C320

MONASH UNIVERSITY
THESIS ACCEPTED IN SATISFACTION OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
ON..... 20 December 2002 .....
.....[redacted].....
Sec. Research Graduate School Committee

Dynamic Reconfiguration under Real-Time Constraints

by

Dean Thompson, BComp (Info. Sys) Hons

Dissertation

Submitted by Dean Thompson

for fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

in the School of Computer Science and Software Engineering at

Monash University

Monash University

December, 2002

Contents

List of Tables

List of Figures

Dynamic Reconfiguration under Real-Time Constraints

Dean Thompson, PhD
Monash University, 2002

Supervisor: Professor Heinz Schmidt

Abstract

Continual development and demands in computer applications has resulted in a need to develop software systems capable of evolving to meet such changes. Traditionally, systems have been handicapped by tightly integrated components restricting their flexibility.

The requirement for such flexibility led to the development of a component based programming paradigm, enabling the interchange of software components both locally and remotely and thus providing the flexibility required to support change. To facilitate the development of components a number of architectures and software systems were developed. Examples of architectures include CORBA, DCOM/COM, .NET and JavaBeans while software systems include DARWIN, REGIS and SOFA/DCUP.

The rapid deployment of systems into real-time environments has led to the need to perform inline software component upgrades while systems are still operating and without causing any interruptions. Significant advances in reconfiguration tools have made it possible to achieve this goal, but little support exists for software developers and end-users to control method calls made on components that are not available. In addition the real-time commitments of the system must be handled. Incorrectly handled situations can result in method calls being blocked which is unacceptable in real-time environments.

This thesis addresses that concern by providing software developers and end-users with a model known as the Component Oriented Reconfiguration Environment and Scheduling (CORES) environment which can be incorporated into existing component environments. This model provides a level of control over method calls and how they are handled within a real-time environment when sent to a component which is not available. CORES allows method calls to either wait until a component is available or to return to the component making the initial call.

The CORES model also examines and provides support for the manipulation of real-time schedules governing the environment and incorporating support for Quality of Service (QoS) characteristics. In addition to the formal definition, an implementation is provided utilitising a radio telescope array at Narrabri, Australia to show its use and advantages. This research solves the problem that software developers and end-users face when operating within real-time environments by giving them back control over how components interact with one another and the method calls themselves.

Dynamic Reconfiguration under Real-Time Constraints

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

[redacted]

Dean Thompson
December 5, 2002

Acknowledgments

This thesis is the result of 5 years work and has benefited greatly from the significant contributions and influences of many people. Before acknowledging those who assisted, I would like to apologise in advance for any names which may have been inadvertently omitted from the list.

I would like to sincerely thank Dr. Chris Ling for his patience and understanding that he showed while explaining the concept of Petri nets and how they can be formally modelled. His assistance and ideas were invaluable when modelling the various Petri net structures.

Thanks to Zelko Karlovic who explained the fundamentals of astronomy, astro-physics and who pointed me in the right direction so as to obtain the answers to the questions that I asked. Without these references and his astronomy knowledge I would have been truly lost in calculating the position of heavenly bodies.

Special thanks to Dr. Damien Watkins for acting as my research mentor and encouraging me to write a number of papers with him on Distributed Objects. It was through many of our conversations that ideas for this thesis were developed and later explored. I would also like to thank him for his constructive criticism which aided the development of this thesis and for ensuring that I remained on track.

I would like to acknowledge the assistance of Rod Simpson and Barrie Thompson who tirelessly proof-read many of the chapters and provided invaluable feedback, direction and corrections. Without these corrections, the chapters would be a mere shadow of themselves.

In addition, I would like to thank Michi Henning for verifying the source code and confirming the assumption that a problem being experienced in the implementation was the result of an error in a third party library rather than the implementation itself, Professors Richard Mitchell and Christine Mingins for their supportive comments and ideas and Dr. David McConnell and the staff at the Narrabri Radio Observatory, Australia for their time, hospitality and knowledge in explaining how radio telescopes and the observatory work.

Thanks go to my supervisor Professor Heinz Schmidt for getting me started, introducing me to the worlds of real-time environments and component technology, providing ideas and approaches, allowing me to try ideas and discuss implementations and his assistance throughout the entire canditure.

I would also like to thank the many people of the School of Computer Science and Software Engineering for their generous advice and support while developing this thesis. In particular I would like to thank those postgraduate students who I have shared an office with or borrowed a terminal from and the staff that have assisted in the production of this thesis. Specifically, I would like to thank: Chee Yeen Chan, Professor John Crossley, Simon Cuce, Michelle Ketchen, Shonali Krishnaswamy, Dr. Jason Lowder, Trent Mifsud, Nick Nicoloudis and Dr. Peter Stański.

Dean Thompson

Monash University
December 2002

For my parents

Chapter 1

Introduction

1.1 Background

The need and ability to develop software systems capable of *evolving* to changing needs is becoming increasingly important as systems are deployed into rapidly changing environments. This evolution is accomplished through the interchanging of software components and can best be illustrated by examining the lifetime of a computer system and/or application. Throughout its lifetime, a system undergoes a series of configuration changes to allow itself to remain relevant to its current environment. Each configuration change that is made is coordinated through a change management system and hence forms part of the systems evolutionary process.

The development of change or configuration management systems can largely be attributed to the need for formal processes having to be established to allow for the management of changes within systems. The need for having to introduce the change management concept as mentioned in Schmerl and Marlin (1996) arose from the problems that large, monolithic systems have when composed of a number of bulky components tightly integrated with one another.

These tightly coupled components led to an increase in the dependencies throughout the entire system (Kramer and Magee 1990). This increase in dependencies makes the process of system development very complex as changes made in one component may affect others scattered throughout the system. To further complicate the process, these effects may not be seen immediately after a change but are experienced during phases of maintenance or at points where there are interconnections with other systems.

With the increase in tightly coupled components comes the decreased ability for systems to provide any sort of component configuration. Such systems are difficult to reconfigure as all dependencies must first be identified before the appropriate flow-on effects calculated.

Distributed systems have been able to reduce the dependencies in the code through the introduction of loosely coupled components. The introduction of these loose components has led to a programming style first mentioned in Kramer (1990) known as the 'Component Based Programming' which is based off the 'Object Oriented' programming paradigm (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991; Meyer 1997). The component based approach allows developers to build self-contained components for the system rather than constructing one encompassing system.

The original concept of component based programming was developed in response to problems which distributed systems suffered when constructed by clumping together a number of small programs and arranging them to form larger systems (Magee, Kramer, and Sloman 1989). Over a period of time these large systems have become quite dense and can be identified by the use of heavily congested networks and systems. To address this issue, component based programming was developed to provide developers with the ability to avoid problems such as scalability, complexity and dealing with the ever continuing problem of general business evolution (Kramer and Magee 1997).

Additionally, while these monolithic systems were being developed there was little agreement on how modules within the system should be structured, communicate with one another or how each should be configured (Magee, Kramer, and Sloman 1989). As a result it was possible to determine that whilst component based programming had managed to lay the foundation for component programming it still did not deal with the issue of configuration management for both static and dynamic systems.

The consequence therefore is that little configuration technology has transpired or migrated from the monolithic mainframe or tightly coupled application development environments to that of distributed systems.

While considerable effort is being placed on the development of systems capable of integrating configuration management along with component creation and management services, there is also a fundamental need to incorporate these concepts into real-time systems. No where is this more evident than the increasing number of systems which are being built with embedded real-time processing units.

With an increasing number of software and hardware components providing real-time support as well as a growing number of already existing systems requiring the ability to reconfigure themselves in a real-time environment comes the need of being able to give software developers and/or end-users control over such activities.

1.2 Thesis Objectives

The key objectives for this thesis are:

- To identify any problems or lack of support which may exist for software developers and end-users to exert some control over what actions should be performed when a component is being reconfigured or otherwise not available in a real-time environment
- To develop a conceptual model which:
  - Provides the ability for software developers or end-users to exert some control over what happens when a reconfiguration is taking place within the system
  - Provides software developers and end-users with timely information within a real-time environment regarding the impact of method calls being made on those components which are undergoing reconfiguration
  - Provides support for objects located in a distributed heterogeneous networking environment
  - Provides an approach giving system developers and end-users more flexibility without having to perform extensive reprogramming of the system
  - Allows Quality of Service (QoS) specifications to impact on the availability of the components and have this availability incorporated into the real-time environment
  - Demonstrates how flexible a system can be when end-users have some element of *control* over it

1.3 Research Questions

To address the objectives mentioned earlier in section 1.2, this thesis puts forth and addresses the following research questions:

RQ1: To what extent do the current reconfiguration management environments provide support for dynamic reconfiguration?

RQ2: What support exists for software developers or end-users to deal with a situation where a method call is made on a component which is in the state of reconfiguring itself or otherwise not available whilst operating under real-time constraints?

RQ3: What are the requirements of a conceptual model capable of addressing real-time reconfiguration and what additional support is needed when deploying it within a non-trivial environment where software developers and end-users can specify what actions should be taken if a method call is made upon a component reconfiguring itself or is otherwise not available?

RQ4: Is it possible to integrate into the proposed model the capability to calculate the impact on the overall schedule and to identify what implications may exist when various sequencing algorithms (best case scenario vs. worst case scenario) are used to schedule tasks and jobs?

RQ5: What impact does the incorporation of Quality of Service (QoS) arguments have on the proposed model so that software developers or end-users can associate QoS characteristics with components and then have those characteristics used in conjunction with the respective real-time commitments of the system to calculate a components availability?

In order to address these research questions, it is necessary to examine the current support that software developers and end-users have with regard to supporting dynamic reconfiguration during both development and the lifetime of a systems execution.

Secondly, after the examination of the software component reconfiguration tools, each tool needs to be assessed with regard to being able to provide software developers or end-users with some support to deal with the situation of having to still operate a system under real-time constraints while components are being reconfigured.

It is the contention of this thesis that although reconfiguration tools do exist to help software developers reconfigure a system, there is no one tool or environment which provides the ability to specify what should happen in the case of a method call being made upon a component being reconfigured or otherwise not available.

To address this situation, a conceptual model is presented which outlines how software developers or end-users can specify commands to a system to instruct it on how to deal with method calls being made on components that are not available, whilst at the same time not requiring the entire system to be totally rewritten or shutdown. The model initially proposed in chapter 4 and further developed in chapters 5 and 6 allows for various actions to take place if a method call is sent to a component being reconfigured within the system.

1.4 Overview of Thesis

The remainder of this thesis directly addresses these research questions. Chapter 2 introduces the concept of the component based paradigm and examines the issue of how the component based paradigm is defined and its objectives. Additionally, this chapter focuses on those techniques and programming aids developed to help software developers embrace such a programming concept.

After an examination of the component based techniques which are available to aid software developers, the chapter addresses those architectures designed to support and promote component and object based development within a distributed system. The chapter also discusses the key benefits that the component based paradigm offers and introduces the background and processes which are

commonly followed when a component is reconfigured, especially when occurring within a dynamic system.

Chapter 3 introduces the criteria used to assess the major attributes of various configuration management systems and determine their suitability as an architecture to provide dynamic reconfiguration services. Furthermore the chapter develops the notion of configuration management systems by examining a number of systems and applying the criteria already introduced. The chapter also highlights how the various configuration management systems can be grouped based on the support they provide for dynamic reconfiguration management. An understanding of the configuration architectures, environments and systems assist in answering RQ1 by highlighting what support already exists.

Chapter 4 examines the support provided by the reconfiguration management systems for performing dynamic reconfiguration management. This examination is further re-enforced by the results from the critique performed on each of the reconfiguration environments presented in chapter 3. The findings from the critique influence the remainder of the chapter which examines the issue of how reconfiguration management environments fail to provide a level of control to either software developers or end-users when dealing with method calls sent to components that are being reconfigured, especially in real-time systems. Identifying this lack of support for dealing with real-time systems contributes to responding to RQ2.

The chapter also provides a background on the impact of having no controls available for managing tasks within real-time systems which are governed by either hard or soft real-time constraints. Issues raised in this chapter are addressed by a proposed extension to provide support for software developers and end-users alike to be able to control the way in which method calls are handled when components are being reconfigured.

Chapter 5 describes a proposed conceptual model to provide software developers and end-users with the extra control needed to handle situations where method calls are made on components which are reconfiguring themselves or otherwise not available within a real-time environment. In addition to providing support for reconfiguring components, the chapter presents a sequencing and scheduling engine responsible for taking tasks and placing them into the overall schedule while at the same time being mindful of the real-time constraints within which a scheduler has to work.

Throughout the chapter the concepts required to provide this level of control for software developers and end-users is introduced. These include algorithms responsible for sequencing and scheduling tasks and the dynamic controls which influence the behaviour of the system when a reconfiguration event arises and the component is not available. Also included is the configuration manager that is responsible for handling the reconfiguration events and performing operations that the dynamic controls have specified. The chapter also identifies the limitations associated with the conceptual model.

Chapter 6 discusses the translation of the conceptual model into an implementation model which can be incorporated into software systems. Specifically, it details how the sequencing of tasks both at the task construction level and at the scheduler l[illegible]el were implemented. The chapter also examines the implementation details relating to how a method call is handled if it is sent to a component that is unable to honour it while at the same time observing any real-time constraints which have been specified by the request.

Both chapters 5 and 6 provide details that can be used to address the concerns raised in RQ3, RQ4 and RQ5.

Chapter 7 demonstrates how the design detailed in chapter 5 and the implementation as specified in chapter 6 can be brought together into a real-world scenario whilst at the same time providing a system with real-time constraints in which to work. The chapter introduces a case study built to simulate an abstracted radio observatory and provides the perfect opportunity for the design principles governing the system to be verified. The case study provides end-users with the ability to control what happens when a method call is made against a component which is reconfiguring or not available while at the same time allowing end-users to dictate how tasks are sequenced and scheduled.

This chapter also provides a brief introduction to radio astronomy and the telescope facility based at Culgoora, New South Wales, Australia which forms the basis for the simulation. To verify the design objectives of the system proposed in chapter 5 a number of scenarios are presented demonstrating how end-users have *control* over the system during a reconfiguration event. The chapter concludes with a brief explanation of the implementation limitations.

Chapter 8 presents the findings from the research as well as providing answers to the research questions proposed in section 1.3. The chapter also presents a section on what future work can be performed based on this current research and how it might be improved and strengthened.

Chapter 2

Introducing Component Based Paradigms

For the last 30 years, as noted in Szyperski (1997), we have seen the development of the fundamental concepts behind the component based paradigm. These developments have been aided by the advances made in the areas of software engineering and the development of architectures which facilitate component construction.

The fundamental ideas of the component based paradigm can be traced back as far as the mid 1960's as detailed in Naur, Randall, and Buxton (1976). It was during this time that the first notion of a component was raised and how components could potentially be linked together to form component based systems. This idea was further embraced within the computer programming language 'Simula' when it was released as it was one of the first to support the notion of what we call today 'objects' (Dahl and Nygaard 1966).

Since then, there have been a number of developments in the fields of component based technology and object based technology. As a result of the advances made in the respective fields there is now some confusion as to the difference between a component and an object. Whilst there is a difference it is common to find many publications use the terms interchangeably.

This chapter aims to address this confusion by examining the differences between objects and components. Additionally, focus is provided on the objectives of the component based paradigm and the additional support required to provide the reconfiguration of components. To assist in the understanding of this paradigm, a number of architectures designed to facilitate the construction of component based systems are discussed.

2.1 Components vs. Objects

The continuing developments in both component based and object based technology has led to an increase in the number of terms used to describe the different frameworks which are currently available.

In some cases the words 'component' and 'object' are used within the same term such as the Component Object Model (COM) (Brown and Kindel 1996) developed by Microsoft. Additionally, there are some frameworks which are described as being able to provide support for both object and component construction. With these terms being seen as interchangeable it is important to actually highlight their differences as well as similarities.

2.1.1 Components

As is mentioned in Szyperski (1997), a component is defined as having the following characteristics:

- Independently Deployable
- Is of a third-party composition/construction
- Has no concept of persistent state

In order for a component to be independently deployable, it must not make any assumptions about the environment in which it is being deployed. Additionally, the component must make no assumptions or have any tight links with any other component, as it can not be assumed that the other component will exist within the same environment. Therefore, for a component to be totally independent it must *encapsulate* both the data and member functions required by itself in order to operate. Access to these data variables and functions is permitted through a series of well defined interfaces. From this conceptual point of view, a component can never be partially deployed, it must be deployed in full, as there is no way of being able to break up a single component.

It is predicted that in the future we will see an explosive growth in the number of companies providing third-party components for systems. To some degree this has already started to happen with the .NET framework as described in Microsoft Corporation (2001) being designed to interoperate with web-services which have been built by third parties. In these cases, a software developer may not necessarily have access to the source code of the component but may receive it in a *binary format*. In some areas of literature this is known as binary deployment as all that is shipped with the third-party component is the binary/byte code. With this in mind it is the responsibility of the component developer to specify what *requirements* this component needs and what this component actually *provides*. Once again, in order for third-party components to operate without any problems they must be fully self-contained (ie. encapsulate all the data members required, functions to perform

and any other components required to be present) so as to allow them to work straight out of the box.

In order for something to become persistent it must first have a handle which can be used to identify it specifically. As was mentioned earlier, components have no perception of persistent data and therefore have no sense of identity within them. A benefit from this is that a component can be independently, removed or activated within a system without having to rely on the transfer of data which is pertinent to it.

Again, with there being no concept of identity there is no real way of being able to differentiate between components. This effectively means that a user has to be very careful when loading multiple versions of the same component into memory as each component within memory will look exactly the same resulting in complications when coming to tell the two apart.

2.1.2 Objects

As is mentioned in Szyperski (1997), a object is defined to have the following characteristics:

- Unit of instantiation which has a unique identity
- Has state which can be persistent
- Provides a method which can encapsulate both state and behaviour

Additionally, as specified in Stroustrup (1994) an object can also support the concept of an *inheritance* structure.

Similar to a component having a concept of being a unit of deployment, so does an object which has a concept of being a unit of instantiation. This effectively means that an object can not be partially instantiated and must be instantiated in full. However, it is unclear as to what would happen in the case described in Meyers (1998) where an error (most typically an exception) is thrown during the object's construction.

A consequence of being able to instantiate an object brings with it the notion of identity. Identity is where an object is uniquely identifiable by a specific reference. No matter how similar two objects are to one another (they can contain the same data within their data members) they will still be unique by virtue of their presence. The specific reference that uniquely identifies objects is constant for its entire lifetime. Using the identity characteristic allows state data held within a particular object instance to be uniquely identifiable. This association between the state of an object and its instance forms the basis for state persistence and allows objects at a later date to refer back to their own state.

The role of encapsulation also plays an important role with each object as each is capable of having its own state and behaviour encapsulated. This differs from a component where encapsulation

exists merely for the component while objects allow for each individual to have their own area of encapsulated data.

The concept of inheritance as defined in Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen (1991) is the grouping together and organisation of classes which are similar to one another in one respect using a tree like structure. An example of inheritance might include having a football object inherit from a ball object.

It is important however to realise that an object is a *run-time instance* (Meyer 1997) of an abstract data type (ADT) known as a class which acts like a *blueprint* (Deitel and Deitel 1997). For an object to be realised and used it must be instantiated (as has already been covered) through the use of an execution plan (Szyperski 1997). This plan is responsible for allocating all of the space required for the object as well as establishing the initial behaviour and attributes as prescribed in the class definition.

2.2 Objectives of Component Based Paradigm

The component based paradigm aims to address the significant issues which exist with current monolithic systems. These systems traditionally lack cohesion, consistency and an overall sense of common purpose in their architecture. In order to address these problems, the component based paradigm encourages systems to be built with modularity in mind and strongly promotes this through its own framework.

This modularity concept is achieved through the construction of software components which can be connected together to form the appropriate system. Such a paradigm brings with it the following benefits:

- Modularity / Flexibility
- Consistency
- Structure
- Software Re-Use
- Testing
- Configuration Management

2.2.1 Modularity / Flexibility

The concept of modularity is achieved within the component based paradigm through the successful deployment of software components within the system. As a system is built within the paradigm,

the software developer will notice that components which are similar in their functionality will be grouped and interact with one another.

In addition, by specifying that each component must be self-contained (refer to section 2.1.1), the developer is left with little choice but to break down a system into its fundamental components.

The flexibility that a software developer has within the system is directly proportional to the granularity of the component. Flexibility aims to allow the software developer to substitute a component within the system without having to require a significant amount of re-programming. This can be achieved if a component is written to a fine granularity level. The coarser a component is, the less flexible the system becomes.

The benefits of having a flexible system include the ability to allow the software developer to move modules around the system and for those modules to be placed wherever the system dictates the need for them.

2.2.2 Consistency

The need for consistency is something which is constantly assessed and valued within the component based paradigm. To aid the software developer with the construction of the system, a number of graphical tools have been developed as mentioned in Kramer, Magee, and Ng (1989) to provide the developer with instant feedback concerning the consistency of the system.

In addition to graphical aids which exist, a number of configuration definition languages have also been developed to aid the software developer during the process of system construction. These configuration definition languages will be discussed in more detail in section 2.4.

However, it is important to note that a full consistency check is normally performed upon the system by the desired component framework architecture responsible for implementing that system. In some cases it is impossible to provide a complete analysis of the system, although numerous developments have been made with Labelled Transition Systems as is mentioned in Magee and Kramer (1999).

2.2.3 Structure

One of the key points that the component based paradigm aims to address is the lack of structure or perceived structure in current monolithic systems. It is well recognised that these legacy systems have been modified for a period of years resulting in tightly inter-connected components and objects. In most cases these modifications which consist of adding additional bits of code to the system to achieve the functionality required (band aid approach) have resulted in the decay of the underlying system structure which is responsible for holding all of the components together. At a future point in time after another modification is made, the system will reach a critical mass point where the entire underlying system structure holding the system together will collapse.

Component based programming solves this problem by requiring the software developer to implement the system in terms of components. Each component represents an independent low level function or function group. By adopting this approach, natural functional structures will form by themselves hence providing the system with some structure. Additionally, the paradigm is also structured in such a way as to provide the software developer with more opportunities to develop their own system structures.

2.2.4 Software Re-Use

One of the key benefits that the component based paradigm supports along with software engineering is the concept of re-use. Both the object oriented and component based paradigms encourage the re-use of components. One of the major problems currently facing monolithic or legacy systems is the constant re-writing of code which performs the same task. The component based paradigm supports the concept of using third party components either written by a local developer or purchased from somewhere else and placing them into the system.

This has a number of significant benefits. One is the amount of time which is saved in not having to rewrite a component which already exists. This reduction in time is achieved by not having to get a local developer to write or in some cases rewrite a component which already exists elsewhere. As is mentioned in Szyperski (1997), the component based paradigm is ready for a component market to be established by software vendors so as to allow developers to be able to purchase and insert pre-written components. This is a vision that dates back to the late 1960's (Naur, Randall, and Buxton 1976).

In addition to saving time through the development or in some cases re-development of software components, a significant period of time can be saved by not having to test a pre-written component[1].

It is envisaged that in the near future, there will be a considerable number of software repositories from which developers will be able to acquire the latest component that they require. There have also been some advances in the architectures (refer to section 2.5) which support component frameworks to provide the system with the ability to fetch the latest version of a component (based on the pre-requisites being met) and to install it into the system.

2.2.5 Testing

As with any system, one of the most important phases is the process of testing. In traditional monolithic systems the process of testing is normally viewed as an arduous task, as large and legacy systems tend to have subsystems inter-twined with one another. This makes it nearly impossible to isolate a section of the system so as to perform isolated system testing.

[1] It is assumed that a third-party component sold by a software vendor would have already performed a rigorous set of tests upon the component.

Additionally, legacy systems normally introduce the problems associated with spaghetti code where one change made to part of the system may have consequences or flow on effects to other parts of the system. In these cases, a system test would result in invalidating all previous tests conducted upon the system. This would therefore lead to having to re-check every component within the system otherwise known as regression testing.

The component based paradigm alleviates this problem by enforcing the modularity concept (refer to section 2.2.1) which provides for components to contain functions and responsibilities within themselves. With these components having minimal connections to others within the system, the complexity and time taken to perform system testing is reduced. This is achieved by only having to test the individual component rather than re-testing the entire system. Additionally, by using this paradigm there is a reduction in the number of bugs introduced into the overall system because components are normally tested thoroughly before being introduced to a component library or directly to the system.

2.2.6 Configuration Management

To date we have covered some of the benefits that the component based paradigm provides. However there is still one benefit yet to be discussed which offers the software developer unparalleled control over the way in which the system is constructed and operates. This benefit as outlined in Kramer (1990) and Szyperski (1997) provides the ability for the software developer to specify the configuration of a system as well as being able to modify the configuration while the system is running.

The configuration/reconfiguration model which is supported within the component based paradigm provides for a mechanism which allows components to communicate with one another at both a functional level and at a binding level. This is traditionally achieved within the component based paradigm by enforcing components to hold onto 'indirect' references.

Other programming models such as the object oriented paradigm are unable to achieve this level of configuration/reconfiguration control as the components and objects deployed within the environment hold 'direct' references to one another. Figure 2.1 illustrates the differences between 'direct' and 'indirect' referencing. The use of direct referencing results in components knowing the exact location of the other component. A change to the component's location results in having an entire system search for all the references to the old component and substituting it with the address of the new location. Although this is feasible, it does provide an extra strain on the system. This still remains a fundamental problem within these programming paradigms.

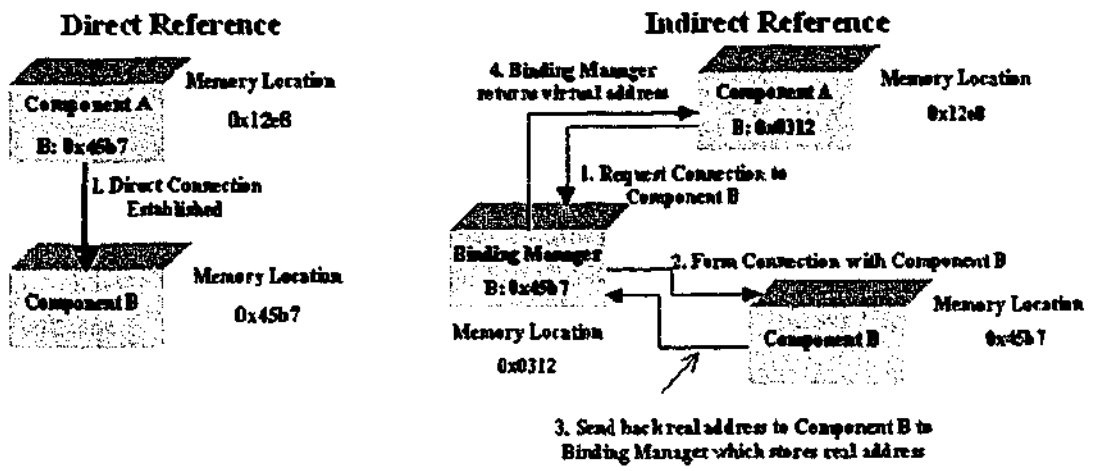


Figure 2.1: Direct Reference vs. Indirect Reference

Additionally, this technique does not lend itself well to the concepts of distribution. In single standalone machine environments the reference to the other object or component is normally the virtual memory address allocated to the component. The problem is that this virtual memory address refers to the memory located on one physical machine. When the concept of distribution is introduced it is possible that the same memory address which existed on one machine will not translate across to any other machine in the system, or worse still in some cases the memory address may point to another object located on another machine somewhere else in the network.

For the most part this is a considerable problem for the development of systems which span across many computers. However, as we will see in section 2.5 there are now a number of architectures which have been developed to address this distribution problem.

2.3 Configuration Management within the Component Based Paradigm

Configuration management was incorporated into the component based paradigm so as to provide a rich framework allowing for system developers and maintainers to interact with the configuration of the system. Additionally, this framework serves as an abstraction layer between the logical associations and physical interconnections found within each component.

When the framework is used appropriately, it allows for developers and maintainers alike to modify aspects of the system in terms of its configuration and to analyse the impact of these changes on the system. Once a configuration has been finalised these changes can then be declared as the

configuration level of a functioning system. This specification is normally made within a specialised configuration definition language (refer to section 2.4).

2.3.1 Objective of Configuration Management

The objective of the configuration management framework is to allow developers to provide a flexible system which is capable of accommodating the changing needs of the system while at the same time modelling the effects of these changes. These changes are sometimes required before the actual change is implemented in the final system.

Additionally, the configuration management system can also be responsible for recommending or slowly implementing the stages of a new configuration to an already existing configuration in a sequence which would cause minimal disruption to the running system. This is similar to incrementally changing the system through a number of stages. The conclusion of the reconfiguration process allows the configuration management system to perform a comparison between the actual results that it has observed in the system and those that were expected.

2.3.2 Explaining how Configuration Management Works

As was mentioned in section 2.2.6, configuration management relies on the use of indirect references to components. One approach as outlined in Kramer and Magee (1990), Magee, Kramer, and Sloman (1989), Sethuraman and Goldman (1995), Kramer, Magee, and Young (1990) is the introduction of a 'Binding Manager'. This arrangement of components and their relationship with a binding manager is explained in the next section. By using a binding manager the concept of indirect references are further embraced.

2.3.2.1 Binding Manager

The binding manager plays a central part in the configuration management system. It is responsible for maintaining the connections between all the components within a system. In addition to maintaining the connections the binding manager is responsible for identifying when it is safe to remove and/or add a binding from/to the system and assess what impact this action may have on the rest of the system.

In order to provide the functionality required to manipulate the interconnections of components, the binding manager must provide some functionality with regards to management. This management is achieved through a sequence of operations which results in the component having its state become quiescent. A complete explanation as to what quiescence is and how it can be achieved within components can be found in Kramer and Magee (1990) and Kramer, Magee, and Young (1990). A brief summary of the key points for quiescence is presented below.

The process of managing an interconnection between components requires a considerable number of operations and must be performed in a logical sequence. These operations include:

- Create — This involves creating the actual component within the system. It is at this stage that the operator can either develop their component interactively with the system or can import it from somewhere else. The configuration manager also registers the object so that it can refer to it at a later date.

- Passivate — This is the process of placing a component into a state which does not make it interact with any other components within the system. This effectively withdraws the object from 'active' service. The first stage in passivating the system is to inform the components (done via the binding manager) that the component is not available. If by an off chance a request is made upon a component which has been passivated, the traditional response will be for that request to be 'frozen' until the component is reactivated.

  Once this stage has been reached the component then withdraws from actually initiating any new interactions with the remaining components in the system. After a period of time, quiescence is reached as the component which is being passivated stops/finishes interacting with the other components. In some cases, most notably those components which do internal processing, there will never come a point in time where the component is totally quiescent. In these cases it is assumed that after a given period of time the component can be said to be quiescent.

- Activate — This operation is responsible for activating the new component with the system. As the component receives the 'activate' operation, it initialises itself and prepares its bindings to be attached to the rest of the system via instructions specified by the binding manger.

- Active — This operation is responsible for informing the binding manager that the component is active and is ready to receive requests via the intercomponent communication channels which have been established (refer to section 2.3.2.2). Additionally, the call to 'active' informs the binding manager to send on those requests which have been in a state of 'deep freeze' while waiting for the component to become active in the system.

- Link — This is one of the more important operations that a binding manager and overall configuration management system can perform. This operation is used to maintain an index of the links flowing from one component to another (ie. which components are linked together). In general terms the operation 'link' is responsible for the addition of links into the central index (maintained by the binding manager). It is this index which is used to determine whereabouts a method request will be dispatched. As can be appreciated, it is critical that the indexing system remains consistent.

- Unlink — In addition to managing the links between components, the binding manager must also have a method of being able to de-register those links which have been nominated by a developer as being no longer useful. The 'unlink' method is used to remove those relationships from the index within the binding manager. Operations such as 'unlink' require the system to perform a comprehensive check to determine any possible side-effects. Side-effects can include the attempted removal of an active binding or the attempted removal of a binding which is currently being reconfigured. Such side-effects if they exist are required to be acknowledged before any reconfiguration takes place. Once the analysis has been completed and the impact to the system has been assessed, the reconfiguration of the relationship index is performed.

2.3.2.2 InterComponent Communications

As previously mentioned, components in a component based paradigm lack any direct communication with one another. Instead, each component communicates with another through the use of a binding manager (refer to section 2.3.2.1).

Each component within the system has a proxy address which acts like a pointer to the required functionality located within another component. This proxy address is actually a pointer to an entry which is held within the binding manager.

Along with each address which is held within the binding manager is a significant amount of other information. This information can be of a security nature which dictates who is able to call the method and when that function can be called, statistical information which holds information such as the number of times the component has been accessed, or other auxiliary information which is required to answer questions regarding the status of the binding.

As is evident the binding manager must then maintain a list of two addresses. The first address is the proxy address which is used by the components as a primary key in the lookup table. By using the first address, the binding manager is capable of determining the real memory address of the desired components interface.

Before the binding manager 'puts through' a request to the desired component, a number of house-keeping functions must be performed. The first of these functions includes checking on the status of the binding and determining whether the binding is actually suitable for use. If the binding is available then the binding manager processes the request and indicates that the link/binding is currently being used. Once the method invocation to the component is complete, the binding manager records that the binding is free once more and updates its records accordingly. However, if during the checking the binding manager determines that the binding is not available, the binding manager 'blocks' or 'freezes' the method invocation and waits for the component to become available once more.

Overall, the binding manager is able to control how a component interacts within the system through the use of a specified set of interfaces. These interfaces were briefly covered in section 2.3.2.1. As mentioned the `passivate` interface allows for components to become quiet while reconfiguration takes place. Once reconfiguration has taken place the `activate` interface allows the component to rejoin the system and to process requests once more.

As a result these interfaces allow the binding manager to exert its authority over the components within the system. In addition the binding manager can enforce its own binding protocols if desired. These protocols can either replace the notion of quiescence within the system or can be used in parallel to allow the system designer to have a lot more control over the components. Some typical examples of what might be added to the configuration manager include the ability to buffer incoming requests to components which are not available at that point in time or the ability to provide the system designer with performance information such as how long is it taking for a method to get to a component or how long is it taking a component to process a request.

Through the use of the binding manager the developer of the system has the capability of being able to tune the way in which requests are dispatched to components within the system. Additionally, with the introduction of the proxy address concept there is scope for the system to redirect the function call to another module located within the system.

2.3.2.3 Manipulating Components

The need to manipulate components is fundamental to the operations of the configuration management system and the binding manager. As has been mentioned, the reconfiguration of components and the redirection of bindings is achieved through the manipulation of the proxy address and the real address held in the binding manager.

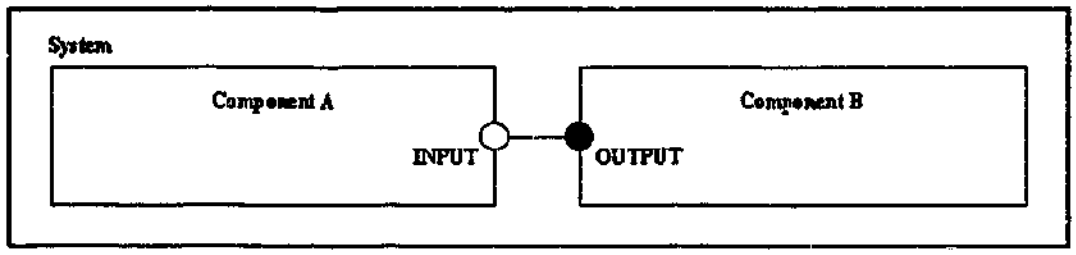
To the 'caller' everything remains static as the communication channel is established with the binding manager and not the direct component. It is within the binding manager where all of the adjustments are made to the proxy address. With regard to the 'callee', the address given between it and the binding manager remains constant. The switching of the component requests is done within the binding manager. This is similar to the way early phone exchanges worked where both parties wanting to talk to one another were connected to the phone network, but had to dial the phone exchange for a link to be made between the caller and the callee.

2.4 Supporting Configuration Management

Although the configuration management system allows for components to be manipulated within the system, there is a need for these changes to be recorded in a declarative manner.

This is achieved through the use of a Configuration Definition Language (CDL). CDL's have been designed as a method of being able to aid the software developer in encapsulating the semantics of a system while it is being analysed and designed.

Traditionally, these languages are purely descriptive and contain no constructs for any implementation. CDL's normally identify all of the components within a system, their interconnections, their location within in a distributed system and any other special relationships which exist. An example of a CDL and the system that it describes is shown in figure 2.2.




```
Component A {
require input;
}
Component B {
provide output;
}
```

```
Component System {
inst
  Component_A: A
  Component_B: B
provide output;
bind
  Component_A.input -- Component_B.output
}
```

Figure 2.2: Sample Configuration Definition Language & System

It is from these languages that a more detailed description of the system can be generated or with the assistance of some automated tools, the configuration of a system specified in CDL can be used to generate the appropriate classes and stubs for an implementation of the system. Section 2.4.1 introduces the DARWIN configuration language which is commonly used to represent the configuration of a system.

2.4.1 DARWIN Configuration Language

DARWIN as defined in Dulay (1992) was developed with the objective of being able to provide a description of a system which details the structure of those components within it, along with their hierarchical arrangement (Pryce and Crane 1996; Magee, Dulay, Eisenbach, and Kramer 1995).

In addition to storing the definitions of structures and their hierarchy, DARWIN also records the interconnections between those components which have been identified within the system.

Most of the concepts that the DARWIN configuration language embraces were originally developed as part of the CONIC project (Magee, Kramer, and Sloman 1989). The DARWIN configuration language was principally developed to act as the configuration language to the REX project (Magee, Kramer, and Sloman 1990; Kramer, Magee, Sloman, Dulay, Cheung, Crane, and Twidle 1991) but since then has been adopted by other systems as a standard for describing the notation of components, system structures and their dynamic interactions.

To achieve this, the DARWIN configuration definition language provides further enhancements to the semantics available to model the dynamic nature of a system. Earlier configuration languages focused on the need to provide static support and did not provide the same level of semantics required for recording the dynamic nature of components. DARWIN addresses this issue through the introduction of a `dynamic` keyword.

Although there have been a number of configuration definition languages and systems developed recently such as CONIC (Magee, Kramer, and Sloman 1989), DARWIN (Dulay 1992), REGIS (Magee, Dulay, and Kramer 1994) and PONDER (Damianou, Dulay, Lupu, and Sloman 2000), it is interesting to note that the most commonly used configuration definition language is DARWIN.

A contributing factor to the success of DARWIN as a configuration definition language is its richness in notation. As mentioned before, DARWIN provides support for the configuration of a system to be specified declaratively and to include support for dynamic systems. However DARWIN also provides the ability for the configuration to be represented graphically. One tool as described in Kramer, Magee, and Ng (1989) which is currently available is called the Software Architect and allows the system developer to specify the configuration of a system in either a declarative or graphical manner. Figure 2.2 shows an example of a DARWIN component declared declaratively and graphically.

DARWIN provides support for components to interact with one another through a concept known as a 'port'. This idea of a port was originally developed in CONIC but has been further enhanced within the DARWIN specification.

However, one problem that DARWIN does not address is the issue of type checking between the different components and at the various ports that exist within the system. The decision not to introduce type-checking at these points was a bi-product of the CONIC project where the issues of type checking were not fully examined nor explored. The decision to keep type-checking out of DARWIN was made on the basis that system developers would make the appropriate decisions with regards to combining components. This approach or lack of type-checking has sparked a number of research groups into investigating the possibility of inserting type-checking into the DARWIN configuration language itself or into a DARWIN configuration language clone.

Another reason why DARWIN is commonly used as a configuration definition language is its ability to abstract lower level concepts of the system into a higher component. By providing this level of abstraction the developer can identify sections within the system and abstract them into a component which can then represent the collection of functions or role within the system. This is very similar to the concept of 'partitioning'.

Additionally, this level of abstraction helps the system developer illustrate the real designs of the system as opposed to a system which is clouded with a number of details pertaining to the lower levels of the system. DARWIN is capable of supporting these abstraction techniques with the help of a number of concepts. These concepts include:

- Importing Services
- Exporting Services
- DARWIN Components
- Services
- Levelling

2.4.1.1 Importing Services

When DARWIN components are constructed they are designed to provide the core functions which are responsible for achieving the task at hand. In certain circumstances, a component may require the functionality of another component to achieve its objective.

Where a component requires a service or functionality that it does not provide itself, it must seek out the required functionality from another component. It is at this point that a relationship or connection between two components must be established. Hence, when the first component requires a service provided by another component to achieve its goal, it is said to have `imported` the function or service.

In order to make this happen, the DARWIN configuration language provides a keyword called `import` which notifies the configuration manager that this component *requires* another component to be present before it can do any processing. The `import` keyword is used by the system to ensure that its dependency tables are kept up to date.

2.4.1.2 Exporting Services

The term 'exporting' a service is similar to the concept which was discussed in section 2.4.1.1 for importing services, except that in this case it happens in the opposite direction. A component which is capable of providing a service to other components is said to `export` that service if another

service *requires* it. By default any functionality that is provided within a component is not set to be exported by default. The component developer must specify that they wish to export the service.

