflows are similar to administrative except for the fact that they tend to be created to deal with exceptions, or where there is no set pattern for moving information among people. Therefore the ordering, and co-ordinating of the activities is controlled by human participants. They are intended to support short-term activities. An important point of this category of WFMS is that activity ordering, and co-ordination decisions are made during the performance of the workflow. An example of this is that when an activity has been completed the WFMS will then see who is available to perform the next activity, and the activity will be placed in the worklist of that participant.

Collaborative workflows, can be classed in the Ad Hoc category, but can also be viewed as a category on its own. This is the extra category that is not mentioned in most literature. Unlike the other categories, which are based on forward-directed tasks, the collaborative category includes those tasks that are iterative over the same step until some form of agreement has been made. An example of this would be the writing of a paper by more than one author, where the final paper must be agreed upon

by all parties and, there might have been several toing-and-froing by the authors in the process of reaching agreement. It would be very difficult to model such a process using tools that are not geared for collaboration since it is impossible to predefine the steps to follow [Alon97b]. Most of the co-ordination is done by the human participant, so it can be argued that these types of processes don't count as being defined as WFMSs.

Production workflows involve repetitive and predictable business processes. They can be classed as the implementation of critical business processes. This means that they are directly related to the function of an organisation. An example of this would be the processing of insurance claims or loan applications. Co-ordination and ordering of activities can be automated, but the automation of a production workflow is complicated due to the fact of, a) information process complexity, and b) accesses to multiple information systems to perform and retrieve data for making decisions. These WFMS tend to be large scale, and have to deal with heterogeneous environments.

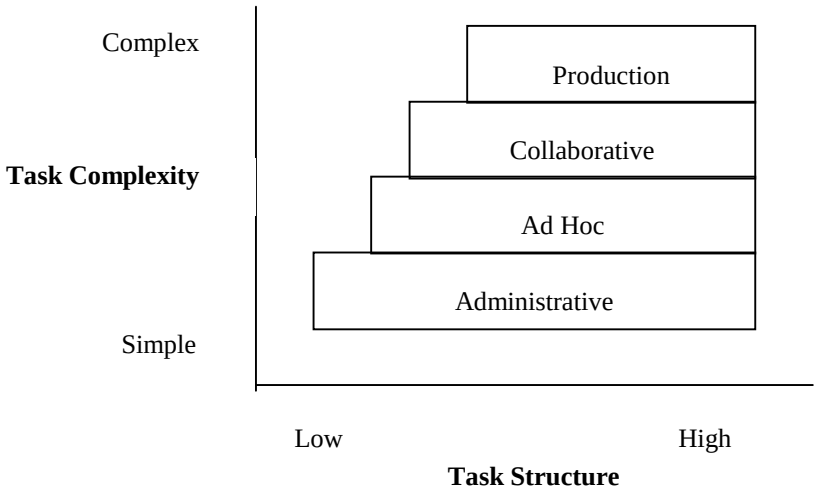

Figure 2: A Rough Characterisation of Workflow

Figure 2, above [Alon97b], shows the four categorisations in relation to each other, based on task structure and task complexity. As can be seen a production workflow is the most complex and the most highly structured. This is due, as stated above, to the

information process involved in the workflow, and the fact that the activities are mostly automated.

2.3 Workflow Model

At the centre of any WFMS is the business process, and a workflow is a computerised model of the business process. There therefore needs to exist a set of rules implemented by a WFMS so that it can execute workflows by use of software that is driven by the computerised workflow model.

2.3.1 WfMC Reference Model

Figure 3 is a graphical representation of the workflow model as given by the Workflow Management Coalition (WfMC) [WfRe94]. This is the standard that the WfMC has set down so that WFMSs can at least have a common understanding, and to prevent the growth of completely unrelated WFMS. Figure 2 illustrates the major components and interfaces within the workflow architecture.

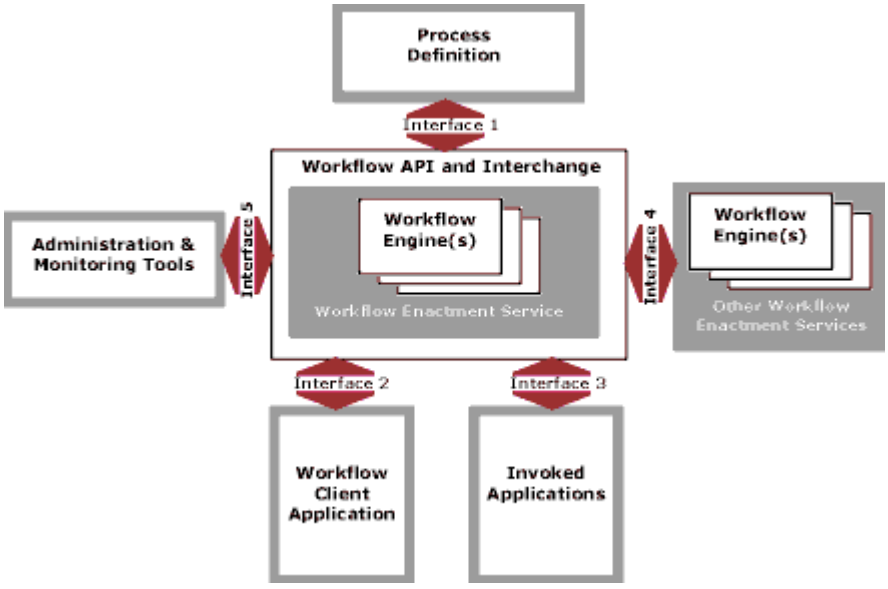

Figure 3: The Workflow Management Coalition Workflow Reference Model

Working Groups	**Objectives**
Reference Model & Glossary	Specify a framework for workflow

	systems, identifying their characteristics, functions and interfaces. Development of standard terminology for workflow systems.
Process Definition Tools Interface (1)	Definition of a standard interface between process definition and modelling tools and the workflow engine(s).
Workflow Client Application Interface (2)	Definition of APIs for client applications to request services from the workflow engine to control the progression of processes, activities and work-items.
Invoked Application Interface (3)	A standard interface definition of APIs to allow the workflow engine to invoke a variety of applications, through common agent software.
Workflow Interoperability Interface (4)	Definition of workflow interoperability models and the corresponding standards to support interworking.
Administration & Monitoring Tools Interface (5)	The definition of monitoring and control functions.
Conformance	To develop the Coalition's policy on product conformance against its specifications and agree an approach to vendor certification.

Table 1: Summary of the WfMC Reference Model

Table 1 [WfRe94] gives a summary of the different parts of the reference model, as shown in Figure 3, that the WfMC have set as standards. An important point to note in this model is that interfaces 2 and 3 are same, although originally they were specified as two different interfaces. Over time they have been recognised as having the same interface.

2.4 Process Modelling

Modelling a process involves capturing the business process. Usually this would involve interviews with experts that have domain knowledge about the process. Once enough knowledge has been gathered about the process the workflow specification is performed to capture the process which requires a workflow model [Geog95]. This workflow model consists of a set of concepts that describe the process, the activities and the dependencies among the activities.

2.5 Survey of Data Exchange Approaches in WFMS Prototypes

This section introduces the prototype WFMS that were investigated, and their methods of data exchange.

2.5.1. ORBWork

ORBWork is a reliable and fully distributed CORBA based enactment system for the METEOR2 WFMS. ORBWork supports scalable software architecture, multi-database access, as well as error detection and a recovery framework that uses transactional concepts. The workflow specification created in the designer is stored in an intermediate format called the Workflow Intermediate Language (WIL) which is similar in structure and semantics to the Workflow Process Definition Language (WPDL) of the WfMC. The WIL specification contains all the dependencies between activities and the data objects that are passed among the different activities. The runtime system of the METEOR2 system is divided into two types of components: task manager (controller/scheduler), and task (executable). [Das97]

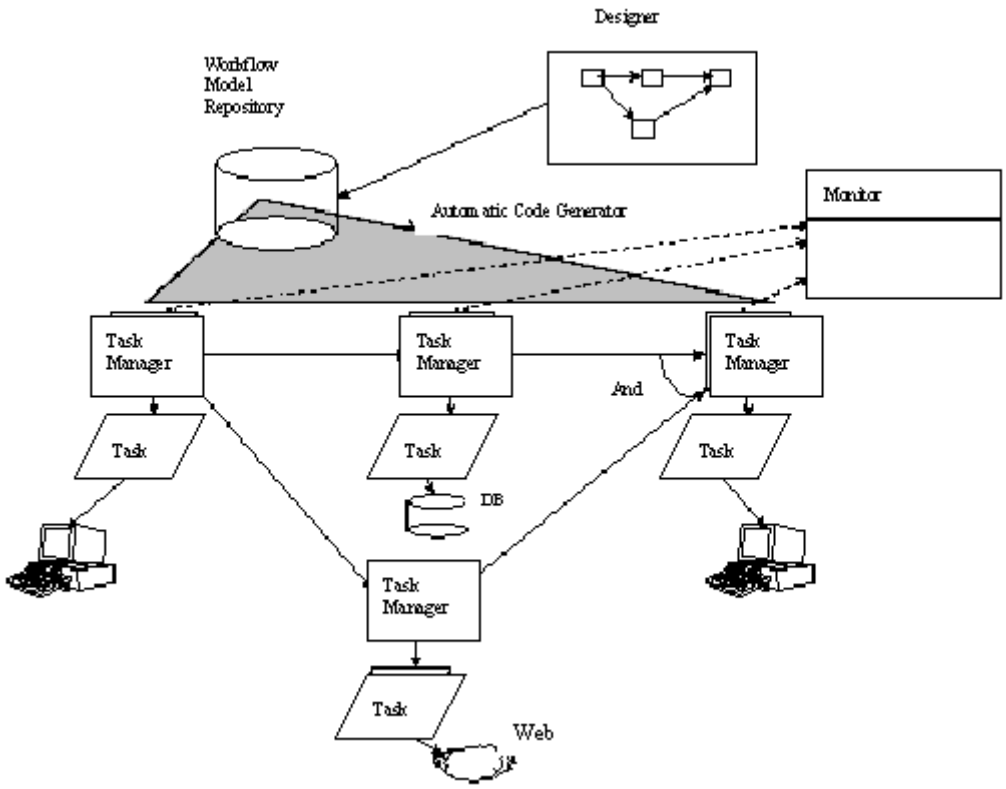

Figure 4: The METEOR$_2$ Architecture

A task manager is started usually by its predecessor by a method called *Activate*. This method starts the task manager, and it also passes it the necessary parameters for the task manager to begin. One of the parameters that it passes is a list of all the data objects that the task manager will need [Das97]. The input parameters are 'unpacked', and once the task/activity has completed, a *Save* method is called to save the output parameters. The save method is provided by the data object, or by using the persistent object services of CORBA. It is the workflow code generator, the designer, that creates an appropriate IDL interface for each data object, as it processes the WIL specification. This WFMS is considered to be a production workflow.

2.5.2. The Mentor Projects

There are two projects that have been developed under the Mentor project. There is the original Mentor project, and arising from that the Mentor-Lite project was developed. The Mentor-Lite project has approached the development and integration of a WFMS from a different aspect to the original Mentor project.

2.5.2.1 Mentor

The Mentor architecture, as shown in Figure 5, is generally designed as an open modular architecture where further components can be added, and components can easily be replaced by alternative implementations. The invoked applications of activities are run at the client sites. An Object Request Broker (ORB), which is part of the CORBA architecture, is integrated to cope with the potential heterogeneity of the invoked applications that belong to the workflow [Muth98]. Orbix is used as the CORBA compliant ORB.

The workflow specification is based on state and activity charts. *Activity charts* specifies the data flow between activities, in the form of a directed graph with the items as arc annotations. *State Charts* reflect the behaviour of a system in that they specify the control flow between activities [Muth98]. The workflow specification is partitioned based on the assumption that for each activity of the activity chart there is a corresponding department or business unit that carries out the activity. Therefore each activity can be assigned to a workflow server of the corresponding department or business unit. States are then assigned to activities. The state chart is then orthogonalised and then the partitions are assigned to workflow servers.

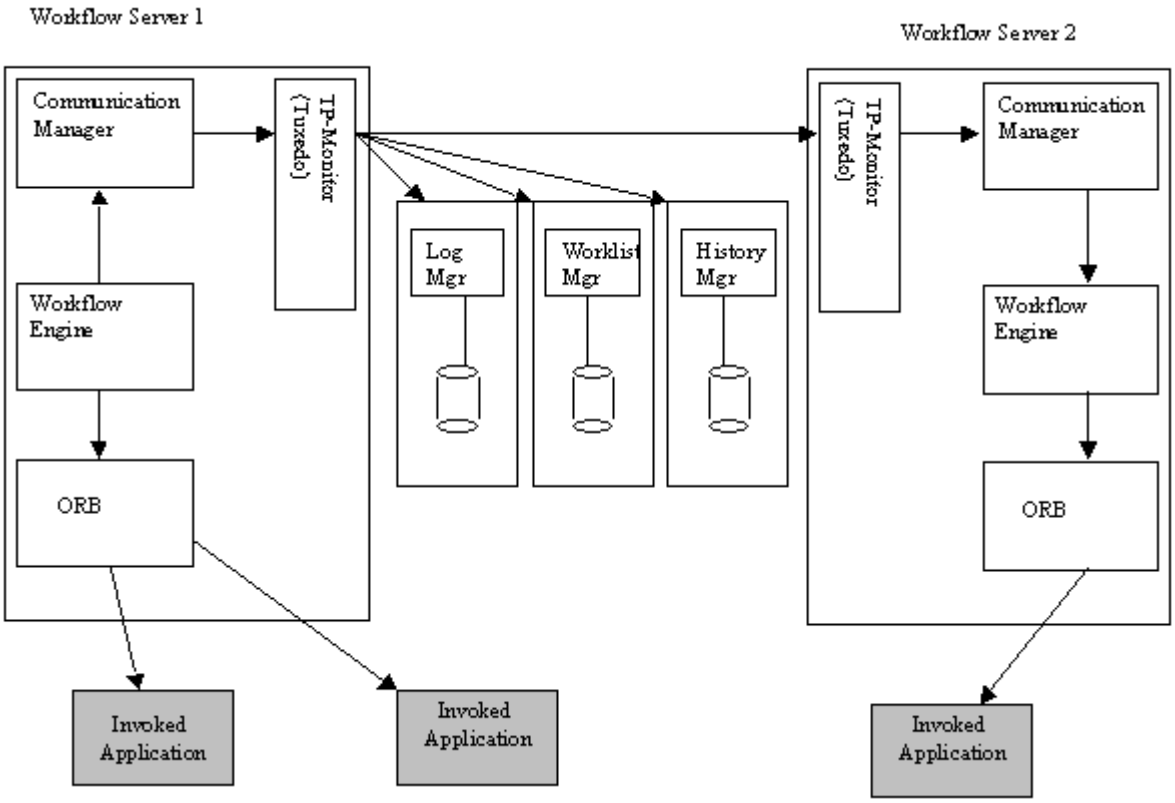

Figure 5: Client/Server Architecture of Mentor

2.5.2.2 Mentor-Lite

The Mentor-Lite project is a lightweight WFMS based on the Mentor WFMS. The view taken is that a WFMS is integrated within environments that already have solutions for implementing control flow. Other WFMS typically involve implementing the application's control flow exclusively by the WFMS; the control flow in this type of WFMS would be specified and implemented from scratch. In most cases it is rare that a business process will be computerised from scratch. There are two architectural requirements of the Mentor-Lite project:

- A stepwise integration of workflow management functionality into existing environments must be supported. This requires the integration and stepwise substitution of the existing control flow implementations.

- A workflow management system must facilitate the implementation and seamless integration of system extensions. Applications must not suffer from runtime overhead or large system footprints caused by system extensions unless their functionality is actually exploited [Muth99].

As shown in figure 6 the basic building block is an interpreter for workflow specifications based on state charts. The communication manager (ComMgr) and the log manager (LogMgr) are closely integrated with the workflow interpreter. These three components make up the workflow engine. The TP-Monitor, Tuxedo, is used to deliver synchronisation messages, but it is hypothesised that CORBA will replace this. Applications are connected to the workflow engine by specific wrappers, and these are basic communication interfaces using CORBA.

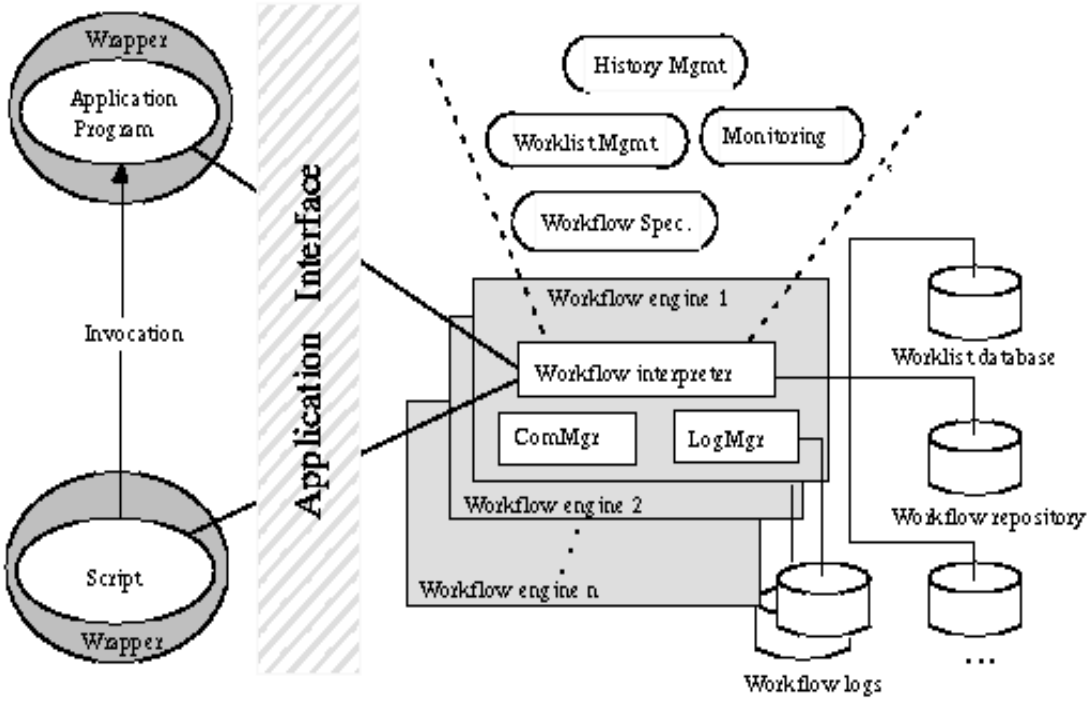

Figure 6: The Mentor-Lite Architecture

It is viewed that data flow is not within the scope of research of this project. The data flow between workflow activities is an orthogonal issue to the control flow handling.