Once again the export keyword is used by the system to ensure that its dependency tables are kept up to date.

2.4.1.3 DARWIN Components

The concept of a component within the DARWIN configuration definition language is exactly the same as those components which were defined in section 2.1.1. Components within DARWIN only need to communicate with one another if they require the services of another component as is outlined in section 2.4.1.1. These components can normally be identified through the use of the DARWIN keywords import or export.

2.4.1.4 Services

The term services is used within the DARWIN configuration definition language to describe the functionality of a component. This functionality can either be imported into another component or exported from a component. The issue of services becomes of considerable importance when the design of the system dictates that a series of components are grouped or clustered together in a manner to abstract part of the system.

2.4.1.5 Levelling

As with most information systems, modelling techniques and concepts such as Data Flow or Entity Relationship diagrams play an important role in the modelling of the business domain (Burch and Grudnitski 1989). Additionally, these techniques are quite useful in identifying errors in the design through a process known as levelling which allows the designer to partition the system into a series of layers, each of which vary in technical specifications.

The DARWIN configuration language when it was being designed provided for this process of levelling. By allowing levelling, the developer can start with a diagram which illustrates a very high level picture. Typically, a diagram at this level could be used to convey the basic ideas of a system to management while a more detailed diagram lower down in the system could be used for analysis, design and implementation. In DARWIN this concept is expressed as components being nested within other components.

Levelling also helps the software developer to identify problems which exist within the system. As one level of the system is developed and then expanded to reveal the layer below it, all of the external connections from that level are carried through to the next level and hence must be matched and accounted for. Use of this levelling process aids the developer in identifying situations where there

are mismatches within the system and provides the opportunity for the appropriate actions to be taken.

It is for all of these reasons that the DARWIN configuration language is used for the REX system which provides support for developing systems which span a distributed network (Magee, Kramer, and Sloman 1990; Kramer, Magee, Sloman, Dulay, Cheung, Crane, and Twidle 1991).

2.5 Supporting Architectures for Component Based Paradigm

To date, section 2.1 has focused on the fundamental concepts and terms which are pertinent to the component based paradigm. However these are terms and definitions which do not deal with the implementation details.

This section examines some of the more common architectures generally available to software developers which facilitate the development of systems by using a component based approach. These architectures are not to be confused with those systems detailed in chapter 3 which actually provide an environment from which systems can be constructed and reconfigured. This section only examines the available architectures.

2.5.1 Distributed Computing Environment

As mentioned in Yang and Duddy (1995), the Distributed Computing Environment (DCE) was developed by a consortium of software companies known as the Open Software Foundation (OSF)[2]. At the time that OSF was formed, there was a considerable amount of tension within the computing industry as alliances had just recently been formed for the development of the first system to allow programming in a distributed domain. This tension as detailed in Bloomer (1992) ended with the creation of the 'Network Computing Architecture (NCA)' as a result of an alliance between Apollo and Hewlett Packard and 'Open Network Computing (ONC)' system developed by SUN Microsystems.

Although both the ONC and NCA systems addressed the issue of distributed computing, they were both of a proprietary nature costing programmers a considerable amount of money to use. Additionally, both systems were inflexible as they were suited to one particular environment from a vendor. After a period of time, the demand for a distributed programming environment grew, and the OSF was formed. OSF decided to address the distributed computing programming paradigm problem by providing both an architecture and an implementation.

While the OSF was in the process of being formed, another battle was looming between AT&T and the rest of the computing industry. This battle was sparked by the dominance of AT&T in the UNIX

[2]This group is now known by the name of XOpen.

market and their successful application to have the word UNIX copyrighted. As a result of these transpiring events, there was even a greater determination for the various software and hardware vendors to band together and develop a truly unified and standardised version of UNIX (Martin 1995). This determination resulted in the first unified UNIX operating system being developed. The operating system was codenamed 'OSF/1'. Since then OSF/1 has been superseded by 'Digital Unix' which has been designed to embody all that OSF/1 aimed to achieve.

To ensure that DCE would be successful, the DCE design committee decided to develop a goal for their system. This goal simply put is "*to provide an architecture which allows developers to build distributed applications independent of the lower level functionality*".

2.5.1.1 DCE Architecture

Figure 2.3 illustrates how the DCE architecture is structured. From the figure it is possible to see that the architecture is modular in nature, allowing for certain areas of the DCE system to be replaced as required. This replacement strategy could be utilised in the instance of a new threads service having to be inserted into the system as a result of an operating system change.

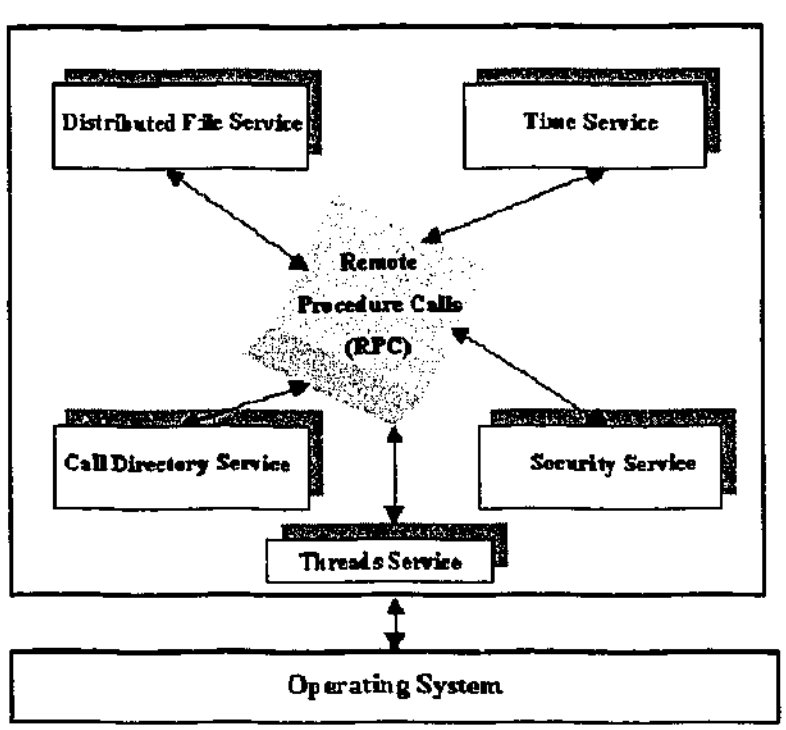


Figure 2.3: DCE Architecture

File Service Module

The file service within DCE is responsible for providing a standardised interface to those files located within the DCE system. As part of its responsibility, the service provides a complete list of files (traditionally represented in a hierarchy structure, similar to that found on PC based operating systems) which spans across multiple systems to the user upon request. Files located on other systems throughout the network appear just like a local file would in the hierarchy. It is the role of the file service to provide transparent access to these files no matter where they might reside. A cut down version of the same file system is available on a number of UNIX platforms and is known as the Andrew File System (AFS).

Additionally, the file service is responsible for ensuring that incoming requests for files have the appropriate security privileges. This is done in consultation with a security service which maintains Access Control Lists (ACL's) that govern the usage of various resources. For example, UNIX makes use of ACL's to control access to various parts of the file system. In brief, an ACL is associated with every resource and contains a list of those users or groups who have privileges to read from the resource where appropriate and those that have write access. It is important to note that an access privilege to read a resource does not necessarily imply that the user also has write access.

Security Service Module

The security service is responsible for controlling access to all components within the DCE system. As mentioned previously, this is also responsible for managing the ACL's which the file service makes use of. Effectively, the security module takes requests from a client known as a *principal* which contain details on the user and the requested service and then performs an authentication process.

If the request can be authorised then it proceeds otherwise it is rejected and sent back to the user. To ensure the security of the information being passed to the security module, the KERBEROS data encryption protocol is used.

Directory Service Module

The directory service should be renamed to the locator service as it is responsible for knowing and tracking the locations of all the resources available within the DCE system. As DCE facilitates the networking together of computers, it is possible to identify that the directory service might have to maintain a large amount of data dispersed over a wide area.

As figure 2.4 illustrates, the directory service is partitioned into a series of levels and sections. At the top of the directory module is the Global Directory Service (GDS) and the Domain Name System (DNS). The DNS component of the directory service is a third-party system and interfaces with the networking components of the installed operating system (Transarc Corporation 1996).

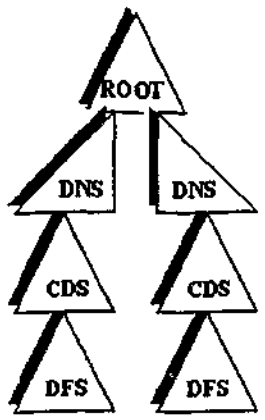


Figure 2.4: DCE Directory Services Namespace

On the GDS side, we can see that the system is partitioned into two lower levels known as the Cell Directory Service (CDS) and Directory File Service (DFS) or the file space as described earlier.

It is important to note that the directory service only stores information pertaining to the name of the resource and contains no information or attributes about the resource itself. Such information can only be retrieved after the directory service has located the object and returned the results to the client. Once the client has the information it can then form a direct connection with the resource required and obtain the relevant information.

Searches within the DCE directory structure are hierarchically based. If a resource can not be located within the current cell, the global directory service is contacted to assist with the search.

Within the directory service a considerable amount of information is stored. Transarc Corporation (1996) provides a comprehensive listing of all the entities which are stored within one cell located within the directory service.

Time Service Module

With the distributed computing environment networking a number of machines, each of which has its own time clock and in some cases spanning a number of time zones, comes the problem of being able to provide a *synchronised* time that all the machines can agree to. This becomes important when entering into time critical functions such as database transactions or accessing and locking disk files or providing consistent auditing information.

To address this problem, the DCE architecture synchronises on one time measured from one point in time. This time is calculated by the number of seconds that have elapsed since 15th October 1582. This date happens to be the start of the Gregorian calendar[3].

[3]Refer to van Helden (1995) for an in-depth discussion on the developments of the Gregorian calendar.

By using this time base, all resources and components within the system are able to maintain a consistent time. Needless to say, that all time calls made by resources go through the DCE time service.

Remote Procedure Call Module

The Remote Procedure Call (RPC) module of the DCE system forms the cornerstone of allowing components and resources to communicate with one another. RPC which was a bi-product of the development of the Network Computing System (NCS) as mentioned in Bloomer (1992), provides the mechanism for components to send *low level* messages to one another. It does this through the aid of an Interface Definition Language (IDL) which allows developers to specify the interface to the resource.

This interface definition file (refer to figure 2.5 for an example of a interface definition file for DCE) is compiled to provide a series of stubs which are then used to marshall the relevant data into a format known as XDR (SUN Microsystems 1987). The resulting stub code is used for handling the transmission and receiving of data to and from these resources.

```
/* Example of a DCE IDL File */
[uuid(6db16365-51d6-472f-b93f-00ab1de29c6c),
 version(0)]

interface subtract
{
 [idempotent] void subtract_op
 (
  [in] handle_t clientHandle;
  [in] long number1,
  [in] long number2,
  [out] long *result
 );
}
```

Figure 2.5: Example of a DCE IDL File

The DCE system was one of the first architectures to introduce an interface definition language responsible for the decoupling of the interface from the implementation. The basic concepts of the DCE/RPC system have now been embraced within the development of other architectures which provide the ability for distributed programming. The most notable architecture which has adopted the RPC mechanism is Microsoft's DCOM (refer to section 2.5.3.1).

Threads Service Module

The threads component of the DCE system is designed to provide the developer with a standardised interface into the threading mechanism used by the operating system regardless of whether the

operating system is using native or green threads and is irrespective of the operating system being used. To ensure future compatibility the DCE thread interfaces were designed to conform to the POSIX.1 standard (Zlotnick 1991).

The thread subsystem was also engineered in such a manner as to provide the developer with some additional flexibility. In some operating systems an interface into the threading system is unavailable. To address this situation the DCE system has also been engineered to allow a threads library or package to be compiled into the system. This library then acts in the place of the threads sub-system within the DCE architecture.

Operating System

As shown in figure 2.3 the lowest level module on the DCE architecture is the operating system itself. Although there is nothing to specifically discuss with regards to the operating system it is important to note that the operating system must have the appropriate hooks in place to allow such DCE services as threading and RPC systems to operate.

The first operating system to have a fully integrated DCE system was OSF's own operating system OSF/1. Since then Digital UNIX which has superseded OSF/1 and other operating systems has continued to provide support for the Distributed Computing Environment (DCE). DCE today is still one of the more popular distributed programming architectures currently available for systems which do not require the full benefits of the object oriented programming paradigm.

2.5.2 Object Management Group's Approach

Although the DCE system provides an architecture for developers to construct distributed systems, it does however lack additional functionality which is becoming more relevant in distributed systems today. This extra functionality can be in the form of services which provide transaction based management, concurrency or persistence. An architecture which provides a solution to this problem is the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG).

2.5.2.1 Object Management Group

In 1989 a number of major software vendors including Hewlett-Packard, SUN Microsystems, Canon and 3COM came together to form the Object Management Group (OMG) in an attempt to be able to provide a standards organisation which could help facilitate the first draft of a complete distributed programming environment[4] (Ben-Natan 1995). Since then the member numbers of OMG have been steadily increasing. Over the last 5 years, member numbers have increased on average by 100 per year. In 1995 when the second edition of the CORBA specification (Object Management Group

[4]It was recognised at the time that DCE provided a good stepping stone for distributed programming but did not provide a complete solution.

1995b) was released the reported number of members belonging to the OMG was 500, in 1996 it was 600 as mentioned in Siegel (1996) and at last count the number of members was well over 800 (Object Management Group 2001).

Just like the OSF, the OMG when being formed developed a charter to help steer the direction of the members. The charter for the OMG as defined in Ben-Natan (1995) is to "*promote the use of object technology and to provide an architecture which facilitates the development and deployment of distributed object oriented systems*".

However, unlike OSF's approach to DCE and Microsoft's approach to DCOM (refer to section 2.5.3.1), the OMG is merely a standards organisation responsible for the development of the architectural concepts only. The OMG does not provide any implementation for the architectures that it specifies. As a result the OMG consists of a number of committees which oversee the design process from a management level while technical committees and working parties are formed to deal with the more specific issues.

2.5.2.2 Object Management Architecture

The cornerstone behind the OMG's architecture is the Object Management Architecture (OMA) as described in Object Management Group (1997). This architecture is broken up into two smaller models known as the Object Model (OM) and the Reference Model (RM). Although both are not implementation models, the OM presents more of a conceptual view of the architecture while the RM builds upon those concepts described in the OM.

Object Model

The role of the core object model within the object management architecture is to provide a convenient location where the concepts and terms relating to object oriented programming can be defined in such a context as to make sense to the clients who will be connecting to the system. Additionally, by having the object model focus on the concepts of object oriented programming rather than implementation, the developers can promote portability as a key component to the object model. Here is a brief explanation of the terms that the object model defines in Object Management Group (1997):

Object: An object within the core object model is an entity which can be modelled within the real world. Traditionally, an object encapsulates both data and functionality which is relevant to the real world entity. It is expected that this object will then make available some of this functionality so that others can interact with it.

Request: A request is an event made by a client to an object for a service. It normally contains information such as the operation requested, parameters which are to be passed to the object and parameters which are to be completed by the object and returned to the client.

Object creation and deletion: It is important to note that there are no special mechanisms in existence for clients to be able to control the creation or deletion of objects. Clients will however see the effect of an object creation through a new object reference being allocated to the object.

Types: The object management architecture identifies a type as being an identifiable entity which has a predicate associated with it. A type is considered to be valid if it satisfies the predicate of that type. The type system is then further refined to differentiate between basic types or constructed types. Basic types as defined in Object Management Group (1997) are considered to be:

- 16 & 32 bit signed and unsigned 2's compliment integers
- 32 & 64 bit IEEE floating point numbers
- Characters which conform to the ISO-Latin 1 standard
- Boolean types
- A binary stream guaranteed not to undergo any transformations during transmission
- Enumerated types
- A string type which consists of a variable length of characters
- An 'any' type which can represent either a basic or constructed type

Constructed types are defined as:

- A record structure defined with the `struct` keyword
- A discriminated union type
- A sequence type which can represent an array of elements with a single type
- An interface type which specifies a set of operations to be performed

Interface: An interface is an operation or a group of operations which can be performed upon an object. This interface is the entry point to the functionality encapsulated within the object.

Operation: An operation refers to the internal functionality which exists within the object. Requests are made to operations contained within objects.

Attribute: The term attribute refers to a piece of data which can be individually associated with an object. Additionally, the term *attribute* is used within the CORBA environment to provide both `get` and `set` methods to manipulate the internal data.

Reference Model

As was mentioned previously, the reference model builds upon the founding concepts which were defined in the object model. Additionally, it is responsible for identifying those components, interfaces and protocols which exist within the object management architecture.

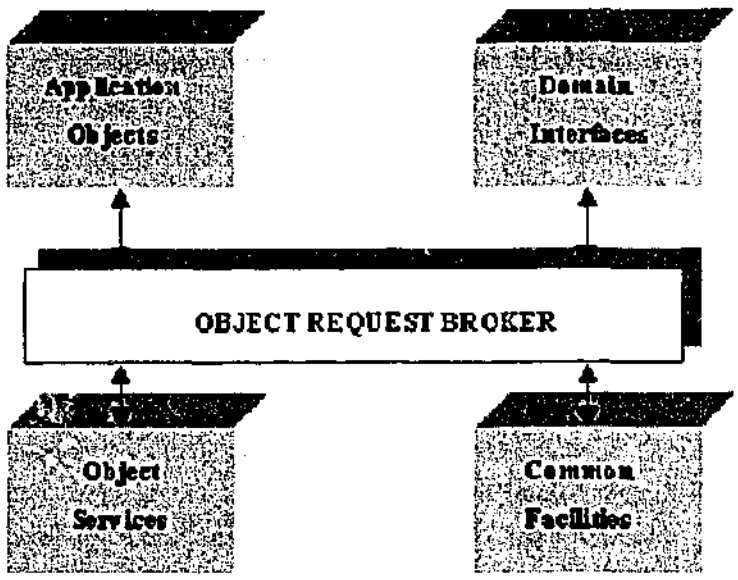


Figure 2.6: OMG's Object Model Reference Model

From figure 2.6 it is possible to see that the reference model has been divided into four discrete units which are interconnected with one another through the Object Request Broker (ORB).

Application Objects

Application objects are those which are constructed by the developer for a problem domain which is specific to the end user. As a result of each application object which is developed being unique for the developers' own domain, the OMG does not specify nor expect any particular object written within this domain to conform to any known standard.

Domain Interfaces

The domain interfaces module is responsible for the grouping together of those interfaces which belong to a specific domain. Examples of these interfaces include finance, telecommunications and electronic commerce. To allow for the reuse of these interfaces within different organisations which operate within the same domain, the OMG specifies a particular standard to which developers must program. This ensures that objects developed for this domain are fully compatible with other systems.

Object Services

Object services are responsible for providing objects with the basic low-level functionality required for an applications development. A sample of the services standardised by the OMG include persistency, transaction control, concurrency, security and timing. A complete list of CORBAServices can be

found in Object Management Group (2001) or Object Management Group (1998)[5]. It is important to note that all services provided to objects are independent of any application domain.

Common Facilities

Common facilities are similar to object services except they are geared more towards the application development rather than providing a service to the object itself. Additionally, common facilities represent a higher level of abstraction than object services. As is specified in Object Management Group (1995a), the OMG has standardised upon the following:

User Interface: The user interface facility is responsible for covering all the aspects related to user interface design. The interface allows the developer to specify the *look and feel* of the user interface with respect to the design tools located on the system. In addition to providing the user with the control elements to the user interface it is also responsible for giving the client easier access to the data and automating some of the tasks required.

An example of the interfaces supported include providing for operations to allow the rendering of data for output to screen or printer as well as providing a framework for allowing the developer to build a compound interface environment. This is an environment in which the user has various pieces of information presented to them on a screen.

Information Management: The information management facility provides an interface to allow data for an enterprise/organisation to be stored, retrieved and manipulated in an efficient manner. The interface provides options for the modelling, storage, retrieval and encoding/decoding of data for transmission as well as performing the transmission itself.

System Management: As the name suggests, the system management facility is designed to allow developers and end-users to control the management of the system. The interfaces supported in this facility abstract such tasks as controlling the system, monitoring on-going processes, providing an access point so that security can be managed as well as providing a method for policies to be specified outlining the level of control provided to an individual developer or group of developers.

Task Management: The task management facility aims to provide an infrastructure which allows for the modelling of tasks while at the same time providing the functionality required to manage users, production workflows, rules governing the system and communication activities between tasks. The task management facility also provides support for agents to examine the tasks within the system.

Vertical Market: The vertical market facility aims to provide users and developers with standards which allow interoperability in specific problem domain areas. This is a relatively new addition to the common facilities module and is constantly under going modification. Some

[5]The OMG is constantly issuing request proposals for new services to be added to the OMA.

examples which have already been specified include imagery, support for the information superhighway, manufacturing processes and distributed simulations. For a complete list of vertical markets, refer to Object Management Group (1995a)[6].

Object Request Broker

The Object Request Broker (ORB) as shown previously in figure 2.6 is responsible for linking together all of the other components which exist within the reference model. In addition to linking components, the ORB is responsible for providing the central communications backbone between objects. One of its tasks is to allow requests to flow between objects within the system while at the same time providing this form of communication in a transparent manner. By achieving this level of transparency, the client only needs to hold onto an object reference and use this reference to contact other objects without having to know where the object is physically located. In addition to coordinating the transfer of requests between components, the ORB is also responsible for the marshalling the data in such a manner as to allow the data to cross machine byte-ordering boundaries. This ensures that big-endian and little-endian systems can communicate with one another without having to worry about primitive data types being corrupted. This is achieved with the help of an interface definition language which provides some separation between the interface and the implementation. The ORB is also responsible for detecting and capturing any errors which may result from a request from an object. If such an error does occur (commonly known as an exception), then the ORB is responsible for marshalling this data and sending it back to the client.

2.5.2.3 Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) object model takes those terms and definitions defined within the core OMA object model (refer to section 2.5.2.2) and extends them in such a manner as to provide a real or tangible concrete model from which an architecture supporting distributed object oriented programming can be formed. As the CORBA object model extends the core OMA model, all of the definitions such as requests, operations, interfaces and types, both basic and constructed continue to carry the same meaning.

Interface Definition Language

The role of the Interface Definition Language (IDL) within the CORBA architecture is to provide a level of abstraction between the interface and the implementation of the object. This abstraction aids developers in the design of the system as it provides a clear separation between the clients responsibilities and the servers. As a consequence, all objects developed under the CORBA specification must have their interfaces expressed in terms of an IDL.

The IDL is a language and implementation neutral environment. Effectively, this means that the keywords used within the language are not dependent on any particular language nor does the

[6] As new interoperability standards are constantly being introduced, http://www.omg.org provides the latest information.

language provide any support for the programming concepts of *iteration* or *selection*. However, the OMG does admit in Object Management Group (2001) that the interface definition language was based on the C++ language and will most likely be modified to follow the standardisation path that C++ takes. Table 2.1 shows keywords which are reserved within the CORBA IDL.

any	attribute	boolean	case	char
const	context	default	double	enum
exception	FALSE	fixed	float	in
inout	interface	long	module	Object
octet	oneway	out	raises	readonly
sequence	short	string	struct	switch
TRUE	typedef	unsigned	union	void
wchar	wstring			

Table 2.1: Reserved CORBA IDL Keywords

Once the interfaces are defined in the IDL, they are compiled through a IDL compiler which generates the appropriate client and server side stub and skeleton files. The stub code which is generated contains routines which are responsible for the marshalling and unmarshalling of data, and the passing of the request to the desired object through the object request broker. The skeleton code generated during the IDL compile process provides the developer with a basis from which they can develop code. The skeleton code normally produced by the compiler contains the method calls, the parameters being passed to and from the object expressed in CORBA types and any headers which may be required for the program to compile.

As is mentioned in Object Management Group (2001) the goal of the OMG is to provide the ability for developers to create systems which span over many distributed environments. In order to provide developers with this flexibility, the OMG has standardised a series of language bindings which allow developers to write objects in any of the languages supported. For example, due to the decoupling of the interface and the implementation, it is possible for Java clients to talk to C++ servers or vice versa. The language bindings currently supported by CORBA as specified in Object Management Group (1999) are:

- Ada
- C
- C++
- COBOL
- Java
- Smalltalk

Components of the Common Object Request Broker Architecture

As can be seen in figure 2.7, the underlying architecture of the CORBA model is very modular in nature and hence provides the basis for a very powerful architecture. The modular approach was adopted to allow components within the architecture to be designed for one specific purpose. This therefore provides the ability for components to be updated/upgraded within the model without causing massive disturbances to the remainder of the system.

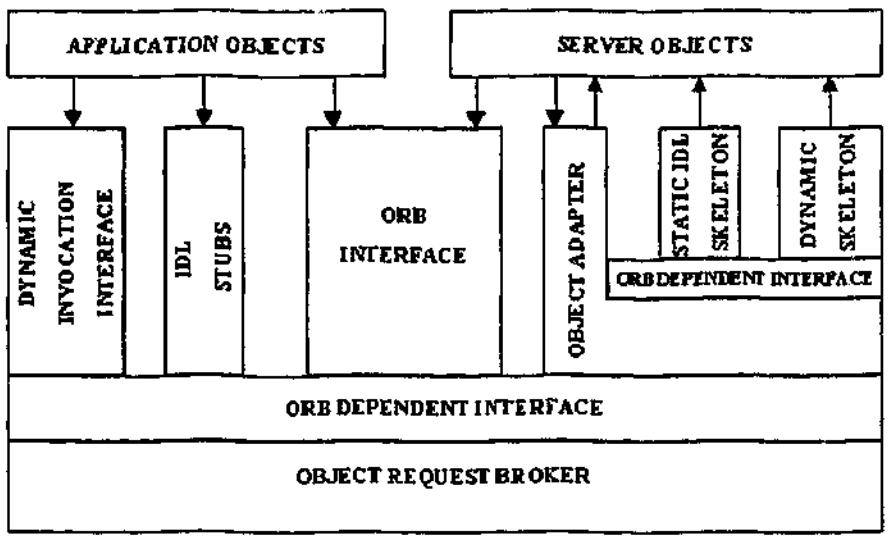


Figure 2.7: CORBA Architecture

The components which make up the CORBA architecture include:

- IDL Stub Interface
- Dynamic Invocation Interface
- ORB Interface
- Object Adapter
- IDL Skeleton Interface
- Dynamic Skeleton Interface
- Object Request Broker

IDL Stub

The IDL stub is responsible for providing an interface between the application object and the Object Request Broker. It is specifically designed to handle those requests which come from a client and are static in nature. Static requests are those requests which were specified in the source code with a specific object type. Traditionally, these operations involve a specific object types which are known to the compiler at compile time.

Dynamic Invocation Interface

The Dynamic Invocation Interface (DII) is responsible for providing an interface between the application object and the Object Request Broker. This interface is specifically designed to provide support for requests which come from clients and are dynamic in nature. Dynamic requests provide developers with additional flexibility as client applications can be programmed to use both objects and method calls which were unknown at the time of compilation. Clients making use of the DII are responsible for discovering a number of parameters required by the interface to process the method call. The parameters include locating the object providing the service, specifying the name of the service to be used, determining the number of parameters and their types expected by the service and determining the flow directions of parameters. The information pertaining to the layout of a request is stored in the Interface Repository which is discussed later in this section.

From the component architecture point of view, this interface is responsible for performing checks on the incoming request to ensure that the request is valid. These checks involve ensuring that the request actually exists and that the parameters are of the right type, right ordering and in the right direction. If a request fails any of these tests then an exception is returned back to the client.

Just like the IDL system, dynamic requests can be either asynchronous or synchronous in nature.

ORB Interface

The ORB interface is responsible for providing an entry point into the internal workings of the ORB for both application and server objects. This interface allows either clients or servers to directly manipulate the controls of the ORB or to make use of functions which are specific to the ORB's operations. The most common method calls made on the ORB include `string_to_object(...)` or `object_to_string()`. These routines are commonly used in conjunction with dynamic requests from clients which have a string reference to an object and need to obtain a handle to an object.

Additionally, the ORB interface provides functionality such as the ability to modify the timeout periods for objects that are not responding to method calls right through to the ability of modifying the behaviour of a protocol while it is connecting to a new server object.

Object Adapter

The Object Adapter Interface (OA) is responsible for providing an abstraction layer between the server objects and the functionality which is provided within the ORB. For most operations performed on the server object, the Object Adapter will be responsible for interrupting and forwarding

the request onto either the server object or the ORB. As is described in Siegel (1996) and Object Management Group (2001). The Object Adapter is responsible for the following functions:

Registering Server Objects: One of the most important tasks that a distributed system must attend to is its knowledge of what objects are registered in the system and where those objects are physically located. The OA provides facilities which allows developers to register their objects within a Implementation Repository. This is a structured storage area which maintains the name of the object, its location and the path where the object is located along with any other additional command line parameters. It is important to note that the registering of an object does not mean that the object has been started. That is a task performed at a later date.

Managing Object Reference: Just as important as knowing where the objects are is the process of being able to reference those objects which are currently running. The OA is responsible for managing all of the object references within the system so that when a client request arrives, the OA knows exactly where to dispatch the method call.

Activation and Deactivation: In an effort to minimise the resources required to maintain a distributed system, objects are activated and deactivated as necessary. The process of activating an object involves finding the location of the object and starting it up based on the information which is available within the Implementation Repository. Once started, the object registers itself with the ORB and the server is deemed to be active. The process of deactivation occurs normally when the server object has not been referenced for a set period of time. Once this period has elapsed, the server object sends a message to the ORB and is unloaded from memory. If another request arrives for the object which was just deactivated, the entire process of reactivating an object must be followed.

Method invocation: As was mentioned in the introduction to the Object Adapter, the OA is responsible for passing method invocations to and from the server object to the ORB.

Integration with other services: As the OA is responsible for interfacing the server objects with the ORB, it makes sense that the OA would provide support so that method invocations could be trapped and referred to other services. A typical example might include a security service which needs to monitor all method invocations going to and from a server object. The OA provides an excellent opportunity to provide redirection services.

IDL Skeleton Interface

The IDL Skeleton Interface is responsible for passing static requests to and from the server object. The nature of this interface is very similar to the IDL stub interface which exists on the client side.

Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) is responsible for passing dynamic requests to and from the server object. The nature of this interface is very similar to the DII which exists on the client side. Just like the DII, the DSI makes use of information located within the systems Interface Repository.

Object Request Broker

The Object Request Broker (ORB) is responsible for channelling all of the communications between components which make up the architecture. It is also responsible for dealing with the lower level issues such as the physical transfer of data as well as dealing with potential low level system faults.

Interface Repository

The Interface Repository (IR) is responsible for storing all of the information pertaining to an individual method contained within an object. Effectively, the IR records a complete *signature* of a method. By recording the signature in a central location, clients at a later date can examine the repository and construct dynamic requests.

In order to make the data recorded in the repository more useful, the IR provides a set of interfaces which allow clients to connect to the repository and to manipulate data held within it. A detailed description of what these interfaces are can be found in Object Management Group (2001).

Implementation Repository

The Implementation Repository is responsible for recording and providing the information required to start a server object. An important role of the implementation repository is to examine all incoming requests which are made to start an object. Each request is checked against the repository to ensure that it has the appropriate access privileges. If the privileges are satisfactory the Implementation Repository will start the server at the nominated location and pass through the appropriate command line parameters. If there are insufficient privileges, the request to start the object is rejected. The Object Adapter and Implementation Repository share a close relationship as the Object Adapter is responsible for sending a large number of requests to activate servers. A typical entry for one server would contain the following information in the Implementation Repository:

- Object Name
- Object Location
- Path to server location
- Additional command line parameters required by server
- Access Privileges

2.5.2.4 Object Management Group and Components

An inherent weakness with the CORBA 2 standard (mentioned above) is its inability to interoperate with components or to provide support for the development of them. Specifically, the CORBA 2 specification does not include any support for the packaging and deployment of components through the entire enterprise. Other disadvantages with the specification include the need for developers to program explicit support for non-functional properties such as security integration, naming services, trading, persistence or transactions. Failing to provide all of these services leads to no overall support for the development of software architectures.

To address these concerns regarding the support of components, the OMG has introduced a CORBA Component Manager (CCM) which is to be incorporated into version 3 of CORBA in the near future. Primarily, the CCM has been developed to address the weaknesses in the CORBA 2 model by providing a Distributed Component Object Model which includes an architecture for defining components and their associated interactions as well as a framework allowing events to be injected into components. Additionally, CORBA's DCOM supports the packaging of components and the deployment of those packages through the enterprise while at the same time remaining compatible with other component environments such as Enterprise JavaBeans.

As detailed in Object Management Group (2002b) the CORBA Component Manager makes use of two models known as the Abstract Component Model (ACM) and the Component Container Programming Model (CCMP) and a framework known as the Component Implementation Framework (CIF). The ACM works by embracing extensions made to the Interface Definition Language proposed in the CORBA 3 specification (Object Management Group 2002a) and the Object Model. The specific role of the ACM is to allow developers to specify CORBA component types and the relationships that the component itself has with other components. These relationships may include what the component *offers* or what is *required*. Other definitions that the ACM provides for component developers include defining which *properties* are configurable and what the business life cycle operations are.

The CIF is responsible for defining a programming model for constructing the component (ie., define how a component should be implemented). While using the CIF the developer is only encouraged to define the business logic that the component will use through a Component Implementation Definition Language (CIDL). The actual implementation of the component is not finalised until the CIDL code has been processed and the appropriate skeleton stub code has been generated.

The remaining part of the CCM is the CCMP which is used to provide developers with the ability to logically view a component from different perspectives. These perspectives may include examining a component from a clients perspective and examining what interfaces are available or it may looking at the component from an implementers point of view. The CCM also provides developers with interfaces which allow the component to be integrated with services like security, persistence and transactions without the need for the programmer to produce copious amounts of code.

2.5.3 Microsoft Approach - COM/DCOM

The year 1995 marked the start of a fierce rivalry between the Object Management Group (OMG) and Microsoft[7]. This rivalry which coincided with the release of the CORBA 2 specification (Object Management Group 1995b) was sparked by Microsoft entering into the market with its product COM/DCOM.

2.5.3.1 Distributed Component Object Model

The Component Object Model (COM) (Microsoft Corporation and Digital Equipment Corporation 1995) was developed in 1995 by Microsoft to provide developers with an architecture to facilitate component construction and to provide a means of allowing components to share data. The Distributed Component Object Model (DCOM) built upon the COM system was designed to provide components with the additional capabilities of distributed shared memory management, network interoperability and transparency and dynamic management of component interfaces. Thompson, Watkins, Exton, Garrett, and Sajeev (1998) provides more detail with regards to the differences between the COM and DCOM architectures. The COM/DCOM architecture was written to supersede the original architecture which Microsoft developed called OLE which is discussed in Brockschmidt (1994). Additionally, the architectures borrow heavily from the RPC method of communication that is used within the DCE architecture.

One important thing to note about the COM/DCOM architecture is that it is a *binary standard*. This means that Microsoft not only specifies the conceptual architecture but also provides an implementation of the system. The binary standard also means that Microsoft specifies how the components are laid out in memory[8]. Just like CORBA, components in the COM/DCOM architecture are all accessible through interfaces.

Interfaces

Within the COM/DCOM architecture every component exports its functionality in terms of interfaces. Interfaces as specified in Microsoft Corporation and Digital Equipment Corporation (1995) are implemented as an array of function pointers which then point to the desired functionality. This approach is commonly used within the C++ compiler to provide 'polymorphism'. Figure 2.8 illustrates the standard memory layout used to provide support for polymorphism.

This polymorphic behaviour is achieved by having the compiler insert a pointer commonly known as a *vptr* automatically into the class when the program is compiled. This *vptr* then points to the array of functions supported by the component. This array of function pointers is commonly known as a *vtable*. To facilitate the execution of a dynamic function, the *vptr* is referenced so that the *vtable* can be found. Once found and accessible, the *vtable* uses an internal index to determine where to

[7] Microsoft at the time was a member of the OMG and still is today.

[8] As yet, Microsoft products such as Visual C++ are the only compilers currently available which match the COM/DCOM binary format.

dispatch the call. A complete discussion on the mechanics behind polymorphism and *vptr*'s can be found in Ellis and Stroustrup (1994).

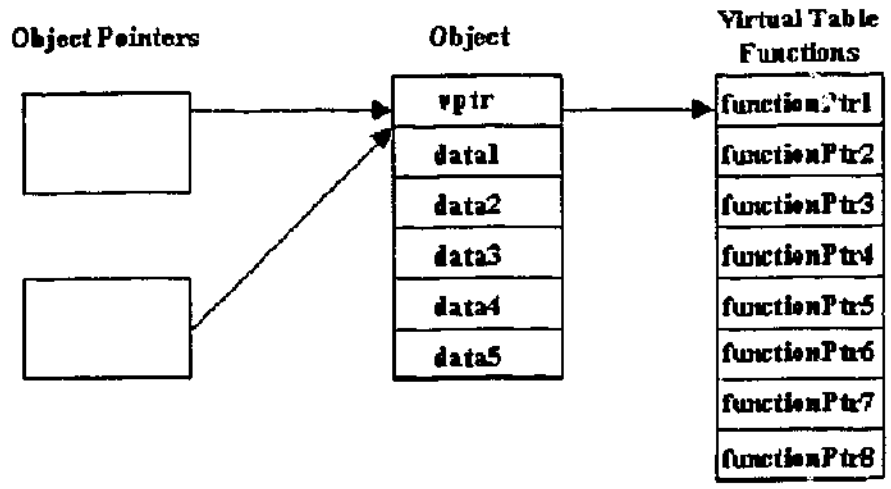


Figure 2.8: Memory Layout required for Polymorphism

When comparing the memory model used to provide C++ with polymorphism to the memory model specified by Microsoft Corporation and Digital Equipment Corporation (1995), it is possible to see one major difference. The difference is in the way the objects support the *vptr*. In the COM/DCOM model, no data can be associated with the structure which supports the *vptr*, as opposed to C++ which allows the *vptr* structure to also contain data.

IUnknown Interface

All components within the COM/DCOM architecture must inherit from the `IUnknown` interface. This interface is responsible for providing critical functionality through the use of three functions to all components within the architecture. These functions include:

QueryInterface(REFIID *requestedInterfaceIdentifier*, void ***returnedObject*;) : In order to access a component within the architecture, the software developer must have an interface to an object. The *QueryInterface(...)* function is responsible for accepting an identifier for the component required by the developer as well as a data member which is nominated to store the returned pointer to the interface. If the component exists, then a valid component reference is returned and the reference count for the component is incremented, otherwise a `NULL` pointer is returned. Kraig Brockschmidt in Brockschmidt (1994) details the various semantics which should be followed when using and dealing with interfaces.

AddRef(...) : The *AddRef(...)* function is responsible for incrementing the internal reference counter within the component. This internal reference counter is used to control when a component is unloaded from the system.

Release(...) : The *Release(...)* function is responsible for decrementing the internal reference counter within the component. Once the internal reference counter reaches a value of zero, the architecture is signalled to unload the component from the system[9].

2.5.3.2 Distributed Component Object Model Architecture

Just like the CORBA architecture, the COM/DCOM architecture shown in figure 2.9 depicts an architecture made up of a number of interlinked components. As can be seen, the architecture is structured in such a manner as to group together components which are responsible for providing a specialised function. Three specific areas within the architecture stand out. These include:

- OLE Compound Documents
- Active X Controls
- Infrastructure Components

OLE Compound Documents

The OLE Compound Document interface is responsible for providing a framework which allows documents to *host* COM/DCOM components within them. The benefit of such a framework allows components of any nature such as word documents, video files, web pages, databases, Active X components or spreadsheets to be included within a document while at the same time still providing the functionality which is native to that component. This is known within the framework as *inplace activation*.

An example of this technology may include inserting a video file describing a product which has been prepared in a word processor. With one click the reader could click on the component, and the video file could start showing them the object that they have been reading. This would all happen within the confines of the word processor.

Brockschmidt (1994) provides a detailed discussion on the fundamental concepts and objectives behind the OLE architecture.

[9] Although a value of zero for the reference counter implies that the component is to be unloaded, this is not always the case. The architecture will unload the component at the next opportunity.

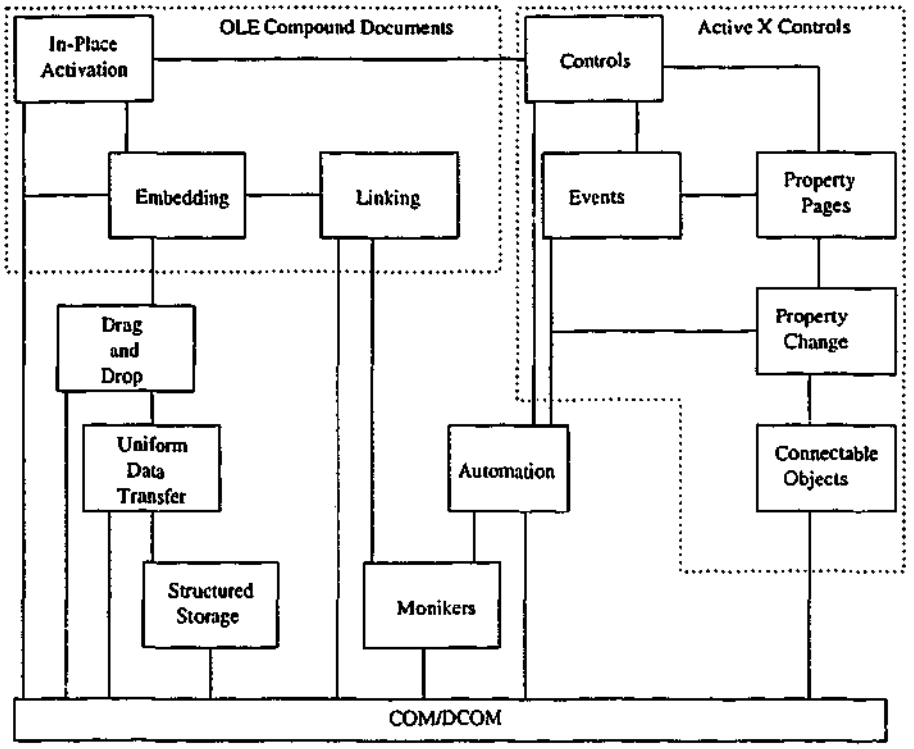


Figure 2.9: COM/DCOM Architecture

ActiveX Controls

The ActiveX programming technology was developed by Microsoft to allow developers to build components which could plug into COM compatible applications. ActiveX components, the most commonly built applications with the ActiveX technology, can be placed anywhere where a COM/DCOM object can be placed. This is possible because ActiveX components are implemented through COM/DCOM components. All ActiveX components contain a series of property pages which allow applications such as word processors to interact with the component. The amount of functionality available to the *host* application is controlled by the property pages and the interpretation the developer placed upon them. A more detailed description of ActiveX technology can be found in Ernst (1996).

Infrastructure Components

Although the term infrastructure components is not really a name used to describe the remaining components of the architecture, it does seem relevant as these remaining interfaces play an important role in controlling the various aspects of the architecture. These interfaces include managing component and object identification, the transferring of data between components, providing support for dynamic functionality by scripting languages and providing a standardised method of storing the data contained within a component so that it can be retrieved at a later date.

Monikers

Monikers within the COM/DCOM architecture are commonly referred to as *smart names*. These smart names are responsible for identifying an object within the system. They are similar in concept to the object references which are used within the CORBA architecture. Monikers can be associated with any COM/DCOM object ranging from a functional object written in Visual C++ to a World Wide Web (WWW) page or a database. In addition to containing information about the objects location, the Moniker also contains additional information to allow a client application to bind to the object. If a Moniker is used on an object which is currently not activated, it is the Moniker's responsibility to start that object and make all of the necessary arrangements in order for that object to receive requests.

The advantage with Moniker's just like the object reference in CORBA, is that they can be passed around or stored on disk. At a later date the Moniker can either be recalled from memory or from disk and be used to reconnect to a specific instance of an object.

Uniform Data Transfer

The Uniform Data Transfer (UDT) interface is responsible for providing a standardised method of transferring data between objects. Previous implementations such as OLE2 (Brockschmidt 1994) used inefficient memory copying techniques for shifting data from one component to another. These techniques normally involved copying data from one component to some global memory and then taking the data from the global memory and then copying it into the desired component memory location.

The UDT allows the developer to specify how the data can be copied from one object to another. With the new interface it is possible to make a direct copy from one object to another, and hence bypass the need for an extra memory copy into the global memory segment. The introduction of this interface now means that it is the developer's responsibility to specify the most efficient way of transferring data.

Automation

The automation interface is responsible for providing the same functionality which can be found in the Dynamic Invocation Interface (DII) component within the CORBA architecture. The automation interface is responsible for exposing the interface of components to clients so as to provide a facility which allows for the dynamic lookup of functions and their various parameters.

This feature is used commonly by scripting languages which require the ability to be able to connect to objects dynamically. This assists scripting languages as the invocation interface provides the ability for them to connect to objects which may not have been developed at the time that the script was designed.

Structured Storage

The structured storage interface is responsible for providing a standardised interface for the persistent storage of objects within the COM/DCOM architecture. The approach to persistent storage differs

from other systems, as COM/DCOM views a file as a hierarchical system containing a number of elements which need to be stored. Other systems use the more traditional approach of treating files as being flat in nature. This requires the entire data stream to be flattened out.

To aid developers in the process of making their objects persistent in nature, COM/DCOM provides a series of interfaces which the developer can inherit to allow the object to become persistent. This is very similar to the Java language (Arnold and Gosling 1998) which allows the developer to inherit from a series of classes in order to provide persistency.

2.5.3.3 Microsoft's Interface Definition Language

Just like the CORBA architecture, Microsoft also provides its own interface definition language (MIDL) to introduce a level of abstraction between the development of a component and the actual implementation. A significant component of the MIDL system was based on the interface definition language used with the DCE system.

MIDL however, does add a few extra keywords to the DCE IDL language so as to provide support for the COM/DCOM programming components. The keyword `attribute` is used to determine whether or not the IDL file being loaded into the system is destined to be a COM/DCOM object or a DCE one. Figure 2.10 shows an example of MIDL code. The keyword `object` is also used to indicate to the IDL compiler that this interface can be made available to other components and hence the compiler should insert into the class an automatic pointer.

```
/* Example of a MIDL File */
[
 object,
 uuid(6db16365-51d6-472f-b93f-00ab1de29c6c),
 helpstring("ISubtract Interface"),
 pointer_default(unique)
]

interface ISubtract : IUnknown
{
 import "unknwn.idl";
 HRESULT subtract_op([in] long number1,
                     [in] long number2,
                     [out] long result);
};
```

Figure 2.10: Example of a MIDL File

A more detailed description of MIDL can be found in the COM/DCOM specification (Microsoft Corporation and Digital Equipment Corporation 1995). One major difference between the MIDL and

DCE IDL system is that MIDL does not support the concept of interface versioning. COM/DCOM adopts a simple approach to interfaces. As the COM/DCOM specification states, once a interface has been published in the *public domain* it can no longer be changed. If a change is required in functionality then a new interface must be introduced into the public domain. Additionally, changes or upgrades to an interface also bring with them certain conditions to ensure that the interface is backward compatible. In some cases this means that new functionality must be written in terms of the old interface.

Interface Error Handling

Unlike CORBA's IDL which allows the developer to specify the return value for a method call, MIDL requires that each interface must return a `HRESULT` value. Results calculated within the function must be returned through an out parameter. The `HRESULT` value is 32 bits in length and provides the developer with all the information required in the case of an error. Figure 2.11 shows the structure of the HRESULT and how it is divided into a number of sections.

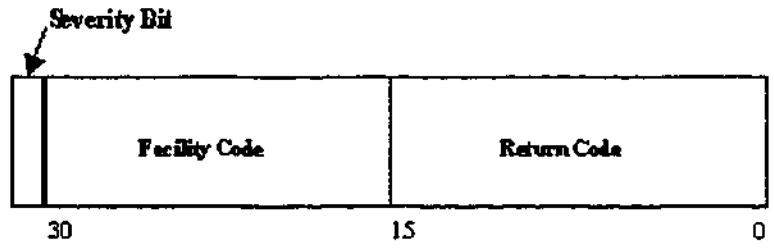


Figure 2.11: HRESULT Structure

As can be seen in the figure, the severity bit which is the most significant bit in the structure indicates whether or not the actual call to the interface was successful. This is similar to the `COMPLETED_YES`, `COMPLETED_NO` and `COMPLETED_MAYBE` statuses which are available in the CORBA implementation model.

The next section of the HRESULT structure is called the Facility Code. This is responsible for informing the developer which part of the sub-system caused an error. The most common return code for this section is `FACILITY_ITF` which indicates that an error occurred while processing the interface. The facility code as described in Microsoft Corporation and Digital Equipment Corporation (1995) is reserved for the use of Microsoft and it is not recommended that developers use this section of the return structure.

The final section of the HRESULT is a 16 bit structure reserved for the developer. The developer can place anything in this field providing it does not exceed 16 bits to represent the return value from the interface.

2.5.3.4 Implementation Repository – System Registry

The introduction of the latest Microsoft operating systems bring with them a new concept in the way that system data is stored. The advent of the system registry which is manipulated by the command `regedit` provides a central storage area that programs can use to save their user settings, system preferences and other related information. Figure 2.12 represents a registry entry as viewed by `regedit`.

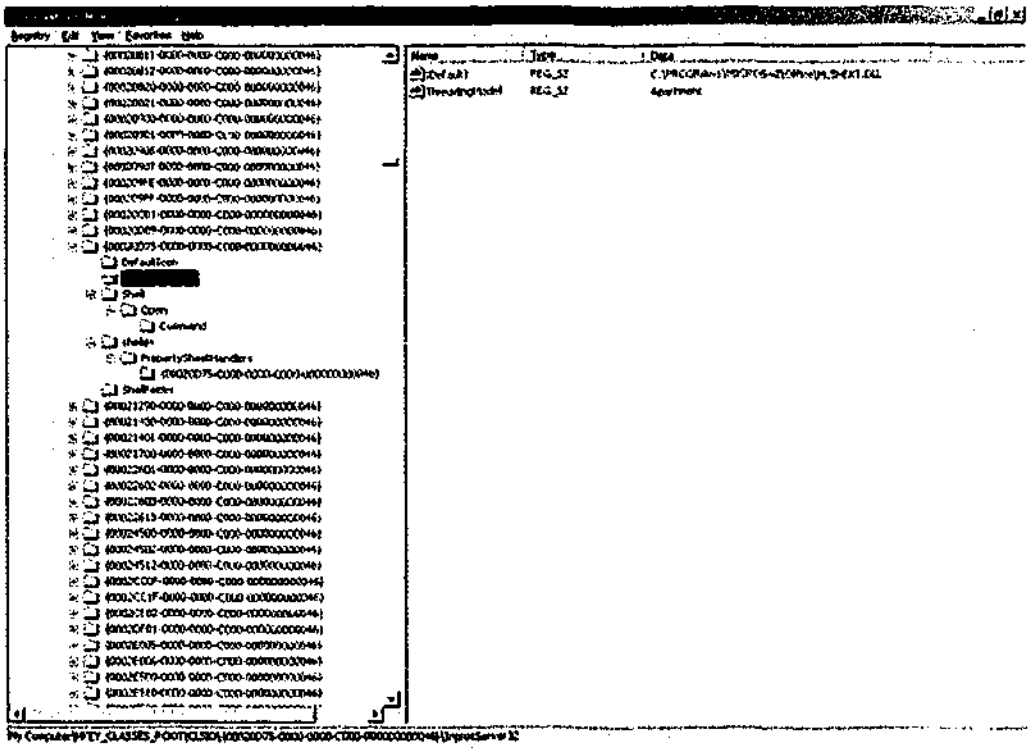


Figure 2.12: COM Component registered within the Registry

In order to minimise the amount of time required to locate an item in the system registry, a tree hierarchy is used to classify data into the appropriate sections. The root nodes of this tree accommodate information which is valid for the current user logged into the system (HKEY_CURRENT_USER), all of the users not logged into the system (HKEY_USERS), information about the current machine (HKEY_LOCAL_MACHINE), information about the current software loaded on that machine (HKEY_LOCAL_MACHINE_SOFTWARE and HKEY_CLASSES_ROOT) as well as dynamic statistical information (HKEY_DYN_DATA) which is calculated by the machine during idle time. The example shown in figure 2.12 illustrates the registration of a server to be used within the family of Microsoft Office products. The data contained with the registry entry provides the servers location on the disk and also specifies the threading model which is to be used when this server is operating.

In the case shown here, the 'Apartment' threading model is used. Additionally, the figure illustrates the unique CLASSID identifier which is stored within the HKEY_CLASSES_ROOT branch of the registry.

Both COM and DCOM use the system registry to store information relating to the location of server objects and what command line parameters need to be passed to those objects to start them. This information is recorded within the section of the registry responsible for registering programs installed on the current machine (HKEY_CLASSES_ROOT). Each application when registered with the system has its own unique 128 bit identifier known as a Global Unique Identifier (GUID). This identifier is based off the DCE Universal Unique Identifier (UUID) system but Microsoft introduced a number of changes to suit the potentially wide distribution of software.

2.5.4 Microsoft Approach - .NET Framework

To illustrate the rapid developments being made with component frameworks, Microsoft is releasing a new component framework known as .NET. The underlying design goals of the .NET framework include providing a simplified approach to component development, providing a unified programming model, providing a framework which has had elements within it standardised[10] and to provide a framework which allows for components to be deployed, executed and maintained without any significant effort. The introduction of the .NET framework is an attempt by Microsoft to encourage software developers to compete with one another at the application level rather than at the platform level (Simmons and Rofail 2002). By adopting the .NET framework and the standards which lie beneath it (especially the web standards), developers will be able to develop components on Microsoft and non-Microsoft systems[11] and incorporate them into the .NET framework. In addition to web standards, the .NET framework also allows the ability for components to be able to communicate with one another using the Extensible Markup Language (XML) allowing components to natively communicate with one another.

2.5.4.1 Architecture

From the .NET architecture shown in figure 2.13 it is possible to see that the architecture has been designed in a modular nature which allows for components within the framework to be interchanged in the future. The figure also identifies the three major components within the system. These components include:

- Common Language Specification
- Base Class Library

[10] Both the C# Language specification in ECMA Standards Organisation (2001a) and the Common Language Infrastructure specified in ECMA Standards Organisation (2001b) have been ratified.

[11] Microsoft has shown interest in porting the .NET framework to other systems.

- Common Language Runtime

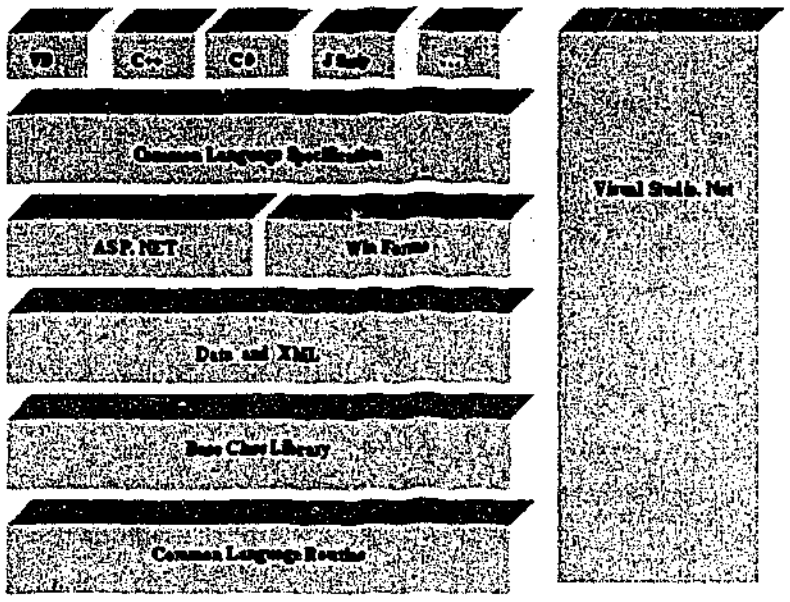


Figure 2.13: .NET Framework Architecture

Common Language Specification

Unlike other component frameworks created by Microsoft, the .NET framework provides developers with the ability to use a variety of languages to build a component rather than forcing programmers into one particular language and style of programming. The Common Language Specification (CLS) provides a series of interfaces which can be used to 'hook' into the .NET framework and allow various languages to make use of the features contained within the framework. Languages identified within Gordon and Syme (2001) provided by Microsoft which can be used with the .NET framework include:

- VB.NET
- C++ also known as Managed C++
- C#[12]
- JavaScript

[12]The language specification for C# as outlined in ECMA Standards Organisation (2001a) has just recently being standardised by ECMA.

In addition to those languages provided by Microsoft being available, a number of participating institutions involved with the development of the .NET framework have provided their own extensions which interface with the CLS. Some of these languages include:

- COBOL
- Eiffel
- Perl
- Scheme

The use of the common language specification will certainly ensure that a wide variety of components are developed for the .NET framework and that these components will be spread over a number of different languages. It is also possible with the help of the Common Language Specification (CLS) and the Common Language Runtime (CLR) to allow components written in different languages to inter-operate with one another.

Base Class Library

The 'Base Framework Classes' library is responsible for providing the standard functionality to the .NET framework. Such functionality can be compared to the Java `java.io` class which is responsible for handling string manipulation and managing the input and output of data, however the base class library in .NET is also responsible for providing security management, thread management, network communication and other functions (Microsoft Corporation 2001). Other classes such as data and XML are responsible for providing support for persistent data connections through the use of SQL interfaces or SQL directives. The XML portion of the data class is responsible for providing functionality which can parse, construct, search and manage XML data streams.

The XML web services section of the framework is responsible for allowing developers to be able to construct distributed components which use the underlying WWW transport system and XML language to communicate with one another. By taking such an approach to component communication, components are able to communicate with one another even within network environments where firewalls are operating (providing that WWW traffic is allowed).

To complement the support already provided for web services, the .NET framework allows the rapid development of components requiring a web interface by providing a series of interfaces to classes contained within the 'Web Forms' component. Applications not wanting to make use of the web technology but still requiring a graphical interface can make use of interfaces located within a section of the architecture known as 'Windows Forms'. These interfaces provide developers with access to set of graphical user interfaces which are tied into the operating system. Using these graphical interfaces gives the application the same look and feel as other programs loaded on the system. The 'Windows Forms' classes also provide support for the internationalisation of applications.

Common Language Runtime

The 'Common Language Runtime' (CLR) is the component which is responsible for interfacing the classes and components developed within the .NET framework with the runtime environment. When the structure of the Common Language Runtime is examined (refer to figure 2.14) it is possible to see that the runtime has to deal with various aspects of a components execution.

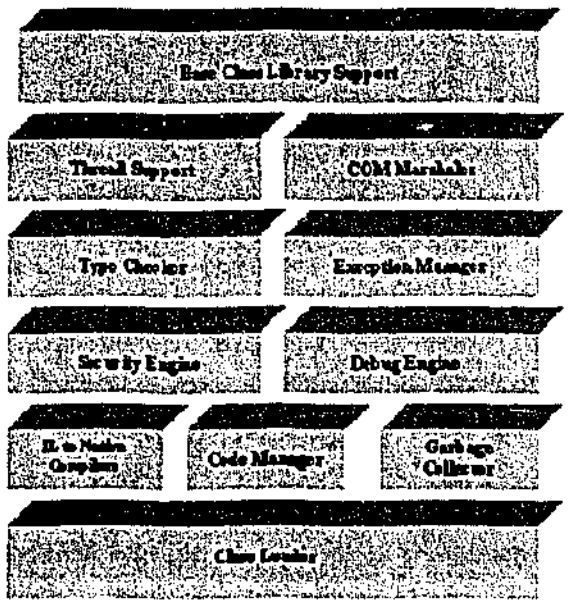


Figure 2.14: .NET Common Language Runtime Architecture

The CLR is responsible for providing the following support to components:

- Threading
- Allow older components developed with COM/DCOM technology to interact with those components which have been developed with the .NET framework
- Type checking components regardless of the language they were developed within[13]
- Exceptions which the developer must now deal with rather than HRESULT's which the user could forget to check
- Security which can include the use of policies based on evidence in the form of a signature or origin
- Debugging

[13]The CLR is able to achieve this as the CLR deals with components after they have been compiled into a Intermediate Language, hence type checking is performed at the IL level.

- Converting the Intermediate Language generated higher up in the framework into native code which can be executed
- Code Manager which may include support for versioning control
- Garbage Collection
- Class Loader which is responsible for actually loading the component into the system and executing it

Further information relating to how languages map to the CLR and how it works at runtime can be found in Gough (2002).

2.5.5 SUN Microsystems - JavaBeans Approach

While both the OMG and Microsoft were busy developing architectures to support the concepts of distributed systems, SUN Microsystems was simultaneously developing a language known as Java. Java was designed to promote the benefits of the object oriented programming paradigm while at the same time providing a basis for an architecturally neutral language. Additionally, as the language evolved, more and more features were added to the system including the development of a framework which facilitates the development and implementation of components. This architecture is known as JavaBeans.

2.5.5.1 Java: The Language

The Java language was released in 1995, but did not become popular until late 1996 and was designed to provide programmers with a simple but powerful object oriented programming language environment. The major objectives of the Java language include:

- Object Oriented Programming Paradigm
- Distributed Programming
- Robust Programming
- Architecture Neutrality

Object-Oriented Programming
As has already been mentioned, Java was designed with the sole purpose to provide the developer with a complete object oriented language environment. This therefore means that everything within Java is considered to be an object and that there is no such concept as a global variable accessible by all classes. Objects are only allowed to access those classes which exist within their visible scope

and can only communicate with one another through the use of messages. It is important to note that classes used within Java take on a *blueprint* role as opposed to objects which actually exist at runtime.

Additionally, Java also supports the object oriented concepts of identity, classification, inheritance and encapsulation as described in Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen (1991).

Distributed Programming

A major advantage in using the Java programming language is the wide variety of class libraries that are available to help support the developer in building applications. One of these libraries known as `java.net` is responsible for providing the developer with a series of methods and classes which assist in the development of distributed and client/server architecture applications.

By providing these classes and hence providing an abstraction over the network layer, the developer can continue to program without having to be concerned about the various networking protocols. Java supports both the TCP and UDP protocols as well as providing comprehensive support for WWW programming.

Robust

Java provides developers with a robust programming environment capable of performing type checking on applications while they are being compiled as well as when they are running. In order to deal with a type checking error as well as other errors which may occur while the program is running, an exception handling mechanism is included within the language. This allows the programmer to capture certain events and to respond to them accordingly.

An example of a typical exception might include trying to open a file which does not exist. In this case, a `java.io.FileNotFoundException` exception would be raised and the programmer would have to specify how such an exception should be dealt with. Failing to deal with the exception would result in the exception unwinding the stack until it reaches the Java Virtual Machine (JVM). If the exception thread reaches the JVM, then the program aborts its execution.

Additionally, Java also provides a memory management model which is responsible for removing references to objects which are no longer in use. This process is known as garbage collection. One final note is that Java does not support the notion of C++ pointers as it provides an environment in which variables are passed around by reference. The lack of pointers avoids the need for pointer arithmetic and hence one less area where programming errors can occur.

Architecture Neutrality

One of SUN's major design objectives for Java was to provide a method where a developer could develop the program once, but run it on many different platforms. This is commonly known as the *write once, run many* principle. SUN was able to achieve this through the introduction of its byte code. This byte code is an architecturally neutral format which is generated as a end product of a Java compile. Once compiled, the byte code can be executed on any Java Virtual Machine (JVM)

without the developer having to perform a recompilation. By adopting the JVM and bytecode principles, SUN provides a language which can run on any platform where a JVM is present.

The only disadvantage using the JVM model is that the byte code must be interpreted before an action can take place. This does introduce a slight performance degradation in the application as an extra level of abstraction must be negotiated.

To ensure that code written within Java is totally portable across the various platforms, the Java ...cation states the explicit sizes of the various types used within the language. These size defini-...s are independent of the actual machine architecture as the JVM is responsible for administering the size of the variables. Additionally, wherever possible, Java makes use of abstractions so as to hide the underlying architecture required. This can be demonstrated with the `java.awt` class hierarchy which abstracts the native GUI controls available to the user. By using this abstraction, the same methods in `java.awt` can be used independent of the platforms.

2.5.5.2 JavaBeans

At the time that Java was released in 1995, both OMG and Microsoft were providing their own solutions to the construction of components in a distributed system. These solutions addressed the immediate concerns of component software construction. However, short comings existed with both approaches.

The CORBA framework although designed to build distributed applications, is not specifically geared to facilitate the easy construction of software components and applications. This may change with the imminent release of CORBA 3 and the CORBA Component Model (CCM). COM however, does provide an object model supporting the complete notion of software components and includes the appropriate infrastructure to allow developers to interconnect components and build applications.

The only problem with the COM approach is that it is Microsoft centric[14], requiring that the developer must have Microsoft tools and a Microsoft environment to work within.

JavaBeans provides an alternative to this problem by providing a framework which allows for the construction of software components while at the same time harnessing the objectives and benefits of the Java language.

Concepts & Objectives of JavaBeans

JavaBeans which was not originally released with Java, was added to the Java 1.1 specification after the event model was re-developed and re-released in October 1996. Since then its popularity as a method of building software components within the Java language has grown.

As is mentioned in SUN Microsystems (1997) the design goal for JavaBeans was to define a *software component model* for Java. A subsequent aim of this goal is to provide a means by which third party

[14]At this stage there are no other compilers currently available other than Microsoft's which facilitate the native construction of software components.

vendors can create and/or ship JavaBean components which will plug into existing applications or software development environments.

Traditionally, JavaBeans normally consist of a set of classes and/or resources which are then made available to an application which interacts with the component. JavaBeans are capable of providing two levels of granularity. The first level supports the concept of using JavaBeans as the building blocks in composing applications. Such an activity would be performed with the aid of a Beans Developer Kit (BDK) which would facilitate the interconnection of JavaBeans to form the component.

The second level is much more coarse and treats JavaBeans as being a full application. Adopting this level of granularity allows third parties to ship JavaBeans as a particular add-on to a development environment or as a standalone application.

JavaBeans have the advantage of being connected to the Java language inherently. This means that the JavaBeans are capable of inheriting all of the benefits associated with the language. These include portability, security and simplicity.

Bean Definition

A Bean as defined in SUN Microsystems (1997) is a *'reusable software component that can be manipulated visually within a builder tool'*. As mentioned previously, Beans can range from being small components such as AWT buttons right through to complex applications which can be used in developer environments or as standalone objects.

However, no matter how big or small a Bean is, it must still provide support for:

Introspection: Allows for a Bean Developer Kit (BDK) to attach to a Bean and to perform an analysis on it so as to determine how it works.

Customisation: Provides support for a Bean's appearance or behaviour to be modified as required.

Events: Events provide the basis for Beans to be able to communicate with other components external to themselves. Events normally form the basis for how Beans inter-communicate with one another.

Properties: Properties provide a basis for customisations to be made to the behaviour of a Bean from a functional perspective. It is important to note that changes made to a property tag associated with a Bean may result in an event being triggered somewhere else within the system.

Persistence: Persistence relates to being able to capture the customised changes made to a Bean and then allowing those changes to be saved and at a later point, re-loaded into the system.

A common distinction which needs to be made is the difference between a class library and a individual JavaBean. Traditionally, JavaBeans have been used to encapsulate visual aspects of a system while class libraries are used to provide more of the functional aspects of a system.

Bean Principles

For every Bean constructed, there are three interfaces which the bean uses to communicate with the outside world. These interfaces include:

- Properties
- Methods
- Events

Properties

Within each Bean, a programmer can specify a series of accessible attributes. These attributes are commonly known as its properties and are stored within a certain section of the Bean where developers can access them. Accessing the properties from within a development environment is achieved with a `read` command while those explicitly defined as being publicly available can be modified via the `write` command.

Methods

Each Bean provides a certain amount of functionality which can be accessed through its method calls. Within the Bean environment, these methods act as a public interface so as to allow applications to make use of the functionality contained within the Bean. By default, Beans export all of their methods to the whole system, allowing any other component to access them. If for some reason this is not the desired result, there does exist suitable functionality within the JavaBeans model to restrict the publicly accessible interface to a subset of functions.

Events

Like any other event programming model, events are commonly used to indicate that a change has happened within the system. They are normally *triggered* by some action taking place, such as the mouse moving over the top of something, a button on the mouse being clicked or some I/O operation completing. Within the JavaBeans environment, events provide the fundamental basis for allowing other components within the system to know what is going on and what action has just taken place. The Java event model has been designed in such a way as to allow objects and beans to associate `EventListener`'s to certain actions. Once these actions take place, the Bean, component or object can take the appropriate action.

Basic Runtime Environment

As discussed in SUN Microsystems (1997), JavaBeans must be designed to be independently deployable executable components. To highlight this point, the specification states that JavaBean components must be able to run in both a designer environment such as the BDK as well as the

runtime environment. The specification does also state that facilities exist to allow developers to split those interfaces required at design time and those at runtime into different classes.

JavaBean Activation

JavaBean components execute within the same address space as their container. This means that the JavaBeans execution thread will run within the same address space as that of the program which is hosting the JavaBeans container. Effectively, JavaBeans will always be activated locally.

Distributed JavaBeans

As mentioned in section 2.5.5.1, the Java language provides a comprehensive class library to help developers build distributed applications. The JavaBeans model is able to make use of this library to assist developers in building JavaBeans which operate in a distributed environment. Additionally, JavaBeans can also be distributed by using:

Java RMI: This allows JavaBeans to use the Remote Method Invocation (RMI) concepts provided by Java to support distribution. The RMI approach is very common for those developers who wish to setup a client/server architecture.

Java IDL: JavaBeans also has the potential to allow interfaces to be generated using the industry standard OMG CORBA IDL. Once generated, these interfaces can be utilised by JavaBeans to contact CORBA-compliant servers to use their functionality.

JDBC: JavaBeans also have the ability to make use of the `java.odbc` library to contact remote databases supporting the JDBC protocol and to perform valid queries upon the data located within the database.

Multi-threading

JavaBeans should always assume that they will be operating within a multi-threaded environment. As a consequence, programmers need to be aware that it is their responsibility to ensure that the JavaBeans are written in such a manner as to be thread-safe. Additionally, developers need to provide the routines, if required, to maintain the data in a synchronised state.

Internationalisation

With Java being widely used, certain issues regarding the internationalisation of names must be addressed. By default JavaBeans can make use of the Java API classes which have been developed to support internationalisation. Developers should build their JavaBeans in such a manner as to limit their exposure to such things. It is important however that the developer pays close attention to the naming of events, methods and properties to ensure that as the application moves around the world that the functionality of the JavaBean will not be compromised as a result.

2.5.5.3 Enterprise JavaBeans

With the development of distributed applications making use of component based technologies increasing and the number of environments supporting the paradigm growing it was necessary for SUN Microsystems to extend its JavaBeans architecture to suit enterprise deployments. To achieve this, a new platform known as Enterprise JavaBeans (EJB) was designed and incorporated into the J2EE environment to simplify the development of components by incorporating a middleware layer capable of providing a number of services. Some of the services supported within EJB include transactions, security and database connectivity. Recently a new EJB standard known as EJB 2.1 SUN Microsystems (2002) has been released which includes additional services capable of providing applications with web services, time-managed container services and improvements to the EJB declaration language (QL) which is used to provide container managed persistence.

The introduction of the EJB platform further enhances component based programming by allowing developers to build distributed applications by allowing components made by various vendors to be combined and providing the appropriate framework to allow them to interoperate with one another. Assisting in the interoperation of components between various environments is EJB's ability to communicate using CORBA's communications protocol. This allows components operating within the Java environment to communicate with other components operating under platforms and languages that CORBA supports.

2.5.6 Architecture Comparison

As has been detailed in the previous sections, there have been a number of architectures developed to provide some assistance to software developers. These architectures have all been designed to address the issue of providing developers with a framework facilitating the development of applications within a distributed environment using the component based paradigm and software components. Of the five architectures which have been described, four support the concept of object oriented programming while DCE provides a framework encompassing the procedural approach to software components and systems development.

From the remaining architectures, both CORBA and COM are normally compared with one another and are traditionally viewed as competing architectures. The next section examines some of the similarities and differences between the CORBA and COM architectures.

2.5.6.1 CORBA vs. COM

Although COM and CORBA are normally described separately and are depicted as being fundamentally different, they do share a number of similarities as described in Exton, Watkins, and Thompson (1997) and Chung, Huang, Yajnik, Liang, Shih, Wang, and Wang (1998). This is not surprising considering that both COM and CORBA aim to address the same issues, it is just that their approach and object models differ from one another.

This major difference between COM and CORBA's object model can be traced to the way in which their architectures were originally developed. The CORBA architecture was originally and is still maintained by the OMG which is a standards body. At the time that the architecture was being developed it was this standards body which insisted that the architecture should be designed to not compound any problems which may have existed in another architecture. The entire premise behind the design of CORBA was to provide a specification which members could use to build their implementation. By providing only a specification the need for creating an implementation is removed, however having only a standard can lead to it never being adopted or implemented. An example of this is the Persistent Object Storage (POS) service which is currently being addressed by the OMG.

Unlike CORBA, COM/DCOM is both a standard and an implementation. During the design process of COM/DCOM a significant number of concepts and ideas were adopted from the DCE architecture. These concepts included the DCE Remote Procedure Call (RPC) method of communication, the concept of calculating a unique identifier to represent the location of an object within the system and the Interface Definition Language (IDL). Although most of these concepts were re-engineered, the basic ideas still operate in the same way. This reuse of concepts allowed the development of COM to proceed in a timely fashion while at the same time providing a solid springboard from which the remainder of the COM architecture could be developed. This springboard has resulted in an architecture which provides a more *complete* solution.

Architectural Comparison

From an architectural perspective it is possible to see that the CORBA object and implementation models (refer to figures 2.6 & 2.7) share nothing in common with the COM architecture (refer to figure 2.9). It is however possible to see that the COM architecture addresses the needs of components more comprehensively than CORBA. This is partly due to the way in which the COM architecture is structured and partly to do with the large number of services which the CORBA standard is yet to implement. It is hoped that with the impending release of CORBA 3 that the OMG will revisit the support needed by components and recommend that the appropriate changes are made to the standard.

Although the architectures have nothing in common, when services promised[15] by the architectures are examined, it is possible to see that both CORBA and COM/DCOM provide similar services.

These services include:

- support for concurrency
- dynamic invocation of requests on components, handling events throughout the system
- providing an interface definition language to separate the interface of the object from the implementation
- providing a means for an object or component to be uniquely identifiable throughout the entire system
- allowing objects or components to be persistently stored and then recalled at a later time when needed
- providing facilities for tracking and supporting transactions which take place throughout the system
- providing a security framework to ensure the data integrity of components and objects within the system
- a mechanism for allowing software developers to register their servers with the system
- providing a means for allowing errors to be reported back to the client

Providing a Portable Framework

A problem that both the CORBA and COM architecture share is providing a framework which is portable across a variety of platforms, operating systems and languages. CORBA tries to address this issue by having a variety of ORB's written for different operating systems scattered over a number of different platforms and through the use of the Internet Inter-ORB Protocol (IIOP) allowing ORB's to communicate with one another over the internet. Additionally, CORBA also provides numerous language bindings (refer to section 2.5.2.3) allowing developers to choose which language suits the problem domain the best. However the OMG and the CORBA model still can not resolve the problem of being able to provide a complete solution to developers for component based programming.

The COM architecture has been specifically designed to operate with the operating systems that Microsoft is currently shipping. These operating systems include Microsoft Windows NT 4.0 and Microsoft Windows 2000. Additionally, COM/DCOM requires Microsoft development tools like Visual C++ in order to ensure complete compatibility with the *binary* standard. By adopting this

[15]The term promised is used due to the OMG providing standards but not a implementation. Hence some services have been designed but do not exist in any implementation.

framework, developers are limiting themselves to Microsoft development tools in order to develop COM/DCOM applications which are only capable of running on PC's with the Microsoft operating system[16].

2.5.6.2 .NET vs. COM/DCOM

The .NET framework was introduced to simplify the already existing architecture of COM/DCOM that Microsoft provided to developers while at the same time taking advantage of new component technology and programming methods. In addition to this, the new .NET framework eliminates the need for programmers to provide in each individual COM component the functionality required to communicate with other components. Inter-component communication is handled by the .NET framework allowing the developer to continue on with the job of writing the component.

The simplification process of the COM/DCOM architecture includes the removal of a component having to register itself before it can be executed (refer to figure 2.12) but rather requires the component to describe itself in a self-contained manner. Other registration requirements which were required by the COM/DCOM system, such as obtaining GUIDS to uniquely identify components and the registration of interface definitions through a IDL file have also been abandoned and incorporated a hierarchical namespace and unified object model.

In addition to modifying the way in which a component registers itself, .NET also introduces a more traditional approach to handling exceptions by abandoning the HRESULT structure and adopting a structured exception handling system such as the termination exception model. The use of this exception handling technique leads to an increase in the components reliability as mentioned in Thompson and Watkins (1997) as the developer is forced to handle error conditions as they occur rather than relying on the return result from a function call.

The .NET framework also simplifies the way in which memory is managed by introducing garbage collection routines which remove references to components which are no longer being referenced or required within the framework. This replaces the `AddRef(...)` and `Release(...)` method calls which had to be explicitly specified and managed within a COM/DCOM architecture.

2.5.6.3 JavaBeans

JavaBeans addresses the issue of portability through the aid of Java. As JavaBeans share a close relationship with Java (they are implemented within the Java language), they are capable of being deployed on any system where a Java Virtual Machine (JVM) is located. This provides the ability

[16]Currently, there are a number of attempts being made by software vendors to port the COM/DCOM architecture across to the UNIX platform. If this port is successful, the COM/DCOM model will probably lack facilities like OLE and Drag-n-Drop technology which has little relevance within the UNIX environment.

for JavaBeans to be implemented on devices ranging from the standard household PC through to mini and mainframe computers running UNIX and portable, smart devices.

Additionally, similar to the COM architecture, JavaBeans provides a complete framework as stated in SUN Microsystems (1997) to facilitate the development and use of components within a system. However, after comparing the JavaBeans specification to the COM or CORBA architecture it is possible to see that JavaBeans does not benefit from a sophisticated set of services such as the transaction service outlined in the CORBA specification.

2.6 Chapter Summary

In this chapter the component based paradigm has been introduced. This paradigm has been designed to assist developers with their construction of applications by providing an infrastructure where components can be developed or purchased from software vendors and where they can be interconnected with one another to provide the required functionality. In order to obtain a better appreciation, the objectives of the paradigm and the overall component based architecture were defined and then compared to the object oriented paradigm.

This chapter also introduced the concept of configuration management and explained the benefits associated with the ability to reconfigure a component within a system especially if it can continue to operate unaffected. To further develop the ideas behind configuration management, a detailed look at a component reconfiguring was presented.

To complement discussion on the component based paradigm and configuration management, section 2.5 presented an overview of the four main architectures developed to provide developers with an infrastructure to build distributed applications.

The next chapter concentrates on identifying the various levels of support that exist within configuration management systems for software developers or end-users to control the management of and reconfiguration of components.

Chapter 3

Configuration Management Systems

In the previous chapter it was identified that the concept of reconfiguration management relies heavily upon the component based programming paradigm. This reliance has led to a number of architecture designs and implementations to assist software developers with the construction of distributed applications using a component architecture. Chapter 2 introduced the notion of configuration definition languages (most notably DARWIN) which can be used to separate the configuration, interface and functionality requirements of a component. Configuration definition languages can also be used to provide a basis for defining the information required to perform static and/or dynamic reconfiguration.

This chapter examines a variety of configuration management systems as opposed to architectures described in section 2.5. Some of these systems are designed to aid software developers during construction while others provide the ability to reconfigure components when the system is not operating. In addition to those systems which manage static components there are more advanced tools and systems that can provide developers and/or operators with the ability to change the bindings between components within systems while it is still operating.

3.1 Critiquing Configuration Management Systems

Before examining a subset of configuration management systems available to the software engineering community, it is important to identify a criteria by which these systems can be measured.

Criterion #1 - Revision Control

Important to any configuration management system is the ability to provide the infrastructure which allows for the identification of differences between component revisions. These differences are commonly referred to as component deltas. A configuration management system is deemed to have revision control if it is capable of being able to identify and store deltas in addition to providing the user with the ability to undo or re-apply changes to the code. This allows components developers to undo any changes that may have been introduced to the component making it behave in a erroneous fashion.

Revision control can be supported at two levels. The first level is locally, which allows for the revision deltas to be calculated on components stored locally on the system. Component deltas are normally stored within the file system on the local system.

The second level operates with revision control remotely. This provides for component deltas to be calculated from components located on both local and remote systems. Traditionally, such systems also provide support for the component deltas to be scattered throughout the network. This approach provides a level of redundancy allowing component deltas to be preserved throughout the system.

Criterion #2 - Replication of Configuration Information

An important aspect of any application development is the role of backups. This sense of importance is not lost on configuration management systems which are responsible for tracking every change made within a system whether during development or through continued maintenance. A configuration management system is considered to support the replication of configuration information if it provides the appropriate functionality enabling the configuration management system to store replicas of component deltas throughout the system.

Criterion #3 - Concurrent Component Development

Throughout the development of large software projects, it is common to find the need for multiple developers to be able to work on the same component at the same time within the same system. This in itself does not cause any problems unless a development team member requires a piece of code which is currently being modified by another team member. As a result, configuration management systems need to be aware of such events occurring and be able to handle such situations with accuracy. A configuration management system is deemed to provide concurrent component development facilities if it provides a process which allows for the integration of concurrent changes. Additionally, the configuration management system may have to flag potential problems when it notices that two or more team members have changed the same section of code within a component.