Only data that is relevant to the control flow behaviour is caught. The assumption is that computerised business processes usually exist when the WFMS is introduced. So, there is also a kind of data flow implementation, e.g. via pipes or temporary database tables. Mentor-Lite proposes to use this existing data flow solution [Gill99]. This WFMS is a production workflow WFMS

2.5.3 Panta Rhei

Panta Rhei's architecture is based on Web technologies. It is a web-enabled system, as opposed to a web-based system. A web-based system is a system where it is entirely

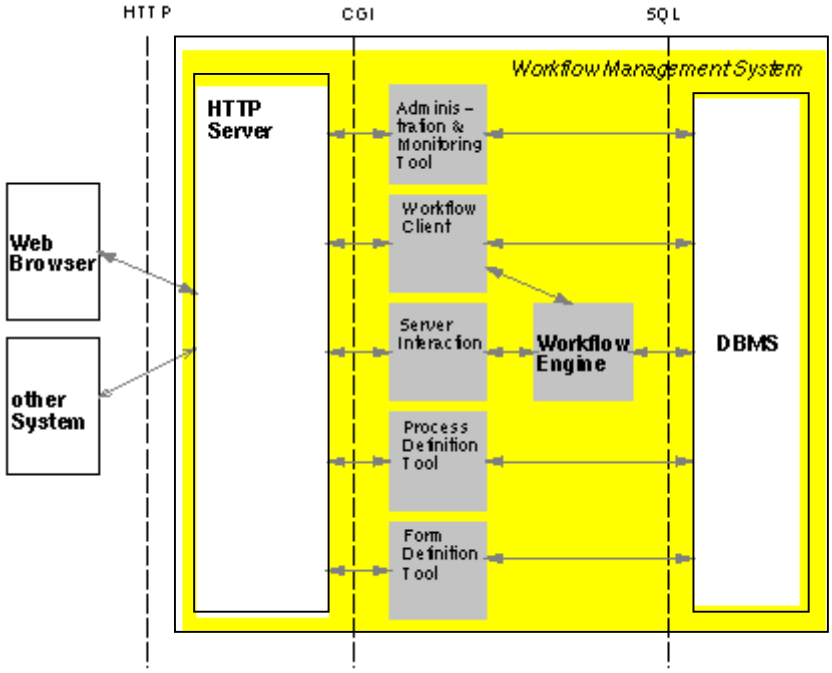

Figure 7: The Panta-Rhei Architecture

run over HTTP. As can be seen from figure 7 there is an HTTP server in the Panta Rhei architecture between the client web browser and the WFMS, and subsequently no other communication occurs over HTTP. The interface of a user to Panta Rhei is integrated in a web browser, therefore allowing any user with a web browser to interact with the Panta Rhei WFMS, and thus participate in a workflow. The Panta-Rhei comes under the administrative workflow category.

This WFMS is a forms based system. The architecture differs from the WfMC reference model (Section 2.3.1), and the workflow engine is a relatively small component containing the process interpreter [Eder98]. As can be seen from figure 7 all the other components in the WFMS are connected directly to the database management system (DBMS). The WFMS is implemented in Java, and connects to the database using Java Database Connection (JDBC).

2.5.4 Exotica/FMQM with Lotus Notes

Another research project that has been carried out does not involve the development of a prototype, rather it pulls together the functionality of two commercial products: FlowMark as the workflow engine, and the replication capabilities of Lotus Notes as the support system for distributed data management. Exotica/FMQM (FlowMark on Message Queue Manager) is the distributed version of FlowMark based on a generic queuing system with recoverable queues [Alon97a].

It was viewed in this research that the managing of data flow has been partially ignored by most commercial products, and the objective was to have a system that took the management of the data flow away from the workflow engine. This enables the workflow engine to concentrate on the scheduling of activities. Added functionality can be embodied in the data manager, which the workflow engine would not be envisioned containing. A definition that must be noted in this sub-section is the use of the term *node*. This is used to represent a physical machine.

The workflow model is enacted by Exotica/FMQM and this is where the business processes are instantiated, and subsequently the activities that make up the business processes are also instantiated. The Exotica/FMQM is a combination of a production and administrative workflow.

Within this system there is a distinction made between the *control node* and the *data node*. The control node is where the activity is carried out, and the data node supplies the data inputs that the control node may need. A data node can supply to more than

one control node. The approach taken in modelling the business process[1] in this research project is that the activities are either manually or automatically completed. This entails there being two manners to carry out the activity and the management of the data needed for the activity.

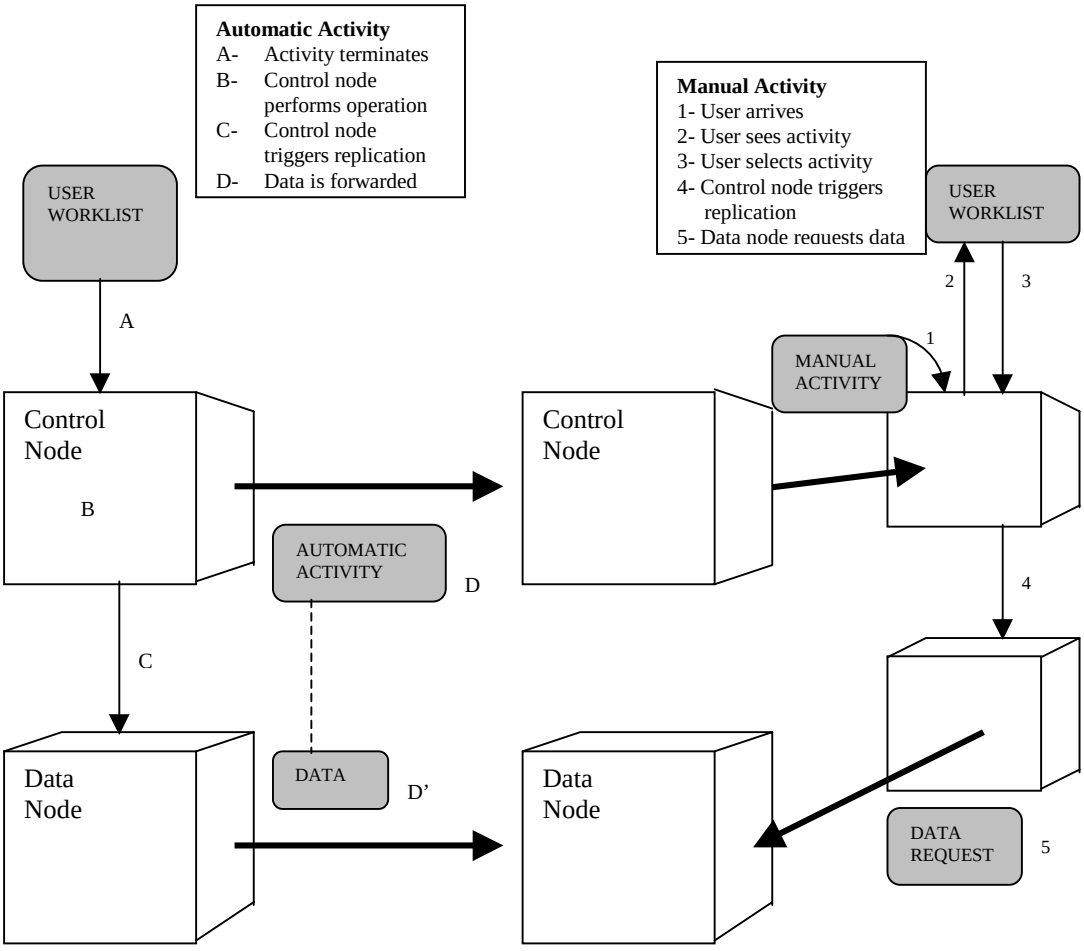

Figure 9: Co-ordination of distributed workflow and data management [Alon97a]

For an automatic activity when it has been instantiated, or put in the context of this project, when a control passes the control forward to another control node the data is also forwarded to this node. This is where Lotus Notes uses it functionality of replication, and the data is forwarded to the next data node through replication. In the case of a manual activity if the data needed for the activity is not automatically forwarded then the user can manually activate the data transfer. Figure 9 gives a

[1] The business process that has been modelled in this project is that of patent claims

graphical representation of these modes of activity completion. Also included in figure 9 is a step-by-step flow of operations that both the automatic handling of an activity, and the manual handling of an activity must follow.

In current commercial workflow systems, data flow is either done externally, with poor co-ordination and little flexibility, or embedded in the control flow, which has a serious impact on performance. The impact of this research project minimises the effect on the control flow, i.e. in the workflow engine, while allowing very sophisticated co-ordination between the activities and the data flow. [Alon97a]

2.6 Serene Workflow Engine

Serene is a workflow driven telecommunications management system based on CORBA middleware. It supports the automation of telecommunication management processes and the integration of service management components [Wade99]. The approach taken in this project is to implement an engine based workflow system. This project is being developed in the Knowledge and Data Engineering Group (KDEG) in the Computer Science department of Trinity College Dublin. Figure 10 shows a schema of the Serene WFMS architecture.

The engine consists of a scheduler that accepts management requests and initiates instances of these requests and quizzes a knowledge base to know which activity should be started. Once the activity to be started has been decided upon then the information that it has been started is stored in the Workflow Information Server (WIS). Then the scheduler passes the information to the dispatcher, i.e. the information that the activity has been started. The dispatcher then invokes the management component that will complete the activity.

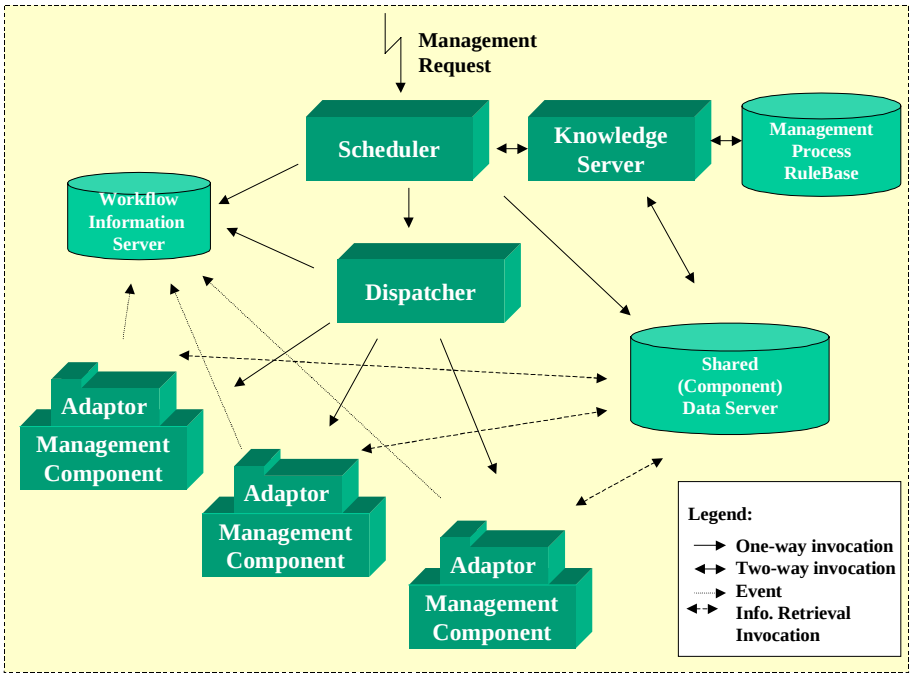

Figure 10: The Serene Architecture

The management component adaptors interface the workflow engine to the components [Wade99]. Between the adaptor and the workflow engine only workflow data is transferred. The data that is needed for an activity to be completed is stored in the Shared Component Data Server (SCDS), which has been renamed the Shared Data Exchange (SDE). It is the job of the adaptor to retrieve this information from the SDE. It is also up-to the adaptor to store any resulting output parameters from an activity in the SDE. The SDE is the focus of research that was carried out for this project. The SDE comes under the problem statement as described in section 1.2. It is the adaptor that lets the workflow engine know that the activity has finished by notifying the WIS.

The scheduler, dispatcher and adaptors were all implemented in Java running on Windows NT. The WIS and Management Rulebase use commercial database and Knowledge based systems. CORBA (OrbixWeb) was used as the distributed platform for the workflow engine [Wade99]. The workflow engine is event driven and also multithreaded.

2.6.1 Knowledge Server

The Knowledge Server supports the representation of management business processes via a rules base, known as the Management Process Rulebase [Wade99]. A unique ID is assigned to every business process instance and then an activity instance is assigned its own ID as well. The rulebase that is used is the Java Expert System Shell (JESS). A business process object is created, and it is the scheduler that interacts with this object.

2.6.2 Scheduling Management Activities

The scheduler ensures the flow of information and controlled interaction between the management activities, in order to accomplish the desired management process. There are three operations that the scheduler supports: start, query, and abort a management process. As this component is multithreaded it can allow multiple management processes to run concurrently.

2.6.3 Workflow Information Server

This server is used by the dispatcher, and scheduler. It is possible to view which management processes are currently running, and the state of the management activities. The data that the WIS uses is stored in a Process Warehouse that the WIS can query. All queries are done using SQL. The data that is stored in the process warehouse is shared and therefore there exists locking methods to prevent inconsistency in the data. Four states exist for a management activity: inactive, active, complete, and abort.

2.6.4 Workflow Dispatcher

The dispatcher is invoked by the scheduler. The dispatcher's objective is to invoke the component application that can complete a management activity. This invocation of a component application is done by creating an adaptor. The dispatcher knows which adaptors have been created. In allowing the dispatcher to create an adaptor rather than allowing it directly to invoke an application, the dispatcher has less load on it. This results in the dispatcher being able to take on more requests from the scheduler.

2.6.5 Component Adaptors

The adaptor is responsible for invoking the component application so that an activity can be carried out. This is as discussed in section 2.4 above, i.e. to take some of the load away from the dispatcher. It is the responsibility of the adaptor to access the SDE so that it can pass any of the input parameters to the component application. It is also one of the functions of the adaptor to store the output parameters of a completed activity in the SDE. The adpator starts a thread, which then binds to a wrapper. Each component application is started using a specific thread that has bound to a wrapper, i.e. each thread is specific to each component application. Input parameters, and output parameters are retrieved and stored respectively by the wrapper.

2.6.6. Shared Component Data Server/SDE

In version 1 the components wrote to files, which were then read by the components. This data relates to the data that is needed for an activity rather than the control data than is needed to allow effective flow control of a management process, or processes. Therefore there was limited functionality.

As stated in section 1.2 the objective is to investigate the issues and propose a solution(s) to support information flow between components in an engine based workflow environment. This means that the SDE must be developed to prove the hypothesis of this project. The functionality of the SDE must be extended with respect to the method of storing and retrieving of data in this second version of the Serene workflow engine.

2.7 Data Integration

The core of this dissertation is to make available a data store in which the components involved in the WFMS can store and retrieve their outputs and inputs respectively. The details will be discussed in the Design section (Chapter 4). Data integration entails supplying the workflow engine with access to a Database Management System that allows persistent storage of the outputs of activities in a business process. An investigation into data exchange resulted in the analysis of the prototypes and their methods of sharing data. These prototypes are those that were discussed in section 2.5. It is necessary now to look at the various ways data is stored in general in a distributed environment, not just specifically in relation to WFMS.

2.7.1 Data Integration of Prototypes

In the analysis of the WFMS prototypes, the consideration of the data flow and the methods of storing and retrieving the data is an issue that is left separate to the development of these WFMSs. In the prototypes investigated the data that was focused on was the control data, i.e. the data that is used to decide whether an activity in a business process has completed, and that the next that activity should be activated. Production data, otherwise know as activity inputs, is not directly addressed in all of the prototypes, and it is viewed as an orthogonal issue to the development of WFMS.

To summarise the issues of data integration in the prototypes researched:

- **ORBWork:** The nature of the underlying persistence media is orthogonal to the functionality of ORBWork [Das97]. The data that is needed for an activity is passed from it's preceding activity and then unpacked. The output is then stored and this is done by a method called 'Save', and this is implemented using an external persistency storage mechanism (such as an Object Oriented Database [Das97]), or the persistent object services of CORBA are used.

- **Mentor:** The data flow is examined using activity charts which are directed graphs, where the data items are shown as the arc annotations. The invoked applications query the databases that store the data that an activity needs to complete its task. Therefore the storage mechanisms are already set up. There are also separate databases that can be accessed depending on the activity.
- **Mentor-Lite:** The data flow in this project is seen as orthogonal to the control flow handling. The system has been built with a view to stepwise integration with already existing applications, and so the data flow solutions that already exist are used. The invoked applications already have a mechanism to store and retrieve data, so it is these methods that are used to carry out the data exchange.
- **Panta Rhei:** The WFMS is connected to a database, and the API JDBC is used to connect to the database. The data that is needed to carry out a business process is stored in the database, along with all other data that is necessary for the WFMS to render a business process complete.
- **Serene Version 1:** There exists a Shared (Component) Data Server which makes use of serialising the objects that an activity needs for completion. These objects that contain the input data for an activity are serialised to a file. This was seen to be a temporary procedure for version 1 development, and functioned well, but it was viewed that the SDE could be developed with more functionality and intelligence.
- **Exotica/FMQM with Lotus Notes:** As described IN detail in section 2.5.4 the management of the data flow is done using Lotus Notes. The data flow is separated from the control flow, i.e. the workflow engine, and so the data flow is better managed, leaving the workflow engine to focus more on the scheduling of activities. The data is passed from one activity to the next by replication. This is why Lotus Notes had been used because of its replication capabilities.

Overall there is less importance given to production data flow. The Serene Workflow engine is, among the above prototypes, the only one that specifically states the need for a component that provides the means for persistent storage of outputs of a business

process' activities, and their related inputs. The Exotica/FMQM is the only research project that set out to focus on how data flow should be managed in a distributed workflow environment, and its conclusion was that two products should be used, one for the workflow engine, and another to handle the data flow.