Such a situation would result in the configuration manager refusing to commit either change until a member from the development team could identify which of the changes made to the component can be committed to the system.

Criterion #4 - Revision History

Although revision control systems exist, some systems do not provide the facility for the developer

to be able to access or use previous revisions of components without having to significantly unwind changes already made to the system. As a result a configuration management system is said to provide a complete revision history if it allows developers to activate or recall sections of code which have previously been superseded. This functionality allows the developer to see and track the changes in the component.

Criterion #5 - Type Checking

As with any two entities which need to communicate, it is important that they share a common understanding. This approach to communication is no different in the world of component based programming where components which communicate with one another *must* be type compatible[1]. Although primarily used to check the types of components during component construction, the type checking system is also used to check the types of connections and ports associated with each component while constructing communication channels to facilitate inter-component communication.

Therefore, a configuration management system is considered to provide type checking facilities if it supports checking the validity of types used within the system and identifying where potential problems might exist within the system with regards to incompatible types.

Criterion #6 - Component Awareness

Since the development of the first configuration management system, there has been a remarkable change in the functionality provided by various systems in addition to continuing the refinement of aims and objectives. This has led to a large number of configuration management systems being built to provide specific services to the software developer. These systems range from simply providing the infrastructure necessary to build components and store their revision deltas through to providing fully integrated development environments allowing for components to be created, tested and inserted into a working system in real-time and to analyse the impacts of such a modification.

This diversity has brought with it a large number of configuration management systems which have been built in such a way that they ignore the fundamental concepts of a component. A configuration management system is said to be component aware if it is capable of distinguishing components within the system and providing them with additional support. This additional support may include the ability to manage inter-component communications, define dependencies between components or perform a consistency analysis on the system.

Criterion #7 - Inter-Component Communication & Management

An important aspect of any system written within the component based paradigm is the way in which components defined within the system interact and communicate with one another. These communication channels form the basis for inter-component communication while at the same time providing the infrastructure required to support the concept of reconfiguring components.

[1] A component may be considered to be type compatible if it is the same type as the originating component or if the type can be matched within an inheritance structure.

As a consequence of providing the infrastructure it is vitally important that a configuration management system provides support for the development and/or management of communication ports between components. This support might include providing the functionality required to create, modify or remove connection ports associated with components as well as constructing or deconstructing the communication pipe.

Criterion #8 - Quiescent Routines

As mentioned in section 2.3.2.1 and discussed in Kramer and Magee (1990) and Kramer, Magee, and Young (1990) the concept of quiescence is extremely important for those configuration management systems aiming to provide reconfiguration services within a stable system. A configuration management system is said to promote quiescence if it provides routines responsible for allowing the system to enter a state of quiescence.

Criterion #9 - Dependency Analysis

Sir. Issac Newton stated *"For each action there is an equal but yet opposite reaction"*. Although not exactly true for the component based paradigm, it is important to analyse and identify what the possible outcomes might be if a certain action is performed. Typically a dependency analysis is used to determine whether or not any deadlocks currently exist or whether any deadlocks may occur as a result of a reconfiguration action.

Additionally, dependency analysis is normally performed to determine whether the system is in a consistent state. Once the proposed configuration changes are known, a separate analysis of the system is performed to ensure that everything will remain consistent after the change has been applied.

Criterion #10 - Component Reconfiguration Control

For an application to continue to be relevant to its problem domain, it is necessary for it to be able to evolve to the changing environment. In order to achieve this, the configuration management system must be capable of supporting the reconfiguration of components. This reconfiguration can either be done *statically* where the system is stopped and components are interchanged or *dynamically* where the system continues to run but where the affected regions by the reconfiguration are *frozen*. A configuration management system which supports reconfiguration control must allow components to be added, modified or removed from the system. Additionally, the system should also provide the developer with the functionality to manipulate the interconnections between components.

Criterion #11 - Abstraction

The process of reconfiguring a component within a system comprises a number of steps. These steps may include such activities as identifying the component, introducing a state of quiescence and stability throughout the system, removing a component, installing a new component, establishing the new component and then slowly bringing the system out of its quiescent state. As has been identified, a number of steps may be required to replace one component within a system. To simplify this,

configuration management systems *abstract* the reconfiguration process into one command and hence subsequently reduce the chance of introducing an error during the reconfiguration process.

Criterion #12 - Specifying Reconfiguration Actions
Traditionally, configuration management systems have dealt with the issue of reconfiguration by unconditionally *freezing* all connections and components associated with the area which is being reconfigured. Although this approach works in many cases, an effective configuration management system would provide the software developer and/or user with the ability to specify what actions should take place in the event of a component not being available due to a reconfiguration.

Throughout the chapter each configuration management system will be compared to the criteria established above to determine the suitability of the tool as a configuration management system. Use of this criteria allows a baseline comparison to be made against the various configuration management systems even though not all of the systems have been designed to satisfy common goals.

The chapter concludes by examining a more sophisticated level of configuration/reconfiguration management designed to allow components to be substituted while the system continues to operate. In addition to this functionality, these systems have to be designed to analyse the consequences of such reconfigurations by performing various analyses including both deadlock and consistency to ensure that the system is consistent.

3.2 Component Configuration Systems

This section focuses upon those systems which provide developers with some assistance during component construction. To facilitate this a framework is provided to developers to produce consistent components, especially those being developed within a distributed environment.

3.2.1 Distributed Revision Control System

The origins of the Distributed Revision Control System (DRCS) detailed in van der Hoek, Heimbigner, and Wolf (1996) and van der Hoek, Carzaniga, Heimbigner, and Wolf (1998) can be traced back to the underlying architecture of the Revision Control System (RCS) from which it evolved. The extensions provided by DRCS bring with them an ability to utilitise the functionality of the RCS system over a distributed domain.

The objective of RCS when designed, mentioned in Tichy (1985), was to provide a system capable of allowing developers to introduce staged releases of information[2] stored within the repository into the system. In addition to allowing developers to perform staged releases it also provides the ability

[2]It is important to note that the RCS repository is capable of storing and versioning many forms of data and not just source code files or documentation. RCS's only requirement for release control is that the data which is to be versioned can be stored in a consistent state.

for an audit trial to be created and maintained. The DRCS/RCS system has been structured in such a manner as to allow many developers who have access to the repository to be able to place exclusive locks on various resources held within the repository. Providing this repository support prevents developers from making modifications to the same resource unless the lock to the resource has been removed.

RCS works on the basis of registering the resources[3] to be controlled and having those resources placed into a separate directory located on the same file system (normally called 'RCS'). As changes are made, the DRCS system calculates the *delta* fragments and stores the *changes* within the repository.

By making use of RCS, software developers are able to review their work and slowly add or remove changes made to the original resource. When a resource is requested from the DRCS repository the original resource is retrieved and then all the deltas associated with that resource are applied to it.

After a period of time and when the resource has had a number of revisions made, the author/operator can chose to *release* the resource. This process involves taking all the deltas for the resource and applying them to it to create a new resource which incorporates all of the previous deltas. After a release has taken place, it is impossible to rollback through previous changes as all the deltas have been merged into one.

The distributed aspect of DRCS caters for a number of clients who wish to make use of the client/server architecture to form a connection with the central server holding the repository. These connections allow distributed clients to manipulate the resource from a remote location while recording the delta resource fragments. Figure 3.1 illustrates the differences between the RCS and DRCS systems.

A problem which exists with the DRCS system relates to the underlying architecture. This architecture requires that all clients connect to a central repository so as to be able to access the desired resources. This approach to handling distributed component construction can lead to major bottlenecks at the central repository or expose the overall system to an unacceptable risk. These risk elements may include a communications fault between a client and the repository or a hardware/software fault at the repository.

3.2.1.1 DRCS as a Configuration Management Tool

DRCS was designed to allow software developers to be able to develop and modify their resources from a number of locations while at the same time providing extra functionality such as revision control and accountability. To this end, DRCS provides no functionality for the decoupling of configuration information from the implementation nor does it have any support for the management of bindings between components. DRCS exists purely for developers and to help them release and

[3] A resource can resemble any entity. Typically this resource is a file.

revision control their resources. From a configuration management perspective the DRCS system only satisfies the first criterion. DRCS provides the infrastructure required to identify differences between resource versions and the appropriate routines to store, retrieve and manipulate those deltas.

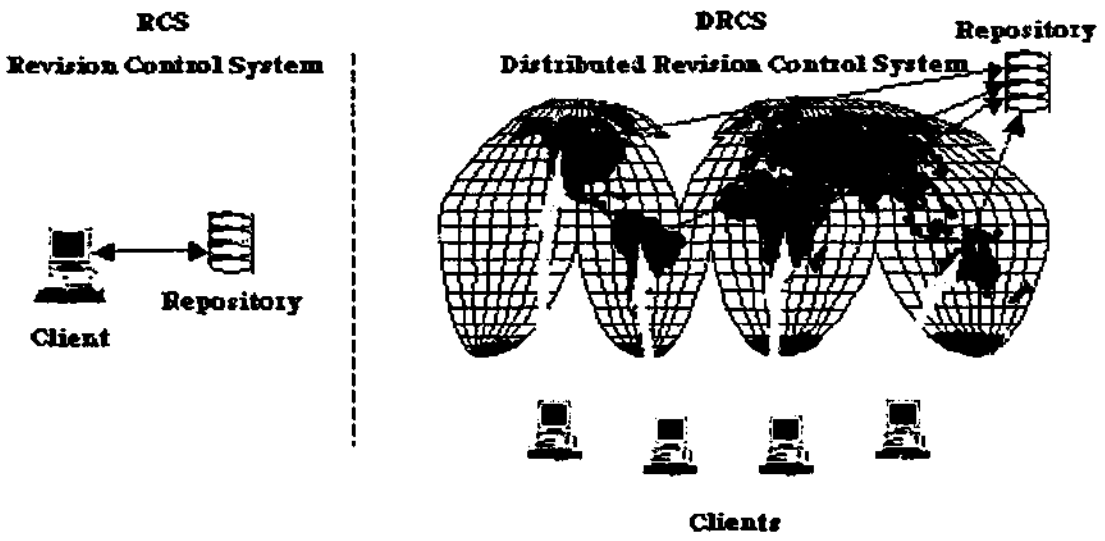


Figure 3.1: Differences between RCS and DRCS

Although DRCS does provide access to previous revision histories of resources, it does not provide support to execute a specific revision. By default, DRCS only provides developers with the resource once all of the deltas have been applied. No provision exists for developers to specify what revision number they wish to use.

3.2.2 Distributed Concurrent Versioning System

Just like the DRCS, the Distributed Concurrent Versioning System (DCVS) is the distributed computing extension to the standalone system known as the Concurrent Versioning System (CVS) which is explained in Berliner (1990) and Cederqvist (1993). The DCVS/CVS architecture is a further refinement to the DRCS/RCS and brings with it the additional functionality of being able to track multiple revisions made to one resource or tracking the various revisions made to a number of resources scattered over a file system within a project. The DCVS/CVS architecture also allows multiple clients to lock regions within the one resource and hence support concurrent development.

The DCVS architecture uses the same approach as CVS with regards to the storage of deltas requiring that all deltas are stored at the central repository. The DCVS architecture however differs when it comes to the transportation of the deltas to a client site. When a client connects to the central

repository and requests a resource, the DRCS architecture exports the entire configuration including the directory structure to the client site.

Once the data is at the client site, developers can make the appropriate changes to the resource and request those changes to be merged back into the central repository. During the merging process, the DCVS will apply each change as an individual change and determine whether or not the same resource has been changed by a previous delta (the granularity of the check can be measured in line numbers within a source code resource).

If multiple changes are detected, the CVS core flags an error and requires either an operator or developer to explicitly allow the change to be admitted to the system. This concurrent change behaviour is dictated by the DCVS architecture. DCVS provides no functionality to determine when two or more developers are modifying the same region within a resource after being exported to the client site because each site receives its own copy of the delta repository. This type of conflict is not detected until the repositories are merged back into the central repository.

Even though DCVS ships out the entire configuration of a system to the client site, it still relies on having the client ship all of the deltas back to the central repository.

Hence, by using the same architecture as the DRCS architecture, both the DCVS and DRCS systems share the same problems and characteristics. These problems relate to the central repository and the processing/merging of the deltas. If for some reason the central repository is offline, then the changes from the client can not be merged and the central repository starts to lose its consistency as it falls out of synchronisation with the clients. However, transmitting a copy of the repository to the client allows changes to still be made although the central repository is offline. The only operation which can not be performed during the downtime is the merging and synchronisation with the central repository.

3.2.2.1 DCVS as a Configuration Management Tool

Like DRCS, the DCVS system was designed to provide developers with a framework to allow large projects to be managed over a wide geographical area. As DRCS and DCVS share the same architecture it should come as no surprise that DCVS provides no functionality for the decoupling of configuration information from the implementation nor does it have any support for managing bindings between components. DCVS exists purely for providing developers with a versioning system which works in a distributed domain and provides an environment which supports concurrent development.

When compared to the configuration management systems criteria presented in section 3.1 it is possible to see that the DCVS system satisfies the first three criterion. In addition to providing support for the identification and preservation of delta information, DCVS also provides the infrastructure for developers to work concurrently on components scattered over a large geographic area.

To achieve this functionality, the configuration repository is shipped out to remote locations and hence provides a backup of the configuration repository which satisfies criterion number three.

3.2.3 Incremental Configuration Engine

The Incremental Configuration Engine (ICE) as detailed in Zeller (1995b) and Zeller (1995a) provides an alternative method to configuration management. In traditional source code configuration management systems the approach adopted is to segment all of the changes into a series of delta components. ICE takes another approach to the problem and places all of the deltas into one file.

Each version or configuration change is represented within the file and is then guarded by a series of preprocessor directives which the configuration manager understands.

This approach allows developers to view previous configuration changes and source code while developing new segments of code or applying new configuration changes. Previous systems such as DRCS and DCVS only allow one patch level to be available at any one time.

In addition to providing clients with the ability to see multiple views of changes, ICE makes use of version sets and feature logic. This allows the configuration engine to keep track of multiple components within the system while in parallel, allowing developers to clearly see all of the paths taken to reach an end component. ICE provides the capabilities to change these bindings during design time to reflect the appropriate configuration.

Through the use of feature logic and version sets comes the extra functionality known as workspaces. A workspace is used within the system to allow developers to trial changes to the system in isolation. Each workspace has a *feature* allocated to it allowing the system to identify that changes made within the workspace should not be incorporated into the overall system. Once a developer is satisfied that the changes are acceptable, the feature is modified and the changes are reflected across the entire system.

The advantage ICE has over other configuration management systems is its use of a specialised file system known as the 'Feature File System' (FFS). This file system has been designed with the express intent to allow all components within a file system to be versioned. Using this file system allows ICE to preserve revision and versioning information relating to access permissions to certain parts of the system, files, directories and groups of directories which make up a configuration[4].

ICE also provides a method of being able to compile all of the deltas for a component into a single module. During this 'compile' the ICE engine is able to use its feature logic functionality and version sets to identify the configurations that are required and is able to identify the required regions within the configuration with the use of the UNIX utility *diff3* to extract the appropriate configuration.

[4]In other systems, the term configuration would be known as a system release or a deployment.

Part of what aids the configuration process is the modified file system, as standard UNIX utilities and commands such as cp and make can be used to assemble the required configuration and build the appropriate configuration without developers having to learn and understand specific tools built for a configuration environment.

3.2.3.1 ICE as a Configuration Management Tool

The Incremental Configuration Engine (ICE) has principally been designed to overcome some of the inherent short-comings in other configuration management systems such as DRCS or DCVS. Even though ICE adopts a different architecture to other configuration management systems, it should be noted that it still only deals with static components during the construction phase.

ICE makes use of feature logic and version sets to maintain the consistency of the system. The consistency check that ICE performs only ensures that the right configuration is selected for the appropriate environment. ICE has no awareness of inter-component communications nor does it deal with issues such as type-checking and ensuring the compatibility between method calls. This work is still left to the compiler. When ICE is compared against the configuration management criteria it can be seen that it shares some similarities between DRCS and DCVS.

These similarities include providing the infrastructure for component deltas to be identified and preserved as well as providing support for the concurrent development of components through the concept of workspaces. However, ICE also provides developers with the ability to execute and assess *any* version which is recorded in the revision history. This marks a departure from DCVS or DRCS which limits developers to only the latest release. Additionally, ICE to some extent, satisfies criterion nine by using feature logic and version sets in an attempt to maintain a consistent static system. The consistency analysis routines however are not fully developed and do not provide a comprehensive analysis of the system when it comes to configuration management.

3.3 Static Component Configuration Management

A static component configuration management system is defined as a system which provides developers with support for the reconfiguration of components and their interconnections. However, the static nature of these systems only permits changes to be performed while the system is in a stable, consistent state which is normally achieved when the system has been stopped. Section 3.4 examines those systems capable of providing dynamic runtime support.

3.3.1 Software Dock

Software Dock architecture was developed to address the concerns detailed in Hall, Heimbigner, van der Hoek, and Wolf (1997) about modern systems being constructed through the use of components necessitating greater care be taken during the assembly and interconnection of those components.

Traditionally, the concept of configuration management has centered upon the development of tools allowing developers to have greater control over the source code and design phase. These tools provide a means of controlling the static configuration of a system from a design perspective. Some of these tools are examined in section 3.2.

Concerns that drove the development of the Software Dock include the lack of configuration management tools developed to support software deployment whilst at the same time taking into account site specific information.

With the software industry moving towards a component based approach come a number of problems. Problems include systems being distributed with missing components or components which are out of date. Hence, these components or lack of, might result in a system not being able to perform certain tasks or providing erroneous results. To address this a system needs to be developed to allow components to be updated and deployed in a non-intrusive manner.

The principal architecture of Software Dock has been designed to operate within a number of environments by making use of a decentralised approach. This approach is normally implemented in the form of networks such as the internet or corporate intra-networks and is used as the underlying carrier for new software components.

The architecture also provides for the appropriate semantics to record the software configuration for each site where it is deployed. This information is similar to the *registry* concept used within Microsoft operating systems. Once recorded, the architecture of a site (both software and hardware) can then be used to plan and develop the most strategic deployment of new software components.

In addition to the software and hardware components being recorded at a site, the architecture also records the configuration information between components. This information is analysed so that the component dependencies can be calculated. Using this configuration information provides a convenient way of determining the constraints on the system.

By recording all of this information it is possible for the Software Dock servers to monitor the environment in which they are operating and be able to identify any changes being made. Once changes have been identified, action can be taken (be it automatic or manual) to adapt to the new and changing environment. By accepting automatic changes, the system to some degree, can adapt itself based on site conditions.

3.3.1.1 Software Dock Architecture

The Software Dock consists of a number of servers. These servers are located at both the client or consumer site as well as at the server or *production* site.

Servers located at the consumer site are known as a *Field Dock*. This dock is responsible for maintaining a registry of the local configuration information (including software, hardware and configuration information). Its role is to propagate events received from remote sites with those components located at a local site.

It is also used as a notification service. When a component has been changed and an update is available, the field dock broadcasts on an appropriate channel. Applications which are susceptible to these changes listen to the appropriate channel. Once a change is detected, components contact the local field dock to find out more information and learn how to adapt to the change in the system.

The field dock is also responsible for providing controlled access to the resources at the site as well as the repository it maintains. It also provides the appropriate resources required by components so that they can subscribe to event channels. Any change made to a local registry is sent to the *Federated Deployment Registry* (explained later) so that it can remain in synchronisation with the local field dock.

A server located at the production site is known as a *Release Dock*. This dock is responsible for informing those who inquire about the current software releases and registry information. In addition, it provides the same event channel mechanisms discussed previously.

A typical software release entry in the release dock will contain a number of artefacts. These artefacts include details such as the executables, libraries associated with them, documentation, dependency information and any constraints appropriate for the installation of the software. It is also responsible for housing the agents which perform the deployment, configuration, maintenance and uninstallation activities.

The field dock is not directly responsible for the distribution and installation of new components but rather to inform other docks located throughout the system that these components have been changed and are now ready to be installed.

In addition to these servers the *Federated Deployment Registry* (FDR) acts as a reference point for both types of docks. The FDR is responsible for providing a global namespace which spans across both the field and release docks. By providing the global name space it allows clients to make standard method inquiries across all installations.

The Software Dock architecture also makes use of the FDR as a point where events can be subscribed too and where properties of the entire system can be queried. In order to provide this type of information, the FDR is organised into a tree like hierarchy (similar to how the registry by Microsoft

is arranged). The schema of the registry is maintained in a consistent manner as it synchronises with all of the changes made at the field dock.

Figure 3.2 illustrates the various relationships between the docks which exist in the system, the Federated Deployment Registry, the agents and the Wide Area Message/Event Service (described later) responsible for passing on and processing all of the various message channels within a deployment.

Agents do most of the work in the Software Dock architecture. They are normally embedded within the docks which are scattered through the system. They are responsible for the registration of specific events as well as performing tasks based upon the events that they receive. Agents within the system can be separated into two categories, those that operate within the scope of a dock, known as *internal agents*, and those that originate outside of the dock known as *external agents*.

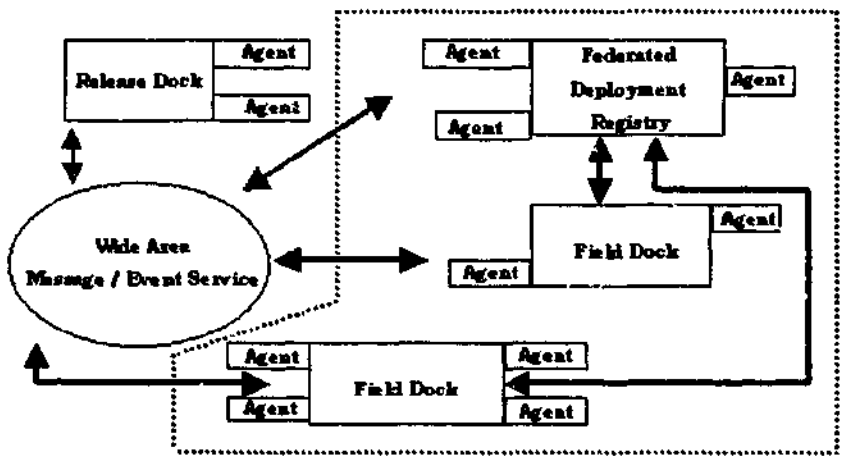


Figure 3.2: The Software Dock Architecture

Internal agents are responsible for extending the functionality of the local dock and to some extent are trusted in their operations at the local dock. Commonly, there are three services that these agents provide. These include:

- Viewing – Is responsible for building a interface which provides a view of the dock.
- Abstraction – Is where abstract interfaces are built for external agents to use. These interfaces are built to hide any site-specific knowledge which may be required to perform activities.
- Isolation – Is where the dock is able to setup a strictly controlled environment for an external agent to work within. Such an environment would allow an external agent (untrusted) to perform updates on a component where both reading and writing operations are required.

External agents are obtained from remote sites and are responsible for performing activities on behalf of a remote organisation. These agents are not trusted and normally require the help of an internal agent to setup an isolated area so that the requested action can be performed.

During the installation of a component, other agents may well be triggered as a result of different events. This can lead to a cascading number of agents being deployed throughout the system.

The communication system used throughout the Software Dock Architecture is known as the Wide-Area Messaging/Event service (WAM/E). WAM/E is responsible for broadcasting all of the events to the various software docks which have been registered within the system. The WAM/E service is also capable of using interfaces provided at the various software docks to inject events into the system.

3.3.1.2 Software Dock as a Configuration Management Tool

As previously discussed, Software Dock aims to provide a method of configuration where components can be interchanged automatically, without user intervention.

However, for such a system to allow the interchanging of components, it must assume that all components are loosely coupled which means reconfigurations can only take place when a system is in a static state as no dynamic consistency management is provided. The lack of a dynamic consistency manager means that the system is unable to provide any dynamic reconfiguration support.

When compared against the configuration management system criteria it can be seen that the Software Dock architecture addresses different areas to those systems found in section 3.2. Software Dock provides developers with an environment capable of identifying components and automatically responding to the needs of those components when their environment changes satisfying criterion ten. These changes and the information required to administer these actions are gleaned from the numerous servers and agents scattered throughout the entire system. Additionally, the architecture provides a simple but yet effective infrastructure allowing the system to automatically manage inter-component communications.

3.3.2 Adele

Adele as explained in Estublier and Casallas (1994) is a database oriented configuration management system. By making use of the underlying database schema it is possible for the individual software components, source code and header files to be individually registered into the system. Adele refers to these entries as elements.

As the elements are entered into the database, the system builds up a profile of the relationships between each and constructs a dependency graph. This graph is then used to ensure that the system remains in a consistent state.

In addition to the dependency graph, the database also allows for the logical grouping of components, source code and headers. Once grouped, global operations can be performed on the groups just as if they were an individual element themselves.

Adele also provides two other facilities which aid the construction and management of system configurations. These facilities include the concept of a *WorkSpace* and *Revision Set.*

A workspace is used within the Adele system to provide developers with an area where bindings and components can be changed with other components. All actions which are undertaken within the workspace are subject to the dependencies contained within the dependency graph. This ensures that the consistency of the system is preserved according to the database schema.

The second facility Adele provides is the notion of a revision set. A revision set is a collection of all the revisions made to a particular system or sub-component of the system. It is normally constructed from a set of revisions made within a workspace. The revision set acts as a superset of all revisions.

Due to Adele being able to file all elements into the underlying database, it is possible for the revision set to be preserved as well. Once an element, it too can have a number of operations performed on it which effects the whole set. The revision set provides the ability for developers to see what changes have been made to the system as well as giving an opportunity to reverse some of them.

3.3.2.1 Mistral

Adele manages configurations but it does not deal with the issues of distribution nor does it address the issue of dynamic configuration management. In an effort to provide a distributed approach the Mistral system as detailed in Gadonna (1999) was developed.

Mistral was built as an extension to the Adele system allowing developers to be able to move Adele databases from one point to another. Mistral provides the functionality of permitting Adele databases to be copied or moved from one location to another. The system also provides conversion utilities which allow a database created in Adele to be imported into other foreign database systems.

The process of copying or moving a database between systems involves three steps. The first step is to identify all of the deltas and then store these in a form which allows the data to be transferred. The most common forms of media include magnetic tape or email.

Once generated the deltas are shipped to the other site where they are loaded onto the new system. Once loaded a status report is generated reporting on objects created, relocated and or re-grouped. This is important data as it forms the basis of a consistency check.

After the data has been loaded and report generated, the report is sent back to the initial site where it is run through the Adele database. Once received, Mistral reads it and makes appropriate changes to the Adele database schema. The changes to the schema form the basis for the system to remain consistent.

Explained in van der Hoek, Carzaniga, Heimbigner, and Wolf (1998), the Mistral update routine, especially in the circumstances where a direct connection between databases can not be performed, relies on a manual approach. By having manual intervention in the system it is possible for the Adele database to become inconsistent. The inconsistency may be introduced as a result of an error in the report generated by the database at the remote site, or human error where the report was never run back through the original system.

3.3.2.2 Adele/Mistral as a Configuration Management Tool

As has already been discussed, Adele was designed to provide developers with the ability to reconfigure components in a static system with the aid of a database management system.

The database backend of Adele is designed to ensure the consistency of the system is always preserved while at the same time preserving the various revisions of components within revision sets. Adele itself does not address concerns about dynamic reconfiguration management nor does it provide the appropriate modules to ensure the consistency of the system while it is operational.

In an effort to address the problems with the lack of distribution in Adele, the Mistral system was developed. Mistral, however, only provides developers with the ability to be able to shift the Adele database around to other sites and does not address the other problem of Adele which is lack of dynamic reconfiguration support. Additionally, neither Mistral nor Adele provide any consistency managers which could be used to keep the components and systems synchronised while running.

Both Adele and Mistral were designed to provide reconfiguration support for components in static systems. When compared to the configuration management criteria (refer to section 3.1) it is possible to establish that both Adele and Mistral satisfy a number of the criterions. These include:

- Providing support for the identification and manipulation of the various component revisions
- Partially providing the infrastructure required through the use of revision sets to allow developers to try out previous revisions
- Providing primitive type checking through the use of the database management system
- Providing consistency of a status nature by making use of the DBMS
- Providing support for component reconfiguration of a static nature

The only difference between Adele and Mistral from a configuration comparison is that Mistral provides support for the database which contains the configuration delta to be exported to another site. This allows the configuration repository to be backed up although great care needs to be taken when integrating the repositories with one another.

3.4 Dynamic Component Management

In this section, those configuration management systems which support the reconfiguration of the interconnections between components and the components themselves are examined.

3.4.1 Surgeon & Polylith

The system Surgeon as detailed in Hofmeister, While, and Purtilo (1993) builds upon the Polylith Software Architecture (Purtilo and Hofmeister 1991; Purtilo 1994) to provide the additional functionality of dynamic reconfiguration management.

3.4.1.1 Polylith

Polylith was developed to address some of the difficulties being experienced with distributed computing. The primary design goal of Polylith was to provide a software architecture allowing developers to construct distributed applications without having to invest a significant amount of time in dealing with the issues of distribution.

To provide this level of abstraction a concept known as a *Software Bus* responsible for coordinating all of the inter-component communications was introduced. The software bus is also responsible for the translation of data structures between the different distributed buses or various methods of sharing processors (eg. shared memory). This process of data translation and marshalling is not that unfamiliar when compared with distributed frameworks such as CORBA (Object Management Group 2001) and DCOM (Brown and Kindel 1996).

In addition to the software bus providing a level of abstraction for developers, Polylith was also designed to make use of a language responsible for separating the interface from the functionality. This language was known as the Module Interconnection Language (MIL).

MIL was designed to be implementation independent allowing developers to choose the language that they wish to develop the module in while at the same time providing a level of abstraction from the implementation and the interface. This is very similar to the notion of a Interface Definition Language (IDL) used in CORBA, COM and many other distributed system frameworks.

The role of MIL in Polylith was to allow developers to specify the semantics of the interface. Once defined, Polylith would analyse the MIL specifications and build a graph that depicts the relationships between the modules in the system. Additionally, MIL contains the ability for developers to specify the translation tables necessary for data flows between different distributed processing schemes and different software buses.

Although Polylith was designed to provide a level of abstraction for developers to develop distributed applications, it did not address the issues relating to component reconfiguration. To address this issue, the system called Surgeon was developed.

3.4.1.2 Surgeon

Surgeon was developed on the foundations of Polylith in order to provide developers with the ability to be able to dynamically reconfigure systems already operating on the software bus.

The approach adopted by Surgeon in dealing with the difficult issue of dynamic reconfiguration management is one of simplicity. As far as Surgeon is concerned, a component is considered to be in a consistent state providing messages or method calls being sent between components continue to flow and reach their desired destination. Even with an assumption like this, the Surgeon system provides no consistency manager to oversee or coordinate the transitions of the various configurations.

Surgeon implements its reconfiguration policy by identifying those components which interact with components that are to be replaced. The connections between neighbouring components and the component to be replaced are *frozen* to prevent any further communication or participation within the system.

Neighbouring components who have had some of their bindings frozen continue to operate as normal except for where they incur an interconnection which deals with the component being replaced. Any data waiting to be sent to the reconfiguring component is held until the object is ready to be restarted again. This approach is similar to the node quiescence scheme detailed in Kramer and Magee (1990). Surgeon however does not implement the quiescent set of routines.

Since the development of Surgeon, further enhancements have taken place as is detailed in Purtilo and Hofmeister (1991), to provide components with the ability to be able to save and restore their state. The ability to manage a components state enables it to retain persistent data through various component revisions.

A disadvantage of the approach taken in Purtilo and Hofmeister (1991) is that it requires developers to nominate the reconfiguration points and specify the methods for restoring and establishing the state of the components. Such an approach goes against what Polylith was originally designed for where simplicity and abstraction were the main aims.

In such an approach, a tool responsible for the reconfiguration would then have to coordinate the exercise of extracting the state from one component and marshalling the data to the other.

3.4.1.3 Polylith & Surgeon as a Configuration Management Tool

As mentioned in section 3.4.1.1 the Polylith system was not designed with configuration management in mind. It was developed to provide a framework for developers to build applications which could run on a variety of distributed architectures.

Surgeon does provide support for dynamic reconfiguration management however it does have a significant problem in the approach it takes to dealing with frozen components. There are also additional concerns with regards to the assumptions made relating to the consistency of components during a reconfiguration activity.

The problem stems from those components which have been marked for a reconfiguration activity. Once frozen they are unable to provide any information about their status and whether they will be able to perform their designated activity. This approach raises many issues with regard to the protocols and communication mechanisms used within the system.

From a configuration management point of view the Surgeon system lacks a consistency manager which is responsible for coordinating the changes within the system and the synchronisation of states between components being removed from the system and those being added.

In addition, the Surgeon system and later systems detailed in Purtilo and Hofmeister (1991) provide no ability for any deadlock analysis to be performed. Putting these concerns aside, it has been identified in Goudarzi (1999) that under certain circumstances the reconfiguration mechanism used can result in failure and in some cases can cause a crash of the replacing component. Overall the Surgeon and Polylith systems do not rate too well when compared to a number of configuration management system criterions. Both systems provide some type checking support satisfying criterion five and in the case of the Surgeon system some support is provided for the reconfiguration of components which satisfies criterion ten.

3.4.2 Programmers Playground

The Programmers Playground introduced in Goldman, Hoffert, McCartney, Plun, and Rodgers (1997) and Goldman, Anderson, and Swaminathan (1993) provides a series of software libraries which have been primarily designed to support distributed multimedia applications. The playground has been designed with the specific intention of separating the implementation of a module from its interface while at the same time embracing the concepts behind the 'I/O Automation' model explained in Sethuraman and Goldman (1995) and Goldman, Anderson, and Swaminathan (1993). The playground also provides limited[5] support for dynamic reconfiguration and includes enhanced support for discrete and continuous data types.

[5]Programmers Playground does not support the instantiation of objects, preserving consistency and connecting them into the system at runtime.

An unique feature of the Programmers Playground is the way the communication channels between each of the modules are specified. Component connections are managed with the aid of a design tool rather than having components explicitly specify the communication channels. Using this component connection manager allows the operator to directly manipulate the external interface of an object and its corresponding communication channels.

3.4.2.1 Overview of the Playground

The playground can be viewed as a workspace for developers. It is here that components are defined, created and linked together to form distributed applications.

In addition to providing an area for component development, the playground provides an abstract environment where applications can run in isolation. This isolation area allows applications to execute with their respective components even though the entire playground is populated with other components running other systems. Not unlike a children's playground, the Programmers Playground does not endeavour to segregate applications into their own domain but instead keeps them in the same area under the supervision of the executive.

As each component is built, it is constructed from a series of primitive data types that the playground supports. Once a component has been built, it is given a presentation layer. This presentation layer provides the separation from the internals of the component and the external links to other components.

In addition to providing a barrier between the internal and external parts of a component, the presentation layer adds extra functionality to the component. The presentation layer is responsible for providing the appropriate `get` and `set` methods to allow data structures contained within the component to be observed and manipulated. Such manipulations are coordinated by the security module of the playground.

By having these interfaces present in each component, observers can inspect the system and see how certain components react with one another. In addition to the observation, the observer can also inject changes or events into the system and see how the components react with the changing environment. Due to the way in which components are linked, it is possible to determine what effect one change to the system has on the rest of the modules.

As mentioned previously, one of the unique characteristics of the Programmers Playground is the way that communication channels between components are not explicitly defined by a developer of the system. This allows the configuration of the system to dictate what communications take place between components and when those connections should be made.

These communication channels are controlled through the use of a binding manager. The role of the binding manager within the system is to supervise and inform all components of relative changes made in the system. The binding manager is also responsible for recording the links between all

components as well as contacting individual components and informing them of changes specific to them.

The Programmers Playground architecture revolves around three sub-systems. These sub-systems control the data types and their usage, the control and coordination of components and the establishment and management of connections between components.

Data SubSystem: The Data SubSystem is responsible for providing to the Programmers Playground a list of data types which can be used within the system. These data types include:

- Integers
- Reals
- Booleans
- Strings
- Tuples
- Aggregate Data Types

A tuple data type provides developers with the ability to store a set of records and pass them around the system as a collection. This is similar to the aggregate data type except that an aggregate contains an array of single elements while the tuple can have many elements in the one record.

The Data SubSystem is also responsible for providing the way in which the data types are used. As mentioned earlier, the Programmers Playground provides support for the direct inspection and manipulation of data structures contained within components. Although inspection is possible it is not mandatory. When a data type is being defined the component developer is provided with the option to make the data element either publicly available or not. If the component developer does wish to make the variable publicly available then a public name is assigned to the data element and the appropriate access privileges are assigned.

Control SubSystem: The Control SubSystem is responsible for determining the responses which need to be taken as a result of a change in the state of a component or a change in the environment. Although a playground application has no direct control, it is capable of manipulating its own data values within the presentation layer to ensure that the appropriate state changes occur. As a consequence, the system ensures components which are effected by the change interact with one another in an appropriate manner.

Connection Management: Connections between components in the playground are controlled through the use of a communications manager. The communications manager is responsible for ensuring that type safety is preserved and that the security requirements[6] are met.

[6]Security requirements refer both to the accessing of the components in addition to making sure that the observers of the system can only modify those data structures that they are entitled to.

Other responsibilities for the communications manager include administering two types of connections. These connection types are classified as *simple* or *element-to-aggregate* connections.

Simple connections are those which bind together two data items of the same type. The flow of data along these connections can be defined by a developer to be uni-directional or bi-directional. As simple connections deal with single elements of data it is unnecessary to provide recording locking operations.

Figure 3.3 illustrates how two components can be connected to one another with the use of uni-directional and bi-directional communication channels. The figure also demonstrates the abstraction which the Programmers Playground provides for components where the computational elements are separated from the published data elements.

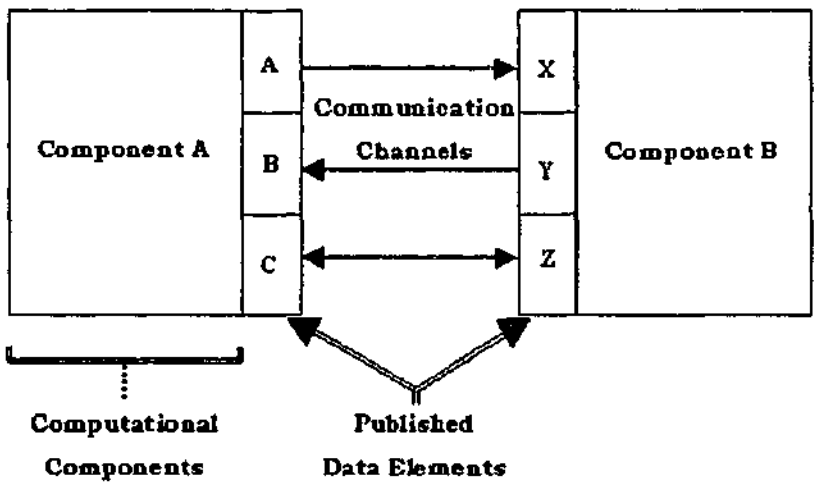


Figure 3.3: Representation of a Component in the Programmers Playground

The other type of connection known as a element-to-aggregate connection allows a collection of tuples to be bound together providing the types of tuples are the same. In such a connection there is a server (the component which initiated the connection) and a client (the component receiving the connection).

This distinction becomes important when you consider what view the client and the server have of the data. The component acting as the server in this situation can see a series of tuples while at the client side the only data which is visible is a single tuple which contains the result of a query.

Due to the aggregate connection dealing with many records at the one time, the element-to-aggregate relationship provides support for record locking operations. This enables the client to lock the record that it is currently using while at the same time allowing the server to manipulate other records.

3.4.2.2 Playground & I/O Automation Model

By making use of the I/O Automation model, the Programmers Playground is able to provide developers with a dynamic environment allowing the modelling of dynamic behaviour of components within an application. This behaviour can be represented graphically with a tool called 'Euphoria' as detailed in Goldman, Hoffert, McCartney, Plun, and Rodgers (1997).

This dynamic behaviour of the playground can be modelled through the various state changes which take place throughout the system. By allowing the observer to make changes to the configuration of the system it is possible to model the dynamic nature of the system.

3.4.2.3 Programmers Playground as a Configuration Management Tool

The Programmers Playground provides developers with a dynamic environment in which to assemble large distributed multi-media applications. To achieve this a system was established where components could be developed and individually tested allowing developers to concentrate on the task at hand.

The playground provides support for components to have bindings established between them at both design and runtime. These bindings are handled by a communications manager that coordinates traffic between components within the system and arranges for the bindings to be changed.

One other important attribute to note about the Programmers Playground is that it makes use of the I/O automation model. This allows the playground environment to introduce the notion of an observer who can manipulate public data variables. Such changes to the system allow the observer to model the interactions between components and document the expected and unexpected changes. By allowing this activity, the Programmers Playground claims to support dynamic reconfiguration as the environment presented to the observer allows for components to be created, the manipulation of public data values and altering the bindings between connections. It does not provide the support required to update components while the system is running. The playground also lacks the appropriate managers required to support concurrency, consistency and deadlocking issues.

As a consequence, the playground is a nice environment in which to model changes to a system, but lacks support in the area of dynamic reconfiguration manager. From a configuration management point of view, the Programmers Playground provides developers with a number of services which have been designed to simplify the process of creating components and interconnecting them to form applications.

Compared with the configuration management criteria detailed in section 3.1 the Programmers Playground satisfies a number of the criterion. Those satisfied include criterion five with the integration of a type checking system for components, criterion six with the Programmers Playground providing

an architecture that is aware and sensitive to the needs of components contained within it and criterion seven with the provision of routines which are responsible for managing the inter-component communications. Compliance with criterion nine is achieved through the introduction of a binding manager capable of allowing developers to use a limited set of tools to identify the dependencies of components within the system while criterions ten and eleven are satisfied by the binding manager providing the functionality to statically reconfigure components while providing developers with an abstracted interface to shield the lower level functions required to effect the reconfiguration.

3.4.3 CONIC

As is explained in Magee, Kramer, and Sloman (1989), CONIC was designed to allow developers to build large distributed applications and at the same time providing them with an architecture allowing the removal of the barriers associated with specifying configurations. CONIC was also built to provide support for a mixed target host environment, to be uniform in its approach and to make it simple to develop distributed applications while at the same time being efficient and portable.

The development of CONIC occurred at a time when there seemed to be little agreement with regard to the concepts of modularity, concurrency, synchronisation, inter-process communication and component reconfiguration.

In an effort to address these problems, a number of approaches were adopted, ranging from completely new development and programming environments through to the modification of operating systems.

The driving force behind these developments stemmed from the concept that distributed applications were considered to be a collection of sequential programs as opposed to individual components working together to solve a common aim. The sequential programming approach translated into many systems where the inter-connections between components became so tight that it was impossible to provide any change management facilities.

The term change management refers to the concepts of evolutionary and operational change. Evolutionary changes are very hard to predict and normally result in program modifications being incorporated into already existing systems or new technology. Operational changes are those which need to be made as a result of an organisational restructure or business rules.

To overcome these problems, CONIC provides its own language based approach to building distributed applications. The principal objective of the language revolves around the separation of the interface from the functional specification.

To achieve this, CONIC introduced two languages. One is used by developers to specify the functionality of components, while the second is used to define the interface and configuration requirements of components. In addition, the information specified in the interface specification language is used to support dynamic reconfiguration.

3.4.3.1 CONIC's Module Programming Language

The CONIC programming language allows for the definition of task modules types. Once built, these types are used as the blueprint for the creation of instances within the system.

Modules communicate with one another through strongly typed ports. These ports are classified as being either *exitports* or *entryports*. As the name suggests, an exitport is where the results of a service can leave one component and be sent to another. An entryport is where data originating from one component can be received and processed.

For every port which exists within the system, whether it is an entryport or exitport, it must have a local name specified in addition to specifying a type.

The process of *binding* entryports and exitports together is handled by the configuration specification. Such a specification can only be performed by sending the appropriate configuration messages. These messages are sent to a configuration manager which uses them to arrange the components into the appropriate configuration.

By having a configuration manager establish the configuration of the system, CONIC modules avoid the worry about configuration issues. Each module is independent and free of all configuration restraints as all references are self-contained.

CONIC provides two approaches to the transmission of data. These approaches allow data to be transmitted in a asynchronous or synchronous manner.

The *Notify Transaction* approach to data transmission provides support for the asynchronous transfer of data. Using this approach allows for the uni-directional transfer of data and also provides the support necessary for the transmission of multi-destination messages. Additionally, the receiver can block the originator of the message until it is ready to accept and process the data message.

The *Request Reply* approach provides the ability for the data message to be passed by making use of a bi-directional synchronous form of communication. When using this technique for data transmission the sender of the message is blocked until a reply is received from the target module. As the possibility exists for the sender to block for an extended period of time, the request reply approach allows for a fail clause. This clause allows the originator to abort the data transmission to the target if a certain time period expires or if the transaction fails.

3.4.3.2 CONIC's Configuration Language

The CONIC configuration language is used explicitly to define each of the configuration tasks for each module. Once the tasks have been defined, the language provides the semantics for the grouping together of configuration tasks into logical nodes. The nodes can then be used to form a hierarchical relationship of tasks.

The configuration language makes use of few keywords. Some keywords that are supported include: `use`, `create`, `entryport`, `exitport`, `reply` and `link`.

3.4.3.3 Dynamic Configuration

CONIC provides its dynamic configuration facilities through the use of a configuration tool. The tool comprises a collection of precompiled logical nodes which can either be run as UNIX processes or as standalone programs. The actual form these nodes take is dependent upon the run-time support that is offered by CONIC.

Parameters specified to the configuration manager are used in one of two ways. The first is to provide configuration information detailing how many tasks should be created at a particular node. This provides developers with the ability to distribute the load throughout the system. The second set of parameters provided to the configuration manager allows command line arguments to be passed to the modules.

3.4.3.4 Runtime Support

In order to provide configurable runtime support the CONIC system allows the user to tailor the change management system to meet the specific requirements of the user. By providing runtime support, CONIC is able to provide the ability to dynamically configure components while the system is operating. This functionality provides the CONIC environment with the flexibility required for applications and developers. Developers are able to make changes to the system whenever they consider it necessary. These changes are made with the assistance of the configuration manager node type 'iman' and the virtual target facility 'vf' which assist in the reconfiguration of remote targets.

Configuration Manager - *iman:* The configuration manager type *iman* is responsible for providing a user interface to the configuration manager within the system. Users can issue reconfiguration commands such as `manage` to redefine parts of the system.

The configuration manager is able to reconfigure the different components by manipulating the various `entryport`'s and `exitport`'s which exist in the system. As the standard communication protocol used in CONIC provides no guaranteed form of message delivery, a special form of the protocol is used to ensure that configuration messages are received by the destined module.

Virtual Target - *vf:* As CONIC provides support for nodes to be located on other systems in a distributed environment, it is important that there be some method available to allow for those nodes to be contacted and manipulated. The virtual target fills this void. It is responsible for acting as the proxy for objects located remotely. It accepts all the instructions for the node and then forwards the requests onto the node at the remote location.

3.4.3.5 CONIC as a Configuration Management Tool

When CONIC was designed, it was built with the purpose of providing a framework allowing developers to build large distributed applications without having to deal with the technical issues associated with distribution.

In addition to providing a framework for the development of distributed applications, CONIC provides a rich set of tools allowing components to be built independently from their interfaces. This separation provides for runtime and dynamic configuration as well as reconfiguration management.

CONIC provides this support through the use of a configuration manager capable of redirecting the connections between components. Additionally, the configuration manager is able to contact components located on remote systems and reconfigure their connections through the use of the virtual target mechanism.

Although CONIC does provide dynamic reconfiguration support and allows for the separation of the interface from the implementation it still suffers from the problem identified in Magee, Kramer, and Sloman (1989). This problem relates to making changes to a running system which is operating the CONIC architecture. CONIC provides no services for ensuring that the system remains in a consistent state when a configuration change is made.

Overall, CONIC does provide a sound basis for other configuration management systems to build upon. Just like the Programmers Playground, CONIC satisfies a number of configuration management criterion. The differences between CONIC and the Programmers Playground from a configuration management perspective include the CONIC system not providing any tools to ensure consistency of the system or perform any dependency analysis on the relationships which exist within the system. However CONIC does provide better support for component reconfiguration and management in both static and dynamic environments as well as providing developers with a number of routines and interfaces which provide an abstraction from the lower level commands needed to process the reconfiguration of components.

3.4.4 Equus

As detailed in Kindberg (1991), Equus is a system built specifically for providing the functionality of dynamic reconfiguration management. Equus is able to provide this through the use of a specialised kernel.

These modifications provide a system capable of creating and migrating components[7], providing components with the ability to communicate with one another and providing a location service to detect where components are located. Additionally it provides support for determining what ports

[7] In Equus, components are referred to as incarnations.

are associated with each component, message routing capabilities, termination of components and provides a service which detects and notifies components of a particular event.

An underlying assumption made by Equus is that all the components defined within the system are loosely coupled. Equus also assumes that components are normally connected to one another through channels. These channels are used as intermediaries between components and can provide both uni-cast and multi-cast communication methods.

When configurations take place under Equus, careful consideration takes place to determine whether the data streams or ports being reconfigured interface with a multi-cast or uni-cast channel. If a data stream is bound to a uni-cast channel and the end configuration does not require the component anymore then the stream is removed. However, if the stream is bound to a multi-cast channel, then the new stream/channel is inserted and no changes are made to the old binding.

The configuration management system of Equus provides three methods in which the configuration of a system can be changed. The first approach is to propagate a port from one component to another.

The process of propagating a port between components involves the creation or identification of the component which is to replace the old component and then to propagate the port across. Throughout all of the configuration activities, the Equus kernel ensures no data is lost as a result of the data streams being moved around.

A second option available to software developers for reconfiguring a system under Equus is the attachment of a new port to an already existing configuration. If used, the approach entails the system locating the replacement component, creating a new port on the desired component and then connecting it to the system. It is important to note that this approach facilitates the creation of a port rather than the propagation. The configuration is then assessed to determine whether the binding is connected to a multi-cast or uni-cast channel and the appropriate configuration changes are made.

The third approach offered by Equus for the reconfiguration of components is stream rebinding. This allows components the ability to change the components they are bound to at runtime. Stream rebinding involves having the new[8] component establish itself and then reposition the bindings to reflect the new configuration. Once the changes have been made and the new binding is in position, the old binding is removed from the system.

3.4.4.1 Equus as a Configuration Management Tool

The suitability of Equus as a configuration management tool clearly becomes evident after examining the architecture of the system and the extra functionality which has been incorporated into it to assist

[8]In this context, new also refers to an already existing object which has been located.

developers. Predominately, this extra functionality focuses on supporting dynamic reconfiguration and allowing for the migration of components between different nodes.

Even though Equus does provide dynamic reconfiguration support to its components, it does make a number of assumptions as to the construction of the system. The first assumption made about the system is concerned with data transmission. Equus assumes that all data being transmitted between components will occur in one packet and hence not be segmented during transfer. Although a desirable assumption to have in a prototype, it is clearly not practical in a production system.

Additionally, Equus bases all of its configuration functionality around the assumption that each component is fully aware of its own consistency information and that the component is able to calculate the impact on the rest of the system if it were unavailable. As mentioned in Goudarzi (1999), no individual component can be totally aware of its impact on the system nor can it totally control its own consistency. This highlights one of the major problems with Equus as there is no comprehensive configuration manager present which is responsible for coordinating the various reconfigurations. Having no configuration manager in the system to coordinate reconfigurations means that there are no modules responsible for performing any deadlock analysis. As a result, the system provides no way of determining whether the configuration changes being performed on the system will result in the system being deadlocked or entering an inconsistent state.

One final concern is the inability of Equus's to deal with the concept known as *frozen time*. This refers to the time where no execution takes place as a result of a reconfiguration operation. Equus provides no way of handling such periods nor does it detail what happens to messages and method calls during this period of time.

Similar to CONIC, Equus shares a number of configuration management similarities. Infact, Equus satisfies the same criterions as CONIC except Equus restricts its type-checking support to the ports used in inter-component communications. The Equus system also provides a limited set of commands to configure the ports on components. These configuration routines are geared more towards the reconfiguration of the ports themselves rather than components.

3.4.5 REGIS

REGIS (Magee, Dulay, and Kramer 1994) was built to further enhance configuration support, especially the dynamic reconfiguration aspects that were developed in REX (Magee, Kramer, and Sloman 1990) and the CONIC system which both introduced the separation of the configuration aspects of a component from its computation responsibilities.

REGIS extends this work by further developing the notion of separating the configuration aspects from the computation components while at the same time providing support for dynamic program structures. To provide this support the REGIS system includes methods for allowing both *lazy* and *direct* component instantiations.

Computation components written to work with REGIS are developed and executed under the C++ object model. REGIS uses the DARWIN configuration definition language as the means for expressing the configuration of a component.

As with CONIC, programs normally exist as a collection of components which are integrated with one another and execute in parallel to achieve the overall goal. Such an arrangement of components requires a tool which can automate the task of performing consistency checks across the entire system.

CONIC, REX and REGIS share many similarities when it comes to dealing with the fundamentals of components. The difference becomes apparent when the general support that REGIS provides for objects and the support given to provide dynamic configurations is considered.

A design objective for the REGIS system along with other component based systems is component reuse. To this end, the REGIS architecture was developed to provide support for user defined and third party components alike and to integrate them into the one architecture. This integration allows developers to save on labour costs associated with the components development and testing time. To allow for this integration components are said to be *context independent*. This independence allows for components to be used in contexts other than the one they were originally designed for. This further reinforces the notion that a component should be independently deployable as stated by Szyperski (1997).

In addition to component re-use, REGIS has been structured to allow developers to reuse the structural aspects or program skeletons for future developments. This is achieved with the DARWIN configuration language as it allows for parameterised types to be specified in templates. These templates allow developers to build generic components (eg. linked lists, binary trees) which can then be used in other components to build upon their structure.

The second design objective of REGIS is to provide support for dynamic configurations and is provided within REGIS through the introduction of two new concepts. These include the lazy instantiations of objects and support for directly instantiating dynamic components.

Lazy instantiations provide a means for a component to be defined within a configuration specification at design time but not to be directly instantiated when the system runs. The 'lazy' component only becomes active when a message is sent to it. The concept of lazy instantiations is normally coupled with recursion which provides a mechanism to allow systems to grow when needed. Components which are dynamically instantiated are indistinguishable from those that were created statically.

Dynamic instantiation allows components to directly instantiate other components (composite or computational) dynamically at runtime. This provides an alternative from lazy instantiations when there is a requirement to specify parameters to the new component to aid in its construction or when a recursion approach is not applicable to the problem domain. The concept of dynamic instantiation also allows components to only be instantiated when required.

3.4.5.1 REGIS as a Configuration Management Tool

The REGIS system provides further advances to those earlier systems such as REX and CONIC with regard to configuration management. REGIS provides these advances through the extra functionality provided in the area of dynamic configuration management. This functionality is achieved by allowing components to create new components when required (lazy instantiations) or by allowing components already operating within the system to directly instantiate others.

In addition to the advances made in dynamic configuration management, REGIS also provides developers with an ability to separate the configuration specifications from the computation requirements of a component. This has been achieved through an improved version of the DARWIN language. The improved functionality allows developers to build generic components capable of taking other components as inputs and to build templates.

To provide the functionality of REGIS, a number of daemons work together in order to supervise the entire system. Daemons provide naming services and the ability to remotely coordinate components located on other systems.

However REGIS lacks an appropriate sub-system which combines the task of keeping the system consistent with deadlock analysis. Having no deadlock analysis means that components can potentially get themselves into an awkward state and jeopardise the consistency of the entire system. Additionally, there is the need for some extra work to be performed in order to provide the components within the system with some level of persistence.

As with most configuration management systems examined, REGIS's type checking supports both inter-component communications and ensures the type-safety of variables used within a component. REGIS also provides a number of routines which aid developers in the management of components and provides the flexibility of reconfiguring components in both static and dynamic systems. All of these routines provide a level of abstraction to developers who are unaware of the underlying procedures taking place to achieve the requested action.

3.4.6 CHORUS

CHORUS detailed in Rozier et al. (1992), is a distributed operating system which has been under development since 1980. Since its inception, there have been a number of significant changes performed on the underlying structure of the system but its original aims have not changed over the last four versions.

The design concepts embodied within CHORUS include the provision of an architecture which supports components communicating with one another through a structured set of messages. CHORUS is also able to provide real-time services to servers. To achieve this level of real-time support a

special module is provided within its *core* dedicated to providing real-time systems with the support required.

CHORUS makes use of a modular approach to facilitate the development and continued running of components and servers. This approach allows developers to add, replace, modify or remove components without causing massive disruptions to the remainder of the system while at the same time providing a clear distinction between the actors[9] (computational component) and the ports which form part of the communication mechanism used.

The principal design aims behind CHORUS includes support for those systems requiring a number of different applications to be running over a diverse environment. This diverse environment could range from different types of machines or architectures through to executing various applications written in any language.

CHORUS also aims to provide developers with a degree of *synergy* by making components appear as if they are located together in the one spot when in reality they are geographically spread over a large number of resources.

3.4.6.1 CHORUS Architecture

The CHORUS architecture has been designed to provide varying levels of abstraction to the underlying hardware as well as providing a modular approach. The underlying architecture of CHORUS is made up of a number of levels. These levels include the CHORUS Nucleus, the middleware or sub-system layer and the application layer.

CHORUS Nucleus: In CHORUS, the nucleus forms the centre of all activities and is responsible for the managing and coordinating the lowest level of resources available on the local machine. To increase modularity and flexibility, the nucleus is broken up into four sub-sections.

The first sub-component located within the nucleus is the *supervisor*. This module is responsible for dealing with the hardware of the system and the various types of hardware exceptions and software traps which are generated.

The next sub-component within the nucleus is responsible for providing *real-time executive services*. These services are primarily concerned with the allocation of tasks to the available processors. Additionally, the real-time executive is responsible for applying scheduling algorithms to the priorities of tasks waiting for a processor.

As CHORUS deals with components scattered throughout the system, it is critical that the nucleus provides a *memory manager*. The memory manager is responsible for managing both the allocation of local and virtual memory within the system.

[9] An actor refers to a component.

An important part of the CHORUS system is the processing of messages between components. To achieve this the nucleus provides a sub-component responsible for coordinating all inter-process communications. The role of the *communications manager* is to provide the infrastructure required by CHORUS to send messages to any component within the system. Through it, both synchronous and asynchronous messages can be sent using the RPC or various IPC communication systems.

CHORUS SubSystems: CHORUS subsystems are a collection of resources working together to provide a means for application programs to access the lower levels of the nucleus through a middle layer. In order to provide this middleware functionality the subsystem provides a number of interfaces to both the application program and nucleus.

Individual Application Programs: Application programs within CHORUS are layered at the top of the CHORUS cell. This allows them to take advantage of the complete CHORUS architecture if they require it and at the same time allowing them to operate normally. It is important to note that although CHORUS provides hooks to the nucleus through the middleware layer it does not force programs to make use of the hooks.

3.4.6.2 CHORUS Nucleus Abstraction

Actors: Within the CHORUS nucleus there are many components communicating with one another. These components are known as 'actors' and consist of many resources. Each actor contains two separate address spaces. One address space is known as the 'user area' and is where application programs execute while the other address space known as the 'system area' is reserved for privileged nucleus operations.

There are three types of actors loaded within the CHORUS nucleus. These actor types include: user, system and supervisor. Each actor type has a varying degree of trust which dictates what operations can be undertaken and whether they can be performed.

A 'user' actor is considered to be unprivileged and untrusted when it comes to performing operations within the nucleus. A 'system' actor is considered to be unprivileged to perform nucleus operations but is able to execute non-privileged commands. The 'supervisor' actor is the only actor able to perform privileged operations on the nucleus.

Threads: Each actor is responsible for performing an action on behalf of the nucleus. In some cases there is a need for one or more routines within an actor to work concurrently. To facilitate this, CHORUS uses an underlying IPC mechanism to provide multi-threading support as well as allowing actors to synchronise themselves with one another. Although threads can communicate with other threads in actors, they are unable to migrate from one actor to another.

Ports: Ports form the fundamental basis for messages[10] to be passed between actors as well as serving to decouple the interface of a component from its computational elements.

[10] A message is a data structure of finite length carrying binary data. A message has a origin and a destination.

Ports defined by CHORUS act as portals to the inner-realms of the actors. These ports are responsible for receiving messages from remote actors and ensuring that the threads located within the local actor pick up the messages. Ports are also used to provide dispatch services for messages heading towards other actors.

Figure 3.4 illustrates how the CHORUS architecture is divided into a number of layers including the application layer, middleware and nucleus. The architecture demonstrates, that the lower the layer, the lower the level of operations being performed. The figure also illustrates how the nucleus is subdivided into three sections responsible for real-time services, memory management and supervision. The relationship between the actors and the lower sub-systems is also highlighted.

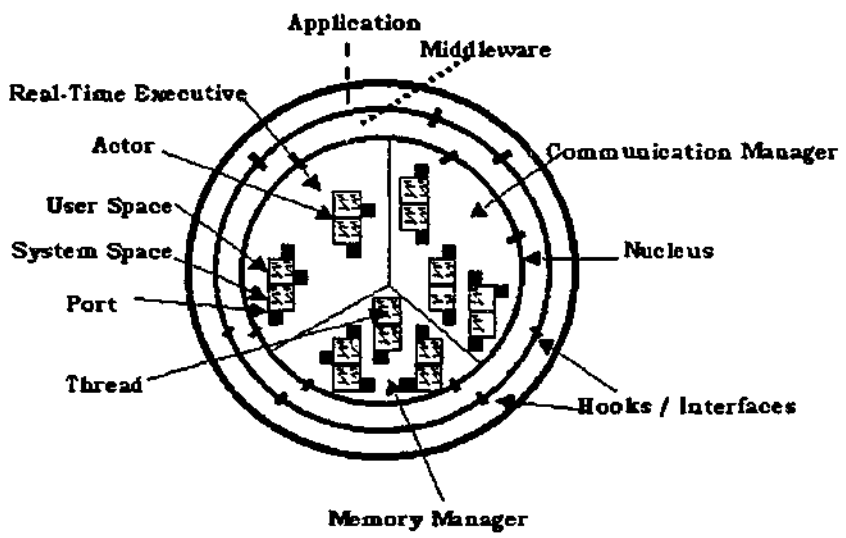


Figure 3.4: CHORUS Architecture

Reconfiguration in CHORUS

Allowing the use of ports to decouple components from their interface gives rise to the concept of dynamic reconfiguration. By using ports as an extra level of indirection it is possible to provide additional functionality such as the migration of ports between actors.

CHORUS provides two approaches to dynamic reconfiguration management. The first is performed through the migration of ports from one actor to another. This approach is normally used when a new server needs to be introduced into the system to provide additional functionality.

Port migration refers to the process of repositioning a port from one actor to another. This process results in the port located at the old actor being removed while the new actor gains an additional port. Once the port is in position, messages which had not already been processed by the old configuration, will be processed at the new configuration without any considerable interruption.

A disadvantage to the port migration approach is that it requires participation of the server to coordinate the migration process.

An alternative approach to dynamic reconfiguration management is through the establishment and manipulation of a port group. Port groups are the grouping together of ports spanning many actors and potentially scattered across many sites. They allow for various messages to be distributed amongst the group and to the various threads contained within each of the actors.

The grouping of ports provides the framework necessary for passive reconfigurations. This form of reconfiguration allows the server to be free from all of the lower level details involved in the reconfiguration process.

The underlying feature of port groups makes dynamic reconfiguration possible and works upon the basis that messages are rerouted or forwarded when one port within the group is unable to receive the message and act upon it.

Adopting this approach allows dynamic reconfiguration to be achieved, through the establishment of another actor in the port group with the corresponding ports. Once established, the old actor is removed from the port group and the new actor takes over the role of processing messages.

All reconfiguration and communication actions are supervised through the use of a protection identifier (a unique identifier which must be known to make changes to the system) and a network manager responsible for making sure messages moving from one actor to another make use of authorised pathways.

3.4.6.3 CHORUS as a Configuration Management Tool

CHORUS was originally designed to provide a comprehensive framework for the development of large distributed systems while at the same time allowing these systems to execute within a UNIX environment. Additionally, CHORUS was designed to provide support for developers through the implementation of a modular framework.

In more recent additions of CHORUS the concepts of dynamic reconfiguration have been explored. These goals have been achieved either through the use of port migration or through the management of port groups.

Unfortunately the approach to dynamic reconfiguration management introduced in the CHORUS system brings a number of problems. One problem is that the CHORUS system seems to ignore the internal state of the server while a reconfiguration action takes place. There appears to be no consideration or forward planning performed, such as preserving internal data members of an actor or a port, or getting a subsequent actor within the system to synchronise the state between the old and new configurations.

There also appears to be little analysis performed to determine what the ramifications are of a message being sent to an actor while the system is in the process of performing a reconfiguration. CHORUS seems to ignore the concept that one message sent to an actor may result in other operations starting up within that actor or elsewhere in the system causing inconsistencies which may complicate the reconfiguration process.

Overall, CHORUS provides a suitable framework for the development of dynamic and large systems but provides very simplistic dynamic reconfiguration tools. There is a considerable lack of support in the areas of dependency and concurrent analysis which could potentially cause a CHORUS actor to deadlock or enter an inconsistent state.

Unlike the other systems examined, CHORUS addresses the issue of configuration management by providing a complete operating system which has been specially designed to provide an architecture allowing components to communicate with one another. This operating system has been refined to allow components to interact within a real-time environment. Although CHORUS is primarily an operating system, it can still be measured against the criteria established in section 3.1.

CHORUS, like other systems provides a limited type checking system which is primarily geared towards checking the various types of ports used within the system. These type-checks ensure that compatible and appropriate types are being used to inter-connect components but requires the aid of a developer's compiler to enforce type-checking within the components. Additionally, CHORUS provides services which allow developers to manage the inter-connections between various components while at the same time providing abstract facilities to manage and manipulate the components within the system in both a static and dynamic nature.

3.4.7 Finite State Machines

In Lim (1993) an alternative approach to dynamic reconfiguration management is presented. This approach makes use of finite state machines coordinating the reconfiguration of basic machines[11] rather than using the traditional transaction based reconfiguration approach.

Throughout his work, Lim identifies a number of problems associated with transaction based configuration management systems. Some problems include the way in which transaction based reconfigurations provide constraints on the way in which processes can be synchronised.

Additionally, there are time constraints placed on transaction based systems which require all effected nodes to become quiescent. Such an action may take a considerable amount of time and hence delay the reconfiguration of the system. As a consequence, Lim derives from this problem that it should not be necessary to have all the components in a quiescent state before reconfiguration commences. Reconfiguration should start to commence as soon as possible.

[11] A basic machine is a term used to describe a component within the system that can be reconfigured.

In order for a finite state machine to work it must first have the system expressed in a notation that the state machine can process. This encourages developers to be very particular about the layout and design of the system so as to simplify the state machine while at the same time maintaining functionality.

3.4.7.1 Process of Reconfiguration

Once specified, a considerable amount of processing is conducted to determine the required recovery paths that are available as well as calculating the possible deadlock situations which may result in the execution of a recovery path. While determining these paths, the state machine considers other conditions which have been specified such as invariants and resource constraints.

The process of reconfiguring a basic machine using the state machine approach involves many steps. To proceed with a reconfiguration, the basic machine must first be operating in a manner which is consistent with its current specifications. Once a reconfiguration of the system is requested, a set path is followed to allow the basic machine to move from one configuration state to another.

As the basic machine is reconfigured, it heads towards a transient configuration. While performing the reconfiguration the state machine applies transient conditions[12] while at the same time slowly removing conditions relevant to the old configuration. It is during this time that the state machine determines whether other basic machines located within the system require reconfiguration or whether a need exists to execute any recovery actions to stabilise the system.

During the process of reconfiguration, a basic machine enters a region known as the transient configuration point. By this stage, the basic machine has had a number of changes made to it. From a configuration perspective the basic machine at this point partially resembles a combination of the old and new configurations. Once at this point a number of transient considerations are applied to the entire system to ensure its consistency.

With the transient conditions having been analysed and the system declared to be consistent the remainder of the reconfiguration can proceed. As the basic machine moves along the reconfiguration path it becomes subject to the conditions which the state machine has identified as being necessary to support the new configuration. It is also during this time that other recovery routines are performed so as to allow the system to remain in a consistent state.

Eventually, over a finite period of time the process of reconfiguring the basic machine is completed and the finite state machine reverts back to maintaining the consistency of the system by executing recovery procedures.

[12]Transient conditions are a combination of conditions which reflect both the initial and final configuration.

3.4.7.2 Reconfiguration Possibilities

Lim's model provides support for three types of reconfiguration operations to be performed to a system. These operations include:

- Replacement of a Basic Machine
- Relocation of a Basic Machine
- Restructing of a Basic Machine

Replacement of a Basic Machine: The replacement of a basic machine allows developers to add extra functionality to the system or to correct erroneous code which may exist. This aids in the evolution of the system.

When replacing a basic machine it is important to note that there are a number of conditions that must be met in order to provide a seamless reconfiguration. To ensure this migration, the new basic machine to be installed must support the same behaviour as the old. This is similar to the COM/DCOM architecture (Brown and Kindel 1996) where once an interface is published it remains fixed. Additionally, when a new basic machine is placed into the system it must provide a mechanism to start at a similar position as that of the replaced machine.

As part of the construction and replacement process of a new basic machine, a state map must also be constructed. This map allows the state machine to map across the different states from the old basic machine to the new.

The overall replacement of a basic machine is performed in four steps.

1. Instruct the old basic machine to specify and store its current state
2. Instigate a reconfiguration transition responsible for issuing the destruction of the old basic machine
3. Instantiate the new basic machine into the system where it awaits a state transfer and the order to start executing
4. Instantiate a reconfiguration transition which sends the state of the previous machine to the new machine and instructs it to commence its execution

Relocation of a Basic Machine: The steps undertaken when relocating a basic machine are very similar to those performed during a replacement process except that with relocations extra attention must be paid to the transfer of state data from one location to another. To start the relocation process, the finite state machine must first determine the new location of the replacement basic machine and then determine whether the state data held within the basic machine to be relocated needs to be preserved or transferred to the new location.

If a basic machine is to be relocated from one processor to another (be it on the same machine or a different one) then consideration needs to be given to the movement of state information across processor boundaries. Depending on the location a transformation process may be required to marshall the data and transmit it to the destination. This process does not need to be considered when replacing a basic machine.

One advantage, with the relocation of a basic machine is that there is no change of state which means that the reconfiguration can be performed at any time.

The only event which requires careful examination is when a relocation is required as a result of a processor failure. In such circumstance, the basic machine and its related state must be transferred. Additionally in such situations there might be a need for recovery conditions to be applied to help retain and possibly regain the consistency of the system.

Restructuring the Application: The process of restructuring an application involves the addition and removal of basic machines. As a consequence of such a restructure there is no need to preserve the state of any old basic machines which are no longer needed. New basic machines which are to be added to the system commence operations from their initial starting point and hence require no state information to be transferred to them.

The only operations which might need to be performed to the overall system while restructuring is taking place are those required for recovery. These routines are used to ensure the system remains consistent while other basic machines are removed.

3.4.7.3 Finite State Machines as a Configuration Management Tool

In order to address problems which have been experienced by traditional transaction based configuration management systems, the concept of finite state machines has been applied.

By making use of finite state machines it is possible for the overall system to provide support for recovery management through the use of recovery paths, comprehensive deadlock analysis (from the examination of the various states in the system) and dynamic reconfiguration management. This functionality is provided through the examination of the various states and the calculation of configuration changes required to move from one state to another.

Finite state machines are capable of providing software developers with the extra flexibility required to commence system reconfigurations without the need for the entire system to enter a quiescent state.

A problem that Lim's finite state model did not address was the behaviour exhibited by the model when method calls or transactions are sent to basic machines that are not available. Although the finite state model does include a comprehensive consistency manager, it is still unable to detect the sporadic nature of method calls being sent amongst basic machines. It is the handling of these

method calls that the model fails to address. From the formal definition provided in Lim (1993) it is impossible to know if method calls sent to these basic machines return with an appropriate error condition or whether they are blocked until the basic machine returns to service. This lack of information impacts on software developers and end-users operating within real-time environments where the success or failure of a method call needs to be known straight away.

It should be noted that the finite state machine approach to reconfiguration provides a comprehensive solution to identifying possible reconfiguration paths that can be taken to safely reconfigure a system when compared to others. After applying the configuration management criteria to the finite state machine model it is possible to see that its strength lies in its dependency analysis. Additionally, the model provides developers with the functionality of being able to manage components and their inter-connections.

3.5 Runtime Component Configuration & Consistency

Throughout this section those configuration management systems which provide software developers with a great deal of configuration flexibility are examined. Each of the systems detailed provide the ability to change the way in which components are operating as well as allowing the manipulation of inter-component relationships while the system is still executing. These configuration management systems do not require the complete system to come to a halt.

3.5.1 SOFA/DCUP: Dynamic Architectures and Component Trading

The SOFA/DCUP architecture described in Plasil, Balek, and Janecek (1998) aims to provide an architecture which allows for the creation of dynamic architectures and provides for dynamically updating components within the system. Its design has been motivated by the need to provide support for future software applications which will comprise a number of reusable components.

SOFA/DCUP aims to provide a framework necessary for the customisation of components to specific requirements as well as providing the appropriate infrastructure for allowing components to be interconnected to one another.

To provide such a framework the SOFA/DCUP architecture relies on a considerable amount of research performed in literature such as Magee, Kramer, and Sloman (1989) and Magee, Dulay, and Kramer (1994) which details the area of interconnecting software components to form functional groups as well as investigating the design of interfaces representing components. It is this interface that provides an abstracted viewpoint of the component and aids in the dynamic reconfiguration of systems.

The process of interface design has primarily been concentrated on the construction of configuration definition languages. These languages are designed to allow the interface of a component to be

defined in the terms of how it interacts with others. Early configuration languages were known as Module Interconnection Languages (MIL's) but have now matured into more sophisticated languages such as those used by the CORBA and DCOM architectures (Exton, Watkins, and Thompson 1997) and the configuration language DARWIN (Dulay 1992) used in the REGIS system.

With most configuration management systems the concept of dynamic reconfiguration is implemented through the disconnection of the component from the system. Unfortunately this introduces two significant problems. These are dealing with a new component which does not provide backward support for the old interface and having to deal with dangling object references scattered throughout the system pointing to the old component after it has been removed.

The SOFA and DCUP architectures address the problem of dangling references by introducing an architecture which does not directly remove the components from the system but performs an in-line update to an already existing component.

3.5.1.1 SOFA Architecture Overview

The SOFA architecture is primarily responsible for the construction of components. It is when all of these components are grouped together and interconnected that an application is created. Each component using SOFA has its interface specified in a specialised component definition language.

The language allows software developers to specify the interconnections between the different components in the system and the various interfaces. These interfaces describe the services that the component *requires* or *provides*. If a component requires a service then the direction of data flow will be into the component while a component providing a service will have an opposite data flow.

When these components have been created they operate within an environment known as 'SOFAnet' which is part of the SOFA/DCUP architecture. A SOFAnet environment can be represented by a Directed Acylic Graph (DAG) containing a list of how each SOFAnode is interconnected with each other.

SOFAnet also provides a self contained environment for components while providing a series of interfaces which the SOFAnet environment can use to communicate with the various SOFAnodes. In addition to providing interfaces, the SOFAnet environment provides a meta-component known as the template repository. This repository has been specifically designed to act as a template reference to all other nodes within the network. As new templates are added to the system they are registered with the repository so as to allow other SOFAnodes to use them.

Associated with each SOFAnet is the *RUN* environment. This provides developers with the flexibility to provide extra parameters to the execution environment. These parameters can be used to control the launching of applications or aid in the construction or creation of templates at runtime.

3.5.1.2 DCUP Architecture

The DCUP architecture is responsible for providing the dynamic reconfiguration extensions to the SOFA architecture. Additionally, DCUP addresses the problem of maintaining object references in a system which is dynamically changing its components.

DCUP addresses this problem by dividing a component into two parts. One part remains statically attached to the system and is responsible for handling the configuration issues while the other part is dynamic and is responsible for integrating the changing implementations[13] of a component into the system. Having part of a component attached to the system permanently ensures that the configuration manager will have a constant object reference for the entire lifetime of the component and will allow clients to always be able to reach it at its published object reference.

Within each DCUP component there are two interfaces. The first is known as the *control* interface and is responsible for coordinating a consistent approach to the management of all DCUP components. The second interface is known as the *functional* interface and is responsible for linking together the components interface to that described in the SOFA architecture.

In addition to providing two interfaces, the DCUP architecture introduces two new implementation objects to each component. These objects are known as the *Component Manager* (`CManager`) and the *Component Builder* (`CBuilder`).

The `CManager` object is responsible for the core component of the permanent section. As the `CManager` resides in the permanent section, it is unaffected by any reconfiguration changes. The principle role of the object is to coordinate the updates to the inner component.

The `CBuilder` object is located within the dynamic part of the component and is responsible for overseeing the integration and removal of various component implementations from the system. The lifetime of the object is governed by how long the particular implementation of the component is in use.

In addition to the construction and destruction of a component, the `CBuilder` object is responsible for providing serialisation operations. This becomes necessary when the component needs to preserve or change its state.

The DCUP architecture also provides an `Updater` component which monitors all requests for updates throughout the system. When an update is requested, the `Updater` component examines the update request and transmits the request to the corresponding `CManager`.

[13] An implementation of a component may change a number of times throughout the lifetime of the system.

Figure 3.5 illustrates both the `CManager` and `CBuilder` components which are responsible for coordinating reconfiguration activities within a SOFA/DCUP component.

Figure 3.5: Overview of the SOFA/DCUP Architecture

3.5.1.3 Interconnecting DCUP Components

Components in the SOFA/DCUP architecture use a similar approach to that used in REGIS (Magee, Dulay, and Kramer 1994) where components exist within an environment with specific bindings established between them. SOFA/DCUP components also have structured interfaces similar to the DARWIN configuration definition language allowing developers to express whether an interface `provides` or `requires` a service.

For components to be able to provide services to the rest of the system, the component providing the service must explicitly export the service. Once exported, components requiring the service can execute the API call `bindtoService(...)` to have a reference pointer returned to them.

The process of obtaining services is slightly more complex. In DCUP/SOFA it is the outer most layer which is responsible for providing the references to the required service. As a result, the inner component provides its requirements through the API call `getRequirements(...)`. It then becomes the `CManager`'s responsibility to obtain the appropriate references. Once the references have been obtained by the outer layer they are sent to the inner layer through the `provideRequirements(...)` API call.

3.5.1.4 Updating a SOFA/DCUP Component

Updating a component under SOFA/DCUP is addressed through the underlying infrastructure defined in section 3.5.1.1. This enables developers to perform an update on a component by using a method call within the `CBuilder` object.

The overall process of updating a component involves sending the method call `onLeaving(...)` to the old component and then capturing the internal state of the component. Once the state has been captured, the new version of the component is created and the `onArrival(...)` method call is sent to the component along with the state data. This operational sequence is performed by the `CManager` through an `updateComponent(...)` subroutine.

Figure 3.6 illustrates the process that a SOFA/DCUP component performs when a component is being upgraded. Specifically, the figure highlights how the `onLeaving(...)` and `onArrival(...)` functions are sequenced within the `updateComponent(...)` method call.

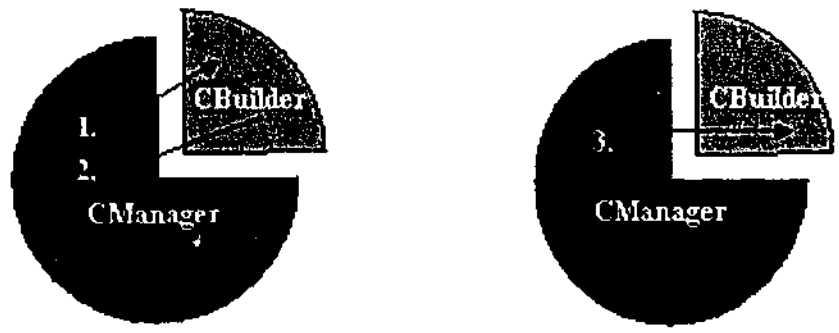


Figure 3.6: Overview of upgrading a SOFA/DCUP Component

3.5.1.5 SOFA/DCUP as a Configuration Management Tool

The SOFA/DCUP architecture was designed to provide developers with an environment in which components could be dynamically updated without a need to shut the system down. Additionally, the system encourages the development of applications through the use of inter-connectable components providing the basis for components to be interchanged at runtime.

The process of dynamic reconfiguration is made available through a component definition language which separates the computational components from the interface definitions. In addition to this language, each component is represented on two levels. One level provides the outer-component functionality to the system and is responsible for managing the component for its entire life-time within the system. The second is the inner-layer providing the functionality of the component and is dynamic in nature as it can be replaced at any time.

However SOFA/DCUP lacks the appropriate components required to provide consistency checks and comprehensive deadlock analysis. The `CManager` and `Updater` components provide only limited support for the actual updating of components and deal more with the issue of notifying the component of the change rather than calculating the ramifications of the update.