2.7.2 Data Integration Options

There are three main ways to integrating data in a distributed environment:

- One central computer that collects and processes all data
- Independent computer systems in each office/region that do not share data with the other offices/regions
- A distributed database system

2.7.2.1 Centralised Database, Distributed Access

A centralised database is one that is located on one machine or server and is accessed by clients across a distributed system. This improves data integrity, but there are problems associated with a centralised database. The database can become a bottle-neck if there is a high amount of traffic to and from the centralised database. If however the centralised database can handle the traffic then it can be considered an appropriate solution vis-à-vis the other two solutions detailed below. Data integrity is improved due to the fact that the data is stored in only one place and only once. There is no replication, or partitioning of the data. This allows for easier management of the data on the database side.

2.7.2.2 Partitioned Databases

The second option is partitioning the database, but this implementation is only possible when there is no sharing of data across the partitions that have been set up. These partitions are generally based on office locations, or regions. It is common to still see this approach in organisations around the world. Any sharing of data is usually transferred by e-mail, or faxes. This is an ineffective procedure if there are no well-defined boundaries in the data, and if there is need to share data across boundaries

2.7.2.3 Distributed Databases

A distributed database is the third option available and this has significant advantages over the other two approaches. One of the major advantages, which is one of the objectives of distributed systems, is that a distributed database system is extensible. A distributed database system is one where most updates and queries are accomplished locally, but anyone in the organisation can access the information stored in any of the distributed databases if they have authority to retrieve and integrate the data. Control over the data is retained locally. One of the major strategies of designing and controlling distributed databases, is to replicate data. This form of database uses an approach that is better suited to the layout of a company, given that most large organisations have different departments, offices, or can be in different regions. Backup and recovery plans are substantially more important when designing a distributed database system. A well-designed distributed database should give the users of the system location transparency.

One approach to distributed databases applications is the use of the Three-Tier Client/Server Model. The three-tier approach adds a layer between the clients and the servers. The three-tier approach is particularly useful for systems having several database servers with many different applications [Pos99]. This use of middleware allows an application to make use of legacy applications. Another advantage of the use of middleware is that an application server can be moved, or changed and the user

need not know. This approach is well suited to object oriented development. Much of this object oriented approach is discussed in the next section.

2.8 Enabling Distributed Object Technology

As described in the introduction (Section 1.1) there is a move in the software world towards componentisation which allows software reuse, and enables organisations build on to existing applications, therefore avoiding the need to develop systems from scratch. There are various standards that have been developed to enable distributed computing such as Distributed Transaction Processing (DTP) defined by X/Open, the Distributed Computing Environment (DCE) defined by the OSF, CORBA defined by the OMG, and DCOM defined by Microsoft.

Component based reuse is seen as an increasingly important software development aid. This allows for the building of systems using components that can interact through well defined interfaces, and can offer a route to reusing software across projects. Two component emergent technologies are addressing component design and development, OMG's CORBA Component Model, and Enterprise JavaBeans. These will be outlined later in this section. It is envisioned that these technologies will not be used in the development of the SDE, but their evaluation is important so as to be able to make recommendations with respect to the Serene workflow engine.

2.8.1 The Component Model

A definition of the Component Model is as follows:

- A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application [EJB98].

The component model is aimed to have a multitier distributed application architecture. A multitier application is one that has been split into multiple application components. This results in having significant advantages over the traditional client/server architectures including improvements in scalability, performance, reliability, manageability, reusability, and flexibility.

In designing components one is allowing the use of reusable software, which is the primary objective of componentisation. A component has it's own interface which is published and so is available to any other component or client in the system. This interface once defined should not be changed, allowing the clients or other components in a system to know what the functions of the component are and the results that they return. This gives them the added advantage of not needing to know or understand how the component carries out any of its functions. It also sets a standard, so that if it is necessary to use a component the developer will know what the component is, and how to reuse it, that is if the developer is familiar with this particular component. Otherwise there would be additional information on what the component is, and what it does. An organisation involved in component technology would have some form of repository, a kind of reference, that would hold details on all the components that it has. This repository would be invaluable for software reuse. Development of applications would be able to reuse previously developed components, therefore saving valuable hours in development time, as it would be simply a task of putting the necessary components that already exist, together.

2.8.2 Workflow and the Component Model

There is a development in workflow that workflow tools could provide a vital element in the co-ordination of distributed components within different domains. In a WFMS new components can be added to provide the necessary functions of new business processes and their related activities. Component integration is therefore an area of importance for WFMS. This integration of components has some key requirements:

- Ability to integrate with legacy systems in a cost effective way

- Ability for components to interoperate even if they have been implemented using different programming techniques.
- Ability for components to interoperate even when they offer interfaces that have been defined in different languages, e.g. IDL, ODL, GDMO or SMI.
- An integration mechanism that minimises the knowledge needed of other potential interoperating components when a new component is developed.
- The need for an integration mechanism that clearly supports the needs of specific business processes in a clearly observable manner.
- The need for an integration mechanism that is robust to changes in technology.
- The need for an integration mechanism that minimises the obstacles to adapting a component to a new application.

2.8.3 The CORBA Component Model

The CORBA Component model is part of the CORBA 3.0 specification. This has got much media coverage within the IT world which has yet to be released. The main extension to the CORBA 3.0 specification is the addition of the CORBA Component Model. This component model specifies a framework for the creation of plug-and-play CORBA objects. The integration of Enterprise JavaBeans is also an integral part of the component model, and the model will also help in the further integration of other object-based technologies. Enterprise JavaBeans (EJBs) will be discussed further in section 2.8.4.

The model is based on EJBs, but it extends the power of CORBA, in that this component model is intended to work with the other major programming languages such as C++, COBOL, Smalltalk, and ADA. This gives increased power to an organisation that has legacy systems developed that are still extremely useful to the organisation, and therefore avoids the need to redevelop these systems, saving time and money, which is of great importance to an organisation. These legacy systems can now be exposed to the developers within an organisation, thereby creating one of the objectives of the component model, which is software reuse.

The Component Model expresses the component as a type in CORBA IDL. The type definition provides both compile-time and run-time information on its external interfaces. This includes new IDL syntax to provide:

- Unique component identification
- Identification of interfaces that the components both provides and uses
- Details on the events that a component both emits and consumes
- Navigation interfaces that allow the above to examined
- Interfaces that allow runtime attribute and property configuration
- Interfaces for managing multiple component instances

2.8.4 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) component model logically extends the JavaBeans component model to support server components. Server components are reusable, pre-packaged pieces of application functionality that are designed to run in an application server [EJB98].

The main advantage that EJB has over the other component models is that it manages the middleware services for the components themselves, and as a result removes some of the development from the application developer. There is no need for the developer to be concerned with services such as lifecycle, state management, security, transactions, and persistence. The EJB model manages these services. Application development is therefore speeded up, as the developer does not have to waste time on the complex middleware. The EJB model is very versatile and is able to integrate and interoperate with environments that are compliant with the EJB model. This reinforces the ‘Write Once, Run Anywhere’ objective of the Java language.

A container is where the components execute and this container manages the components. In practical terms a container provides an operating system process or thread in which to execute the component [EJB98].

3. Requirements

The development of a prototype for this dissertation was implemented in conjunction with the development of the Serene Workflow Engine Version 2. The architecture of this version is as described in section 2.6. The requirements that were stated at the outset of the project were therefore requirements that were essential for the integration of the Shared Data Exchange (SDE) with the Serene Workflow Engine. Initially the investigation into data exchange in a componentised workflow environment focused on the prototypes that have already been developed, and then a prototype data exchange server was developed.

3.1 General Requirements for Integration of Data

Analysis of Data Flow

For any integration of a data store the initial step is the analysis of the actual data flow. This entails identifying activities, and then the input data they use and the output data that the activity produces. There are various methods for accomplishing this, and the best form of analysis is a diagrammatic representation of the data flow which the experts, that are familiar with the activities and flow of data in a system, can render comprehensible to the developer. A data flow diagram shows how activities depend on one another for their information. From the data flow a data model can then be designed. The objective is that the data store should always be consistent with the data model. It is essential that this be done correctly as an incorrect data model will make it more difficult to rectify later on during the development of an application.

Integrating the Application

The limitations of a data store must also be acknowledged. The objective of a data store is to supply applications with a mode of storing data and retrieving data. There is only a certain amount of intelligence that can be built into the data store, and this so-called ‘intelligence’ must be derived from the data model. It is possible that if

middleware is being used to develop an application to allow for the retrieval and storage of data, then there can be intelligence built into the application code that separates the middleware from the Database Management System (DBMS). The DBMS itself is limited in its functionality in providing intelligence. Intelligence can be defined as the manipulation of the data so that it can be returned to the requestor in the format, or type, that the requestor might need it in, and also the storage requirement might mean that the data would need to be manipulated in some way so as to make storage possible. One of the reasons for the necessity of such requirements is brought about by the use of object-oriented languages, and the DBMS that are also used in the integration. They might not be interoperable and some manipulation might be essential to make the applications work together.

Need to be Able to Store Outputs, Retrieve Inputs and Retrieve Outputs

The major and defining requirement of Data Integration is to be able to store data and then retrieve then data when necessary.

Provide SDE with Application Code for Storing and Retrieving Data

One of the requirements is to provide a server that implements the above requirements of storing and retrieving data. This is to act as a bridge between the chosen DBMS and the components that invoke the methods to store and retrieve data. This implementation should be opaque to the components that call it, that is to say there is no necessity that the components see the implementation code, or understand how it carries out the storing and retrieving of data. All they require is that the implementation works.

Provide Interface to the SDE

This follows on from the above requirement. The components will only see an interface that will allow it to call methods for storing and retrieving. The requirements for the interface will be discussed further in section 3.2.

Provide a Means of Preventing Concurrency Problems

In a distributed environment occasions arise when two users might try and update the same record at the same time in the database. This can cause problems in relation to

the data integrity. This must be avoided so that any updates, or storage, or retrieval of the data must include a blocking function so that only one update occurs at a time.

Avoiding Deadlock in a Distributed Database

Deadlock must be avoided at all cost. Deadlock occurs when a user could hold a lock on a table on one computer, and be waiting for a resource on a different computer. Another user could be waiting for the resource that the first user has locked, and they themselves have locked the resource that the first user is waiting for. The second user is unable to release until the first releases. The deadlock problems have to be identified so that they can be avoided.

Hardware Requirements

The platform that the data integration is going to be carried out on must be taken into account. The DBMS should be chosen depending on the platform used.

3.2 Data Integration in the Serene Architecture

The Serene architecture has already been described in section 2.6. At the outset of the project there were specific requirements that were given to allow the integration of a Shared Data Exchange (SDE), which were important to adhere to, as the integration of the SDE would have been all the more difficult if these requirements had not been followed.

Hardware Requirements

The application is to be built on the Windows NT platform, as the Serene workflow engine has been developed and implemented on NT.

IDL to Provide the Interface to the Server

The first step in the development is to design an IDL interface so that the other components would know the interface to use in order that they could implement the storing and retrieving functions of the SDE. This interface allows the implementation of the SDE to remain opaque to the components of the workflow engine

Store the Output of an Activity

When an activity is completed in a business process there are generally outputs, and these outputs should be stored in some form of data store. This is one of the major requirements, as the project is based on data exchange, so a mode of storing was important to achieve the goal of data exchange.

Retrieve the Input of an Activity

An activity in general needs some parameters so as to complete, so there was a prerequisite that there be some method in the interface to be able to retrieve the inputs of an activity.

Retrieve the Output of an Activity

On occasion it is necessary to retrieve the output of an activity that has previously been stored and it was requested that there be an interface that reflected this.

Data Flow Model

In order to gain a better understanding of the data in the business process under examination, a data flow model would have to be extracted. This involves sitting with the people who understand the data and can then provide expertise on the data in the system that will be stored, and retrieved. The data flow model is therefore to be implemented in the chosen DBMS. This data flow model should reflect the data store limitations, and aid in the storage and retrieval of data.

Develop the Interface Using Orbixweb

As the middleware used in the Serene workflow engine is OrbixWeb, which is based on the CORBA standard, the IDL's produced must be compiled using OrbixWeb, and any communication across networks, or between the components is to be based on the

CORBA standard. The CORBA specification used in this project is 2.3. All coding on the server side has to take into account that OrbixWeb is the middleware.

Provide DBMS

The DBMS that is to be used is Microsoft Access 97. This requirement was specified at the outset of the project.

Provide SDE that Implements Store and Retrieve Methods

The server that is to be developed must provide the implementation of the methods described in the interface. These methods are the three that are described above, i.e. store output, retrieve inputs, and retrieve outputs. The implementation of these functions should not concern the components that invoke them.

Need to be able to Return the Inputs as Required by an Activity

The inputs of an activity may have to be built on the server side so that they are compatible with the required input of an activity. This may require hand-coding the retrieved inputs of an activity. This should also be opaque to the components that invoke the methods defined in the interface.

3.3 Different Approaches to Data Integration

3.3.1 File Based Storage Approach

The SCDS of version 1 of the Serene workflow engine allowed components to invoke methods on a server implementation class that stored and retrieved data. This server was called the Shared Application Data (SAD) Server. The function of this was limited to storing only objects that were serialisable. The objects that were stored and retrieved were CORBA objects of type ‘Any’, which is a class defined in CORBA and also in the API’s of Java. These objects of type ‘Any’ allow any other type of object to be packed into them. This allows for a generic mode of transferral from a component to the data store. The objects themselves were serialised to a file by the SAD server, where they were stored.

The retrieval method that was invoked would read the serialised object from the file where they were stored, and then the SAD server implementation class would unserialise them, and pack them into the container type CORBA ‘Any’. The object of type ‘Any’ would then be passed back to the component that invoked the method. The component could then unpack the object and use the input to complete an activity. The details of the type ‘Any’ will be examined in greater detail in the implementation section.

One major requirement for this approach to data storage is that all objects that were to be stored had to be serialisable. This meant that all the objects needed to implement the *serializable* class defined in Java. This meant editing all the classes that defined these objects. Another requirement for the SAD server to work meant that the objects passed to the SAD server were to be of the type CORBA ‘Any’. The server had to have the ability to accept CORBA objects, and also know what types were packed in the ‘Any’.

3.3.2 DBMS Storage Approach

Once an activity in a business process has completed the data that it generates is considered to be the output of the activity. There must exist some mechanism to allow for this output to be stored persistently. Persistency storage, in this project, entailed using a DBMS to provide this functionality. By providing persistent storage the Serene workflow engine would be enabled with a data store that can be used in the future for further querying and investigation. The obvious prerequisite of persistency storage is that each output of an activity must be stored.

A component must be able to pass in an object to the server that is implementing the storage function. This object will be a CORBA object of type ‘Any’. On storage of an output the method that is invoked on the server side must avoid concurrency problems, so that when one component is storing an output, no other component can store an output of an activity to the DBMS. The DBMS that is used for persistent storage must be able to store the CORBA objects with an index, or key that is unique for each object. This requires there to be some mode of creating keys or indexes for

each CORBA object. The CORBA objects that should be assigned the key are the CORBA objects that contain the output of an activity.

3.3.3 Memory Cache Approach

A memory cache approach is an intermediate solution, between a transient data solution and a persistent solution. The main requirements are similar to the requirements in the above section, for the persistent DBMS storage approach. When a call is made to the server that implements the storage functionality then the output of an activity should be stored in the memory cache.

The memory cache must be able to store objects of type CORBA ‘Any’. These are the types that are passed in from the components that wish to store the output of an activity. The output of an activity is contained within the CORBA ‘Any’. These objects must have an index, or key, that is unique for each activity.

3.3.4 Cache & DBMS Approach

The combination of the persistent storage approach, and the memory cache approach is seen as beneficial to the development of data storage. It can increase the speed up retrieval and have persistent storage of the data. If there are any problems with the memory cache then the DBMS will always contain the data that is being searched for. The DBMS is almost a backup to the memory cache. The storage of data occurs in both the memory cache and the DBMS. If data is being retrieved and it is located in the memory cache then the DBMS is not searched, thereby saving valuable time.

4. Design

4.1 Overview

The overriding requirements for this project is that for each activity in a business process it is possible to store the output of a completed activity, and an activity is also able to retrieve its input so that it can complete its activity. An extra requirement that developed during the project was that a component should be able to invoke a method that would return the output of a completed activity that is stored in the data store.

The basic design is as shown in figure 11. As can be seen from the diagram a component invokes the store interface on the server. The server then implements this method and stores data to the database. Then for the retrieve method the component invokes this method on the server. The server, as was done for the store method implements this method and retrieves data from the database.

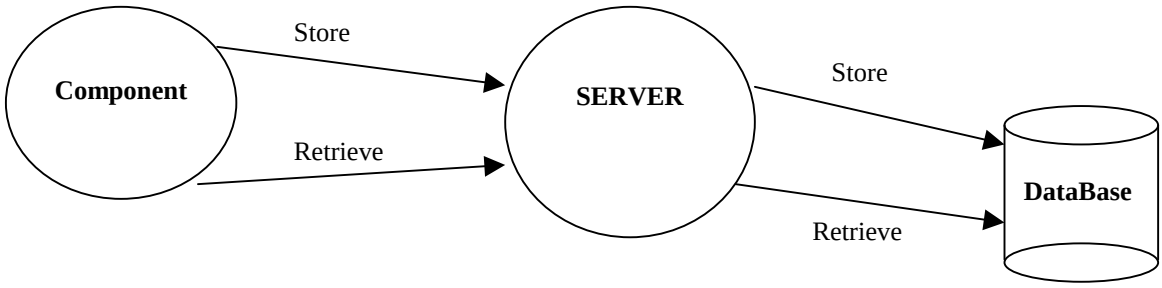

Figure 11: Basic design of the SDE

In figure 12 a diagram of the data flow is shown. This data flow represents the flow of data for the store function. It is the output of an activity that is being stored to the database.