The overall problem with the approach taken by SOFA/DCUP is that individual components seem to have the power to update themselves without much consideration being given to the consistency of the system. Although this approach does provide a solution to the dynamic reconfiguration problem, Goudarzi (1999) states that no individual component is ever in a position to determine when it is in a safe state to update or replace a component. This action is normally coordinated from a configuration manager responsible for performing the dependency and consistency checks for the entire system and required to guarantee the safety of the system. The problem with the `CManager` in this architecture is its close relationship to the implementation and hence can not possibly be in the correct state to control configurations.

From a configuration management perspective the SOFA/DCUP system provides developers with a system which is component aware (refer to section 3.1) and allows them to control the reconfiguration of components within the system.

3.5.2 DynamicTAO

The DynamicTAO system, described in Román, Kon, and Campbell (1999), states that over the last decade there has been a growing importance on the use of middleware to aid the development of large systems. The DynamicTAO system has been designed to provide more flexibility to those developers who rely on the use of middleware and required ability to dynamically reconfigure that middleware while the system is operating. The DynamicTAO architecture extends the TAO ORB discussed in Schmidt, Levine, and Cleelan (1999).

Applications which make use of middleware infrastructures suffer from having a static middleware layer. Although applications themselves have been designed to take into consideration the ever changing computing environment, middleware responsible for the coordination of these applications does not. This means that as the environment changes, the applications implemented below the layers of middleware are unable to realise their full potential. This can lead to the inefficient use of resources as these applications designed to take advantage of changing conditions may have to continue using legacy functions as the middleware is unable to support anything else.

To overcome this, DynamicTAO allows developers to reconfigure the internal middleware while the system is running. In achieving this DynamicTAO makes use of reflective techniques to provide dynamic reconfiguration support. In addition to making use of reflective techniques, DynamicTAO allows developers to pick appropriate strategies for their applications.

3.5.2.1 D_ aicTAO Architecture

As mentioned, DynamicTAO provides reconfiguration services through a process known as *reflection*. This process is achieved through a collection of entities known as component configurators. Each

component configurator is responsible for containing the dependency information between the current component and the remainder of the system. These component configurators play an important role when the consistency of the system is being analysed.

Each process that runs a DynamicTAO component has a `DomainConfigurator` that is responsible for maintaining all of the references to those instances known to the ORB and its servants. Additionally, each component instance of DynamicTAO contains a `TAOConfigurator`.

These configurator modules are responsible for providing hooks or entry points into the running system allowing developers to determine what strategies are currently in use as well as allowing them to be modified. Each strategy comes in the form of a dynamic library which can be linked into the system while still running.

To provide the ability of this extra support for dynamic reconfiguration, three new major components were added to the original TAO architecture. These components include:

- Network Broker: Responsible for receiving reconfiguration requests from the network and forwarding requests onto the appropriate components.
- Persistent Repository: Responsible for storing all of the category information on different implementations available to the DynamicTAO system and providing methods for developers to be able to manipulate these implementations. Operations that are supported include the creation, modification and deletion of implementations.
- Dynamic Service Configurator: Contained within this component is the `DomainConfigurator` which in itself is responsible for providing operations to allow the reconfiguration of the system at runtime. Additionally, the `DomainConfigurator` delegates some of its responsibilities for specific components back to the component configurators.

The structuring of this architecture allows for separate components to be inserted into the system and for different network brokers and persistent repositories to be developed and used with the dynamic configurator service.

3.5.2.2 Dynamic Reconfiguration

As already referred to, the underlying operations for dynamic reconfiguration are provided through the dynamic service configurator. There are two ways in which developers can issue instructions to reconfigure the system. One is through a well defined CORBA compliant interface which specifies all of the operations that can be performed by the dynamic service configurator.

The other approach is to make use of a *network broker*. This service is provided so developers can make dynamic connections to the system and architect changes. Additionally, some reconfiguration

operations which are available to developers are too hard to express in terms of a CORBA IDL and so are only provided and implemented via the network broker.

Some of the operations available from the dynamic service configurator include:

- `load_implementation`
- `hook_implementation`
- `upload_implementation`
- `delete_implementation`

In addition, there are a number of other operations which are responsible for providing feedback information on the systems performance as well as management routines that allow strategies to be suspended and resumed.

3.5.2.3 ORB Consistency

As discussed in the last two sections, it can be seen that where there is a need for dynamic reconfiguration there is also a need for infrastructure to be available to support the consistency of the system. The DynamicTAO system is no different.

The DynamicTAO system approaches the issue of consistency though minimisation. At the center of the system is a small core which is responsible for maintaining only the most critical functions of the system. By adopting this approach the core allows itself to remain relatively free from all of the configuration work which is done.

The process of change, especially dynamic change, requires a considerable amount of fore-thought and planning in areas such as keeping the overall system consistent and deadlock analysis. Before any change is implemented in the DynamicTAO system a series of dependency checks are performed across the entire system to determine whether any new dynamic libraries that may be introduced into the system will conflict with any others that are already loaded.

Additionally, in a system such as DynamicTAO which deals with other ORB's, dependency checks need to be performed to make sure that changes performed to one ORB will not effect communications with other ORB's. In the event that such reconfigurations do have wider ramifications, other alternative actions may be considered including the deferment of the reconfiguration or coordinating an upgrade of the affected components on the other ORB at the same time.

After performing the appropriate analysis, the correct state of a new strategy must be determined. In some cases the initial state of a new strategy often depends on the final state of the previous strategy. Planning is essential to ensure that the consistency of the system is maintained between reconfigurations.

Once the correct state has been determined, the underlying architecture of DynamicTAO provides what is necessary for the state to be preserved or transferred to another strategy. To ensure that the most up-to-date strategy is used, each strategy is given a unique version number. This versioning approach allows DynamicTAO to select the most recent strategy available even if there are two or more strategies working concurrently in the same section of the ORB.

3.5.2.4 Automatic Reconfiguration

As addressed earlier, most applications are developed to take advantage of the changing conditions, but typically middleware lags behind due to its firm foundations. With the reflective nature of DynamicTAO the central middleware itself is now able to adapt to the changing circumstances as well. This leads to systems which can *self-heal* and adapt to take advantage of the environment.

Román, Kon, and Campbell (1999) identified three key areas which have benefited greatly from the introduction of automatic reconfiguration.

Optimisation: Through the use of automatic reconfiguration, the components running within a system can be significantly optimised. This is achieved by analysing the environment in which the system is running and applying the knowledge gained through a set of heuristics to the configuration. Once implemented, the system can maximise the use of its environment.

Customisation: Customisation provides for a system to be changed to meet the ever changing needs of business. Permitting change, the system can allow for the incorporation of new business practices without the need to shutdown the entire system and bringing the business to a halt.

Error Recovery: Due to the system continuously monitoring itself, it is capable of determining erroneous components. Once detected, the system can identify and carry out the necessary steps to repair the affected components dynamically. Adopting this approach reduces the amount of downtime that the system would otherwise experience during its lifetime.

3.5.2.5 DynamicTAO as a Configuration Management Tool

The DynamicTAO system presents a new approach to the concept of dynamic reconfiguration management of middleware. The architecture has been specifically designed as a *wrapper* allowing developers to control the internals of the TAO ORB's middleware.

It provides developers with the flexibility to customise the middleware, or in some cases have the middleware reconfigure itself, upon detection of a set of changing circumstances.

In addition to providing the underlying architecture for dynamic reconfiguration management, attention has been given within the confines of the system to check the consistency of the system when a new component is being planned or introduced.

However despite its advances with dynamic reconfiguration management, the system provides no support for deadlock analysis and appears to lack detail with regard to supporting method calls which have been made on components currently being reconfigured. From the overall architecture of the system, DynamicTAO provides a considerable amount of functionality expected within a configuration management system. This functionality however is geared more towards the internal management of the middleware. The architecture nonetheless does provide a form of dynamic reconfiguration even if it is only the ability to reconfigure the middleware which governs the operation of the system. The manipulation of middleware allows developers to reconfigure the manner in which memory is allocated, the way that tasks are queued and new functional components introduced to handle the internal operational details of the system.

Apart from handling the system internals, the architecture provides partial support via the use of version numbers to identify components used within the middleware. The system also supports component awareness through its ability to activate automatic reconfigurations in order to *heal* components which may have been damaged. DynamicTAO also provides developers with an abstracted environment allowing access to the underlying low level details of the system such as the threading or memory management model being used.

3.5.3 Preserving Consistency with REGIS

It can be seen from the systems already detailed that a considerable amount of effort has been placed on the construction of systems which allow components to be dynamically reconfigured. However in each system described it is possible to identify that each has its own determination of what constitutes dynamic reconfiguration management.

In Goudarzi (1999) it is suggested that there are very few systems which provide developers with a framework for dynamic reconfiguration management. This statement may well be motivated by the fact that there are very few systems which provide the total solution.

To overcome the problem, a new system outlined in Goudarzi (1999) is proposed which builds upon the work already carried out into dynamic reconfiguration by the Distributed Computing Group at Imperial College, UK and their REGIS project (Magee, Dulay, and Kramer 1994). The proposed system aims to provide a framework allowing for the development of re-configurable components while at the same time including support within them so that they can be managed throughout their entire lifetime.

In addition to providing lifetime support for the management of the components, the framework has a special emphasis on preserving the consistency of the system and minimising interruptions to the remainder of the system. This was identified in Goudarzi (1999) as one of the most common areas overlooked by other systems but one which is critical to the smoothness of reconfiguration operations.

The extensions to the REGIS architecture were proposed as a result of an examination conducted into REGIS's ability to support dynamic reconfiguration management and to preserve the consistency of components while performing reconfiguration operations. The conclusions from the review highlighted the need for extra support to be given to the dynamic reconfiguration module and for the introduction of a system capable of preserving consistency during a dynamic reconfiguration.

The extensions to the system took the form of four additional entities known as the *Reconfiguration Manager*, *Configuration Database*, *Consistency Manager* and the *Event Composition Service*. The role of these managers and the architecture used for these extensions is explained in section 3.5.3.1.

An inherent benefit of extending the REGIS system is the flexibility of being able to make use of the underlying model while at the same time being able to provide individual enhancements. This means that the extensions allow for the separation of the configuration aspects from behavioural ones.

Some of the enhancements include extending the configuration programming paradigm detailed in Kramer and Magee (1990) and allowing the original configuration descriptions to be merged with the reconfiguration behaviour. Additionally, the extensions further promote the idea supported in the reconfiguration model designed by Kramer for components to be placed into a *safe* or *quiescent* state, allowing a component to be modified without causing any significant system disturbances.

3.5.3.1 Architecture behind the Extensions to REGIS

Throughout the development of the REGIS extensions, great care was taken to ensure the extensions would communicate with one another and interface with the client and underlying REGIS architecture. It was decided that the connection oriented approach (ie. components have a direct communication channel to one another) would be used to provide the distributed runtime reconfiguration management facilities. The use of the connection oriented approach ensures compatibility with most systems.

The extra dynamic reconfiguration management functionality is coordinated through the addition of four new managers to the REGIS architecture. A detailed explanation of these managers can be found below. In addition to the managers, REGIS makes use of specialised hooks to enable the dynamic reconfiguration manager to control the state invariants contained within the individual components.

Component Managers

Throughout this section each of the new component managers introduced into the REGIS environment will be examined together with the role that the manager plays in keeping the system consistent and providing the dynamic reconfiguration functionality.

Reconfiguration Manager (RM)

The reconfiguration manager plays a pivotal role in the reconfiguration of components. It is the component responsible for coordinating all of the reconfiguration activities throughout the system.

The involvement of the reconfiguration manager starts when it receives an event message either generated by the *Event Composition Service* (ECS), described later on in this section or from another event generating source such as a change in the outside environment where the system is operating.

With the events being received, the manager examines the type of message as well as the content and with the aid of change designers (programmable filters) determines which reconfiguration script should be executed. Before any of the reconfiguration scripts selected by the manager are executed, the manager must first ensure that the proposed changes or actions to be made to the system will not result in any component being placed into an inconsistent state nor will the configuration process interfere with any other reconfiguring operations.

Due to the constant need to ensure the consistency of the entire system, the reconfiguration manager maintains a close relationship with the data held within the *Configuration Database* (CDB) and the *Consistency Manager* (CM). To further highlight this relationship, when an event is detected which results in a reconfiguration of the system, the reconfiguration manager will contact the configuration database in an attempt to get the most accurate representation of the system.

Additionally, the information contained within the CDB helps the reconfiguration manager select the most appropriate configuration script. Once the script is selected, both the CDB and CM are consulted to determine what impacts and consequences these operations will have on the system. After the consistency checks are performed and the deadlock analysis complete, the reconfiguration manager instructs the consistency manager to make the appropriate changes and authorises the scripts to be performed on the basis that there are no errors preventing them from being executed successfully. While updating the system, relevant changes are made to the CDB in parallel to ensure that the CDB and the system are consistent with one another.

The reconfiguration scripts used throughout the system actually perform two separate roles. The first is to make the appropriate changes to each of the components, while the second and more important role of the reconfiguration script is to preserve, set or reset the invariant data which is being held within each component.

By preserving the internal invariants within a component, the reconfiguration manager is capable of resuming or starting a component from the point at which the reconfiguration of the system was started[14].

With the REGIS extensions taking care of the mechanisms behind the manipulation of the system and application invariants, the software developer only has to include code which provides access to

[14]In some cases it is not possible to restart the component at the same exact location, so the change designer must provide an entry point which is directly translatable from the previous component.

the entry point, or hooks to the invariants contained within the components sub-system.

Consistency Manager (CM)

The consistency manager is responsible for ensuring that all components within the system remain in a consistent state for the duration of their lifetime. Consistency is even preserved during reconfiguration periods where under certain circumstances the invariants within a component may become invalid. All such occurrences are carefully controlled and reset at the end of the reconfiguration.

The consistency manager is contacted after the reconfiguration manager has determined that there is a need for a change and when the appropriate reconfiguration script has been selected. Upon receiving the changes, the consistency manager examines the implications of the script and determines whether the change can proceed smoothly without disrupting other components or reconfiguration actions currently happening throughout the system.

Once a reconfiguration path has been achieved, the consistency manager sends a message back to the reconfiguration manager signalling that it can proceed with the execution of the reconfiguration scripts.

To ensure consistency within a system at all times, the consistency manager must adopt an approach which allows for the detection and handling of inconsistencies. To facilitate this detection there are two approaches available. These are: *recovery* and *avoidance*. From the relationship between the consistency manager and the components within the system it is possible to see that there is a great deal of interaction between the two. The extent of this interaction is governed by the avoidance scheme used.

Recovery Approach to Consistency

The recovery approach to consistency focuses on providing an underlying framework allowing for the identification of those transactions which might be affected by a pending transaction. In such circumstances an exception mechanism could be used to interrupt the current transaction and to allow the reconfiguration to proceed. Once completed, the transaction could be restarted at the same point if support for check-pointing, rollback or setting system invariants was in place.

However, the recovery approach to consistency management could be made redundant if the system which the REGiS extensions are operating within provides support for an exception model. If such support within a framework existed then a series of special hooks could allow for the manipulation of the invariants located within the component.

Avoidance Approach to Consistency

Instead of approaching the consistency problem from a heavy handed perspective like the recovery approach, the avoidance approach takes the more serene view of placing components into a consistent

state. The approach works by having the system identify those components that would be affected by a reconfiguration and having them moved into a *safe state* which isolates them from the rest of the system. This minimises the disruption to the remainder of the system.

In order to effect such a change on a component, the consistency manager requires the ability to be able to determine the current configuration of the system[15] and make use of the special management hooks located within each component to manage its invariants.

During the process of reconfiguration each of the components affected in the system is handled by the use of a specialised management interface. As the reconfiguration routines proceed the configuration database is updated to reflect the changes made.

Configuration Database (CDB)

The configuration database is responsible for containing up-to-date information on the configuration of the system. As each reconfiguration operation is performed, whether it be a `link`, `unlink`, `create` or `remove` operation, the appropriate record in the database is updated.

Additional to the reconfiguration operations, records within the database are updated whenever the reconfiguration system makes use of the specialised hooks to manipulate the state of a components invariants.

The CDB is one of the most important components in the extensions to the REGIS architecture as its information is in constant demand from software developers and other manager modules which need to know the current configuration. The records contained within the database assist developers when a system change is planned. In order to access this data, the database provides a set of interfaces which enables software developers or automated reconfiguration agents to obtain the information they need and make the appropriate decisions from it.

To ensure that the data within the database is as current as it can be, all modifications to the system are centrally coordinated through the configuration database. This linkage is demonstrated by the reconfiguration manager which consults the configuration database everytime it receives an event. Additionally, the reconfiguration manager uses the data that it obtains to determine which reconfiguration script should be selected or with the help of the consistency manager determine what the implications are for applying a certain set of reconfiguration actions to the system.

From the interfaces provided by the database it is possible to extract all of the structural information contained within the system. The data recorded within the configuration database includes a unique identifier representing the component and a value indicating whether the component in question is a primitive one (ie. it is a basic building block) or a composite (ie. there are other components contained within it). The interfaces within the database are also capable of listing all the bindings that a specific component is connected to. The database even contains the host that the component is currently active on.

[15]This is achieved through the use of a configuration database such as the one described in Goudarzi (1999).

By forcing all reconfiguration operations to be performed through the database, it is possible to ensure that the database maintains the most accurate description of the system configuration. This can be very important when there are multiple reconfigurations being performed within the system.

As the database is central to all reconfiguration operations, it is necessary for it to be able to support concurrent updates. These updates to the database minimise disruption to the system and to other components. The actual routines responsible for providing the concurrency within the database are embedded within the database management system.

Additionally, the database management system includes support for other concurrency strategies used while updating the system. These strategies include reconfiguring a system with a single reconfiguration script or reconfiguring a system consisting of multiple reconfigurations.

Reconfiguring with a single script

As the name suggests, this approach involves reconfiguring a system with the aid of one reconfiguration script. Upon the execution of the script, all of the reconfiguration operations are identified and grouped together.

Those operations which can be performed in a concurrent manner are processed in such a manner, while those routines requiring sequential processing are ordered and executed in the specified order. Both the reconfiguration manager and the configuration database take great care to make sure that any reconfiguration operation performed in a concurrent manner does not conflict nor introduce a state of inconsistency to any other component.

Performing multiple reconfigurations

This form of reconfiguration involves making changes to the system through a number of reconfiguration scripts. Similar to the method by which a single reconfiguration script is performed, all of the reconfiguration scripts are read into memory and then analysed. Those routines that can be executed in parallel are performed in parallel while those requiring sequential processing are once again executed in that particular sequence.

When using this method, the configuration database and the concurrency system take great care to preserve the consistency of the system and ensure that only one reconfiguration action is being performed at any one time on any one component. This eliminates the chance of one reconfiguration action interfering with another.

The configuration database is then updated once a transaction has been successfully committed otherwise there is a potential risk of the database containing invalid data and reflecting an inconsistent configuration.

Event Composition Service (ECS)

The Event Composition Service (ECS) serves as a gateway for all incoming events. Events may come

from components located throughout the system or from outside influences including the result from an interface being used or some other external circumstance such as a server failing.

Although the ECS receives a number of events from various sources, the system is only responsible for analysing the implications of configuration events. These events are normally triggered by one or more configuration/reconfiguration events. These events include: `create`, `remove`, `link` or `unlink`. If such reconfiguration events are detected, the ECS will pass the reconfiguration event and the component that raised the event to the reconfiguration manager for further analysis.

The identification of change in the outside environment is harder to determine in some cases as the system has to give significant consideration to events arriving over sometimes a random and extended period of time. It is at the ECS that these events are checked and cross referenced with a list of events contained within a configuration database and maintained by a system development team. If the event is deemed to be configuration related then the event and a handle to the component which raised the event are passed to the reconfiguration manager.

One of the differences that has been adopted by this system and makes it different from other configuration management systems is the decision to separate the specification of a component from the evaluation of the changing conditions. By shifting the evaluation mechanism away from the component, developers can focus on the task of creating components while allowing the ECS to deal with the triggering of reconfiguration events.

This is achieved by having the reconfiguration language only provide support for defining how components are to be reconfigured giving developers more flexibility in creating or modifying a component as well promoting a much simpler language for developers to use.

3.5.3.2 Reconfiguration Programming

The concept of *reconfiguration programming* was inspired by the configuration programming paradigm first described in Kramer and Magee (1990). The configuration paradigm allows a designer to be able to describe all of the *initial* components within the system as well as their *bindings*.

The reconfiguration programming paradigm extends this notion by allowing designers to specify what comprises the initial system (in terms of the components and their bindings) as well as providing support to continually update the system. This occurs by matching a type-description specification located within a reconfiguration language with the computational component.

Once defined, these type-description instances can be manipulated to reflect changes to the system structure through an event driven interface. As a type-description instance is manipulated the corresponding computational component is also altered to reflect the changed state. Of course, as these changes are made they are subject to the rules and limitations expressed within the reconfiguration and consistency managers.

As mentioned, when a reconfiguration occurs, both the computational and configuration components are changed concurrently in their respective runtime systems. This allows the changes to occur simultaneously to the system and ensure that the configuration database is kept up-to-date.

Primitive State-Access Methods

In order to provide the dynamic reconfiguration facilities, each computational component must provide a set of access methods which allow external agents to query and modify the internal invariants. The scope of these interfaces is determined by software developers.

In most cases the interface provides `get(...)` and `set(...)` routines. These operations are responsible for maintaining structures within a component that control the behaviour specification and synchronisation controls of an individual component. The actual manipulation of these routines and management hooks is coordinated by the reference manager.

Primitive Configuration Descriptions

In addition to containing information such as the communication interface, instantiation parameters and the configuration specification, the reconfiguration paradigm records the signatures of the method calls.

These signatures allow software developers to incorporate these functions within reconfiguration scripts and hence enable developers to actively restore the system invariants.

As reconfiguration scripts can alter the invariants of a component, great care must be taken by the reconfiguration subsystem that the appropriate data types are selected and that they are passed into the component correctly. To address this issue the reconfiguration manager performs type checking and data marshalling/unmarshalling routines where appropriate.

Composite Configuration Descriptions

Composite configuration descriptions embody everything already mentioned in the primitive configuration description as well as providing support for two new method types. These method types provide reconfiguration methods and composite state access methods.

Reconfiguration Methods

Reconfiguration methods are responsible for altering the internal configuration of a composite. They normally include management routines to dynamically adjust internal values which are directly related to the configuration. An example of such a routine may be to increase or decrease the size of a memory buffer or thread pool.

Such reconfiguration methods can be accessed via one of two ways. The first is to make use of an event based approach. This is normally the interface that reconfiguration components like the ECS will use to affect change on a component. This approach also allows for the loading of dynamically linked libraries.

The second approach is to make use of a procedural entry point. This approach is taken when a component needs to perform an operation. The use of this interface is not normally encouraged and all reconfiguration operations are directed towards the reconfiguration manager.

Composite State Access Methods

Composite state access methods are similar to other access methods except the information provided from these routines relates to a composite component rather than a primitive one. This means that the return values from certain method calls performed upon a composite object might actually be constructed by a cascading method call down all of the composite components.

Those access methods which are used on a composite component are not permitted to change the internal configuration or structure of a component. All of the reconfigurations performed within the system are referred to the reconfiguration manager so that the appropriate changes can be recorded and made throughout the system.

3.5.3.3 Guaranteeing Mutual Consistency

One of the important concerns with the extensions to the REGIS system was the ability to provide mutual consistency throughout the entire system. To ensure that the consistency approach was going to be useable, a conscious decision was made not to follow the same decisions used in other systems. In the past, system such as Lim (1993) have used recovery systems which have placed a considerable overhead on the system.

To overcome the problem, extensions to REGIS make use of an avoidance scheme that reduces the load on the overall system while a reconfiguration is being performed. Additionally, this approach was selected due to the specific requirement of minimising the disruption to the system. The avoidance scheme is careful not to overburden software developers with extra code which must be placed in components to facilitate reconfiguration.

Safe-State for Reconfiguration

With most systems, the driving force behind any distributed component based application is the components themselves. During normal operation, these components are inter-linked with one another so as to be able to pass data. As the components are performing their roles, they are said to be in a *consistent* state.

However, great planning must be undertaken in order for a reconfiguration of the system to occur without posing any threat to the bindings between the components or introducing an inconsistent state into the system. To assist with this planning a set of reconfiguration rules were used to ensure that certain actions can only be performed after other actions have completed. These rules are

explained in detail in Kramer and Magee (1990) and Goudarzi (1999).

Achieving the Safe State

In order for a reconfiguration to occur within a system, it is necessary for those components involved to move into a safe state. Apart from moving the components into a safe state comes a problem of what to do with method calls made on components involved with the reconfiguration.

The architecture addresses this by having the method calls blocked when the corresponding component is undergoing a reconfiguration. The method calls are then unblocked on the completion of the reconfiguration. The use of the blocking and unblocking approach to handle component reconfiguration is explained in Goudarzi (1999) as is the criteria used to define what suitable approaches there are for a component to enter a safe state.

When moving a series of components into a safe state there are two blocking techniques that the transition can use. The first involves the identification of how a safe state can be applied to all components within the system while the second approach refines the first by determining how a safe state can be brought about on only those components which are effected by the reconfiguration request.

Blocking States

Below is a brief overview of both blocking algorithms which can be used to identify and isolate components which are to undergo a reconfiguration. The extensions to REGIS mentioned in Goudarzi (1999) make use of the optimistic blocking algorithm.

Pessimistic Blocking Algorithm

The pessimistic blocking approach is basically designed to bring all components within a system to a safe state irrespective of their involvement in the reconfiguration.

The algorithm works by sending a blocking message to all components registered within the system. Those components which are already in an idle state move into a blocked state, while components busy with transactions continue to process as normal. Upon the conclusion of the transaction, the component involved is placed into an idle state.

In some cases, a component may require the services of another component which has already moved into a blocked state. In these circumstances a component can move from a blocked state back into an idle state for the duration of the transaction. At the end of the transaction, the component moves back into a blocked state.

All components remain in this state until the reconfiguration has taken place, at which time they are all instructed by the reconfiguration manager to unblock and restart processing transactions.

Optimistic Blocking Algorithm

The optimistic blocking algorithm differs from the pessimistic approach by only concentrating on those components which are affected by the reconfiguration.

The algorithm sets out to build a set of nodes directly affected by the reconfiguration request. Once identified, these components are blocked. However, the system then goes on to examine whether there are any sub-transactions which may occur, resulting in other nodes to change state. Instead of allowing these nodes to change state, they are added into a new set which is known as the extended blocking set.

Once all the nodes in the primary blocking set are blocked, the nodes left in the extended blocking set are released. Changes are then made to those components located in the blocked set and upon successful reconfiguration are released back into the system.

3.5.3.4 REGIS Extensions as a Configuration Management Tool

From the description of the extensions to the REGIS system it is possible to see that the issue of dynamic reconfiguration manager has been addressed. This can be seen by referring to table 4.1 illustrating how the extensions to REGIS satisfy a considerable amount of the criterions detailed in section 3.1.

Table 4.1 also shows that the extensions to REGIS address such functionality as providing a type-checking system through the reconfiguration manager, the provision of routines which can be used to coordinate the inter-connection of components and their communication channels, a comprehensive dependency and consistency analysis system and the provision for introducing a state of quiescence over components which may be affected as a result of a reconfiguration.

However, even though the extensions provide the extra functionality for an abstracted environment, the system provides little support for the operator in respect to the status of a reconfiguration request made to the system.

Additionally, there is no support within the system for a method call to be rejected or dealt with appropriately if it can not satisfy a certain criteria. This lack of support makes it incredibly hard for real-time systems to operate effectively as each transaction requires a commit or no commit action before it will proceed.

3.6 Chapter Summary

Throughout the chapter a number of configuration management systems which have been developed over a period of time to address the concerns raised by the software engineering community have been examined.

Based on the functionality and support these systems provide to software developers it is possible to categorise the systems into four separate areas. Namely,

- Component Configuration Systems
- Static Component Configuration Management
- Dynamic Component Management
- Runtime Component Configuration & Consistency

3.6.1 Component Configuration Systems

From the systems which fit into this category it has been possible to determine that they provide support to software developers during the construction of software components. Traditionally, these systems provide support for multiple revisions of a component to be stored. In certain systems such as ICE, extended functionality exists to aid in the generation of components which can be configured for certain circumstances and environments.

3.6.2 Static Component Configuration Management

Static component configuration systems have been identified as those which provide support for the reconfiguration of components. However these systems lack the ability to be able to perform changes to the configuration of the system while it is still operating. In order for these systems to provide any reconfiguration support they must first be placed into a 'safe' or consistent state by shutting them down.

3.6.3 Dynamic Component Management

Dynamic component management systems provide the extra functionality required to manipulate the configuration of a system. The dynamic nature of these systems permits software developers to change the internal configurations and bindings between components while the system is still operating. These changes however, are made when the components are placed into a consistent state and are often performed on running systems which do not provide any comprehensive deadlock or consistency checking system.

3.6.4 Runtime Component Configuration & Consistency

Runtime component and consistency systems once again enhance those services provided by those systems grouped into the dynamic component management category. Systems grouped into the

runtime component configuration and consistency group provide the functionality required to reconfigure components during runtime while ensuring that the consistency of the system is preserved during reconfiguration.

This chapter conducted an examination of various configuration management systems and compared them with the criteria established in section 3.1 which represents the features expected in an ideal configuration manager. The following chapter provides a comparison of the features contained within the various configuration management systems and discusses the results with regard to the support given to components operating within real-time environments and the control that both the software developer and end-user have in circumstances where a method call is sent to a component being reconfigured.

Chapter 4

Real-Time Dynamic Component Reconfiguration

Throughout the previous chapter, a number of configuration management systems designed to facilitate the development of component oriented systems and provide support for the reconfiguration of components were introduced. As seen from the functionality provided by those configuration management systems introduced in chapter 3 and the varying levels of support provided for reconfiguring components, it is possible to classify configuration management systems based on their functionality and support.

Assisting the classification of configuration management systems was the criteria established in section 3.1. Making use of the criteria allowed for the identification of key features contained within systems which can be used as a basis for classification. When the criteria was applied to the configuration management systems introduced in chapter 3 it provided the basis for a short description on the suitability of each system with regard to the support provided for reconfiguring components, especially those operating within a real-time environment.

Chapter 4 concentrates briefly on the functionality provided by these configuration management systems but examines in more detail the classifications identified in the previous chapter. The chapter also focuses upon the issue of dynamically reconfiguring components within a real-time environment and why dealing with the real-time commitments of a component is so important. Additionally, the issue of justifying the need for supporting the dynamic reconfiguration of components in real-time environments within configuration management systems is discussed.

The chapter concludes by further examining the reconfiguration support provided by each of the configuration management categories and assessing how much control they provide for components

being reconfigured in a dynamic environment. As can be seen from the various architectures presented in chapter 3, little support is provided to enable software developers or end-users to control the execution of methods calls made upon components which are undertaking reconfiguration actions or are not available. This is especially relevant in real-time systems.

4.1 Configuration Manager Comparisons

The configuration management systems examined in chapter 3 reinforce the conclusion that each system brings its own unique functionality and perspective as to what a configuration management system should provide.

With each system providing its own interpretation of the functionality required to handle the management of components, it is impossible to directly compare reconfiguration environments. To address this problem and to provide a means of being able to compare the various configuration management systems, a criteria detailed in section 3.1 was developed. Using this criteria allows comparisons between the various environments based on the support provided for managing components rather than the systems underlying functionality. As shown in chapter 3, the application of the various criterions led to configuration management systems being grouped together based on common functionality.

Table 4.1 provides a list of the configuration management systems reviewed as well as the criterions that each system supports. A score which is the end result of assessing how well a particular system satisfies the overall criteria has been used to predetermine the order in which the configuration management systems appear in the table. The scoring system works by allocating each criterion a value representing its overall importance in the criteria. The values assigned to each criterion were equivalent to its own criterion number (ie. criterion eight has a weighted value of eight). With the individual weighted values assigned, the total score for a given system can be calculated by totalling each of the criterion points associated with it.

The results from table 4.1 illustrate how configuration managers which share their functionality and provide the same level of support for reconfiguring components can be grouped into the same category.

System \ Criterion	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
Distributed Revision Control System	✓	✗	✗	①	✗	✗	✗	✗	✗	✗	✗	✗
Distributed Concurrent Versioning System	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Incremental Configuration Engine	✓	✗	✓	✓	✗	✗	✗	✗	②	✗	✗	✗
Software Dock	✗	✗	✗	✗	✗	✓	✓	✗	✗	③	✓	✗
Adele	✓	✗	✗	①	④	✗	✗	✗	✓	⑤	✗	✗
Mistral	✓	✓	✗	①	④	✗	✗	✗	✓	⑤	✗	✗
Surgeon	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗
Polylith	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Programmers Playground	✗	✗	✗	✗	✓	✓	✓	✗	⑥	✓	✓	✗
CONIC	✗	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗
Equus	✗	✗	✗	✗	⑦	⑧	✓	✗	✗	✓	✓	✗
REGIS	✗	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓	✗
CHORUS	✗	✗	✗	✗	⑦	✗	✓	✗	✗	✓	✓	✗
Finite State Machine	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✓	✗
SOFA/DCUP	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗
Dynamic TAO	✗	✗	✗	①	✗	✓	✓	✗	✗	✓	✓	✗
REGIS Extensions	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗

Table 4.1: Comparison of Configuration Management Systems

①: The system provides support for revision histories but does not allow the developer to execute previous revisions of the code.

②: ICE provides limited consistency checks through the aid of feature logic but does not provide an overall dependency analysis solution.

③: The Software Dock architecture provides for components to be reconfigured automatically as the environment changes, however, it does not provide for the manual intervention or reconfiguration of components by the user.

④: The system provides primitive type checking through the use of a DBMS but does not provide any comprehensive type checking system.

⑤: The system supports reconfiguration operations, but only when the system is stable or static.

⑥: The Programmers Playground provides a means of checking the bindings between components but not for the entire system.

⑦: The system provides type checking on the ports used within the system but does not provide any type checking on the entire system.

⑧: Equus identifies that components exist within the system but provides no functionality to manipulate them.

4.1.1 Component Configuration Systems

As discussed in chapter 3, component configuration systems are primarily concerned with the development of computational units and combining those units to build entire systems. To achieve this functionality, the systems commonly provide supporting tools and the infrastructure required to aid developers in their construction efforts. As an example, these systems may include support for maintaining a revision history of each component or provide developers with the flexibility of being able to arbitrary recall previous revisions of a component and execute the code contained within it. The ability to manipulate revisions during development allows software developers to form various configurations and to assess them. Component configuration systems may also provide additional support for allowing projects to be developed concurrently.

Although component configuration systems provide for the development of computational units, they do lack support for both internal and external type checking and do not provide any functionality to allow interactions with the unit itself. Additionally, these systems do not facilitate communication between components because at this stage the components exist only in a source code perspective and have not been instantiated. As components in these systems exist in a meta format, source code or non-instantiated form, there is no need for these systems to provide any comprehensive reconfiguration support. The only reconfiguration service that may be provided is limited to the management of a components construction. Consequently, there is no need for any deadlock or consistency services to be incorporated into the system.

This concentration of functionality designed to aid the developer with the construction of computational units is shown in table 4.1 where the Distributed Revision Control System, Distributed Concurrent Versioning System and Incremental Configuration Engine all exhibit functionality consistent with component construction. By focusing on the construction of components, these systems do not satisfy any of the more substantial criterions developed to assist reconfiguration management systems.

4.1.2 Static Component Configuration Management

Static component configuration management systems are designed to allow developers a level of flexibility enabling reconfiguration of components to occur providing that conditions are satisfied. These conditions ensure the system remains in a stable state for the duration of reconfiguration. The term *static* in this case refers to the system being at a complete stop and hence allowing developers to be able to insert either source code, object code or third-party software components into the system.

By providing this functionality, the end-users of the system gain more flexibility as it can accept or receive new or updated components at any point when the system upgrade criterion is met. Once

ready for configuration, the system can proceed to calculate and implement the desired reconfiguration path. It is this path that allows the system to evolve and accommodate the domain in which it is operating. These evolutions are normally brought about by adopting organisational changes or addressing the issue of an erroneous component.

Table 4.1 illustrates that static component configuration management systems are not primarily concerned with the fundamental development of software components. This is highly evident as the table shows these systems to have little support for source control, revision management, concurrent component development or the ability for the developer to select what revision of the code to execute.

Additionally, due to the static nature of reconfigurations within the system, there is no support for on-going consistency checks while the system is running. The only consistency checks which may be available within these systems is through specialised development environments responsible for the loading and unloading of components or by a third-party tool such as a compiler that identifies type-checking errors during the systems compilation.

By not having to support a dynamic environment there is no need to provide services which are responsible for ensuring that components reach a state of quiescence nor is there a need to provide any on-going consistency or deadlocking services while the system is operating.

It should be noted that static component configuration management systems do provide an environment in which components can be substituted into a system, provided the system is in a static or stopped state. These environments either consist of a database management system or a fully integrated development environment.

4.1.3 Dynamic Component Configuration Management

Dynamic component configuration management systems differ from static component counterparts by providing an environment that allows components to be introduced, modified or removed from a system while it is still performing operations. Other dynamic component configuration systems might provide a complete environment specifically constructed to allow the replacement of components.

In such environments as the Programmers Playground, these activities may actually be performed while the system is performing an action or carrying out a number of tasks. Such reconfiguration actions however should be performed sparingly as these systems are not designed to support continual configuration changes whilst operating but rather when they are idle. Section 4.1.4 introduces those systems which are capable of integrating configuration changes into the normal activities of the system.

Just like the static component configuration management systems, dynamic component management systems do not provide support for the development of components. Functionality, such as revision management or supporting the development of a system concurrently, is left to component configuration systems such as those mentioned in section 4.1.1.

When compared to the static component configuration management systems, it can be seen that dynamic component configuration systems provide additional support for type-checking. These type-checking systems ensure a type match between components, the interconnections which join components, as well as providing type checking services for variables located within components. The actual level of type-checking is dependent on the implementation of the system. Additionally, dynamic component management systems typically provide extra functionality to manage and facilitate the communications between components.

It is important to note that although dynamic component management systems provide for the manipulation and reconfiguration of components, they do not include services responsible for ensuring the consistency of components. By not providing this support, software developers themselves need to make sure that they incorporate their own consistency checks into the component when performing reconfiguration operations or that they have adequate checks in place to monitor the stability of components. In addition to providing no consistency management, these systems do not necessarily include support for software developers to be able to perform a comprehensive dependency analysis when components are being reconfigured.

4.1.4 Runtime Component Configuration & Consistency

Runtime component configuration and consistency systems further enhance the dynamic environment by providing a means for developers to manipulate, add or remove components from a system while it is still performing unrelated tasks. Reconfiguration tasks are normally processed in parallel while the underlying architecture required to support the seamless interchange of components within the system processes the configuration change.

Table 4.1 shows that runtime component configuration and consistency systems have comprehensive consistency checking systems which are responsible for determining the impact that a configuration change will have on a system. Determining the components that will be affected by such a change is an important step to ensuring the overall consistency of the system, as components directly and indirectly involved in the reconfiguration must be handled with care.

In addition to the strengthened consistency checks, there are a number of systems responsible for determining the dependencies that exist between the various components within the system. These systems provide a method for being able to identify those components effected by a reconfiguration and to 'isolate' them while components are being restructured, updated, replaced or removed.

Overall, runtime component configuration and consistency systems provide a basis for a system to be able to deal with changes in the workplace and evolve to the user requirements while at the same time providing the appropriate safeguards to insulate the changes being made on the system from other components. In the past, systems have been constrained by their inability to adopt to the ever-changing needs of an organisation. This failure to accommodate change has either led to

organisations investing large amounts of money and resources in system upgrades or discarding one solution which has served the needs of the organisation and seeking another. Through the use of runtime component configuration systems it is possible for an organisation to maintain one system and to change it when the need arises.

However, after examining the various configuration management systems shown in table 4.1, it appears that none of the listed systems provide any support to software developers or end-users to control what happens to a method call when it is sent to a component under-going a reconfiguration operation. This is especially critical in systems where components are operating within real-time environments. The introduction of a real-time environment brings with it a new dimension as these systems need to know if a method call can or cannot be performed on time. A method call unable to be performed may result in consequential actions or recovery paths having to be performed (if specified) rather than blocking and waiting for the method call to be processed when the component becomes available again.

4.2 Real-Time Dynamic Component Updates

Before examining the impact of having no control over how method calls are handled when components are unavailable and operating within a real-time environment, it is important to consider the definition, roles and responsibilities of real-time systems which are fundamentally effected by any non-anticipated availability issues.

4.2.1 Real-Time Systems

Since the development of time critical circuitry and systems, the need for and the complexity of real-time systems has steadily increased. Today, most pieces of electronic equipment concerned with or operating within timing constraints contain embedded real-time systems. These real-time environments are normally defined as systems which are bound by a series of timing constraints (Bennett 1990; Levi and Agrawala 1990).

Typically, a real-time environment consists of two parts known as the *controlling sub-system* and the *controlled sub-system*. The controlling sub-systems refer to an abstract entity modelling the physical device being controlled except that the device is emulated in software. These sub-systems are typically hardware managers responsible for coordinating the actual signals sent to the physical device. The controlled sub-systems refer to the actual physical device being controlled by the system.