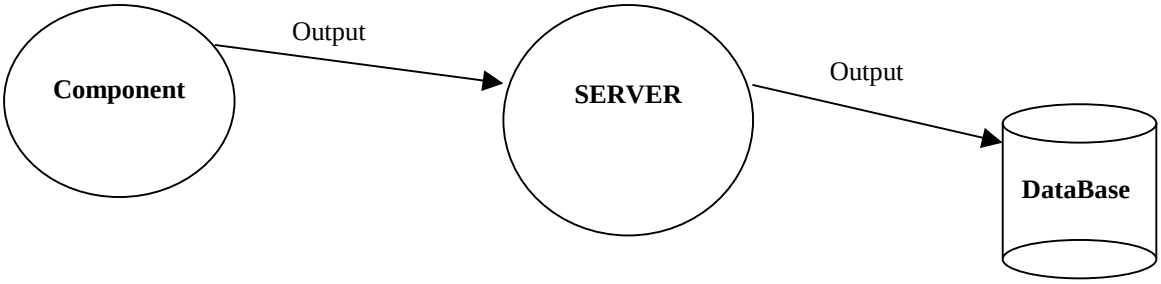

Figure 12:Data flow of the Store Method

In figure 13 the data flow is shown for the retrieve method. There are two forms of the retrieve method: 1) retrieve the input for an activity that a component is carrying out, and 2) retrieve the output of an activity that has already been completed. It is important here to distinguish between the two because the first retrieval function returns the inputs for an activity, whereas the second retrieval method is distinct in that it retrieves the output of an activity that has already completed and stored its output. Accordingly the input of an activity is not generally the output of a preceding activity. The input for an activity may take some of its variables, not all, from the output of the preceding activity, and/or other activities.

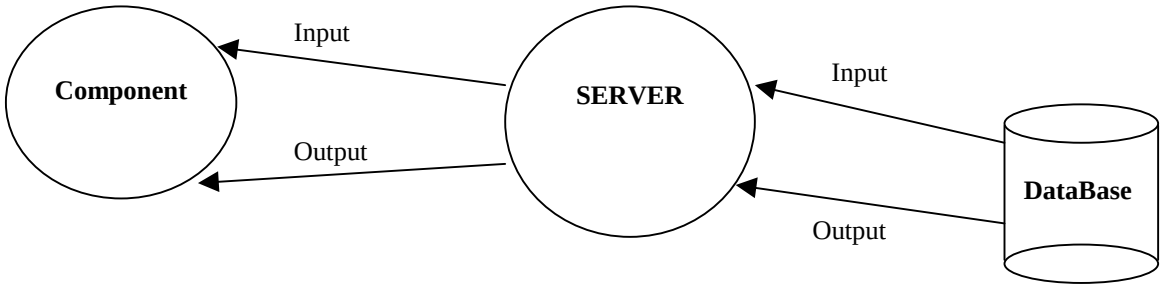

Figure 13:Data flow of the Retrieve Methods

A summary of these three functions is as follows:

- store
- retrieve
- getActivityOutput

It was also an objective to build intelligence into the SDE rather than the Adaptors in the Serene WFMS. See section 2.6.5 for an explanation of the Adaptors. These invoke the component applications and call the SDE to request the data that a component will need for an activity. The more intelligence that was built into the SDE the less the Serene workflow engine would have to do, and this was one of the requirements of the SDE. So the design had to attain this objective.

4.2 Comparison to Related Research

The research that was carried out into data exchange in workflow engine prototypes turned up very little information on the design of the DBMSs that support the workflow engines or the handling of production data. The weight given to the production data, i.e. the data that an activity needs to complete is small in comparison to the emphasis given to control data in a workflow engine. Indeed the prototypes Mentor, Mentor-Lite, and OrbWork view the area of production data as orthogonal to issues that they believe to be important in the design of the workflow engine. Mentor-Lite was developed as an integrable system, so that any of the invoked applications that would carry out an activity already had their own forms of data stores set up. There was no discussion on what these data stored were, but one would assume that they were standard DBMS.

OrbWork uses the persistent object services of CORBA, and again there was no discussion on what type of data store there was. Panta-Rhei has a backend DBMS that passes the production data along with all the other data that an activity might need, but the method of data exchange was never discussed in the research papers available. The only prototype that was relevant to the area that this project undertook to investigate was the Exotica/FMQM prototype that is discussed in section 2.5.4. Lotus Notes is used as the data store and the production data is separated from the control data that activities use. The production data is distributed to suit the Exotica workflow engine, which operates in a distributed environment. The Serene workflow engine also operates in a distributed environment. The Serene project also views that the production data the WFMS needs should be a separate component to the other

components in the WFMS. This has resulted in the SDE being designed as a separate component to the workflow engine, which itself is a componentised application. As each new activity in the Exotica/FMQM project is initiated then the data that is required for the activity is replicated and passed to the next node of the WFMS. This is not the view of the design that covers the SDE for the Serene workflow engine. The data is centralised in a DBMS so there is no method of replication, but it works within a distributed environment, in that the data can be called from any other component that needs the input to an activity in a distributed environment. The design of the DBMS for the SDE does not exclude the fact that it can be developed as a distributed database. The SDE has been designed with the basis for extensibility, and it can be viewed that the extensible function would be resolved by using a distributed database, or a replicated database. The replication of the data would involve looking at each component to analyse the data that each component would use and replicating only the data that the component would need to use. However the design for the SDE in this project though was a centralised database, with distributed access.

So overall there are very few benchmarks that this project can be evaluated against. The Exotica/FMQM prototype was the only one specifically developed to investigate distributed data management in a workflow environment. The data though was based on Lotus Notes, which used forms to pass data.

4.3 Accounting Business Process

The SDE has been designed with the Accounting Business Process as the basis for development; therefore it is important that one has a better understanding of this business process. The Accounting Business Process is a business process that was developed to support the billing of multimedia telecommunications services. It was specified in an EU research project and called Flowthru. It is a business process that contains six different activities. Figure 14 is a graphical representation of the Accounting Business Process.

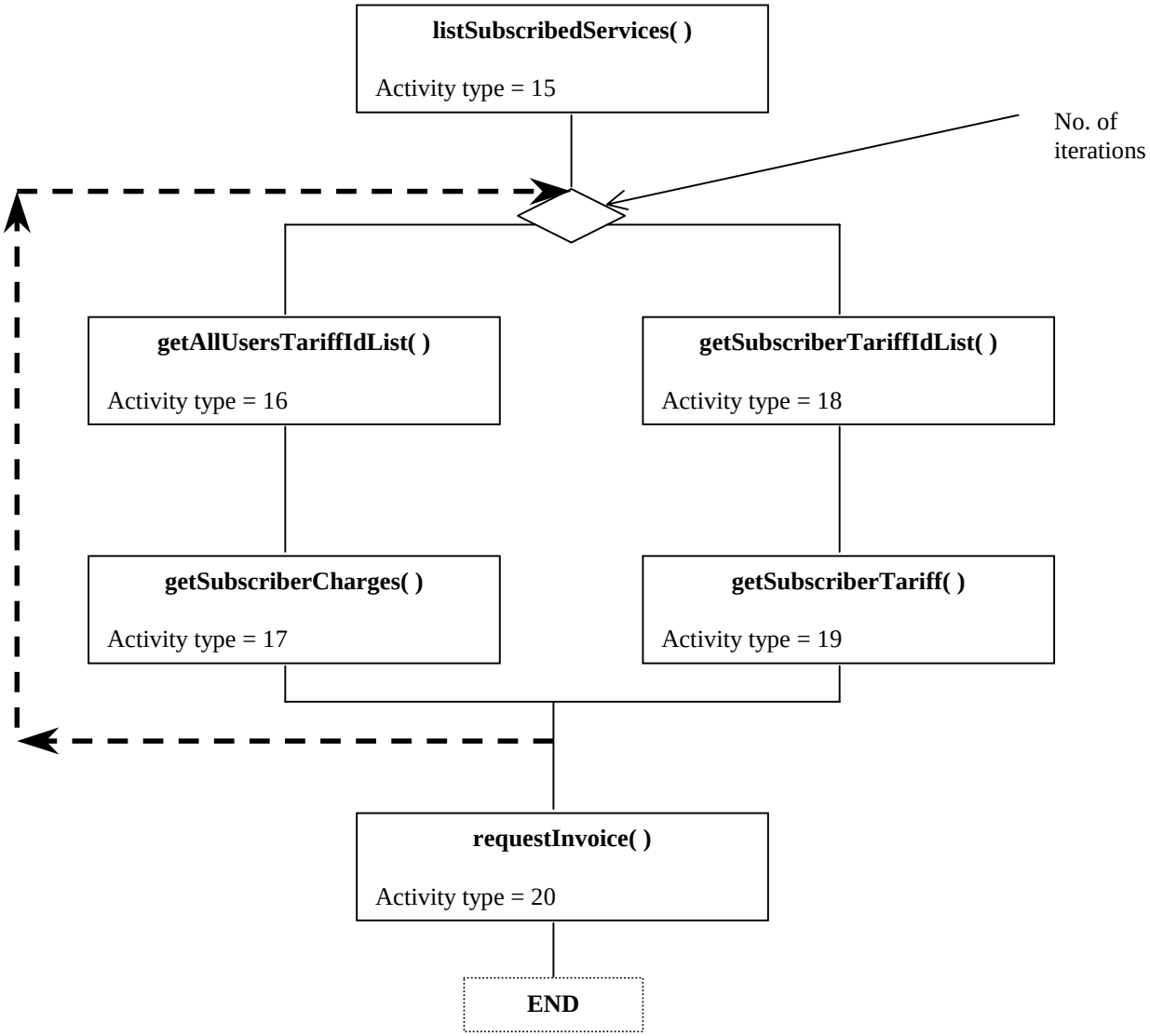

Figure 14: The Accounting Business Process (Flowthru)

Following is a description of each of the activities:

- **listSubscribedServices( ):** List of subscribed services for a particular subscriber (company).
- **getAllUsersTariffIdList( ):** A tariff ID list is returned for each user in this subscriber list for a particular service.
- **getSubscriberCharges( ):** The tariff is input to get the charges for a subscriber.
- **getSubscriberTariffIdList( ):** Get the tariff list for a particular subscriber for this service

- **getSubscriberTariff( ):** Get the tariff information
- **requestInvoice( ):** Generate full invoice

4.4 Data Flow Model

Figure 15 represents the data flow in the Accounting Business Process, and it is important for this data flow diagram to be understood so that the complexity of the data flow in the business process is perceived and also the complexity of the data itself. One bonus at the outset of the project is that there was no requirement to propose a indexing for the data. Previous to any development there was an indexing method set up. This was a simple, and easily understood model. When a business process is instantiated by the Serene workflow engine, it is assigned a unique ID, the Process Instance ID (PIid). This is a numerical code and is unique for every instance of a business process. Also when an activity is instantiated it is assigned a unique ID, called the Activity Instance ID (AIid) which is also a numerical value.

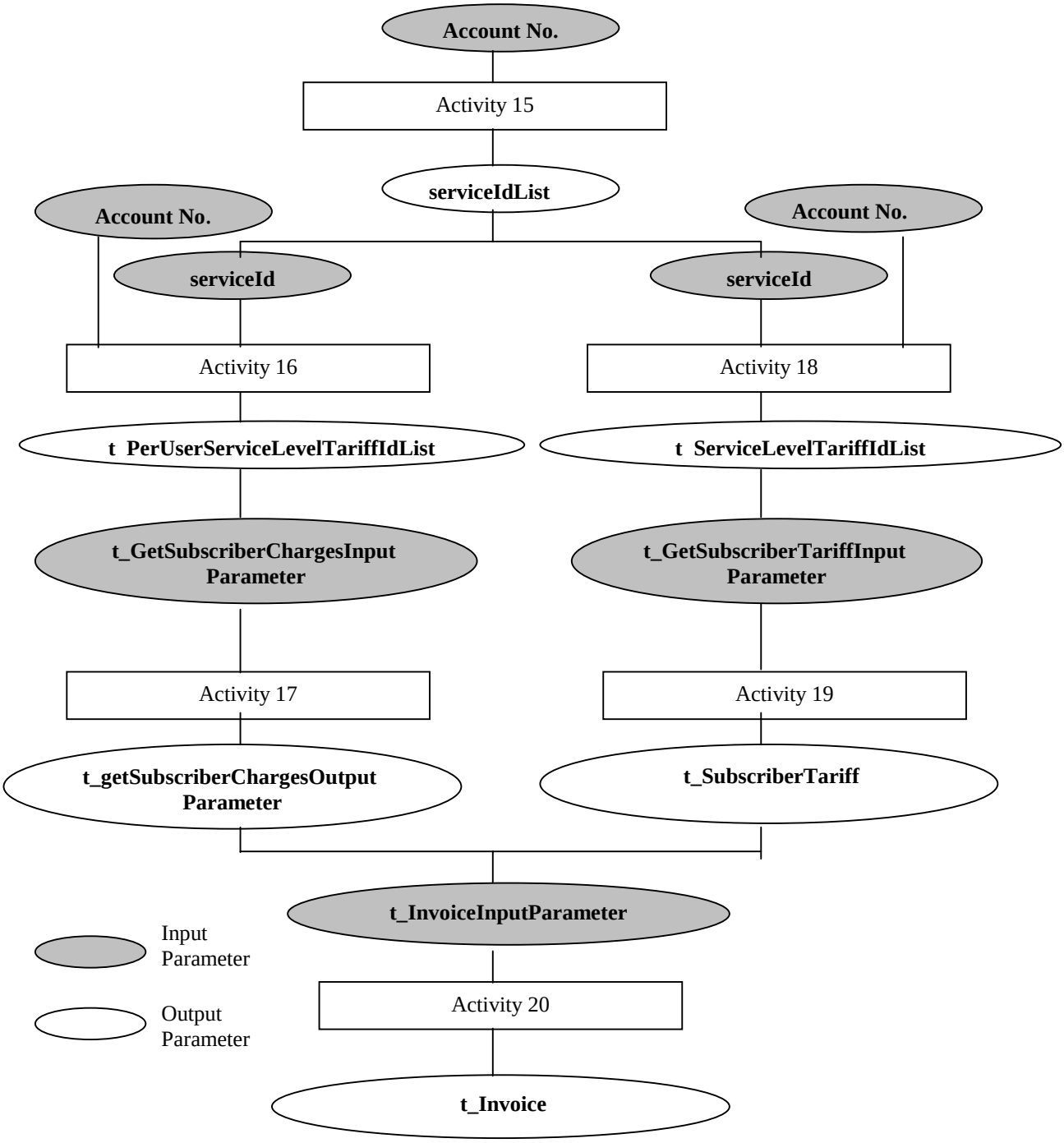

Figure 15: Data Flow Analysis of the Accounting Business Process

As can be seen from figure 15 for each activity there exists an input, and an output. These input and output parameters are user defined user defined types. These types can be viewed in Appendix A. They are defined in IDLs, and were not part of this project but were developed separately with a view to be used by the SDE. An example of one of the input parameters is t_GetSubscriberChargesInputParameter, which is the input for activity type 17. This is made up of the variables as shown in table 2.

Variable	***Type***
SubscriberId	String
TidList	output of activity 16
ServiceId	iteration (integer)
FromTo	Complex date/time type structure
CurrencyUnit	String

Table 2: Variables that make up the input parameter for activity 17

So the input for activity 17 is not a simple string or integer, but a complex data structure. As can be seen from the table 2, the variable *tidList* is the output of the preceding activity, which is activity 16. The variables *subscriberId*, *fromTo*, and *currencyUnit* are three inputs that must be input by the user at the beginning of the business process. They are called the Business Process Inputs, and are those inputs that are not contained in any of the outputs, of any of the activities in the business process. They are not generated by any activity and must therefore be generated by a user of the system. The variable *serviceId* represents the iteration that the activity is on, and is another parameter that is not generated by any activity. This variable will be gathered from the SDE database, and the design of this will be discussed later.

These data types that have been predefined in IDLs are called CORBA *structs*, and they allow for nesting of types. CORBA *structs* are defined in the IDL interfaces so that a developer of an application can create their own types. They are user-defined types and can have deep nesting structures if desired. A *struct* allows one to form an aggregate structure of variables, which may be of the same or different types [Iona98].

This allows for more complex variables, and a bit more flexibility for the application developer. More details of the complexity of the data types that have been defined for the accounting business process will be discussed, and demonstrated in the implementation section (Chapter 5).

4.5 Design of the Shared Data Exchange (SDE)

The overview of this chapter gave a brief description of the design of the SDE. Figure 16 gives a more detailed design. The major difference is that there is a memory cache included in the design.

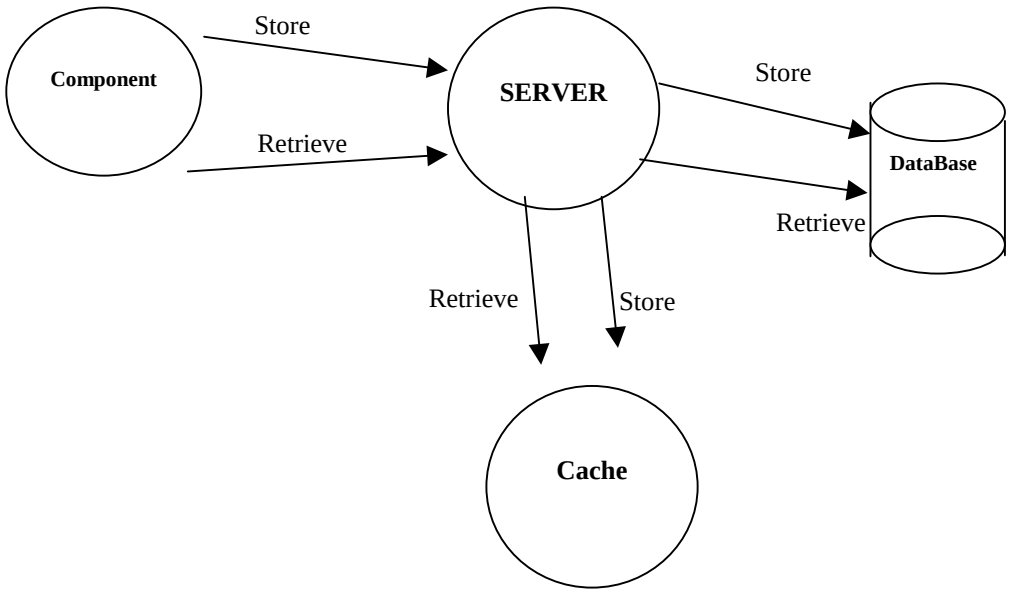

Figure 16: Design of the SDE

The introduction of the memory cache allows for a speed-up in performance. When the server is passed the data that it has been requested to store, this data is stored in the database and the memory cache. So, when the server is invoked by a component to retrieve data, the memory cache is searched first for the data. If the data that is being requested resides within the cache, then this data is returned to the component and the database is not searched. If for some reason, e.g. the server goes down resulting in the

memory cache being deleted, the retrieval method can now fall back on the database. The cache should always be checked first, and if there is no data matching the index that is being used to search the cache, then calls should be made to the database, and if the data is there, it is returned to the component. Calls to the database take more time than calls to a memory cache, so it is viewed that the cache will improve performance rates in the system.

As discussed in section 4.3 the introduction of a business process that has activities involving iterations introduces complexity into the data flow model. Also as discussed in section 4.4 the data flow model is further complicated by the fact that the original view of data flow from one activity to the next is an incorrect picture of the actual data flow. The data for the input of an activity can come from various activities, indeed some of the data needed must be manually inserted into the database, so that an activity can use that data to proceed. There exists as a result two more design requirements, which are to have (1) some functionality that can keep a track of the iteration that the loop in the business process is on, and (2) a bridge between the

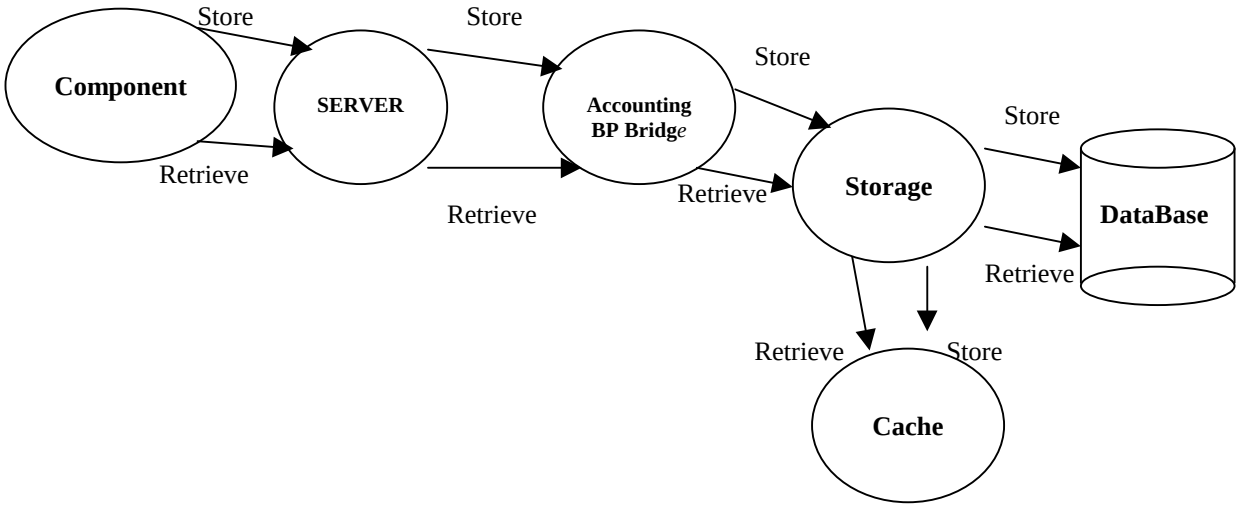

Figure 17: Design of the SDE including the Accounting Business Process Bridge

server and the persistent storage so that the inputs for an activity can be built up. This new design is shown in figure 17.

In order to keep track of the iteration that the Accounting business process is on, there were two options that were available on storing this data. The first was to store the iteration number in a memory cache. The second was to store the information in the database. The second option was decided upon. It was seen as an advantageous choice over the memory cache. Choosing the second design over the first was due to the fact that if the server crashed, or any failure occurred in the application, at least in the database there would be a permanent record of what iteration the Accounting business process was on. Therefore it would be possible to recover from a point of failure. If this information remained in the memory cache there could be the possibility of losing it, and this might result in the whole business process being restarted, which would waste time and require a clean-up of the data that had already been stored in the database. This clean-up would be as a result of the fact the data is stored under a unique index, which is the Process Instance ID (PIid) and the Activity Instance ID (AIid). If data already stored had not been deleted then the indexes, that were already used to identify the data, could not be used again. At least with the second option the business process can be rolled back to the last activity completed, with the metadata remaining in the database. There would be an obvious reduction in performance time due to DBMS access rates but this is a necessary handicap in gaining extra functionality.

The Accounting BP bridge in figure 17 must be able to take in various outputs that have been stored in the database, and extract from them the data needed for each activity. This is only necessary for the retrieval of inputs for an activity. So for each activity as shown in figure 15 the data input needs to follow this process:

- retrieve the outputs of activities that contain the essential data;
- extract the data from these outputs;
- insert the data into the predefined input types for each activity.

This is where the generic store and retrieval methods in the original design fall down. Originally it was hoped that there would be no need for the SDE to know what was been stored in the data store. Data was to be passed in and stored, and then retrieved. The design involves storing the data in a generic fashion, therefore adhering to the

original generic view. Unfortunately a generic retrieval method was then found unattainable, as the data flow model didn't allow it, and so as stated above the inputs for an activity have to built up for each activity.

It is important to point out that the persistent storage mechanism and the memory storage mechanism have been designed separately. The database can be considered as a separate component itself, leaving the possibility that it can be replaced with some other persistent storage service. This allows for further development of the storage mechanism.

4.6 Database Design

In section 2.7.4 the different types of data integration were discussed. It would appear from this analysis that the preferred objective would be a distributed database. However for the development of a prototype a centralised database with distributed access was chosen to be the grounding for investigating the data exchange between components. There is distributed access to the database over the network from the different components that make up the Accounting Business Process. Having a centralised database is not the ideal solution. A distributed database would have been preferable, but for the sake of prototype development the design of a centralised database was chosen. As the traffic to the centralised database is low then the centralised database solution can handle the traffic involved in the Accounting Business Process. Another factor in having a centralised database is the fact that the testing of the SDE was going to be in conjunction with the Accounting Business Process. So there was only one business process that was going to be tested. If there were more business processes then the complexity of the database design would have to be increased, and a distributed database should then be taken into account in the design. The database design itself is not very complex, as there are two functions that it really has to serve. The first, just to reiterate, is a store function and the second is a retrieve function. As the DBMS is a relational DBMS (RDBMS) the tables that would hold the data and the metadata had to be carefully designed.

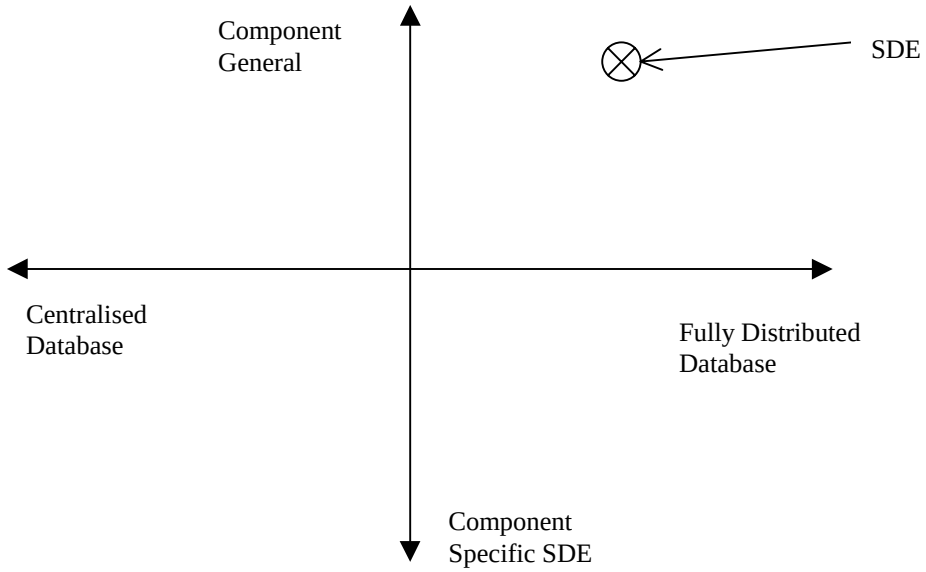

Figure 18: Graphical Representation of Database Design Solutions

Figure 18 is a graphical representation of the solutions available in the database design. The SDE is placed on this graph, and it can be described as a Component General, Centralised Database with Distributed Access. Component General relates to the fact that the SDE is designed to store an output to the database and can remain ignorant of the component that is invoking it, i.e. it is not component specific. As discussed before it was decided to use a centralised database, with distributed access.

The data model section discloses the fact that there is already an indexing system in place. This is a combination of the Process Instance ID (PIid), and the Activity Instance ID (AIid). So one table should contain these two indexes as the key, and therefore the combination of the PIid and the AIid is considered unique. This is all that is needed to satisfy the requirements of the store function. Remember that the function will store the output of an activity along with its PIid and AIid. A single table represents this.

The retrieve function is somewhat more complex, and the added fact of having iterations in the business process complicates the retrieval function even further. The iteration factor, which is the number that represents the number of iterations that have been completed in the business process, must be brought into the design of the database, as this is where the metadata is going to be stored. For the SDE to be able to

query the iteration factor, is one table to represent this factor. Another table is necessary though, and this table will hold metadata that will allow the SDE to know what activity instance belongs to which iteration. These tables and their implementation will be described fully in the next chapter. The goal was to try and build as little complexity into the database so that the minimum number of calls would be made to it, and build the intelligence into the application code. This should improve performance of the SDE.

4.7 Summary

The persistency storage component of the SDE has been designed in such a way to allow for its replacement in any future versions of the SDE. Having both a memory cache and a DBMS improves the functionality and the design of the SDE. The data that is stored in the data store, be it the memory cache or the DBMS is complex, and consists of nested data structures, so the SDE needs to be able to interpret these data structures. The business process that was chosen to be used as the test-bed for testing the Serene workflow engine and the SDE proved to be a complex business process because of the iterative loop in the process itself. Therefore this complexity must be taken into account in the design of the SDE.

5. Implementation

This section will bring the design and requirements together, outlining the choices that have been taken in order to develop the prototype the way that it has been developed for this project.

5.1 Technologies Used in the project

In section 1, the Introduction, the technologies that are used in this project were outlined. All the technologies chosen for the project are given below and the reasons for choosing them are also given.

Java™

All application code is developed in the Java programming language that was developed by Sun Microsystems. This was specified as a requirement of the project, as the Serene workflow engine has been developed in Java. Java allows the workflow engine therefore to be written once, and can run anywhere. This is one of the major advantages of Java, and results in it being a very useful language in enabling distributed object technology.

JDBC

Java Database Connectivity (JDBC) is an API of the Java programming language that allows a Java developer execute queries and update tables in a variety of DBMSs. It is a new API developed for Java and the only database connectivity API that allows communication of the application code with a database. Therefore it was compulsory to use this technology. JDBC is a ‘‘low-level” interface, which means that it is used to invoke (or ‘‘call”) SQL commands directly. It works very well in this capacity and is easier to use than other database connectivity APIs, but it was designed also to be a base upon which to build higher-level interfaces and tools. A higher-level interface is ‘‘user-friendly,” using a more understandable or more convenient API that is translated behind the scenes into a low-level interface such as JDBC [JDBC99].

Microsoft Access 97

This DBMS was chosen to act as the data store for the SDE. One of the reasons that it was chosen was because of its portability, and also it is not necessary to buy in a new DBMS. Also most organisations would have Access installed on their networks, so again this would generally add no extra cost in using this DBMS. As the SDE is to be developed on Windows NT, Access therefore has a very good management interface on this platform.

JDBC-ODBC Driver

The JDBC-ODBC Bridge is a JDBC driver that implements JDBC operations by translating them into ODBC operations. To ODBC it appears as a normal application program. The Bridge implements JDBC for any database for which an ODBC driver is available. The Bridge is implemented as the sun.jdbc.odbc Java package and contains a native library used to access ODBC. [JDBC99] This Bridge allows JDBC to connect to an Access database, and therefore can query the database and store to it. The JDBC-ODBC driver must be installed on the server side to allow the SDE to access the database.

SQL

Structured Query Language (also known as SQL) allows users to access data in RDBMS, such as Oracle, Sybase, Informix, Microsoft SQL Server, Access, and others, by allowing users to describe the data the user wishes to see. SQL also allows users to define the data in a database, and manipulate that data. SQL is used in the prototype in conjunction with JDBC. JDBC allows a developer to create an SQL statement, which is then sent to the DBMS, in this case Microsoft Access, over the JDBC-ODBC driver. The DBMS then executes the SQL statement and returns the result as an object to the application code that called it. JDBC interprets this object.

Orbixweb

OrbixWeb is the middleware that was specified as a requirement at the outset of the project. In order to keep the development of the Serene WFMS homogeneous it was preferable to use this application. OrbixWeb allows clients to connect to servers, and

communicate with the server. It allows connectivity across networks, and the power to access applications written in different languages.

OrbixWeb is based on the CORBA standard. CORBA can be thought of as a communication bus for client-server objects. It is a three-tier distributed mechanism, and the terminology applies to a specific request. That is, if object A invokes a method on object B, A is the client and B is the server. If B then calls A, the roles are reversed. Exported server interfaces must be specified in the CORBA standard Interface Definition Language (IDL). An IDL interface description is then mapped using an IDL compiler to the native language bindings, which in this case is Java. The ORB is the mediator, responsible for brokering interactions between objects. Its job is to provide object location and access transparency by facilitating client invocations of methods on server objects. A client can connect – or bind – to a server.

5.2 SDE Implementation

5.2.1 CORBA Process

There are several steps that must be followed so that a CORBA server and client can be created.

1. Specify the server interface in IDL.
2. Run the IDL description through the IDL compiler which generates a native language interface, server stub, and client stub.
3. Implement the server.
4. Compile the server program and link in the server stub that was generated by the IDL compiler. The result is an executable server program that can accept method invocations via CORBA.
5. Register the server in the implementation repository. The server is now ready for activation.
6. Implement the client.
7. Compile the client and link in the client stub that was generated by the IDL

compiler.

8. When the client is executing, it uses the ORB to bind to the server object and obtain an object reference.
9. Using the object reference, the client invokes server object methods.[Segu98]

In this project it was necessary to develop clients that would bind to the SDE server. These clients were the components that had already been developed by the other members of the Serene team.

5.2.2 IDL Interface

The interface that is presented to the components that will use the Shared Data Exchange (SDE) is defined in CORBA IDL. This hides the implementation of the SDE itself from the components. There are three interfaces, defined in the IDL, matching the requirements of the storage and retrieval functions:

- `short store(in long PIid, in long AIid, in any o);`
- `short retrieve(in long PIid, in long AIid, out any i);`
- `short getActivityOutput(in long PIid, in long AIid, out any o);`

Each interface takes in as parameters the Process Instance ID (PIid), and the Activity Instance ID (AIid), so this is what the component that is calling any of the interfaces must pass in. The *store* function also takes in the data that the component wants to store. This is passed in as a CORBA ‘Any’. The two retrieval functions, *retrieve* and *getActivityOutput* also have an extra parameter in their parameter lists. These are defined as *out* parameters. This is a CORBA standard of passing back a result to the component/client from the server. They are passed back as a CORBA ‘Any’ in an ‘AnyHolder’. The ‘Any’ and ‘AnyHolder’ will be explained in more detail in the next section.

These interfaces should not be changed, and were not changed during the development of the prototype, and that is an important element of the IDL. Any other

developer that use the interfaces is required to add in the calls to the interface only once in a client and then the method calls that they used need never have changed. The IDL, called SDEServer.idl is compiled using the OrbixWeb IDL compiler, which generates the stubs in the Java programming language. These stubs allow for the communication between the client and the server.

5.2.3 CORBA 'Any'

There has been mention throughout this document of the CORBA type 'Any'. It is used to indicate that a value of an arbitrary type can be passed as a parameter or a return value. A client can construct an 'Any' to contain any type that can be specified in IDL. The client can then pass the 'Any' in a call to an interface defined in the IDL. Conceptually this class contains the following two instance variables:

- type
- value [Iona98]

An Any object must always be constructed using the ORB class, e.g.

```
Any a = ORB.init().create_any();
```

The AnyHolder is used in the *retrieve* function and the *getActivityOutput* function. Remember that these functions have an *out* parameter in their parameter lists, so that the data that a component has requested based on the PIid and the AIid can be sent back. The AnyHolder class is used in the server application code to pack an Any object into it and then send it back to the client. To use this method of returning a result in the parameter list the AnyHolder class must be used on the server side. This is a compulsory requirement of using the Any class, and the *out* parameter in IDL.

5.2.4 Data Description

It is worthwhile to describe the type of data that is being passed to the SDE and stored in the DBMS and the cache, which should result in a better understanding of the next section. When the SDE has to save the output, which is a CORBA 'Any', the component that wants to store it passes it to the SDE. It is stored in the cache as a CORBA 'Any', and the SDE converts this CORBA 'Any' to a string and saves this to

the DBMS. It is the stringified value of the object within the 'Any' that is saved and not an Interoperable Object Reference (IOR) to the 'Any'. If the 'Any' contains a base type or a CORBA *struct* the string will contain the values stored in the base type/*struct*. However if the 'Any' contains an object then the string will not contain the value of the object but rather its IOR, that is the IOR for the object stored in the 'Any', not the IOR of the 'Any' itself. A *struct* allows one to form an aggregate structure of variables, which may be of the same or different types [Iona98]. The *struct*s that were used to describe the data structures that the activities needed as inputs and the outputs for the Accounting Business Process are defined in the m_SdxTypes.idl as shown in Appendix A.1.

An example of the data that was stored is given is section 4.4. The data input is for activity type 17, and it is a *struct* of type t_GetSubscriberChargesInputParameter. When this is inserted into a CORBA 'Any' and then converted to a string to be stored in the DBMS it looks like this:

R~m_SdxTypes::t_GetSubscriberChargesOutputParameter~charges{S{R~m_Charge
Types::t_Charge~chargeId{0},userId{0},amountDue{f},currencyUnit{0},descriptive
Text{0}},0},currentSubscriberSLAId{0}CDR:0000000300000010736 16d706c65436
8617267654964310000000 0e73616d706c655573657249643100000 0004188cccd0000
0007657363756 46f0000000000126465736372697470697665207465787431000000000
000001073616d706c6543686172676549643200000000e73616d706c655573657249
643200000041899 99a000000066672616e6300000000000012646573637269747069 7
665207465787432000000000000010736 16d706c6543686172676549643300000000 0e
73616d706c655573657249643300000041 8a66660000000570756e740000000000000
012646573637269747069766520746578743300000000000000f736572764c6576656c
416772656500

As can be seen it is a nested data structure. Starting at the top of the tree of this data structure there is the type t_GetSubscriberChargesOutputParameter, which contains the type ChargeTypes, and then other types, which are strings and integers. Then there is also another variable currentSubscriberSLAId, which is a string and represents the subscriber – company – to the service.

5.2.5 Description of Classes

Figure 19 shows the classes that have been implemented for the SDE. A client or component binds to the server, and the server then carries out the component invocation. Following is a description of each class, and its function.