Between these sub-systems a number of interactions take place. These include data sampling from the controlled device, processing of data received from the device and responding to the conclusions reached from the processing stage. The consequences of these processing actions can range from

taking no corrective action through to transmitting a series of commands to the device from where the data originated or to other components and devices scattered throughout the system.

To this point, real-time systems have been identified as being responsible for coordinating and executing tasks based on a strict set of timing constraints. Real-time systems however can be further classified into two sub-categories known as 'hard' or 'soft'. The difference between these classifications depends on what the implications are for tasks which are unable to meet their deadline and whether 'slips'[1] in the deadline for starting tasks are permissible.

4.2.1.1 Hard Real-Time Systems

Hard real-time systems provide an environment in which an event scheduled for a particular time ***must*** be executed. Failure to execute the task at the specified time normally results in dangerous circumstances or a condition being raised (eg. car not starting as a result of the fuel injection system not working due to an error with the timing of the fuel into the system). To facilitate the execution of these tasks at their specified deadlines, a hard real-time system (under certain circumstances), will re-organise its own internal priority scheduler to provide the best opportunity for that event to be executed. To provide such a flexible and dynamic environment for hard real-time operations requires a considerable amount of programming as thought must be given to each instruction and the time it will take to complete. In addition to calculating the time it takes to complete an instruction, concern must be given to factor in the implications of tasks running later than expected and what flow-on effects such situations might introduce into the remainder of the system.

4.2.1.2 Soft Real-Time Systems

Soft real-time systems provide an environment in which the system tries to satisfy all of the tasks scheduled within the relevant time schedule. However the failure of a task to execute at a nominated time does not result in any significant circumstances occurring but rather introduces a slip into the schedule. To recover from a slipped deadline schedule a recovery path which normally consist of a series of alternative actions (consequential actions) is performed. For this reason, a soft real-time system is considered to be more flexible in nature when compared to hard real-time systems.

As mentioned, real-time systems (more often than not, soft real-time systems) support error recovery paths which can be undertaken when a task slips its deadline schedule or is not executed. In Tsai, Bi, Yang, and Smith (1996) the notion of time being important to real-time systems is further reinforced by taking into account the following aspects of time: Clock Access, Process Delay, Timeout Handling and Deadline Specification and Scheduling.

[1] The failure of a real-time system to perform a task at a designated period of time is known as a slip.

4.2.2 Real-Time Component Control

As explained, real-time systems, especially hard real-time systems, rely heavily on the timing of their operations to ensure that tasks do not miss their deadline. To further illustrate the notion of how important timing is to real-time systems, it is not uncommon for software developers to calculate both the worst case and best case scenarios for the time it will take to process a single instruction or to perform a number of instructions contained within a loop. In addition to calculating the time required to process instructions, a number of other factors including the speed of the processor and load on the machine have to be considered when calculating the speed at which an instruction can be executed.

With a heavy dependence on timing issues, real-time systems are very susceptible to untimely events such as spontaneous reconfigurations or dealing with components that are unavailable. Such events when incurred can cause an unexpected delay in the return of a method call sent to a component not available. Delays such as these may have disastrous consequences in a hard real-time system if the delay is not identified straight away or may result in soft real-time systems executing an error recovery path to try and recover from the situation of the task not being performed within the corresponding time frame.

Table 4.1 provides an overall summary of many available and well known configuration management systems. The table clearly shows that none of the configuration management systems reviewed provide any support for software developers or end-users to control how a method call operating under a real-time environment is dealt with when a component is unavailable. Goudarzi in Goudarzi (1999) identifies this problem in his proposed extensions to the REGIS system but dismisses the delay or blocking of method calls as being an inevitable part of the reconfiguration process.

This approach to delaying method calls within a system may be acceptable in a standard computing environment but in a real-time system (especially in hard real-time systems) such a delay must be planned for and factored into the operation of the system. As a consequence, lack of support for controlling the way in which a method call is handled by reconfiguring components results in:

- Systems not being able to identify that a problem exists when making method calls on components being reconfigured until the method call has actually been made and the resulting blocked method call has caused the real-time system to miss its deadline schedule
- Systems which are unable to provide appropriate procedures to software developers or end-users to allow them to specify what behaviour be undertaken should the system request a method call to be performed upon a component which is reconfiguring itself or is otherwise not available
- Systems not being flexible enough to suit the needs of the organisation or end-users

- Systems incapable of ensuring that a certain level of Quality of Service (QoS) is maintained as a result of the system being unexpectedly interrupted or blocked by unforeseen reconfiguration changes

4.3 Benefits of Introducing Control

As mentioned in the previous section, lack of controls available to software developers and end-users brings with it a significant number of problems to those real-time systems responsible for coordinating a large number of activities over a strict and finite timing period (eg. controlling the tracking system for a radio telescope array). To address this, the thesis presents a concept within the framework of a real-time system allowing software developers and end-users alike to be empowered with the ability to control and manage the execution of method calls made on components which are in the process of being reconfigured.

By harnessing the concept of providing control to the user it is possible to develop systems that are capable of providing a considerable amount of support in dealing with reconfiguring components. These benefits include providing a real-time environment in which method calls, which can not achieve their deadline specifications as a result of a system reconfiguration or a component not being available, are identified and handled. As explained in chapter 5, the Component Oriented Reconfiguration Environment and Scheduling (CORES) model provides developers with more flexibility providing a wide variety of actions which can be performed upon method calls effected by reconfiguring components.

These actions include:

- Allowing a method call to block while the component is being reconfigured and not to proceed until the component has been deemed safe and is accepting new requests (typical system behaviour)
- Allowing a method call to return to its caller to indicate that there was an error condition preventing the execution of the request and enabling the system to take immediate action (default behaviour for CORES)
- Providing the ability to wait until a certain Quality of Service (QoS) agreement is satisfied before allowing the continuation of the request

As can be seen, all of the approaches to handling reconfiguring components within CORES have been designed to provide software developers with control over how to deal with method calls which may be interrupted as a result of reconfiguring components. CORES also provides support for identifying how QoS requirements within the system can impact on the execution of method calls made within the confides of a real-time environment.

4.4 Chapter Summary

Throughout the chapter the focus has been on examining the various strengths and weaknesses that exist between the configuration management systems reviewed in chapter 3. To illustrate the point, table 4.1 showed the differences in the functionality based on the criterion developed to allow configuration management systems to be compared with one another.

The table illustrated how configuration management systems that are grouped together based on the support they provide for reconfiguring components share a common functionality. The various support categories for configuration management include:

- Component Configuration Systems
- Static Component Configuration Management
- Dynamic Component Configuration Management
- Runtime Component Configuration & Consistency Management

In addition to analysing the various strengths and weaknesses of configuration management systems, this chapter also examined what impact lack of control has on developers or end-users in a real-time system where the execution of tasks is time critical. Finally, the chapter concludes by identifying the benefits of systems adopting the concepts presented in this thesis. The following chapter introduces the CORES model which is responsible for ensuring that software developers and end-users alike are capable of dealing with components being dynamically reconfigured in an environment which supports real-time operations. The CORES model also provides the opportunity for software developers or end-users to specify what should occur when a method call can not be processed by a component.

Chapter 5

Component Oriented Reconfiguration Environment & Scheduling System

Previous chapters have raised a number of issues which directly relate to the answering of research questions RQ1 and RQ2 (refer to section 1.3). To answers these questions a number of configuration management systems and proposed architectures specifically designed to facilitate the development and reconfiguration of components were evaluated. In addition to providing support for the construction and reconfiguration of components it was found that such architectures allowed software developers and end-users to exercise limited control over components and to a lesser degree the handling of method calls. From the systems identified in chapter 3, an examination was conducted (refer to chapter 4) to determine what support if any exists for dynamic reconfiguration (refer to RQ1) and what support exists for handling method calls made on components[1] operating within real-time environments and participating in reconfiguration activities (refer to RQ2).

In response to RQ1 and RQ2, the examination discovered that attempts have been made to develop systems capable of providing dynamic reconfiguration services but none were able to combine the requirement of providing an environment that supports the dynamic reconfiguration of components in addition to the strict timing concerns of real-time systems.

From the conclusions drawn by RQ1 and RQ2 it is clearly evident that a lack of support exists for those systems which have components operating within real-time constraints. Specifically, this lack of support impacts on software developers and end-users by preventing them from being able to control what happens when a method call is made on a component that is unavailable. This level of control

[1] Referred to as servers in the implementation.

is vital and is required in systems operating under real-time constraints (ie. scheduling systems) where a components inability to execute a task needs to be immediately identified and handled. To address this lack of support a conceptual model needs to be developed which provides additional capabilities to allow software developers or end-users to specify what actions should be taken if a method call is sent to a component which is unable to process it. This lack of availability might be the result of a component being deactivated or a component being involved in reconfiguration activities.

Such a model forms the basis behind research question RQ3 which asks what are the minimal requirements for a conceptual model to be able to address the lack of support for real-time environments and the components operating within them. The question is also asked as to whether additional support can be integrated into the model to enable software developers and end-users to control the behaviour of method calls. This chapter provides a solution to the concerns raised in RQ3 by proposing a model, its requirements and a number of algorithms capable of allowing software developers and end-users to control how components and method calls are managed. Although the algorithms presented throughout this chapter are based on already existing algorithms for scheduling, they do provide an approach for the sequencing and scheduling of tasks and jobs operating within a real-time environment where specific attention must be paid to when tasks and jobs are performed and whether they are capable of being performed in parallel or sequentially. In addition to RQ3, the model incorporates the concerns of RQ4 and RQ5 by including support to vary the way that tasks are sequenced and scheduled as well as introducing the ability for Quality of Service (QoS) characteristics to dictate when components are available.

Significant advances have been made in controlling method calls and objects with the development of the design by contract approach discussed in Meyer (1997). Since the initial development, further work has been carried out which allows contracts to be implemented in the distributed computing paradigm as explained in Watkins (2000). The concept of allowing a contract to be established between a client and a server and having the contract govern aspects of the method call provides a basis for developing a system that allows software developers or end-users to control how a method call is processed. Additionally, Goudarzi (1999) illustrates how an environment can be established where the reconfiguration of components is possible with the assistance of quiescence routines to stabilise the areas requiring reconfiguration. Despite these advances, there is still no available architecture or system to handle the reconfiguration of components or control method calls operating within a real-time environment.

Presented here is the Component Oriented Reconfiguration Environment and Scheduling model (CORES) designed to provide software developers, end-users and operators alike with an environment for components to be dynamically reconfigured while concurrently handling the requirements of a real-time system. CORES also provides a scheduling and sequencing system allowing tasks contained within a job to be optimised and placed into a scheduling environment which is aware of real-time constraints. The sequencing component of CORES provides the ability to sequence tasks

using either a best case or worst case scenario. The ability to calculate the worst case scenario is important to real-time systems which have a scheduler operating in a conservative mode. The CORES model also introduces the functionality for software developers or end-users to handle and control method calls when a component is being reconfigured.

5.1 CORES Algorithms

To provide functionality, CORES makes extensive use of a variety of algorithms. These are responsible for sequencing tasks within a job, calculating the optimal path in a job containing multiple paths and scheduling a job into the schedule. The following sections introduce the most commonly used algorithms by CORES. The algorithms, wherever possible, are explained in generic terms however in some cases it may be necessary to refer to those tasks performed by the case study presented in chapter 7 to provide some clarification. Throughout the chapter, a number of jobs and tasks are graphically represented with the use of Petri nets.

Petri nets were first proposed in Petri (1962) and allow for the graphical representation of places, transitions and flows within a job. Reisig (1985) explains that a Petri net consists of three elements. These include: a place (referred to as a state in the thesis), a transition and an arc. A state is a defined point within a job (ie. a task contained within a job) and a transition represents an event happening which results in a state change. An arc represents the link from a state to a transition or vice versa. It is through the elements of a Petri net that the relationships between the various tasks can be represented. The graphical nature of Petri nets allow for the implications of those jobs containing multiple paths to be illustrated. The Petri nets and algorithms presented in this chapter do not provide support for jobs containing cyclic dependencies.

The algorithms, specifically those introduced in section 5.1.1 and 5.1.2 do not seek to provide an alternative to the depth-first traversal and shortest path algorithms found in graph theory as detailed in Sedgewick (2002) or Gross (1999) but rather to incorporate them into a model which supports real-time requirements. With the combination of re-sequencing tasks and graph theory algorithms, it is possible to identify the permutation which requires the least amount of time to complete. Identifying the shortest path becomes critical when operating within real-time environments.

5.1.1 Sequencing Tasks within a Job (Single Path)

The algorithm is responsible for taking a series of tasks contained within a job and re-sequencing them into a specific order. This may involve minimising (worst-case scenario) or maximising (best-case scenario) the amount of time required to complete a series of tasks. By default, the algorithm makes use of the best-case scenario.

5.1.1.1 Requirements

The sequencing of tasks plays an important role in the overall CORES model as the scheduler uses the calculated sequences later on in the system to form the overall schedule. To achieve this sequencing, the algorithm performs a number of steps.

The first algorithmic step is to acquire information about each task associated with the job. This is used to assist in determining the placement of the tasks within the sequence and at what time a task can be executed. As presented in chapter 7, the case study makes use of radio astronomy and therefore all tasks defined within a job in this chapter will be assumed to be astronomical in nature. On this assumption, three key pieces of astronomical information need to be obtained for each task. In general the data obtained at this point relates to information which can be used to determine a possible start time.

The pieces of astronomical information which need to be obtained include the right ascension of the radio point source, the declination of the radio point source (provide coordinates for the point source in the sky nominated for an observation) and the total amount of time that the astronomer wishes to spend observing a radio point source. Given this data and a time reference it is possible to calculate a time at which a task can be started. For the case study this dictates the time at which an observation could possibly begin. The mathematical steps involved in calculating the position of a radio point source, given its right ascension, declination coordinates and time can be found in Fitzpatrick (1970) and Duffett-Smith (1988). Part of the start time calculation relies on determining when a task and job can actually be performed successfully. In the case of radio astronomy it refers to calculating a start time where a radio point source would be located above the horizon. For other scenarios this may involve checking task relationships to ensure that preparatory tasks have been performed before the main task is executed.

Having this information for a task allows the algorithm to proceed with its calculations (if requested) to generate the possible permutations that exist for a job given the number of tasks contained within it.

After calculating the total number of permutations, the algorithm uses them to construct an array of possible job scenarios. This array is then used as a starting point to identify those scenarios which are not viable and hence ineligible for any further assessment.

A scenario is considered not to be viable if a sequence of tasks can not be performed in the order specified or there exists a point in the scenario where upon the completion of one task, another task can not be started. This is known as the continuous task test (ie. checking to make sure a task can run when another finishes). By performing this test it is possible to identify and eliminate those scenarios which are impossible to perform or if performed would result in resources remaining idle.

Using an observation schedule as an example, figure 5.1 represents a valid scenario while figure 5.2 illustrates how a gap or idle point in the scenario between tasks causes the resource (the telescope array) to undergo a period of idle time. Wherever possible, CORES tries to actively avoid this situation.

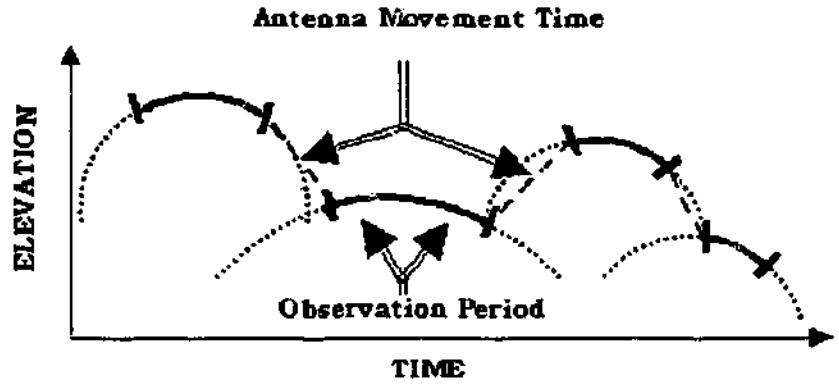


Figure 5.1: Valid Task Sequence

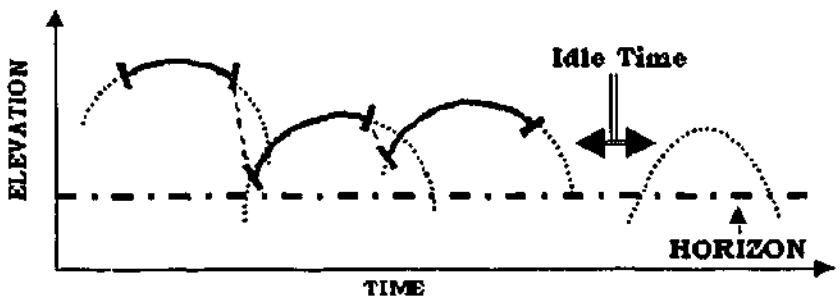


Figure 5.2: Invalid Task Sequence

With the non-viable scenarios removed, the algorithm may proceed to determine which scenario contains the least amount of time spent completing infrastructure tasks so that the main tasks can be performed. In the case study this refers to the time spent moving an antenna from one position where an observation finished to another point where the next task starts. The infrastructure time is illustrated in figure 5.1 and shows the time taken to move from one position to another.

The calculation of the time spent on setting up the infrastructure requires careful planning as the entire scenario must first be represented in such a manner as to identify the various states resulting from the execution of tasks. Once the states are identified, the algorithm can determine what infrastructure tasks need to be performed to allow the main task to be performed and how long it

will take to setup the infrastructure. To construct the plan of a job, the algorithm iterates through all of the valid scenarios and assembles each in memory while making sure that additional tasks such as those needed to setup the infrastructure are factored in. Once constructed, the algorithm will determine the best point in time for the scenario to start. The starting time calculated by the sequencing routine is used internally to confirm a viable scenario. Where a scenario being assessed is selected for scheduling, the scheduler will recalculate the start time.

The sequencer has to be extremely careful when it is laying out the tasks because even a small task, such as rotating the antenna receiver, can change the dynamics of the entire scenario. Figure 5.3 illustrates what a group of tasks belonging to a job may look like after the algorithm has examined the tasks and assembled them into a sequence.

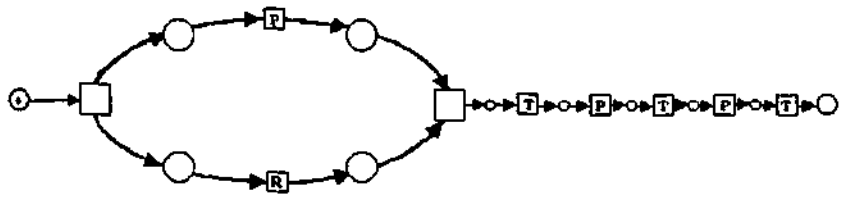


Figure 5.3: Sequenced Job showing Node Relationships

The tasks shown in figure 5.3 denoted by the letter 'P' are transitions and are responsible for establishing the infrastructure before the task is performed. In the case study these tasks are responsible for positioning the antenna before a tracking operation and contribute to the scenario infrastructure setup time. Those tasks labelled 'T' represent transitions performing tasks. The case study uses these tasks to track a radio point source through the sky. These transitions can take a considerable period of time as observations or tasks can take a number of hours to complete. The remaining task labelled 'R' represents additional infrastructure tasks, such as rotating the antenna receiver turret into position. Although not explicitly defined in the job definition file, the model automatically inserts these tasks to ensure that the infrastructure is capable of supporting the task at the requested time.

It is important to note that when a particular scenario for a job is being constructed all tasks that can be performed in parallel are grouped in such a way as to allow the algorithm to identify them so that they can be assembled in memory. Figure 5.3 represents two tasks being executed in parallel and for the execution to occur concurrently, the architecture must support concurrent processing.

Once the scenarios have been planned, calculated and assembled, the algorithm proceeds through them all to determine which has the least amount of associated infrastructure setup time. This involves concentrating on the time difference between the end of one standard task and when the next standard task starts. By determining this, the algorithm is capable of calculating the total

amount of infrastructure setup time required for the scenario. Infrastructure tasks are not assessed by the algorithm.

After the infrastructure setup times have been calculated, they are totalled and compared to a best candidate path time. If the setup infrastructure time for a given scenario is less than the time offered by the best candidate then the scenario being examined is recorded along with the sequence of tasks to form the new best candidate path. If the time calculated is greater than the recorded time for the best candidate path, the scenario is ignored and the next is assessed.

This process of examining all of the possible scenarios continues until all have been examined. Upon the completion of the algorithm, the model is capable of returning a scenario which has the least amount of setup infrastructure time.

5.1.1.2 Formal Definition

With the aid of mathematics (specifically set notation) this section formally defines the algorithm described in section 5.1.1.1 which is responsible for identifying the optimal sequence of tasks contained within a job given that there is only one path. However, before examining the algorithm it is necessary to outline some of the definitions used throughout this algorithm and others that the CORES model uses in sections 5.1.2 and 5.1.3.

To concisely represent the main objectives of this algorithm, the sections pertaining to the generation of the possible permutations of a job and the calculation of the optimal start time have been removed. Chapter 6 details how these algorithms were implemented.

Definition 5.1.1.1.

A radio point source (task element) is a pair such that rps = (r,d) with r,d $\in \mathbb{R}$.
The elements within rps are as follows:

- *r is the right ascension coordinate and is denoted by rps.r*
- *d is the declination coordinate and is denoted by rps.d*

Let Rps denote the set of all radio point sources as defined above.

This defines the structure of a radio point source (from a mathematical point of view). An important note with this definition is that the *rps.r* and *rps.d* values are independent of time and remain constant for the radio point source. These values can be thought of as a unique identifier describing the 'orbit' that the radio point source is in. The *rps.r* and *rps.d* values are used in conjunction with other information to locate the precise point of the radio point source in the sky.

Definition 5.1.1.2.

A radio point source observation (task) is a tuple such that rpsObs = (p,r,s,t,f) where p is a radio point source and r,s,t,f ∈ ℝ. The elements within rpsObs are as follows:

- *p is the radio point source which is to be observed and is denoted by rpsObs.p. To access the right ascension or declination values the following notation should be used: rpsObs.p.r or rpsObs.p.d*
- *r is the rise time which is expressed as the number of seconds elapsed since epoch (1st January 1970) where the radio point source becomes visible above the horizon*
- *s is the set time which is time expressed as the number of seconds elapsed since epoch where the radio point source sets below the horizon*
- *t is the track time which is expressed as the number of seconds elapsed since epoch where the tracking of the point source commences*
- *f is the time that the tracking finishes and is expressed as the number of seconds elapsed since epoch where the tracking of the radio point source ceases*

Each of the times recorded within a radio point source observation is bound by the following:

$$0 \leq r \leq t < f \leq s$$

For the remainder of this thesis, RpsObs denotes the set of all radio point source observations as defined above.

The definition of the radio point source observation (*task*) illustrates how only a segment of an entire observation window is used for a radio point source. This may not always be the case, as other scenarios such as manufacturing process may require that a task be performed as soon as it can and finish as late as possible. Figure 5.4 illustrates the various elements of a radio point source observation (*task*) and shows how only a subset of the total amount of time is used.

With the radio point source (*rps*) and the radio point source observation (*rpsObs*) having been defined, it is possible to define an observation schedule which contains a sequence of radio point source observations (*tasks*).

In addition to those definitions previously mentioned, it is necessary to define an assumption enabling the algorithm to calculate the minimum amount of time required to perform the infrastructure tasks. This time can be calculated by summing the difference in time between one task finishing and another starting.

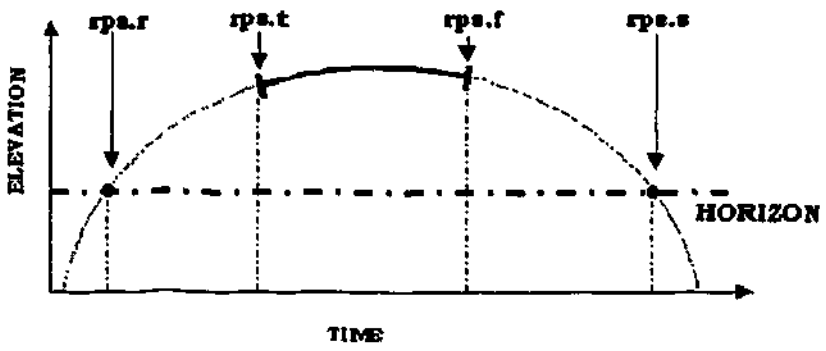


Figure 5.4: Radio Point Source Elements

Assumption 5.1.1.1.

To calculate the optimal observation schedule[2]*, it is necessary to call upon an external function which returns the number of seconds it takes for the infrastructure to be setup to allow the execution of a subsequent task (ts2) given that the preceding task (ts1) has been completed. The external function is defined as minSetup: RpsObs* $\times$ *RpsObs* $\rightarrow \mathbb{R}$ *such that minSetup*$(ts1, ts2) = t$ *where* $(t \geq 0)$ *is the minimal time required to setup the infrastructure.*

The parameters of minSetup include:

- *ts1 is tracking source 1 which is the radio point source observation (old task) which was originally being tracked*
- *ts2 is tracking source 2 which is the radio point source observation which is scheduled to be tracked next (new task)*

The function *minSetup*(...) calculates the total amount of infrastructure setup time required by the subsequent task after identifying what condition the preceding task left the infrastructure in. Once the condition of the infrastructure has been determined, the function can identify what actions need to be performed and the time required before the subsequent task can be supported. If necessary, the results from the *minSetup*(...) function can be used to recommend the adjustment of starting times for subsequent tasks so as to take into account the additional time required to setup the infrastructure.

[2]It is assumed that the operator wishes to calculate the optimal path.

Problem Statement 5.1.1.1.

Given a sequence of tasks to be performed, find an appropriate sequence of tasks which represents the minimal amount of time required to perform the infrastructure tasks. More formally, let: $n \in \mathbb{N}$ *and* $s = o_0,\ldots,o_{n-1}$ *be an observation schedule with* $o_i \in RpsObs$ $(0 \leq i < n)$*, find a sequence* $s' = o'_0, \ldots, o'_n$ *such that:*

1. s' *is a permutation of* s *that is, there is a sequence of indices* $i_0, \ldots, i_{n-1}$ *with* $o'_{i_j} = o_j$ *(for all* $0 \leq j < n$*)*
2. s'' *is equal to* s' *except that the corresponding starting and finishing times for two successive observations are updated inductively:*
   - 2.1 $o''_0.t = o''_0.r$ *(the first observation is initially started as soon as it is above the horizon)*
   - 2.2 $o''_{i+1}.t = max(o''_{i+1}.r, o''_i.f + minSetup(o''_i, o''_{i+1}))$ *(for* $i > 0$*)*
   - 2.3 $o''_{i+1}.f = o''_{i+1}.(o'_{i+1}.f - o'_{i+1}.t)$ *(ensure that the observation is of the required duration)*
   - 2.4 *The sequence* s' *is realisable iff* $s'' \in RpsObs$*, ie. iff the starting and finishing times precede the radio point sources setting time*
3. *The infrastructure setup time expressed by the following sum, is minimal:*

$$(\sum_{i=1}^{n} minSetup(o''_{i-1}, o''_i))$$

Notes:

- *The sequence* s' *is the same as sequence* s *except that the tasks are re-ordered*
- *Each task in the sequence* s *already has the constraints* $(0 \leq r \leq t < f \leq s)$*, a realisable order of tasks must ensure that the end of one task proceeds the beginning of the next*
- *Note that the induction rule 2.1 connects* s'' *and* s' *because* $o''_0.r = o'_0.r$ *since the rise times are unchanged by definition*

Algorithmic Solution 5.1.1.1.

```
Let SS = {s'' | s'' ∈ RpsObs, s'' is a permutation of s,
          s'' is realisable (according to problem statement 5.1.1.1) }
Let n ∈ N and op = o_0,...,o_{n-1} be an observation schedule with o_i ∈ RpsObs (0 ≤ i < n)

bestCandidatePathTime ∈ R (0 ≤ bestCandidatePathTime ∈ R);
bestSequence ∈ RpsObs;
currentSetupTime ∈ R (0 ≤ currentSetupTime ∈ R);

If SS ≠ {φ} then
     op := SS_0;
     Let n ∈ N and op = op_0,...,op_{n-1} be an observation schedule with op_i ∈ RpsObs (0 ≤ i < n)
     in
        if (op'_i.f + minSetup(op'_i, op'_{i+1}) + op'_{i+1}.d) <= op'_{i+1}.s then
           bestCandidatePathTime := (Σ_{i=1}^{n} minSetup(op'_{i-1}, op'_i));
        end
        else
           continue; // Skip this permutation and move onto the next
        end
     end
end

// assert:
// If there are no realisiable sequences, then the observation can not be performed
// In the event that there are no realisiable schedules, leave the op and bestTime undefined

for all elements s' ∈ SS:
     Let s' = s'_0,...,s'_{n-1} be the given permutation and
        if (s'_i.f + minSetup(s'_i, s'_{i+1}) + s'_{i+1}.d) <= s'_{i+1}.s then
           currentSetupTime = (Σ_{i=1}^{n} minSetup(s'_{i-1}, s'_i)) be the minimal infrastructure setup time
        in
           if bestCandidatePathTime > thisTime then
              bestCandidatePathTime := currentSetupTime;
              bestSequence := s';
           end
        end
        else
           continue; // Skip this permutation and move onto the next
        end
     end
end
```

The solution is achieved by iterating through each radio point source observation (*task*) contained within a scenario and performing the following calculation:

$$(\sum_{i=1}^{n} minSetup(s'_{i-1}, s'_i))\ (0 < i < n)$$

As can be seen, the algorithm is responsible for going through each *task* and calculating the minimum amount of required infrastructure setup time before the next *task* can be performed (ie. using the case study as an example would result in calculating the time required to move the antenna from the end of one task to the start of the next).

The results from the *minSetup*(...) function can be used to adjust the remaining start times of other tasks contained within the scenario so as to take into account any changes which may have been made earlier. Once the time spent performing infrastructure tasks is known, it is compared to the best candidate path time. If an optimal sequence has been calculated, the tasks in the scenario and the total amount of infrastructure setup time is recorded after which the remaining scenarios are evaluated. If the time calculated is not optimal then the scenario being examined is discarded and the next scenario is evaluated.

On examining algorithm 5.1.1.1 it is clearly evident that it relies on a number instructions and loops to satisfy its objective. Based on its structure it is possible to calculate the overall complexity of the algorithm by determining the growth in the number of instructions that are executed when the number of tasks and observation schedules that need to be completed are incremented.

Given that each instruction including assignments, comparisons and summations carry a complexity value of one and that three instructions (not shown in algorithm 5.1.1.1) are needed to co-ordinate a looping operation, it is possible to express a formula capable of calculating the number of instructions that need to be performed (in a worst case scenario) given the number of tasks and observation schedules. This formula is expressed as $4t^2 + 6t + 4s^2 + 8s + 2$ where t represents the number of tasks to be performed and s represents the observation schedules.

t	s	Instruction Count	Growth
1	1	24	0
2	2	62	38
3	3	116	54
4	4	186	70
...	...	...	...
5	20	1892	-

Table 5.1: Complexity of Sequencing Tasks in a Job

The overall complexity of algorithm 5.1.1.1 is clearly identified in table 5.1 which shows the growth rate of instructions that need to be performed as the number of tasks and schedules requiring processing are incremented. Examining this growth rate allows for the order of magnitude to be

identified as $O(n^2 * s^2)$ although the values used within table 5.1 indicate a $O(n\log_n)$ magnitude. Closer analysis of the figures in the table indicate a quadratic relationship as the growth rate increase at a constant rate however such a relationship is only identifiable as a result of the number of tasks and schedules being equivalent. The last row within the table represents a realistic example which CORES processes. As the instruction count indicates, a considerable amount of processing is required to sequence tasks and observation schedules.

5.1.2 Sequencing Tasks within a Job (Multiple Path)

This section presents the algorithm responsible for navigating and evaluating multiple paths within a given job and returning the optimal sequence of tasks while ensuring that the job can be completed successfully.

5.1.2.1 Requirements

From the algorithm defined in section 5.1.1, it is possible to see how tasks belonging to a job can be sequenced to minimise the amount of infrastructure setup time required when moving from one task to another. Although this approach is satisfactory for the case study (refer to chapter 7) and those scenarios where there is only one path to follow from start to finish, it is not satisfactory for those where multiple paths are in existence. Figures 5.5 and 5.6 illustrate the difference between a job which has only one path and a job containing multiple paths to reach its destination.

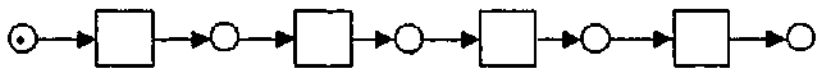


Figure 5.5: Job containing One Path

The job presented in figure 5.6 demonstrates that for an algorithm to be able to deal with generic situations it must be constructed to allow jobs with one or multiple paths to be processed equally. From the illustration it is also possible to see that the tasks contained within the scenario are structured in a similar manner to a Directed Acylic Graph (DAG) which is commonly found in graph theory. By adopting some of the principles of graph theory as explained in Diestel (1997) and Swamy and Thulasiraman (1981) it is possible to construct a graph representing the tasks and interconnections while ensuring that there are no repeating loops or backward flowing paths within the graph. Additionally, using graph theory it is possible to use a variety of algorithms as mentioned in Sedgewick (1988) and in Sedgewick (2002) to assist in the traversal of tasks.

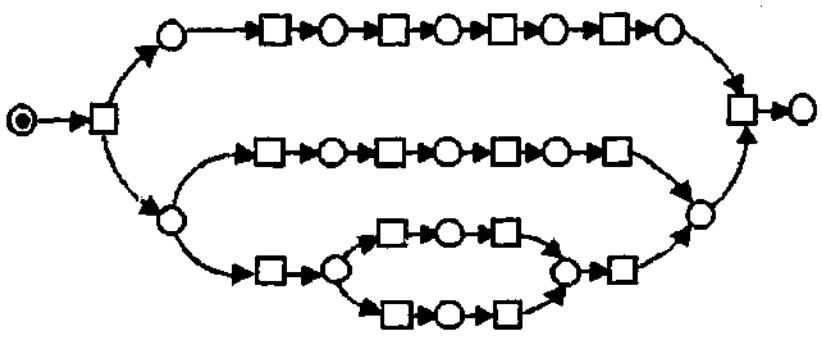


Figure 5.6: Job containing Multiple Paths

Figure 5.6 also illustrates that any generic algorithm developed must be aware of decision nodes. These decision nodes are used to define the direction and actions that an algorithm must take and are known as either an 'OR' task or an 'AND' task.

An 'OR' task when incurred results in the algorithm processing one of the paths attached to the task until another 'OR' task is encountered. The discovery of the second 'OR' task signals the end of the first path. With the first 'OR' path processed, the algorithm steps back through the path taken and assesses subsequent 'OR' paths. Once all 'OR' paths have been evaluated, the algorithm determines which of the paths traversed is optimal, appends that path and the traversed time associated with the path and continues on.

The other type of task is known as the 'AND' task and is responsible for allowing paths or operations to be executed in parallel. Just like the 'OR' path, the 'AND' path is traversed until the second 'AND' task is incurred. At this point, the time taken to reach the second 'AND' task representing the end of the sub-path is calculated. This process continues for each 'AND' path, then after all the paths have been examined, the path which took the longest time to traverse is selected. This path is selected because no matter how fast the other paths are performed, the remaining paths will still be waiting for the slowest path to complete.

With an overview of how paths are structured within a scenario and how they are navigated it is possible to examine in more detail the mechanics of the traversal routine and the algorithm capable of calculating the total traversal time.

When the algorithm is first presented with a scenario, it is provided with both an origin and destination task. The destination task is provided so that the recursive traversal routine knows when to stop recursing upon itself. In addition to this check, other checks can be performed on the task to see whether or not it is the destination. These checks include the examination of the minimum and maximum times recorded within the task. If both are zero then the destination task has been identified as no other task can be completed in zero time.

With the origin and the destination specified, the algorithm navigates the tasks within the job scenario. The tasks and their relationships throughout the scenario are identified by examining each predecessor and successor that the algorithm discovers. This process begins by examining the successors of the origin task which is passed in. While a task is being examined its internal state is modified to 'Visiting'. The internal state is used by the algorithm to determine whether a task has already been or is being subjected to a path discovery examination.

Before performing actions contained within the task, the internal state is inspected and if necessary (when the task is already in the state of 'Visiting') the investigation of the current task is stopped and the algorithm returns to the task which was the immediate predecessor. If the state of the current task can successfully be changed and there are no outstanding actions to be performed by the task then processing continues.

If the predecessors have not been satisfied (ie. visited) then the algorithm records the path and the time taken to reach this point in the scenario and returns to the task's immediate predecessor and continues with identifying possible paths from that point.

Once the task confirms that the predecessors have been completed, they are counted and used to identify whether the task is deemed to be normal or whether it is being used as a collection point for many paths to come together. If the predecessor count is greater than one then the algorithm examines the traversal times associated with each predecessor path and makes a decision on calculating the optimum path.

The decision is based on whether the task being examined is of an 'OR' (conjunction) type or whether it is of an 'AND' type. If the task is of type 'OR' then the algorithm selects the predecessor path with the lowest traversal time and adjusts the overall traversal time and path to reflect the optimal path. Where it is of an 'AND' type, the algorithm concatenates the various predecessor paths traversed and eliminates any duplicate path information. In the event that there is no concurrency support available, the algorithm will concatenate the 'AND' predecessor paths resulting in the summation of all predecessor traversal times and the result added to the overall time for the scenario. If concurrency is available the predecessor path with the highest amount of traversed time is added to the overall traversal time. In concurrent situations the algorithm will make any necessary adjustments to common time spent by the paths as they executed concurrently.

If the task is deemed normal, the algorithm adds only the time associated with the task to the overall traversal time.

After the calculations have been performed on the task, the task's label is added to the path traversal and the focus of the algorithm shifts to examining the successors recorded within the task. The processing of the successors is achieved by establishing a loop and processing each successor contained within the task. If successors do exist then the algorithm needs to once again identify what type of task is being examined as successors need to be handled in differing ways depending on whether they are of type 'AND' or type 'OR'.

If the task is identified as being normal or an 'AND' task, the algorithm will take the first successor and start to traverse the path according to the successor. To assist in the traversal of tasks, a recursive function named *traverse*(...) is used. In order for the function to operate successfully, it must be provided with some information relevant to the traversal. This information is expressed in the form of parameters. The first two parameters include the successor of the current task being examined and the destination task which the traversal routine is seeking. The remaining parameter is a value representing the time spent traversing the job. If the successor to be passed in is the first to be examined, then the function is passed in the total time spent traversing the job. If the successor is a subsequent, the time passed into the function is the total traversal time minus the time required to complete the current task. This subtraction from the overall time is performed so as to avoid the algorithm and function from double counting. The processing of the successors continues until all have been examined.

The only differences between the 'AND' and 'OR' tasks is the way in which they process predecessors and successors. Just like the 'AND' task, the 'OR' task loops through all of its successors providing them with the total traversal time. This is different from the processing regime used for 'AND' tasks where total traversal times have to be adjusted so as to avoid any double counting which may occur when paths are combined. This is not the case with 'OR' tasks as only one path is selected based on the optimal time for the various 'OR' paths.

Once the successors have been examined the status of the task is modified from a state of 'Visiting' to 'Visited'. Changing the state of the task prevents the algorithm from trying to re-examine and process a node which has already been processed by the algorithm via a different path. If the algorithm does discover a node which has been previously examined, the algorithm will reject the task and return back to the preceding task.

The traversal algorithm then continues to process each of the tasks within the scenario until the destination task is reached. To further understand how the algorithm traverses through a job scenario, the following section presents a formal definition. It explains how the algorithm analyses all of the possible paths within a job scenario and how it calculates the optimal path.

5.1.2.2 Formal Definition

The algorithm discussed here is responsible for identifying the optimal path from a job scenario which may contain a number of paths. The algorithm is expressed with the assistance of a mathematical induction ruleset.

Before examining the algorithm from a formal point of view, it is important to understand how it complements others that have already been defined. Performing such an examination assists in clarifying the role and purpose of this algorithm.

Figure 5.3 illustrates a job (observation schedule) which has been celestially calculated and as can be seen resembles a simple directed path. This directed path outlines a number of tasks which need to be performed at specific times to ensure that tasks are completed successfully. Typically, a job such as the one shown consists of two tasks which act as the starting and finishing points and contain a series of tasks in between which are linked to one another where the successor of one task is the predecessor of the next. Note that the observation schedule contains no branches in its flow and as a consequence the execution of the job becomes merely a sequential series of tasks.

This series of execution tasks is satisfactory for simple jobs where there are no branches, but as explained the algorithms defined thus far have not provided any support for those situations where there exists more than one possible way of achieving the goal. The algorithm presented in section 5.1.1 confirms this by only providing a solution which minimised the amount of time spent performing infrastructure setup tasks by altering the sequence of various tasks within the scenario.