SDEserver1.java

This is the server program that instantiates the implementation class SDEMgt.java.

SDEMgt.java

This is the implementation class, which contains the three methods that have been defined in the IDL, i.e. *store*, *retrieve*, and *getActivityOutput*. This object is instantiated by the SDEserver object. The SDEMgt object instantiates the AcingBP object, and binds to the Workflow Manager server. The Workflow Manager is a server that allows access to a database that holds metadata for the Accounting Business Process.

The ***store*** method invokes a method on the Workflow Manager server, and the interface for this method is:

```
Get_Instances( in short AIid, in string client_info,
out short Pinst, out short ActType );
```

This method returns the type of the activity (ActType) that matches the AIid passed to the interface. Then a method on the AcingBP object is invoked and it passed the PIid, AIid, ProcType, ActType, and the output of a completed activity. The ProcType is a dummy value but has been left in for possible extensions of the SDE and additional business processes. It represents the Process Type of the business process represented by the PIid.

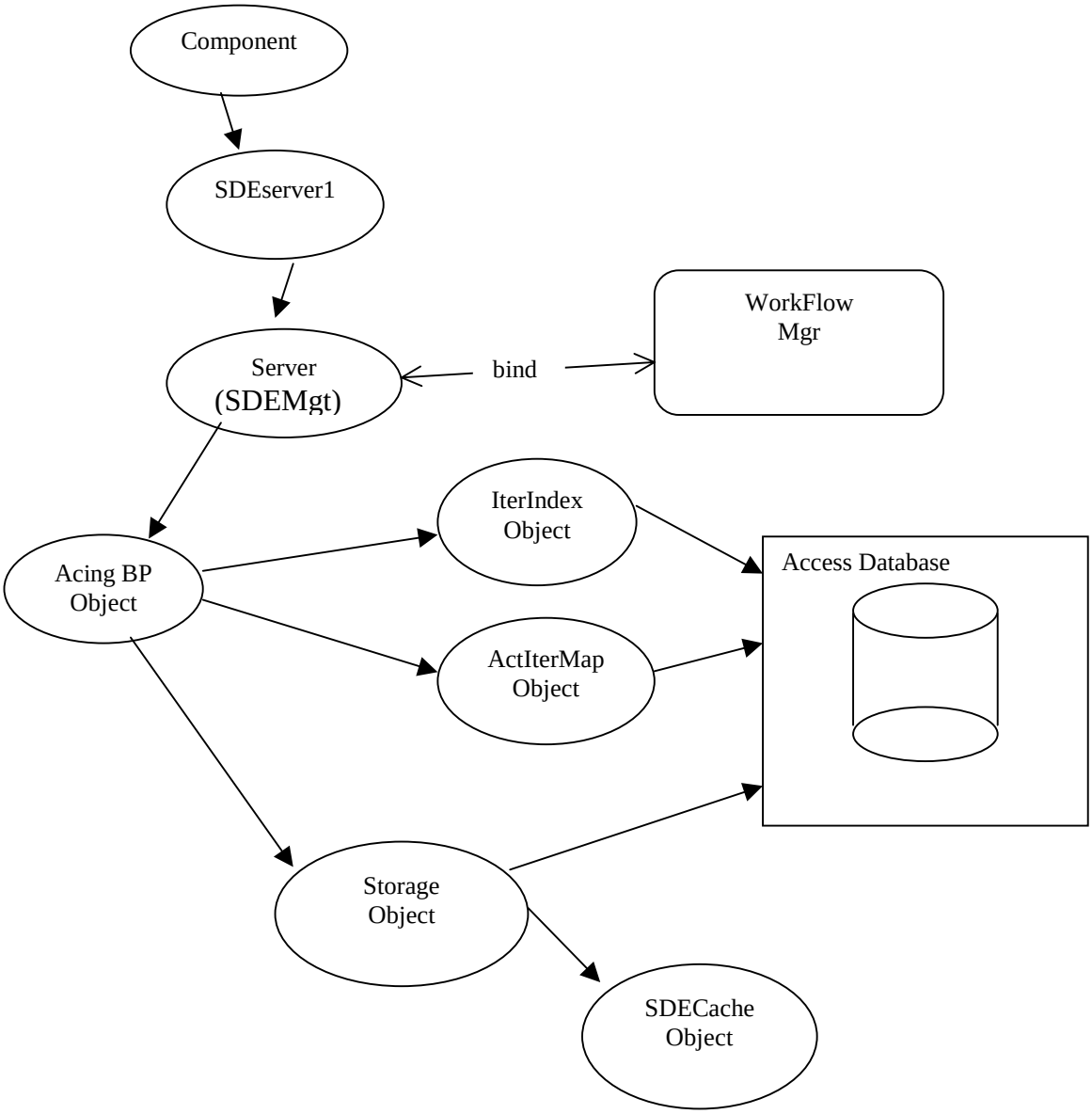

Figure 19: SDE Implementation – all classes

The ***retrieve*** method is passed in the PIid, the AIid, from the component that invokes the method, and this method sends backs to the component the input that it requested. An invocation on the Workflow Manager also occurs here. The ActType is again being requested for a particular AIid, as in the store method described above. The retrieve method on the AcingBP object is invoked and passed in the PIid, AIid, ProcType, and ActType. The returned result from this method is returned as a CORBA 'Any'. This is the input that has been requested by the component, and the SDEMgt inserts the CORBA 'Any' into an AnyHolder, which is returned to the component.

The ***getActivityOutput*** is the only method that doesn't invoke any methods on the Workflow Manager. It receives the PIid, and the AIid from the component invoking it. It then invokes its equivalent method on the AcingBP object (getActivityOutput), and passes in the AIid, and the PIid. The returned result is the output of a completed activity that has already been stored, and it is returned as a CORBA 'Any'. This result is then inserted into an AnyHolder, and this is sent back to the component that invoked the method.

AcingBP.java

This class instantiates three objects: IterIndex, ActIterMap, and Storage. It also has three methods: *store*, *retrieve*, and *getActivityOutput*.

The ***store*** method is passed in the parameters PIid, AIid, ProcType, and ActType. The ProcType is the type of the business process that is represented by the PIid (at the moment this is a dummy value, but it has been retained for further extension of the application). The ActType is the type of the activity that is represented by the AIid. A method is called on the IterIndex object to find out what iteration that the activity is on. This search for the iteration number is only made for the activities of type 16, 17, 18, and 19, as these are the only activities that are iterative. For a refresh of the business process refer to section 4.3. The equivalently named *store* method on the Storage object is then called.

The ***retrieve*** method is somewhat more complicated. This is where the inputs for an activity have to be "built". First off though a method on the IterIndex method is called, so as to increase the iteration number by one. This is only done for the activities of type 16, 17, 18, and 19, as these are the only activities that are part of the loop in the accounting business process. Then another method is invoked on the IterIndex object to find out the new iteration number, i.e. what iteration of the loop the activity is on. This iteration number is called the servcieId in the Accounting Business Process. Then there is a switch statement that has six different cases, one for each of the six different activities. Each case has one thing in common, namely the Business Process inputs that are retrieved from the database. Note that the Business Process

inputs are input into the database before the actual business process has begun. They are contained within a *struct* defined in CORBA IDL. The AIid that was chosen for the Business Process Inputs was zero, so as to distinguish it from the other AIids that are generated by the Serene workflow engine. The Business Process inputs have the same PIid for the business process that they are intended for, e.g. 1001. The PIid and AIid key is therefore 1001 and 0 - in this example - which uniquely identifies the business process inputs. The Serene development team agreed upon this ID.

For each of the activities, they then have their own data that needs to be retrieved from the DBMS, or from the cache if it exists in the cache. Basically the output of another activity is sometimes needed so there is a method that is used to find out the AIid of this output. If the activity is part of the loop in the business process then the AIid must be found using the map that exists in the database, and is accessed by the ActIterMap object. This mapping will be detailed further on. So once the AIid of the output is found, then the AcingBP object makes a call to the Storage object by invoking the getActivityOutput method on the Storage object. The output is retrieved as a CORBA 'Any', and then the necessary data is extracted from it. Once all the data that is required to build up the input for the activity has been retrieved it is then inserted into the input type that has been created for each activity. This input is then inserted into a CORBA 'Any', and then passed back to the SDEMgt object.

The ***getActivityOutput*** method is passed in two parameters, the PIid and the AIid. It then makes a call to the getActivityOutput method on the Storage object. It receives a CORBA 'Any' from this object, which is the output of an activity that has already been stored, and has the index of the combined PIid and the AIid. Then the output is passed back to the SDEMgt object, which then sends it back to the component.

IterIndex.java

This object is instantiated by the AcingBP object. It makes a connection using the JDBC API to the database, called SDE, which contains the tables that need to accessed by this class. It has two methods, setIndex and getIndex. These methods are used to keep a track of the number of iterations that have passed for an activity of type 16, 17, 18, and 19. It also stores an index for the two activities of type 15 and 20, but

as they are only completed once for each instance of a business process their iteration index will be stored as zero. The setIndex method updates the database table Tbl_IterIndex, which contains the iteration index. It checks the database by using the getIndex method and then adds one to the index. The index in the database table is then updated to this new value. Again this increase is only done for activities of type 16, 17, 18, and 19. The getIndex method retrieves the iteration index for an activity. It queries the database table Tbl_IterIndex also and using the PIid and the activity type retrieves the iteration index and passes this back. Both methods pass SQL statements to the database, which are used to query the database. This is where JDBC comes into play. The API java.sql has methods, which are used to send these SQL statements to the database, and methods that can extract from the results of the query sent back from the database.

ActIterMap.java

This object is instantiated by the AcingBP object and it makes a connection to the SDE database using the JDBC API, java.sql, across the JDBC-ODBC driver. There are two methods, setActId and getActId, and they update and query the database table Tbl_IterMap respectively. This is the class that is used to keep a map that contains the PIid , the activity type, the iteration that the activity is on, and the AIid. The setActId method inserts these parameters into the table Tbl_IterMap. The getActId method is passed in the parameters PIid, the activity type, and the iteration that the activity is on. It then queries the table Tbl_IterMap, using JDBC to send an SQL statement to the database, and retrieves the result from this query, the result being the AIid. It sends this AIid back as a result to the Acing BP object. This AIid is used in the AcingBP to find the output that has been stored with this AIid.

So the ActIterMap object is used to keep a store of a mapping that contains the iteration that an instance of an activity is on and its type, so that the AcingBP can find out the AIid of the activity that it is searching for.

Storage.java

This object is instantiated by the AcingBP object. This object carries out the storage to the DBMS and the memory cache. It also handles the retrieval of activity outputs that

have already been stored to the cache or DBMS. The constructor creates a connection to the SDE database (using JDBC to make a connection across a JDBC-ODBC driver), and creates the SDECache object. There are four methods in this class: store, store_p, getActivityOutput, and getActivityOutput_p.

The store method is passed in the PIid, the AIid, and the output that is to be stored. The output is received as a CORBA 'Any'. Then these three parameters are passed to the SDECache object, by using a method called addOutput on the SDECache object. Then store_p method is then called, and it is also passed the same three parameters. It is within this method, store_p, that the conversion of the CORBA 'Any' to a string occurs. Then the PIid, the AIid and the stringified value of the object within the CORBA 'Any' are stored in the database table called Tbl_Datastore. This storage to the database is done using a JDBC method call, so as to send a SQL INSERT statement to the table Tbl_Datastore. The output is received as type org.omg.CORBA.Any, but to convert to a string this output has to be cast to IE.Iona.OrbixWeb.CORBA.Any type, as the method toString() is defined within the IE.Iona.OrbixWeb.CORBA.Any package, and not in the org.omg.CORBA.Any package.

When the AcingBP object calls the getActivityOutput method, it is passed the PIid and the AIid. It first makes a call to the SDECache object to check the memory cache to see if there is an output that has been stored in the cache that has an index that matches the combined index of PIid and AIid passed in. If the output has been detected it is returned as a CORBA 'Any'. There is no need for this method to do any conversions as the output has been stored in the memory cache as a CORBA 'Any'. If however the output is not stored in the cache then the method getActivityOutput_p is called. It is passed the PIid and the AIid and uses these parameters to query the table Tbl_Datastore in the SDE database. Again JDBC methods are used to send SQL statements across the JDBC-ODBC driver to the database. The returned result is extracted from the result set that is retrieved from the database using methods defined in the JDBC package java.sql. The result is a string, which as stated above is the stringified value of the object within the CORBA 'Any'. The 'Any' is reconstructed in the following manner:

```
output.fromString (anyRef);
corOutput = ( org.omg.CORBA.Any )output;
```

The anyRef is the string for conversion. The resulting ‘Any’ that has been reconstructed is of type IE.Iona.OrbixWeb.CORBA.Any, and so needs to be cast to an ‘Any’ of type org.omg.CORBA.Any. This is necessary as it was agreed that the CORBA ‘Any’ that was to passed should be of type org.omg.CORBA.Any, as this is what the component originally passes in to the SDE. The output whether it has been retrieved from the SDECache object or the database is then returned to the AcingBP object as a CORBA ‘Any’.

SDECache.java

The SDECache object is instantiated by the Storage class. The SDECache class extends the java.util.Hashtable package. The cache is a hashtable of hashtables. The first hashtable contains the PIid as the index and the other field contains another hashtable. This second hastable has the AIid as the index, and the other field in this second hashtable contains the output of an activity that matches this combined index of the PIid and the AIid. This is demonstrated in figure 20.

The method addOutput stores the output and its associated PIid and AIid in the hashtables. The method findOutput is passed in the AIid and it searches the hashtables to find the output that has an index with the same AIid. If the output is found then it is returned as a CORBA ‘Any’, but if it doesn’t exist then a value of null is returned to the Storage object.

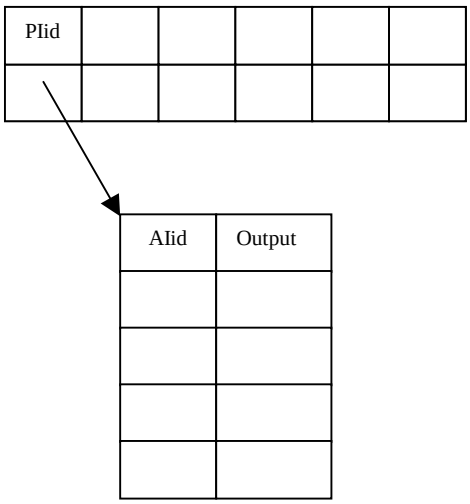

Figure 20: Hashtable of Hashtables that represents the memory cache of the SDE

5.2.6 Database Implementation

The database is called the SDE.mdb and has been implemented in Microsoft Access97 on the Windows NT platform. It was chosen for its portability.

There are three tables that are used by the application code. The objects, as shown in figure 19, IterIndex, ActIterMap, and Storage all make calls to the database using JDBC to send SQL statements across the JDBC-ODBC driver. These tables that are used by the SDE are Tbl_Datastore, Tbl_IterIndex, and Tbl_IterMap. Refer to Appendix B for the database schema.

Tbl_Datastore:

The Tbl_Datastore is shown in table 3. Each record contains the PIid, the AIid, and the Data. The Data field - the output of an activity - is the stringified value of the output. The PIid field, and the AIid are both defined as Long Integers. The Data field is defined as the data type Memo. As the string that is saved to this field is of indeterminate size, and can be a very long string, it was decided to use the Memo data type as this can contain up to 65,535 characters, and only text and numbers will be stored in it, not binary data. The Storage object inserts data and queries this table.

Plid	Alid	Data
1001	0	R~m_SdxTypes::t_Invoi
1002	0	R~m_SdxTypes::t_Invoi
1003	0	R~m_SdxTypes::t_Invoi
1001	5101	S{ul},0
1001	5201	S{R~m_TariffIdTypes::t_
1001	5202	S{R~m_TariffIdTypes::t
1001	5203	S{R~m_TariffIdTypes::t
1001	5301	R~m_SdxTypes::t_GetS
1001	5302	R~m_SdxTypes::t_GetS
1001	5303	R~m_SdxTypes::t_GetS
1001	5401	S{R~m_TariffIdTypes::t_
1001	5402	S{R~m_TariffIdTypes::t
1001	5403	S{R~m_TariffIdTypes::t
1001	5501	R~m_TariffTypes::t_Sub
1001	5502	R~m_TariffTypes::t_Sub
1001	5503	R~m_TariffTypes::t_Sub

Table 3: Tbl_Datastore that is implemented in the SDE Database

Tbl_IterIndex

This is where the iteration index for the Accounting Business Process is stored, e.g. for business process 1001, and Activity Type (ActType) 16 it is on its 3rd iteration, as shown in

Plid	ActType	Iteration
1001	15	0
1001	16	3
1001	18	3

Table 4: Tbl_IterIndex that is implemented in the SDE Database

table 4. For ActType 16 or 17 the iteration that either activity is on is determined by the record for ActType 16. For ActType 18 or 19 the iteration that either activity is on is determined by the record for ActType 18. The IterIndex object inserts data and

queries this table. The three fields in the table, PIid, ActType, and Iteration are defined as Long Integers.

Tbl_IterMap

PIid	ActType	Iteration	AIid
1001	15	0	5101
1001	16	3	5203
1001	16	2	5202
1001	16	1	5201
1001	17	3	5303
1001	17	2	5302
1001	17	1	5301
1001	18	3	5403
1001	18	2	5402
1001	18	1	5401
1001	19	3	5503
1001	19	2	5502
1001	19	1	5501

Table 5: Tbl_IterMap that is implemented in the SDE Database

For each Process Instance ID (PIid), activity type (ActType), and iteration, the associated Activity Instance ID (AIid) is entered in this table by the ActIterMap object.

This mapping is used so that, if during a retrieval method the output of an activity is needed then this table can be referred to, and the AIid can then be found. An example of this could be: an activity instance of type 17 with PIid 1001, on its third iteration needs the AIid of activity type 16 also on its third iteration, which from the above table is 5203. This method of querying is used in the AcingBP object when the input for ActType 17 is being built, as it needs the output of the preceding activity in the same iteration of the loop that activity 17 is in.

6. Evaluation

This chapter involves the evaluation of the SDE in relation to the research that was carried out during the project. The tool itself is then evaluated and the benefits and weaknesses of the application will be detailed.

6.1 SDE and Data Exchange in a Component Based Workflow Environment

The SDE proves that it is possible to provide a design to allow for the exchange of data between components in a workflow environment. It is difficult to find a benchmark to evaluate the SDE against, as there is very little literature available on this area of data exchange, in relation to workflow environments. This is due to the fact that the data flow is usually considered an orthogonal issue in the projects as discussed in section 2.5.

Further investigation was carried out into two component models that are being specified, or that are already in use, those being the CORBA Component Model and Enterprise JavaBeans respectively. These models carry out their own storage functions so that a developer doesn't get too involved on how to store the data. The SDE would be greatly helped with the use of the CORBA Component model but as yet the specification of CORBA that contains the CORBA Component Model has not yet been released. The CORBA Component Model would be preferable to Enterprise JavaBeans as the data is being passed as CORBA Objects. The CORBA Component model is based on Enterprise JavaBeans. This leads to the reasoning that the technologies chosen for the SDE were adequate to provide the function of data storage, and data exchange for components, but that a component model would greatly improve the functionality of the SDE.

The DBMS that was chosen to store the data was Microsoft Access. One of the main challenges in using this application was the difficulty of storing a CORBA object to the database. It simply was not possible to store the object itself, so a string

representation was used instead, and this was stored to the database. Access is a Relational DBMS (RDBMS). The investigation into the DBMS that are on the market should have entailed the investigation into the use of Object Oriented DBMS (OODBMS). These DBMS have been designed to store objects, and therefore using an OODBMS would have removed the investigation into how to store objects to Access. This would have saved the time spent on researching a method of storing a CORBA object to an Access database.

The objective of developing a prototype to demonstrate effective component data sharing within the confines of the existing workflow engine Serene has been achieved. This is what in theory and in practice the SDE accomplishes. There are areas of improvement that have been pointed out above, and the relevant conclusions that can be drawn from the above evaluation will be given in the next chapter.

The database was implemented as a centralised database with distributed access, and is considered to be component general as described in section 4.6. It could have been developed component specific but this would have reduced the functionality of the SDE. A component general approach meant that the SDE would not be concerned about which component that was making invocations to it. If it was component specific then the opposite would have been the case. This advantageous feature of the SDE can allow for extension of the business process. The SDE has no need to understand the components. All it needs is a data model for the business process.

Intelligence has been built in the SDE rather that the adaptors (section 2.6.5) that are used to invoke the component application. This was one of the objectives of the project, and was successfully accomplished. The intelligence has been taken away from the adaptors that invoke the components. The adaptors request the data and pass it to the component. The intelligence that resides on the SDE is its ability to store the data, and retrieve the data so that it can be passed back in a generic way, instead of having this function built into the adaptors.

6.2 Data Storage & Retrieval

One of the major advantages to the design of the SDE is the fact that the persistency storage mechanism can be replaced. The SDE has been designed this way so as to allow for the replacement of the persistency storage mechanism. The componentised development of the SDE is shown in figure 17. If it is removed the memory cache will remain in place, and therefore be unaffected. The memory cache is implemented so as to allow a speedier response in data retrieval, and if it had to be removed when the persistency storage mechanism was removed, this would have downgraded the functionality of the SDE considerably. It is the intention that data is exchanged between components in a distributed workflow environment, and one of the major objectives is to do this in an efficient manner. The cache is a major tool in improving retrieval rates of the components requesting data. Retrieval rates are improved with the cache as calls made to DBMS are slower than that of the calls made to the cache.

The actual data to be stored is stored in the database as a string. Saving the string version of an object to a database saves space in the database, as it is easier to save the string. Unfortunately if the string was an Interoperable Object Reference (IOR) this would prove to be a more useful method but it isn't. Then the objects that a component is looking for may be passed by reference to the database, and then retrieved by reference. This would be a great improvement, rather than passing the object itself around or the values that the object contains. As described in the implementation section, the data that is used for the activities are of CORBA type *struct*, which complicates the mode of storing to the database. The data is received as a CORBA 'Any', and converted to a string so that it can be saved in the database. This string is the stringified value of the object within the 'Any', not an IOR to the 'Any'. If the 'Any' contains a base type or CORBA *struct* the string will contain the values stored in the base type/*struct*. Only if the 'Any' contains an object then the string will contain the IOR of the object stored in this 'Any'. All outputs of the activities in the Accounting Business Process are *structs* so there are no IORs represented by the strings stored in the database.

One result of the above discussion is that the CORBA ‘Any’ that is used to pass the output of an activity into the database should only contain objects rather than *struct*s. If there has been a greater understanding of the data that the components needed then these objects could have been created at the outset of the project. A much more detailed analysis is needed on the data that is flowing through the Workflow Management System (WFMS). In other words the data flow needs to be better understood. This data flow is a separate analysis to the control flow in the WFMS. An aid to this would be to use a Computer Aided Software Engineering (CASE) tool to analyse the data flow in the WFMS. A notation for data modelling is contained in the Unified Modelling Language (UML). UML CASE tools are individual tools that aid a software developer in any phase of a project. The data analysis for this project occurred in parallel with the development of the SDE, and some of the data was not fully detailed at the beginning of the project. The data flow of the Accounting Business Process turned out to be quite detailed and complicated. Section 5.2.3 describes the complexity of the data structures that are stored to the database. This is where a deeper analysis of the data flow would improve the understanding of the data flow in a business process. It would also help to investigate if the data analysis for each different business process should be done separately, i.e. if there are greater differences between different business processes than might be expected.

As the complexity of the business process developed during the project so too did the data structures, and the need for the SDE to understand them. The complexity resulted in the AcingBP Bridge being developed as shown in the design section in figure 17. The object that was developed, as described in the implementation section, needed to build the inputs for each activity in the Accounting Business Process. This meant taking the outputs already stored in the database, extracting all the values from them and then reassembling them into the predefined CORBA *struct*s and then inserting these resulting inputs into a CORBA ‘Any’. This therefore means that the SDE needs to understand the data that is contained in the ‘Any’. One of the objectives of the project was to be able to store a CORBA ‘Any’ and retrieve the ‘Any’ from the database without the SDE needing to know what was contained within the ‘Any’. For the store function this objective has been retained, but the objective for the retrieve interface was impossible to attain in this project. The AcingBP object represents this

inability to maintain this objective for the retrieve interface. In other words the knowledge of the data is contained within the retrieve method of the AcingBP object.

An improvement in the method of storing and retrieving could have been made if it was possible to serialise the CORBA object to the database and store it in this manner. Then the retrieval would have consisted of unserialising the object from the database. To have all the objects capable of being serialised to the database would involve editing all the stubs that are generated from the CORBA IDLs by the idl compiler. The stubs that represent the objects that are to be stored. So each time that the IDLs that define these objects are recompiled then the stubs generated would have to be manually edited so that they implement the Java *serializable* class, and this would be a very time consuming process, and inadvisable to carry out. Reasons for avoiding this is that the large quantity of stubs that are generated, and would then need to be edited, would have to be done each time an IDL is compiled, or recompiled.

Blocking is implicit in the database connectivity JDBC package that Java uses. A requirement of the SDE was that once one component was storing to or retrieving from the SDE there would be no conflicts, and that no two components would be making invocations to the data store. When JDBC executes an update or query by sending an SQL statement across the JDBC-ODBC driver there is an auto-commit function in JDBC that will not allow any other update to the table that is being accessed until with that table.

6.3 Application Evaluation

There is no error handling built into the application code that has been developed in Java, and this is seen as a weakness of the system. Exceptions are caught in the application but they are not handled appropriately. If there are any problems with the application and it catches an exception, in most cases the program will crash and the server must then be restarted, and the component must invoke again the interface that it attempted to invoke. This is viewed as a major disadvantage to the application.
Other business processes can be added to the application but it would be necessary to create another object that will need to build the inputs for the activities in the business

process. This object would be similar to the AcingBP bridge, as shown in figure 17 in the design section. This would be necessary for any new business process introduced. The generic quality of the SDE is lost in this case, as it was envisioned that to add a new business process would involve adding the data model for the business process to the DBMS, and that any storing or retrieving would also be generic. It wasn't envisioned that a bridge would be needed to be created for each new business process. Only if the activities in the business process use the output of the preceding activity as their input and nothing else, would it be possible to have generic storage and retrieval methods. This was what the starting point of the project had been, but as stated before the complexity of the data flow model changed and so therefore the method of storing and retrieving the data.

6.4 Accounting Business Process Evaluation

The Accounting Business Process as a business process is complex. This complexity introduced more challenges in the development of the prototype. As these complexities arose in the project the design of the SDE had to be changed. As discussed in the above section the inputs to activities had to be built in the SDE. Another challenge that arose was the fact that there was an iterative loop in the business process. This created the requirement that it was necessary to keep an index of the number of iterations that had been carried out. The SDE didn't decide on the number of iterations itself rather it just needed to keep track of the iteration number. This was handled well by the SDE as it had it persistently stored in the DBMS. A memory cache would have sufficed, but having the iteration information stored persistently was a superior requirement, as the persistency function allows for recovery. The recovery is possible because there is a table in the DBMS that contains the AIid and the iteration that the activity is on. There is therefore no need to return to the beginning of the business process and restart the whole business process. The business process can restart from the point before it failed, i.e. the last activity that was completed and the data regarding the iteration index is still stored in the DBMS, and the business process can resume. This may slow the application but the reduced speed of the application outweighs the fact that there will be a persistent data store with this information. Also for any ad hoc queries that need to be carried out on the

metadata in relation to the number of iterations that have been carried out it is simply a matter of referring to the DBMS, and creating the necessary management interfaces.

6.5 Testing Evaluation

The SDE has been tested by two means. The first entailed creating clients for the SDE to create dummy outputs of activities, and then clients to retrieve inputs from the SDE for the activities in the Accounting Business Process. The outputs of the types defined for use by the activities in this business process were the actual types defined. It is the values that the data structures contain that are dummy values. These tests were completed for the activities 15, 16, 17, 18, and 19. There were no problems encountered. The data was stored to the database with the relevant PIids and AIids. The iteration indexing and the mapping of iteration index to AIid performed well and encountered no problems. The clients that were created in order to retrieve the data completed and functioned correctly, retrieving the dummy values that were stored to the database. The other method was the testing that the SDE underwent in parallel with the Serene workflow engine. This was undertaken by the Serene team. The engine was only tested with the activities 15, 16, 17 and 18. The SDE functioned and complied with the tests. Further testing with the SDE is to be carried out so that the other activities can be tested fully. Also the number of business processes that the Serene workflow engine is scheduling is to be increased so as to load test the SDE and the Serene workflow engine itself.

7. Conclusions

7.1 Achievements

The objectives of the project were

- Research into how other Workflow Engines exchange component data.
- Design of the integration of a Workflow application data exchange
- Implementation of the design
- Evaluation of the implementation

The first objective was accomplished, but it was evident from the information that was investigated that there was very little focus on production data in the workflow environment. Only one project detailed the approach to the data exchange in a WFMS. The second objective was also accomplished and a good design was found as is detailed in the design section – chapter 4. The third objective, the implementation was also accomplished, and this implementation proved that data can be exchanged between the components of a distributed workflow environment. The fourth objective, to evaluate the implementation of the SDE was accomplished, but given more time the evaluation could be much stronger.

The work that was completed, and the prototype that was developed is the best solution that was available given the choice of technologies, and the development stages of the project.

The most difficult area of the project was investigating a method of storing a CORBA object to the database. Eventually it was necessary to store a string to the database. The string was a representation of a CORBA object, and this string can be regenerated back into a CORBA object. The focus of difficulty was trying to store a CORBA object to Microsoft Access. Using Access with the technologies, Java, JDBC and OrbixWeb, did not allow for the storage of the actual object to the Access database. This was the reason for storing the string. To store an object to the database Microsoft

Access is not a suitable DBMS, but if it is acceptable that the string representation of the object is stored then Access is a suitable application.

The design of the SDE is considered a success, because it worked, and the persistency storage mechanism is removable and replaceable. This is a useful aid if it is intended that the prototype will be re-used and the persistency storage mechanism is replaced. This gives the SDE almost a component-based design, which is advantageous for further development of the prototype. This leads to a conclusion that there should be an investigation into other persistency storage mechanisms that could replace the Access DBMS used in this project.

The memory cache that was developed was a beneficial addition to the method of retrieval in the SDE. The memory cache allows for fast retrieval rates, because if the data that is being searched for is in the cache, then the actual persistency storage, the Access database, does not need to be searched. On failure of the application the memory cache will be deleted but this will always be backed up by the persistent storage. The memory cache should remain in use and should be retained in the design of the SDE.

The functionality of the SDE is an improvement on the SAD server that was developed for the first version of the Serene workflow engine. This stored objects to a file, and had to have knowledge of the data when it was storing it. The SDE has a generic method of storage and there is no need for knowledge of the data for the store interface. In addition there was no memory cache in the SAD server, as it was a simple design only to be an intermediary solution to data storage.

7.2 Personal Achievements

It is viewed that the personal learning curve for this project was high. The knowledge gained of CORBA, OrbixWeb, Java, and JDBC was immense. Starting from a basic knowledge there is now a deep understanding of these technologies and issues relating to software development.

7.3 Remaining Work

The analysis of the data was not carried out as it should have been. The data model that was designed developed in line with the development of the prototype, and while there was a deeper understanding of the Accounting Business Process at the end of the project, this caused changes in the design of the SDE during the project. There needs to be a return to the data flow and a much deeper analysis completed. The analysis of the production data in the Accounting Business Process should be considered separate to any other data flow, i.e. control data flow.

Exception handling in the SDE is as yet undeveloped. The ability of the SDE to be able to adequately deal with exceptions and errors is important. This was not completed in the project.

Other persistent storage mechanisms can be tested with the SDE. The design of the SDE allows for the replacement of the persistency storage mechanism, and so it leaves it open for testing with other mechanisms. There would be very little changes necessary to the application code, if any, if the DBMS presently used is replaced by a different storage mechanism.

7.4 Recommendations

It would be very worthwhile to test the SDE with additional business processes, to observe if the problems that arose from the use of the Accounting Business Process would occur with different business processes in relation to the data model. The complexity of the Accounting Business Process tested the SDE but it is only an example of one business process.

An area that is recommended to investigate further is the use of OODBMS. These OODBMS appear more appropriate in the storing and retrieving of objects. OODBMS

have been developed for the storage and retrieval of objects. Two products that would be worthwhile investigating are Versant and ObjectStore.

Distributed databases with good replication qualities should also be investigated. This would help in distributing the data to the components that need the data. Also the workflow engine is operating in a distributed environment, and a distributed database would work well in the distributed environment. It could also be an alternative to an OODBMS if the software is not available.

The CORBA Component Model is due for release this year as part of the new CORBA specification. The CORBA Component Model is based on Enterprise JavaBeans, and deals with persistent storage. The CORBA Component Model will take care of the persistent storage and the detail that is involved in trying to store data to a database, rather than the developer having to undertake the coding of this. It is recommended that the CORBA Component Model is investigated further.

A proper study of the data is required. The data to be analysed is the production data involved in the Accounting Business Process. The study of the data should be carried out using a CASE Tool. There exists a deeper understanding of the data in the Accounting Business Process now, but the production data flow has not yet been formalised. An analysis of the data flow using a CASE tool would allow for a more generic SDE, and the SDE should be as generic as possible.

Appendix A IDLs

This section contains the IDLs that contain the CORBA s*truct*s that were defined for use by the SDE. They are the inputs and output *struct*s that are used for the Accounting Business Process. The activities in the Accounting Business Process are also defined in IDL.

A.1 m_SdxTypes.idl

```
#ifndef __m_SdxTypes__
#define __m_SdxTypes__

#include <TINASubCommonTypes.idl> // t_AccountNumber
#include <TINAScsAmcCommon.idl>   // t_fromTo
#include <TINAUserInitial.idl>    // t_ProviderId
#include <m_ChargeListener.idl>
#include <m_ChargeTypes.idl>      // t_ChargeList, t_CurrencyUnit
#include <m_TariffIdTypes.idl>
#include <m_TariffTypes.idl>

module m_SdxTypes {

  struct getAllTIdListInputParameter {
    TINASubCommonTypes::t_AccountNumber subscriberId;
    TINAAccessCommonTypes::t_ServiceId     serviceId;
  };

struct t_InvoiceInputParameter {
    TINASubCommonTypes::t_AccountNumber subscriberId;
    m_ChargeTypes::t_CurrencyUnit       currencyUnit;
    TINAScsAmcCommon::t_fromTo          fromTo;
  };

 typedef unsigned long t_Second;

  struct t_RegisterUserListenerInputParameter {
    m_ChargeListener::i_UserChargeListener objRef;
```

```
 TINACommonTypes::t_UserId            userId;
 TINAAccessCommonTypes::t_ServiceId   serviceId;
 TINACommonTypes::t_SessionId         serviceSessionId;
 m_ChargeTypes::t_CurrencyUnit        currencyUnit;
 t_Second                             updateInterval;
};

struct t_RegisterSessionListenerInputParameter {
 m_ChargeListener::i_SessionChargeListener objRef;
 TINAAccessCommonTypes::t_ServiceId      serviceId;
 TINACommonTypes::t_SessionId            serviceSessionId;
 m_ChargeTypes::t_CurrencyUnit           currencyUnit;
 t_Second                                updateInterval;
};

struct t_GetSubscriberChargesInputParameter {
 TINASubCommonTypes::t_AccountNumber              subscriberId;
 m_TariffIdTypes::t_PerUserServiceLevelTariffIdList tidList;
 TINAAccessCommonTypes::t_ServiceId               serviceId;
 TINAScsAmcCommon::t_fromTo                       fromTo;
 m_ChargeTypes::t_CurrencyUnit                    currencyUnit;
};

struct t_GetSubscriberChargesOutputParameter {
 m_ChargeTypes::t_ChargeList                    charges;
 m_TariffIdTypes::t_ServiceLevelAgreementId     currentSubscriberSLAId;
};

struct t_StoreUserTariffInputParameter {
 TINAUserInitial::t_ProviderId   providerId;
 m_TariffTypes::t_UserTariff     userTariff;
};

struct t_StoreSubscriberTariffInputParameter {
 TINAUserInitial::t_ProviderId     providerId;
 m_TariffTypes::t_SubscriberTariff subscriberTariff;
};

struct t_RemoveTariffInputParameter {
```

```
  TINAUserInitial::t_ProviderId providerId;
  TINAScsAmcCommon::t_TariffId  tariffId;
 };

 struct t_GetUserTariffInputParameter {
  TINAUserInitial::t_ProviderId    providerId;
  m_TariffIdTypes::t_UserTariffId userTariffId;
 };

 struct  t_GetUserTariffListInputParameter {
  TINAUserInitial::t_ProviderId       providerId;
  m_TariffIdTypes::t_UserTariffIdList userTariffIdList;
 };

 struct t_GetSubscriberTariffInputParameter {
  TINAUserInitial::t_ProviderId        providerId;
  m_TariffIdTypes::t_SubscriberTariffId subscriberTariffId;
 };

 struct t_ListTariffIdsOutputParameter {
  m_TariffIdTypes::t_UserTariffIdList      userTariffIdList;
  m_TariffIdTypes::t_SubscriberTariffIdList subscriberTariffIdList;
 };

struct t_AllTariffIds {
  m_TariffIdTypes::t_UserTariffIdList      userTariffIdList;
  m_TariffIdTypes::t_SubscriberTariffIdList subscriberTariffIdList;
 };

 // these are the ones that are actually used by the WFE and SDx
struct t_PerServiceChargeListSLA {
  m_ChargeTypes::t_ChargeList               charges;
  m_TariffIdTypes::t_ServiceLevelAgreementId  currentSubscriberSLAId;
 };

struct t_PerServiceChargeListSLASubscriberTariff {
  m_TariffTypes::t_SubscriberTariff subscriberTariff;
  t_PerServiceChargeListSLA        perServiceChargeList;
 };
```

```
typedef                                    sequence<t_PerServiceChargeListSLASubscriberTariff>
t_PerServiceChargeListSLASubscriberTariffList;

struct t_SubscriberInvoiceDetailsandPerServiceCharges {
   TINASubCommonTypes::t_AccountNumber          subscriberId;
   m_ChargeTypes::t_CurrencyUnit                currencyUnit;
   TINAScsAmcCommon::t_fromTo                   fromTo;
   t_PerServiceChargeListSLASubscriberTariffList perServiceChargeListSLASubscriberTariffList;
  };
};
#endif
```

A.2 RETSubM.idl

This idl contains the activities listSubscribedServices( ), getAllUsersTariffIdList ( ), getSubscriberTariffIdList ( ), which are activity types 15, 16, and 18 respectively .

```
//                    Copyright (c) 1999
//                      Alcatel Bell
//                    All Rights Reserved
//
//          LICENSED MATERIAL - PROPERTY OF VITAL PROJECT PARTNERS
//    Possession and/or use of this material is subject to the provisions
//         of a written license agreement with the VITAL concortium
//
//              ACCESS AUTHORIZED TO FlowThru PROJECT PARTNERS
//
// ___________________________________ IDL definitions ___________________________________
//
//   FILE: RET_SubM.idl
//
//   AUTHOR: Koen Daenen (Koen.Daenen@alcatel.be)
//
//
//   VERSION: 5
//   DATE: 14/04/1999
//   DESCRIPTION:
//     IDL definition of SubM (Subscriber Manager) for FlowThru.
//
```

```
//   COMMENTS:
//
//   MODIFICATIONS:
//
//   Cliff Redmond (Cliff.Redmond@cs.tcd.ie) 13/7/1999
//   - made #includes relative not absolute
//   - imported t_UserIdList from m_AccCommonTypes - also used by SM
//   - changed names of tariffId return parameters to refect fact that they are
//     tariff identiiers and not tariffs; used m_TariffIdTypes.idl
//
//   IDL INTERFACES
//      Supported:
//               Name: RET_SubM::i_SubscriptionTariffInfo
//
//      Required:
//
//   RELATED DOCUMENTS:
//
// _____________________________ END DESCRIPTION HEADER _____________________

#include <TINACommonTypes.idl>
#include <TINASubCommonTypes.idl>
#include <TINAScsAmcCommon.idl>
#include <m_AccCommonTypes.idl>     // t_UserIdList
#include <m_TariffIdTypes.idl>      // t_ServiceLevelTariffIdList, t_PerUserServiceLevelTariffIdList

module RET_SubM {

  interface i_SubscriptionTariffInfo {

    exception e_OperationFailed {
      string reason;
    };

    // all structs previously here an shared with the ChargeControl are now
    // contained in m_TariffIdTypes.idl

    // operations
    // ----------
```

```
void getUserIdList( in  TINASubCommonTypes::t_AccountNumber accountNumber,
                    out m_AccCommonTypes::t_UserIdList      userIdList)
 raises (e_OperationFailed);
/**
 *  Lists all usersId's of an given accountNumber.
 **/

void listSubscribedServices( in  TINASubCommonTypes::t_AccountNumber accountNumber,
                         out TINASubCommonTypes::t_ServiceIdList serviceList)
 raises(e_OperationFailed);
/**
 *  Lists all serviceId's to which a subscriber is subscribed, based on a given accountNumber.
 **/

void getUsersAccountNumber( in  TINACommonTypes::t_UserId          userId,
                        out TINASubCommonTypes::t_AccountNumber accountNumber )
raises  (e_OperationFailed);
/**
 * Returns the accountMumber of a given userId.
 **/


void getSubscriberTariffIdList( in TINASubCommonTypes::t_AccountNumber
accountNumber,
                        in  TINAAccessCommonTypes::t_ServiceId         serviceId,
                        out                                    m_TariffIdTypes::t_ServiceLevelTariffIdList
serviceLevelSubscriberTariffIdList)
raises (e_OperationFailed);
/**
 * Returns all tariffIds for a given accountNumber and a given serviceId.
 **/
// out parameter changed to t_ServiceLevelTariffIdList

void getUserTariffIdList( in  TINACommonTypes::t_UserId                userId,
                      in  TINAAccessCommonTypes::t_ServiceId         serviceId,
                      out m_TariffIdTypes::t_ServiceLevelTariffIdList serviceLevelUserTariffIdList)
 raises (e_OperationFailed);
/**
```

```
    * Returns all tariffIds for a given userId and a given serviceId.
    * Note: The userId is unique in the context of a retailer.
    **/
   // out parameter changed to t_ServiceLevelTariffIdList

   void getAllUsersTariffIdList( in TINASubCommonTypes::t_AccountNumber
accountNumber,
                        in  TINAAccessCommonTypes::t_ServiceId serviceId,
                        out m_TariffIdTypes::t_PerUserServiceLevelTariffIdList
perUserServiceLevelTariffIdList)
     raises (e_OperationFailed);
   /**
    * Returns a list per userId of all tariffIds for a given serviceId;
    * and this for off userIds of a given accountNumber.
    **/
    // out parameter changed to t_PerUserServiceLevelTariffIdList

  };
};
```

A.3 m_ChargeContol.idl

This idl contains the activity getSubscriberCharges( ), which is activity type 17.

```
#ifndef __m_ChargeControl__
#define __m_ChargeControl__

#include <TINACommonTypes.idl>  // t_UserId, t_SessionId
#include <m_ChargeTypes.idl>    // t_ChargeList, t_CurrencyUnit
#include <m_TariffTypes.idl>    // t_UserTariffIdList
#include <m_ChargeListener.idl> // i_UserChargeListener, i_SessionChargeListener

module m_ChargeControl {

  enum t_ChargeControlOperation {
    invalidObjectReference,
    invalidServiceSessionId,
    invalidUserId,
    invalidCurrencyUnit,
    invalidUpdateInterval,
```

```
  invalidAccountNumber
};

exception e_ChargeControlOperation {
  t_ChargeControlOperation   error;
  string                     reason;
};

interface i_ChargeControlInit {

  void initialise (in TINAUserInitial::t_ProviderId     providerId);

};

interface i_registerChargeListener {

  typedef unsigned long t_Second;

  enum t_ChargeListenerType {user, session};

  // I split this operation and added t_UserId and t_SessionId;
  // I need to know what session I'm producing the charges for!
  void registerUserListener(in m_ChargeListener::i_UserChargeListener objRef,
                            in TINACommonTypes::t_UserId              userId,
                            in TINAAccessCommonTypes::t_ServiceId     serviceId,
                            in TINACommonTypes::t_SessionId           serviceSessionId,
                            in m_ChargeTypes::t_CurrencyUnit          currencyUnit,
                            in t_Second                               updateInterval)
    raises (e_ChargeControlOperation);

  void registerSessionListener(in m_ChargeListener::i_SessionChargeListener objRef,
                            in TINAAccessCommonTypes::t_ServiceId       serviceId,
                            in TINACommonTypes::t_SessionId             serviceSessionId,
                            in m_ChargeTypes::t_CurrencyUnit            currencyUnit,
                            in t_Second                                 updateInterval)
    raises (e_ChargeControlOperation);

  void deRegisterListener(in m_ChargeListener::i_ChargeListener objRef)
    raises (e_ChargeControlOperation);
```

```
  };

  interface i_ChargeControlQuery {

    void getSubscriberCharges ( in  TINASubCommonTypes::t_AccountNumber              subscriberId,
                                in  m_TariffIdTypes::t_PerUserServiceLevelTariffIdList tidList,
                                in  TINAAccessCommonTypes::t_ServiceId               serviceId,
                                in  TINAScsAmcCommon::t_DateTime                     upTo,
                                in  m_ChargeTypes::t_CurrencyUnit                    currencyUnit,
                                out m_ChargeTypes::t_ChargeList                      charges,
                                out m_TariffIdTypes::t_ServiceLevelAgreementId       currentSubscriberSLAId)
      raises (e_ChargeControlOperation);

  };
};
#endif
```

A.4 m_TariffControl.idl

This idl contains the activity getSubscriberTariff( ), which is activity type 19.

```
#ifndef __m_TariffCtrl__IDL__
#define __m_TariffCtrl__IDL__

#include <m_TariffTypes.idl>
#include <m_TariffIdTypes.idl>
#include <TINAUserInitial.idl>      // t_ProviderId

module m_TariffControl {

  enum t_TariffControlOperation {
    tariffControlAlreadyInitialised,
    tariffAlreadyExists,
    tariffDoesNotExist
  };

  exception e_TariffControlOperation {
    t_TariffControlOperation   error;
    string                     reason;
```

```
};

interface i_TariffControlInit {
  void initialise (in TINAUserInitial::t_ProviderId providerId)
    raises(e_TariffControlOperation);
};

interface i_TariffControlManagement {

  void storeUserTariff( in  m_TariffTypes::t_UserTariff     userTariff,
                  out m_TariffIdTypes::t_UserTariffId userTariffId) // dummy id in
    raises (e_TariffControlOperation );

  void storeSubscriberTariff(in m_TariffTypes::t_SubscriberTariff subscriberTariff,
                    out m_TariffIdTypes::t_SubscriberTariffId subscriberTariffId)// dummy id in
    raises (e_TariffControlOperation);

  void removeTariff(in TINAScsAmcCommon::t_TariffId  tariffId) // goes to show, typedef is a type
of semantic inheritance!
    raises (e_TariffControlOperation);
};

interface i_TariffControlQuery {

  void getUserTariff( in  m_TariffIdTypes::t_UserTariffId userTariffId,
                out m_TariffTypes::t_UserTariff     userTariff)
    raises (e_TariffControlOperation);

  void getUserTariffList(in  m_TariffIdTypes::t_UserTariffIdList userTariffIdList,
                    out m_TariffTypes::t_UserTariffList     userTariffList)
    raises (e_TariffControlOperation);

  void getSubscriberTariff(in m_TariffIdTypes::t_SubscriberTariffId subscriberTariffId ,
                      out m_TariffTypes::t_SubscriberTariff    subscriberTariff)
    raises (e_TariffControlOperation);

  // if no tariffs are stored then two empty lists are returned
  void listTariffIds (out m_TariffIdTypes::t_UserTariffIdList       userTariffIdList,
                out m_TariffIdTypes::t_SubscriberTariffIdList subscriberTariffIdList)
```

```
    raises (e_TariffControlOperation);
  };
};

#endif
```

A.5 m_BillControl.idl

This idl contains the activity requestInvoice( ), which is activity type 20.

```
#ifndef __m_BillControl__
#define __m_BillControl__

#include <TINASubCommonTypes.idl> // t_AccountNumber
#include <TINAScsAmcCommon.idl>   // t_fromTO
#include <TINACommonTypes.idl>    // t_UserId
#include <m_ChargeTypes.idl>      // t_ChargeList, t_CurrencyUnit
#include <TINAUserInitial.idl>    // t_ProviderId
#include <m_BillTypes.idl>        // t_Invoice, t_InvoiceId

module m_BillControl {

  enum t_InvoiceManagementOperation {
    invalidAccountNumber,
    invalidTimePeriod,
    subscriptionAccessDenied,
    subscriptionAccessError,
    invalidInvoiceId,
    incorrectAmount
  };

  exception e_InvoiceManagementOperation {
    t_InvoiceManagementOperation error;
    string reason;
  };

  interface i_InvoiceManagement {

    void previewInvoice(in TINASubCommonTypes::t_AccountNumber subscriberId,
                        in m_ChargeTypes::t_CurrencyUnit       currencyUnit,
```

```
                in TINAScsAmcCommon::t_DateTime       upTo,
                out m_BillTypes::t_Invoice            invoice)
    raises(e_InvoiceManagementOperation);
  // The invoice number will be blank here

  void requestInvoice(in TINASubCommonTypes::t_AccountNumber subscriberId,
                in m_ChargeTypes::t_CurrencyUnit      currencyUnit,
                in TINAScsAmcCommon::t_DateTime       upTo,
                out m_BillTypes::t_Invoice            invoice)
    raises(e_InvoiceManagementOperation);

  void payInvoice ( in m_BillTypes::t_InvoiceId invoiceId, in m_ChargeTypes::t_Amount amountDue
)
    raises(e_InvoiceManagementOperation);

 };
};
#endif
```

Appendix B Database Schema

This appendix details the schema of the SDE database that was used to store the data, and hold the metadata.

Tbl_Datastore

Field	Type
PIid	Integer
AIid	Integer
Data	Memo

Tbl_IterIndex

Field	Type
PIid	Integer
ActType	Integer
Iteration	Integer

Tbl_IterMap

Field	Type
PIid	Integer
ActType	Integer
Iteration	Integer
AIid	Integer

Piid: Process Instance ID

Aiid: Activity Instance ID

ActType: Activity Type

Data: Output of an Activity

Iteration: Iteration of loop

Abbreviations

AIid	Activity Instance ID
ActType	Activity Type
BPR	Business Process Re-Engineering
CASE	Computer Aided Software Engineering
DBMS	Database Management System
EJB	Enterprise JavaBeans
IDL	Interface Definition Language
IOR	Interoperable Object Reference
OODBMS	Object Oriented Database Management System
PIid	Process Instance ID
RDBMS	Relational Database Management System
SDE	Shared Data Exchange
SCDS	Shared Component Data Exchange
WfMC	Workflow Management Coalition
WFMS	Workflow Management System
WIS	Workflow Information Server

Bibliography

[Alon97a] "Distributed Data Management in Workflow Environments"
Gustavo Alonso, Bethold Reinwald, C. Mohan

[Alon97b] "Functionality and Limitations of Current Workflow Management Systems"
G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan
IEEE Expert, Vol12, No. 5, Sept/Oct 1997

[Alon97c] "Workflow Management Systems: The next Generation of Distributed Processing Tools"
G. Alonso, C. Mohan

[COR99] CORBA Components (final submission - orbos/99-02-05)
March 2, 1999
http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html

[Das97] "ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METER$_2$"
S. Das, K. Kochut, J. Miller, A. Sheth, D. Worah
Technical Report #UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, February 1997

[Eder98] “The Workflow Management System Panta Rhei”
Johann Eder, Herbert Groiss, Walter Liebhart
In: Workflow Management Systems and Interoperability,
Springer-Verlag, 1998

[EJB98] Enterprise JavaBeans Technology, Server Component Model for the Java Platform (White Paper)
Anne Thomas
Revised December 1998, Prepared for Sun Microsystems, Inc.

[Full99] "Survey of Workflow Management Technologies for Integrating Fine-grained Services into the Virtual Enterprise"
John Fuller
Department of Computer Science, Trinity College, March 1999

[Geog95] "An Overview of Workflow Management: From Process Modelling to Workflow Automation"
Diimtrios Geogakopoulos, Mark Hornick, Amit Sheth
Distributed and Parallel Databases, 3, 119-153 (1995)

[Gill99] "Workflow query" - e-mail
Michael Gillmann
University of the Saarland

[Iona98] OrbixWeb Programmers Guide
Iona Technologies PLC
September 1998

[Jabl96] "Workflow Management: Modelling Concepts, Architecture and Implementation"
Stefan Jablonski, Christoph Bussler
ISBN: 1850322228, International Thompson Computer Press, 1996

[JDBC99] JDBC Data Access API
http://java.sun.com/products/jdk/1.3/docs/guide/jdbc/index.html

[Muth98] "From Centralised Workflow Specification to Distributed Workflow Execution"
Peter Muth, Dirk Wodtke, Jeanine Weissenfels, Angelica Kotz Dittrich, Gerhard Weikum
March 1998
In: JIIS - Special Issue on Workflow Management, Volume 10, Number 2, March 1998, Kluwer Academic Publishers
http://www-dbs.cs.uni-sb.de/public_html/papers/JIIS97.ps.Z

[Muth99] "Integrating Light-Weight Workflow Management Systems within Existing Business Environments"
Peter Muth, Jeanine Weissenfels, Michael Gillmann, Gerhard Weikum
Proc of 15th International Conference on Data Engineering, Sydney, Australia, March 1999
http://www-dbs.cs.uni-sb.de/public_html/papers/ICDE99_WF.ps

[Post99] Database Management Systems: Designing and Building Business Applications
Gerald V. Post

ISBN 0-07-289893-3, Irwin/McGraw-Hill, 1999

[Segu98] A CORBA Primer – Technical Overview – White Paper
Segue Software
http://www.segue.com/html/s_solutions/pdf/wp_corba_primer.pdf

[Wade99] "Flexible Automated Enactment of Process Driven Telecommunications Management"
Vincent Wade, Sinead Muldowney, John Fuller

[Weis96] "The Mentor Architecture for Enterprise-wide Workflow Management"
Jeanine Weissenfels, Dirk Wodtke, Gerhard Weikum, Angelika Kotz Dittrich
NSF Workshop on Workflow and Process Automation in Information Systems, Athens, Greece, May 1996
http://paris.cs.uni-sb.de/public_html/papers/mentor.html

[WfMC] Workflow Management Coalition Homepage
http://www.wfmc.org

[WfRe94] "Workflow Management Reference Model"
Workflow Management Coalition (Nov 94)