The approach presented here addresses this issue by providing an algorithm capable of taking a job scenario containing a number of valid paths and calculating the optimal path.

In examining the algorithm it is first necessary to add to the definitions outlined in section 5.1.1.2. These definitions are used to assist with the ordering of tasks within a job.

Definition 5.1.2.1.

Let scenario = (s,t,f) be the representation of a job scenario which reflects the structure of a Petri net. The elements within a scenario are as follows:

- *s is a finite set called states and is denoted by scenario.s*
- *t is a finite set called transitions and is denoted by scenario.t. The transitions set is bound by the following condition:* $s \cap t = \phi$
- *f is a set of flows which can move in either direction between states and transitions. The flow set is denoted by scenario.f and is bound by the following condition:*

$$f \leq s \times t \cup t \times s$$

For the remainder of this thesis, let Scenario denote the set of all scenarios as defined above.

Expressing the scenario as a Petri net brings the ability to specify both the origin and destination as well as those tasks which need to be accomplished in between. Additionally, the Petri net structure allows for the richness in semantics needed to represent the predecessors and successors and demonstrates how a task may require or have a number of preparatory tasks which need to be processed before the task itself can be performed. Figure 5.7 provides a graphical representation of a sample scenario and associated dependency information.

The figure displays that the job scenarios which the CORES model makes use of, contain an element of symmetry within them. For each point within the scenario where a 'fan-out' degree is greater than

one (ie. more than one pathway exists), a respective 'fan-in' point is present. This 'fan-in' allows various paths to converge and for the traversal algorithm to perform any calculations necessary to determine the path travelled and to identify the path at which the converging point is considered to be optimum.

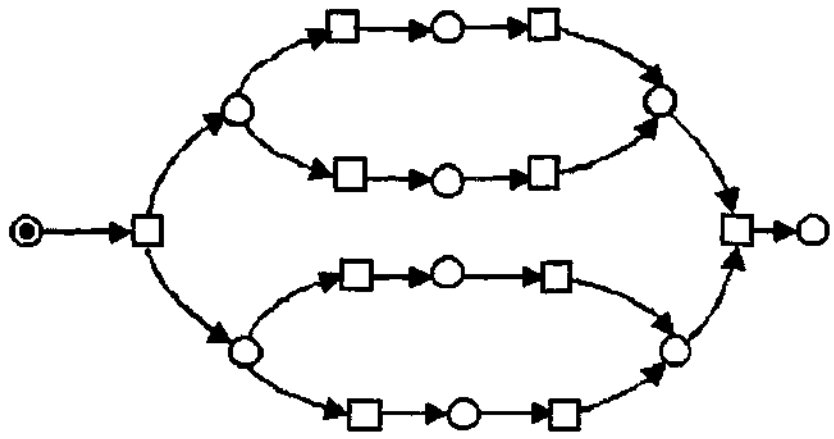


Figure 5.7: CORES Structured Job containing Multiple Dependencies

Having points within the scenario being responsible for controlling the 'fanning-out' and 'fanning-in' of flow paths, forms the basis behind solving the problem of having to correctly navigate multiple paths within the scenario. The following statement addresses this issue by introducing the function *find*(...) which identifies the various nodes within a job which have a fan degree greater than one.

Assumption 5.1.2.1.

To identify the various paths which exist within a job scenario, it is necessary to call upon an external function capable of identifying a node which has a fan-in degree greater than one and matches the same 'type' as the fan-out node given.

Let $js \in Scenario$ *such that* js *reflects a viable job scenario containing multiple paths and the sequence* $seq = js_0, js_1, \ldots, js_k$ *with* $k \in \mathbb{N}$ *be a directed path from* js_0 *to* js_k *iff* $\forall\ i,$ $0 \leq i < k : (js_i, js_{i+1}) \in f$.

A path is deemed to be elementary if all tasks within it are pairwise different. The set of tasks $js_0, js_1, \ldots, js_k$ *is a path sequence and is denoted by* $a(seq)$. *The set of all elementary directed paths from* js_0 *to* js_k *is denoted by the expression* $E(js_0, js_k)$.

The definitions of an elementary path and a set of all elementary directed paths allows a job scenario (refer to figure 5.8) containing a number of paths to be broken down into a series of elementary paths. Figure 5.8 illustrates one of the possible four maximal paths and demonstrates how a maximal path

is constructed from a number of elementary paths. The identification of paths within a scenario allows the traversal time required to navigate the entire scenario to be easily calculated.

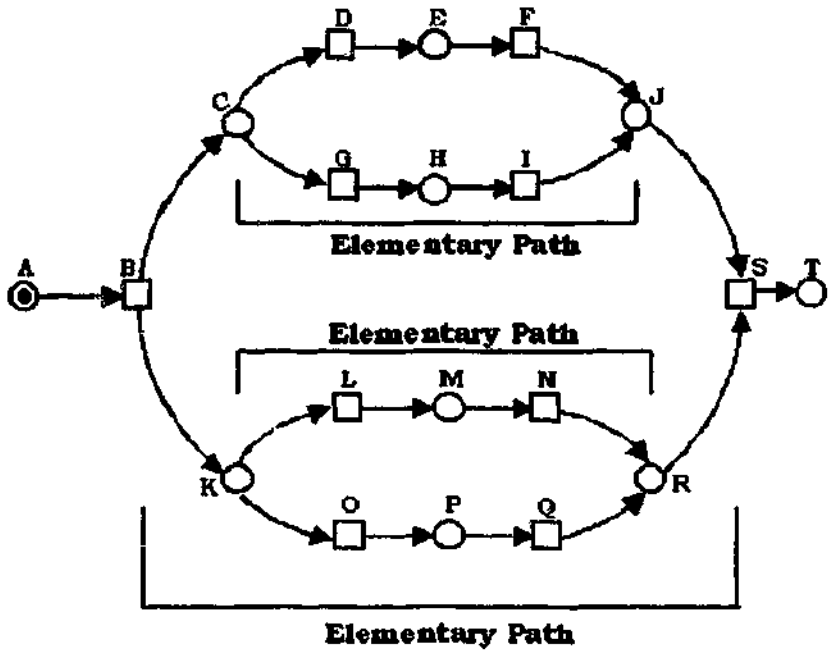


Figure 5.8: Identification of a Direct Path using Elementary Paths within a Job

Problem Statement 5.1.2.1.

Given a node representing a point within a job scenario which has a 'fan-out' degree greater than one, find the corresponding 'fan-in' node which has a degree greater than one and is of the same type. More formally, let js ∈ Scenario which reflects the scenario to be examined and x,y ∈ s or x,y ∈ t.

Algorithmic Solution 5.1.2.1.

The 'fan-in' node is identified by defining the function find: $s \cup t \rightarrow s \cup t$ *such that* $find(x) = y$ *with:*

$find(x) = y$
  where: $x,y \in s \cup t$ and $\mid {}^{\bullet}y \mid > 1$ and
  there exists $seq_0, seq_1 \in E(x,y) : seq_0 \neq seq_1 \wedge a(seq_0) \cap a(seq_1) = \{x,y\}$

Using this function allows the set of all elementary directed paths to be constructed for a particular scenario. With each of the directed paths identified, other functions can be developed to identify the optimal path through a scenario and calculate the total amount of time required to navigate such a path.

Problem Statement 5.1.2.2.

Given a job scenario and two nodes representing a starting and finishing point, find an optimal path through the scenario from the specified starting point to the finishing point while ensuring that each nodes predecessors are satisfied.

More formally, let $js \in Scenario$ *be a scenario to be traversed while the nodes:* $task,dest \in s \cup t$ *represent the respective starting and finishing points.*

Algorithmic Solution 5.1.2.2.

Using mathematical induction rules, it is possible to express the traversal algorithm as:

1. if $task \neq dest \wedge \mid task^{\bullet} \mid = 1$,
   traverse($task,dest$) := <task> $\wedge$ traverse(w, $dest$)
   where $w \in task^{\bullet}$
2. if $task = dest$,
   traverse($task,dest$) := <$task$>
3. if $task \neq dest \wedge \mid task^{\bullet} \mid > 1$,
   if $task \in t$,
   traverse($task,dest$) := <$task$> $\wedge$
   seq (min(traverse(w,x))) $\wedge$
   traverse($findMatch,dest$)
   where $\forall w \in task^{\bullet}$, $\forall x \in {}^{\bullet}findMatch$, $findMatch$ = find($task$)
   if $task \in s$ and processorCount > 1,
   traverse($task,dest$) := <$task$> $\wedge$
   seq (max(traverse(w,x))) $\wedge$
   traverse($findMatch,dest$)
   where $\forall w \in task^{\bullet}$, $\forall x \in {}^{\bullet}findMatch$, $findMatch$ = find($task$)

   if $task \in s$ and processorCount = 1,
   traverse($task,dest$) := <$task$> $\wedge$ traverse($findMatch,dest$)

```
            where findMatch = find(task)
4.  optimal(traverse(w,x)) = min {optimal(traverse(y,z))}
        where y,w ∈ task•, z,x ∈ •findMatch
5.  optimal:  seq s ∪ t → ℝ is defined as:  optimal (σ) = ∑_{n∈σ} min_n
```

From the mathematical induction ruleset, the number of additional steps needed to be performed when dealing with scenarios containing multiple paths can clearly be seen. To deal with such situations it is necessary to identify and process each path using recursion.

This is achieved by having the algorithm use a number of induction rules. Each rule provides for a certain circumstance which may be encountered by the algorithm while traversing the various paths contained within a job. Briefly the rules pertaining to the algorithm are as follows:

The first rule is responsible for processing normal tasks (ie. tasks with one successor) and appending them to the traversed path before continuing. The algorithm advances by calling the *traverse*(...) function upon itself with the tasks *task* and *dest*. Before the actual task is processed it is checked to ensure that it is not the destination. As with any algorithmic approach which makes use of recursive techniques, the algorithm must contain a condition which identifies when recursion is no longer required.

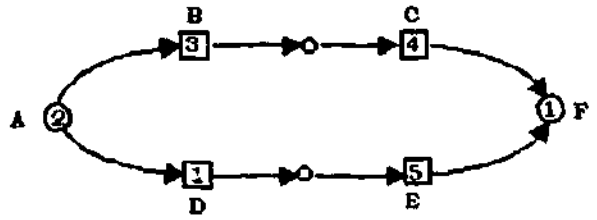
The second induction rule allows the algorithm to stop recursing when the task being examined is equivalent to the destination task. When a match is detected the destination is appended to the sequence of tasks traversed and returned to the optimiser routine.

The third induction rule is responsible for identifying and dealing with those tasks which have more than one successor. These tasks represent branches within a job leading to different paths. How the algorithm handles these paths depends heavily on the type of the task containing the multiple predecessors or successors.

If the task is of an 'OR' type, then the various directed paths which can be derived from the branch are examined. With the examination complete, each sequence is evaluated and the sequence requiring the least amount of time to be processed is selected and appended to the overall traversal path.

If the task is determined to be of an 'AND' type, the algorithm must then determine whether the model was implemented on a system capable of evaluating and processing a series of paths concurrently or sequentially. Knowing this level of concurrency support enables the algorithm to make a decision as to what approach will be taken to process the various paths.

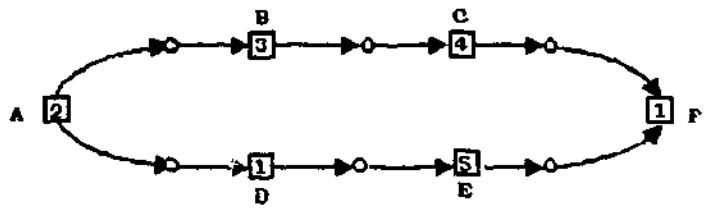
If concurrency support is available the algorithm is able to process the multiple paths in a manner which is similar to the optimum path for an 'OR' task. The difference between the processing of an 'AND' path and an 'OR' path relates to the way in which the converging traversed paths are handled.



PATHS AVAILABLE	PATHS AVAILABLE	SELECTED PATH & TIME	
A,B,C,F	2+3+4+1 =10	A,D,E,F	9
A,D,E,F	2+1+5+1 = 9		

Figure 5.9: Processing an 'OR' Path

For an 'OR' branch the optimum path is calculated by assessing each path (in parallel if available) and determining the path that took the least amount of time to process. For an 'AND' branch the optimum path is calculated by assessing each of the paths (in parallel if available) and determining which path took the longest to complete. The longest path to traverse is the one which will be selected as every path has to be completed before the traversal can continue.



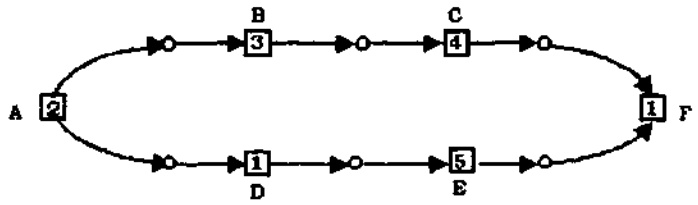
PATH COMPONENTS	TRANSVERSAL TIMES	RECORDED PATH & TIME	
A,B,C,F	2+3+4+1 =10	A,B,C,D,E,F	10
A,D,E,F	2+1+5+1 = 9		

Figure 5.10: Processing an 'AND' Path with Concurrency Support

If concurrency support is not available for the processing of the various 'AND' paths, the algorithm resorts to calculating the total time required to traverse all paths when placed from end to end. The order in which the paths are assessed is irrelevant as all the tasks need to be processed after each other. When the traversal is complete, the traversal time for each of the 'AND' branches is added to the total traversal time for the path. This time and the entire path taken by the 'AND' branches are then appended to the sequence of tasks which have been traversed.

The difference between the ways in which the algorithm calculates the various optimum paths can be seen in the following diagrams. Figure 5.9 depicts how the algorithm moves through a branching sequence considered to be of an 'OR' type. Figure 5.10 illustrates how the algorithm is able to

process a sequence of paths which occur after an 'AND' branch with the use of the CORES model being implemented on a system capable of supporting the concurrent processing of the paths. Figure 5.11 shows how an 'AND' branch is processed when there is no concurrency support to examine the paths. It also shows how all the tasks are assessed by placing one 'AND' branch after another.



PATH COMPONENTS	TRANSVERSAL TIMES	RECORDED PATH & TIME	
A,B,C,F	2+3+4+1 =10	A,B,C,D,E,F	16
A,D,E,F	2+1+5+1 = 9		

Figure 5.11: Processing an 'AND' Path without Concurrency Support

Induction rule four shows how the *optimal*(*traverse*(*task*, *dest*)) function acts as a wrapper by performing various traversals through the job and then selecting the optimal path defined as being the path requiring the least amount of time to traverse.

Induction rule five presents the formal definition of the optimal path. The function *optimal*(...) is defined as taking a collection of 'AND' and 'OR' tasks from a job and returning a value representing the minimum amount of time required to process the optimal path.

5.1.3 Assembling the Overall Schedule

To this point the chapter has concentrated on algorithms responsible for analysing, sequencing and optimising tasks within a specific job. This section introduces a demand-scheduling algorithm which is responsible for examining the broader issue of scheduling a series of jobs and formulating an overall schedule. In the context of the case study presented in chapter 7, the schedule is responsible for coordinating the use of the radio telescope array and its associated resources.

Although a demand-scheduling (first fit) algorithm is used to construct the schedule, it is important to note that other scheduling techniques such as priority or precedence scheduling can be performed to assist with the construction of the schedule. The demand scheduling technique although inefficient when dealing with NP-Hard problems as explained in Krause (1973) was selected due to its ability to allocate resources to time periods when they are first identified as being required. This becomes increasingly important when dealing with scheduling periods that contain extended periods of time where no job can be scheduled (ie. maintenance periods or scheduled downtimes). The scheduling

algorithm presented in this section provides a means for optimised jobs conforming to real-time constraints to be placed into a schedule to allow for their execution.

5.1.3.1 Requirements

With the successful completion of identifying, sequencing and optimising tasks within a job, comes the important task of combining those jobs into the overall schedule. The construction of the schedule brings with it many challenges including the need to ensure that each job can be scheduled successfully and within the specified scheduling time period.

In addition to the initial job scheduling, it may be necessary to schedule the same job a number of times. In providing this functionality the scheduler has to be careful not to allow the scheduling of multiple job instances to starve other unique jobs from being placed into the schedule.

The scheduler is also responsible for calculating a feasible start time where a job can be completed successfully whilst meeting its objectives. This feasible start time is calculated by determining the first available time a job can be successfully run within the scheduling period. Start times of repeated jobs may vary based on the availability within the schedule and the required resources. With regard to the tasks used throughout the case study, consideration to such issues as the position of the radio point sources in respect to the observatory and the resources required by the job impact on the start time. After a start time has been proposed for the job, the business domain rules are applied to ensure that it is scheduled according to the organisations policy. The application of this rule may result in a slight starting time adjustment of the job. The magnitude of this adjustment is dependent upon scheduling commitments already in the schedule and the initial calculated start time.

Domain rules provide end-users or operators with the ability to provide some control over job and task scheduling. Principally, the rules are designed to ensure that jobs are scheduled in accordance with policies dictated by the organisation. An example of a business domain rule implemented for the case study ensures that scheduled jobs begin and finish at the top of the hour (ie. 12:00pm -> 2:00pm). It should be noted that domain rules like the one implemented for the case study can introduce idle periods into the schedule.

To begin the job scheduling process, the scheduler must first carry out an initial appraisal of the data which is going to be placed into the schedule. This appraisal involves calculating the total number of jobs as well as the allocation of the total number of job slots[3]. With preliminary calculations completed, the scheduler is ready to initialise its looping constructs and commence the process of assigning start times to each job instance awaiting scheduling.

The use of the looping constructs ensures that each job has an equal opportunity of being scheduled. This is achieved by having the scheduler's outer loop iterate through jobs while there are still job instances awaiting scheduling. Each time the loop is evaluated the remaining number of job instances

[3]A job slot refers to the number of times an individual job is repeated.

awaiting scheduling is re-calculated. The number of job instances awaiting scheduling is calculated by summing the number of instances left to be scheduled for each job.

Once the scheduler has been passed in a sequence of jobs and has initialised its outer loop, it can start scheduling the jobs instances. This is performed by having the scheduler position itself at the beginning of the job instance sequence and establishing another loop which iterates through each of the jobs.

The second loop allows the scheduler to iterate through each job and determine whether there are any instances awaiting scheduling. Upon detecting that an instance is awaiting scheduling, the model will acquire the required information from the job and calculate the first available time that it can be slotted into the schedule based on resource requirements. After a proposed job start time is calculated, it is assessed to determine whether it can be placed into the schedule. If the proposed starting time corresponds to an available time frame within the schedule, then the job can be scheduled. If for some reason the job can not be placed into the schedule, the scheduler will return an error message.

Before any job is placed into the schedule, the scheduler will first apply its optional business domain rules to the proposed start time to ensure that the job conforms to the organisations policy. After having had timing corrections made, the job instance is added to the schedule and the respective resources are allocated for the period of the job. The successful loading of the job instance into the schedule results in the number of associated instances with the job being decremented by one.

In the event that a job can not be placed into the schedule, the corresponding instances associated with the job are set to zero. If such an action is performed the scheduler will notify the operator that the job can no longer be scheduled and that there will be no more attempts to schedule it.

When the scheduler has finished dealing with a job, it advances to the next in the sequence and checks to see whether there are any instances awaiting scheduling. This automatic advancement takes place so as to avoid any one particular job from starving the resources from those remaining. The looping through each job continues until all have been visited.

Once the scheduler has iterated through all the jobs, it returns to the point of calculating the number of job instances remaining to be scheduled. It then continues until there are no more instances to be placed or scheduled.

The next section concentrates on formalising the algorithm, the information provided to the scheduling algorithm and the external functions utilitised that allow the scheduler to build the schedule.

5.1.3.2 Formal Definition

In order for the scheduler to be able to successfully schedule jobs it is necessary for it to use a number of external functions. This section deals with the algorithm responsible for building the

overall schedule whilst at the same time formally defining the various external functions that the scheduler uses.

The examination of the scheduling routine begins with one of the most important functions for any scheduling activity. This is the function *duration*(...) which is defined externally and is responsible for calculating the duration of a job awaiting scheduling. Such a calculation is performed by determining the total amount of time spent performing tasks within the job. The job which is provided to the *duration*(...) function contains both the planned tasks and those responsible for ensuring that the infrastructure is setup correctly.

Assumption 5.1.3.1.

To calculate the duration of a job, it is necessary to make use of an external function which returns the number of seconds spent performing all of the tasks. More formally, let $n \in \mathbb{N}$ *and* $j = o_0, \ldots, o_{n-1}$ *be the observation schedule to perform with* $o_i \in RpsObs$ *and where the external function is defined as duration:* $RpsObs \rightarrow \mathbb{R}$ *such that* $duration(j) = t$ *where* $(t \geq 0)$ *is the time required to perform the job. The duration*(...) *routine calculates the sum:*

$$\sum_{i=0}^{n} j_i.f - j_i.t$$

Although the *duration*(...) function calculates the total amount of time required to complete a job, the algorithm needs to be mindful of situations where the job contains tasks which can be performed in parallel. Where a path is detected to contain tasks that have to be performed in parallel or contains multiple pathways, the duration routine will make use of the traversal routine outlined in section 5.1.2 to assist in calculating the path which should be taken. The path determined allows the *optimal*(...) function to identify the time required to traverse the path.

Apart from calculating the job duration, the scheduler is responsible for determining a time at which it can be placed into the schedule. Determination of the start time requires the scheduler to find a time at which the tasks within a job can be completed successfully while having regard to the resources specified in the job request. After a proposed starting time has been calculated, the scheduler ensures there are no other jobs or tasks within the schedule which would stop the proposed job from being inserted. If it is determined a resource is not available, the scheduler will ignore the proposed start time and attempt to find another capable of supporting it.

The calculation of the starting time for the schedule is handled by an external function known as *determineJobStartTime*(...). This function takes the job to be performed, the schedule and a set of resources and determines a time at which it can be successfully placed into the schedule ensuring that all the requirements of the job and schedule are met.

In examining the algorithm, it is necessary to provide some additional definitions. It is desirable to consider how the resources within the model are defined and how the scheduler represents its own scheduling information.

Definition 5.1.3.1.

For every task performed, a subset of the resources available to the model are associated with the job for its entire duration. This resource information is stored within the scheduler along with the job details to ensure that those resources requested are available. Let the model resources be defined such that Resources $= r_0,\ldots,r_{n-1}$ *where* $r_i \in \mathbb{N}$ *represents a unique resource identifier and where* r_i *(*$0 \leq i < rl$*,* $i \in \mathbb{N}$*) and where* $rl \in \mathbb{N}$ *signifies the resource limit for the model (ie. number of resources available).*

Each of these resource identifiers is responsible for identifying to the scheduler's real-time execution engine the appropriate resource which is charged with the responsibility of executing the task.

Definition 5.1.3.2.

For each job executed by the scheduler, a corresponding entry in the schedule co-ordinates the resources required and the time at which the job is to be executed. Let the schedule entry be a tuple such that scheduleEntry $= \{j,r,t,d\}$ *with* $j \in$ *RpsObs,* $r \subseteq$ *Resources,* $t \in \mathbb{R}$ *(*$0 \leq t <$ *Max_Period) where Max_Period* $\in \mathbb{N}$ *represents the bounds of the schedule and* $d \in \mathbb{R}$ *(*$0 < d <$ *Max_Period). The elements within schedule include:*

- *j is a sequenced observation schedule containing tasks to be performed*
- *r denotes the resources required by the job*
- *t represents the time when the job is to be performed. The time is expressed as the number of seconds which have elapsed since epoch (1st January 1970)*
- *d is the duration in seconds that the job will run*

For the remainder of this thesis, ScheduleEntry denotes the set of all schedule entries as defined above.

Assumption 5.1.3.2.

In order to place a job into the schedule, it is necessary to first determine a time at which the job can be placed into the schedule. The time calculated also needs to ensure that all the tasks within the job can be completed successfully, that the resources are available and that the schedule has enough free space to fit the job in.

To determine this time an external function is used to calculate an appropriate starting time for the job given all of the preconditions mentioned above. The function is defined as determineJobStartTime: $RpsObs \times ScheduleEntry \times Resources \times \mathbb{R} \rightarrow \mathbb{R}$ *such that determineJobStartTime(j,s,r,d)* $\rightarrow t$ *where* $(t \geq 0)$ *is the time nominated as the starting time for the job.*

The *determineJobStartTime*(...) calculates the respective time by taking the time at which the scheduling period begins and searching for a time where the job can be executed successfully and

where the start time does not exceed the schedule's Max_Period. The start time, once determined, allows the scheduler to calculate the time at which the job will finish. With an approximate start and finish time, the scheduler is able to check the resources required by the job and ensure that they are available for the specified time period.

If the resources are available and there are no other jobs scheduled for the same period, the scheduler returns the calculated starting time to the calling process so that the optional business domain rules can be applied. If the business rules invalidate the start time, the scheduler will advance to the proposed finishing time and continue the search for another suitable time where the job can be performed. This search for an appropriate start time concludes when the scheduler either identifies that there is no more time left in the scheduling period to perform the job or another suitable time is found. If a suitable time can not be found, an error condition will be raised and the operator notified.

With both the start and finishing times calculated, the scheduler applies the optional business domain rules before scheduling the job. The domain rules are applied in the function known as *prepareJobTimes*(...) which takes two parameters. One parameter is the job to be performed and the other is the calculated start time provided by the scheduler. Given these parameters the function performs a number of calculations and provides both the start and finish times for the job as well as returning a modified version of the job which conforms to the organisations policy.

For the case study referred to in chapter 7, the Australian Telescope National Facility (ATNF) specifies the business domain rule that all jobs (observation schedules) must start and finish on the hour to ensure schedule cohesion. It should be noted that the application of the ATNF business rule may in certain circumstances result in an extended period of idle time as jobs may not actually be scheduled to start until the very end of the hour. In such situations the resources allocated to the job may remain idle for a considerable portion of the hour.

Assumption 5.1.3.3.

Before a job can be placed into the schedule, the determined start time must first be assessed by the scheduling process to ensure that it meets any business domain rules which might be enforced by the scheduler. To assess this time and if necessary make adjustments to both the starting time and the job itself, an external function is provided. This function is defined as prepareJobStartTime: $RpsObs \times Resources \times \mathbb{R} \rightarrow RpsObs \times \mathbb{R} \times \mathbb{R} \times RpsObs$ *such that* $prepareJobStartTime(j, r, startTime) \rightarrow modifiedStartTime \times modifiedFinishTime \times j'$ *where* $(modifiedStartTime \leq startTime < modifiedFinishTime \leq Max\_Period)$.

The *prepareJobTimes*(...) function works by examining the proposed starting time for the job and calculating the time difference in seconds between the start time of the job and the top of the hour. Once the number of seconds past the hour has been determined, a process of examining each previous second and the resources being used at that second is performed until the algorithm works its way back to the start of the hour. If all of the resources requested by the job are available within this period, the scheduler will insert an **IDLE** task at the beginning of the job which forms the modified job and informs the scheduler runtime engine[4] that the modified job and the resources associated with it are required from the top of the hour.

In addition to the task being inserted at the beginning of the modified job, a similar task is appended to the modified job to ensure that no other jobs are placed into the schedule. The idle tasks are used to expand the time allocated to the job in the schedule. Within these **IDLE** tasks is a period of time where the task remains idle before it can continue to process other tasks.

Problem Statement 5.1.3.1.

Given a sequence of jobs to schedule, build a schedule which allows each job to be completed successfully while taking into consideration the respective resources that a particular job uses and any organisational rules which the scheduler may be employing. More formally, let: $n \in \mathbb{N}$ *and* $w = j_0, \ldots, j_{n-1}$ *be the work which is to be performed with* $w_i \in RpsObs$ $(0 \leq i < n)$, *build a schedule such that:*

1. *jr is the set of resources required for each job, that is* $jr = r_0, \ldots, r_{n-1}$ *with* $r_i \subseteq Resources$ *and where each indice corresponds to a respective job within the work set* $(0 < i < n)$
2. *jt is the set of instances that a particular job is to be repeated within the schedule, that is* $jt = c_0, \ldots, c_n$ *with* $c_i \in \mathbb{N}$ *and each indice corresponding to a job which is to be performed*
3. *schedule is the set of schedule entries indicating the schedule, that is* $schedule = e_0, \ldots, e_n$ *with* $e_i \in ScheduleEntry$

[4] Determines when tasks are to be executed and the resources that they are to be dispatched to.

4. $determineJobStartTime(j_i, schedule, jr_i, duration(j_i)) \geq 0$

Algorithmic Solution 5.1.3.1.

On – Demand Scheduling Algorithm:

```
startTime, startTime', finishTime ∈ ℝ;
modifiedJob ∈ RpsObs;

for all jobs in:
     w.j_i, ..., w.j_{i-1}  (0 ≤ i ≤ n)

schedule := {φ};
while ((∑_{l=0}^{|jt|} jt.c_l) > 0)
     for all i:
        if jt.c_i > 0 then
           startTime := determineJobStartTime(w.j_i, schedule, jr.r_i, duration(w.j_i));
           if startTime < 0 then
              jt.c_i := 0;
           end
           else
              prepareJobEntry(w.j_i, jr.r_i, startTime) → startTime' × finishTime × modifiedJob;
              if finishTime < Max_Period then
                 schedule := schedule ∧ {modifiedJob, jr.r_i, startTime', duration(modifiedJob)};
              end
              jt.c_i := jt.c_i − 1;
           end
        end
     end
end
```

Like algorithm 5.1.1.1, it is possible to calculate the level of complexity in algorithm 5.1.3.1 by determining the relationship between the number of inputs provided and the number of times that instructions have to be executed. Specifically, the complexity is determined by the looping elements contained within an algorithm, especially those containing nested loops.

With each individual instruction being given a complexity of one and looping constraints requiring three instructions (ie. loop construction, evaluation of the iterator, incrementing the iterator), it is possible to derive a formula which allows express the complexity of an algorithm. For algorithm 5.1.3.1 the complexity is defined as $4t^2 + 4j + 1$ where t represents the maximum number of tasks in a job and j represents the maximum number of jobs which have to be performed.

t	j	Instruction Count	Growth
1	1	19	0
2	2	65	46
3	3	139	74
4	4	241	102
...	...	...	...
8	14	313	-

Table 5.2: Complexity of Constructing a Schedule

Table 5.2 illustrates the complexity of the algorithm by representing the growth in the number of instructions performed as the number of jobs and tasks increases. The last row in the table represents typical values used throughout the simulation of CORES and is indicative of the number of tasks and jobs which might be scheduled over a fortnightly period.

As the growth rate and complexity algorithm indicate, the number of tasks needed to be performed impact heavily on the overall performance. A significant contribution to the complexity of the algorithm is attributed to the $4t^2$ element of the complexity formula which represents the scheduling of the tasks contained within a job. The overall magnitude of the algorithm however can be expressed as $O(t^2)$ although the complexity could potentially increase if external functions which are referenced introduce additional looping instructions.

5.2 Configuration Manager

Apart from the manipulation of tasks and the scheduling of jobs, the model proposed in this thesis is also responsible for interacting with registered components and providing them with reconfiguration management services while ensuring that they conform and operate within their real-time constraints defined in sections 5.1.1 and 5.1.2. This is achieved through the establishment of a configuration manager which is responsible for coordinating, controlling and manipulating the model and the configuration information associated with each component.

5.2.1 Role within the Model

In addition to being able to interact with components, the configuration manager enables end-users or operators to interact with the model and control the manner in which method calls are handled when a request is made for a service located on an unavailable component.

Situations resulting in the component being unavailable include a component undergoing a reconfiguring operation or a component which has been or is being disabled. The current version of the CORES model only enables support for dynamically updating components on the same machine

or allowing the relocation of components within the realm of the model. The ability to update a component provides software developers with the opportunity to introduce additional or improved functionality which in turn allows the implementation of the model to meet the users requirements or the organisations needs. The only restriction imposed by the model on updating a component is the introduction of additional functionality. New functions or improved functionality which is introduced to the component has to implemented in terms of the old component's interface definition.

This approach is similar to the manner in which the COM/DCOM models works ensuring compatibility with older clients which may be operating throughout the model. Using this model dictates that once an interface is published, all future implementations of the server will honour the same interface. During all reconfiguration activities, the configuration manager is responsible for handling the exchange of state information between the old and new components. In the event of a component being relocated from one location to another, it is the responsibility of the configuration manager to receive all of the method calls which arrive at the old components location and redirect them to the new location once the component is initialised and enabled. The forwarding of method calls only takes place upon the successful completion of all the reconfiguration operations.

One final situation that the configuration manager may encounter is the possibility of a component being disabled. In this event, the configuration manager must deal with all of the inbound method calls and perform the appropriate actions which match the dynamic controls specified within the method call when the component is unable to process them (refer to section 5.3).

In addition to handling the various conditions that might arise with each component, the model also provides for the configuration manager to act as an intermediary between the application program (client) and the component (server). This relationship allows the client, the configuration manager and the component to exchange information pertaining to the current configuration of the component.

The bi-directional communication channel established between the configuration manager and the client allows for the transfer of the operational status of the component, the method calls and parameters that the server is to execute (relaying the method calls). It also allows for exceptions which the configuration manager may receive and needs to send onto the client (relaying of exceptions) or the results of method calls which have been executed by the server and are ready to be sent back to the component. The other bi-directional communication channel established between the configuration manager and the server is used to transfer component configuration data, method calls and parameters from the client which are to be performed on the server, results of method calls that the client has requested the server to run or exceptions destined for the configuration manager and/or client.

The configuration manager also provides a number of interfaces which clients can use to specify reconfiguration commands. These commands include being able to place a component into a quiescent state, allowing the configuration manager to perform reconfiguration changes on it as well

as providing a means to manipulate the internal state data held within. Another reconfiguration interface supports the ability to allow the client to specify actions that should be taken when a component is unable to accept or process an incoming method call.

During the initial CORES model startup sequence, the configuration manager is responsible for instantiating a connection between itself and the component that it is to manage. This process is only performed when CORES is first initialised or when a new component (server) is introduced into the system requiring the need for the component to obtain where possible the relevant state data. In cases where state data is required the instantiation process will arrange and coordinate the loading.

5.2.2 Architecture of the Model

The CORES model extends the traditional two tiered client/server approach by introducing an additional tier represented as the configuration manager. This tier is inserted between the client and the server and is used to form a hybrid two tiered architecture. Using this architectural approach allows the client and configuration manager to work together and for the configuration manager and component to work together in two separate groups. Figure 5.12 illustrates how the CORES model introduces the hybrid architecture.

The introduction of the third tier brings with it a reformed approach to the transmission of data to and from the client and component. Data destined for the component from the client must now pass through the configuration manager which examines the incoming data stream for commands that may require specific reconfiguration actions. If the data stream does not contain any instructions which are specific for the configuration manager, the data stream is forwarded on by the configuration manager to the component for processing. The same process is followed when data is being returned to the client from the component.

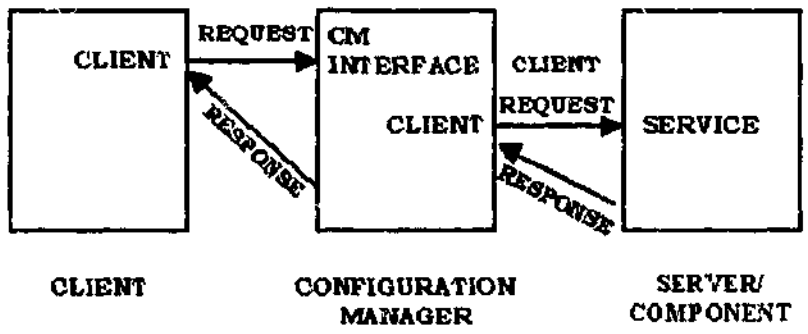


Figure 5.12: Client/CORES/Server Architecture

As figure 5.12 depicts, the configuration manager plays an important role in allowing data to flow between the client and the component. With a component being a dynamic object and capable of being removed or relocated within the model at any point in time (without informing the clients) comes the need to provide a mechanism where clients can send method calls to a component regardless of its location. To achieve this, the CORES model, along with the underlying distributed object framework which handles objects throughout the entire model, ensures that a configuration manager for a component is always located at the same 'well known' virtual memory address even though the component itself may be shifted. Providing the virtual memory address allows clients to be able to interact with the configuration manager regardless of the state that the component itself is in. Once these method calls arrive at the configuration manager, it is the responsibility of the manager to forward them onto the component and to handle the various states that the component may be in. Discovering the component in a unusable state results in the execution of appropriate operations dictating what happens in the event that method calls cannot be processed.

5.2.3 Interfaces provided by the Model

The integration of the CORES model with a distributed object framework allows numerous services to be provided to clients. One of these services mentioned previously is the ability to be able to redirect incoming method calls to the component regardless of where the component is located. The only piece of information the client needs to know is the 'well-known' address of the configuration manager. The configuration manager takes care of redirecting the method call to the component or handling the method call during reconfiguration operations.

From an interface perspective the configuration manager operates in two distinct modes. The first and most common is known as the 'transparent mode'. This mode is responsible for passing data between the client and the server and does not require the configuration manager to actively manage or manipulate the configuration information associated with the component. The second mode of operation is known as the 'configuration mode' which is where the configuration manager is specifically performing operations relating to the ongoing management of configuration data associated with the component. This mode may be triggered by a request from the client or from the model itself.

As part of the 'configuration mode', the configuration manager makes use of a number of interfaces responsible for inquiring and controlling the internal state of a component. These interfaces control whether or not the component is put into a logical state supporting reconfiguration operations. The configuration mode also contains interfaces available to manipulate the internal state of the component.

Figure 5.13 illustrates the various interactions which take place between the client, configuration manager and the implementation of a component known as a server when a configuration manager receives a request to replace the current server with another located elsewhere within the model.

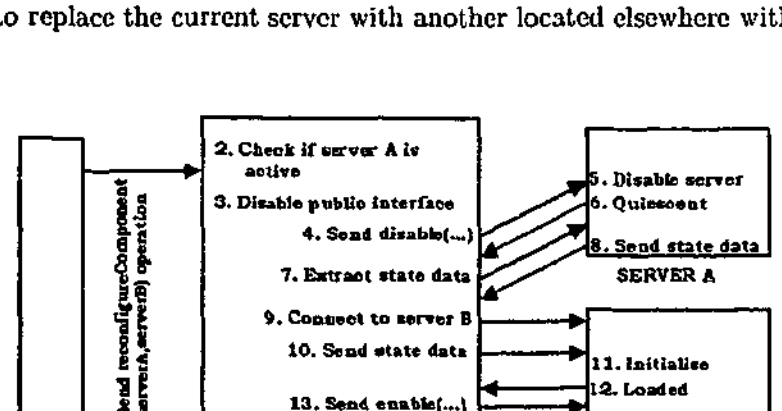


Figure 5.13: Configuration Manager performing a Component Reconfiguration

The process of replacing a component already in use with another begins with the configuration manager receiving a request (from a client) informing it that the component providing the service needs to be deactivated and replaced. Contained within the initial reconfiguration request is the component which will replace the one currently in use. Upon receiving the request, the configuration manager checks the status of the current component to determine whether or not it has already been deactivated for other reasons. In the event it has been deactivated the configuration manager will not attempt to issue any instructions to put it into a quiescent mode.

Where it is marked as being active, the configuration manager will take the appropriate action to place the component into a quiescent mode. Before this is undertaken, the configuration manager disables its own public interface to the component. Performing this operation allows the configuration manager to trap and handle (if needed) all of the incoming method calls directed towards the component.

After disabling the public interface, the configuration manager turns its attention to the component to be replaced. As it can be performing autonomous operations or processing a method call from a client, it too must be informed of the pending shutdown and replacement operation. This becomes a significant issue when state data has to be transferred from one component to another. To facilitate this exchange, the configuration manager sends a *disable*(...) method call from itself to the component. On receiving this, the component slowly moves towards a quiescent state as explained in Kramer and Magee (1990) and is used in such architectures as described in Goudarzi (1999) where

the internal workings wherever possible are brought to a graceful standstill so that the state data can be transferred or preserved.

After a state of quiescence has been achieved, it transmits a confirmation message to the configuration manager to confirm that all of the functions have entered a quiescent mode. If a confirmation message is not received within a certain period of time, the configuration manager assumes that a total quiescent state could not be achieved. This does not prevent the configuration manager from extracting the data, it merely increases a time delay before the extraction operation can commence.

When the configuration manager receives confirmation of quiescence or that the timeout for quiescence has expired, the configuration manager will transmit a request for the state data to be either stored in a persistent store (for future access) or for it to be streamed back to the configuration manager so that it can be sent to the new component. Figure 5.13 illustrates how the data can be sent from the component to the configuration manager as a stream.

With data now available, the configuration manager is able to connect to the replacement component (this may require instantiating the component if it has not been referenced before) and transfer the state data to it. Once the component replacement has loaded the state data, it may have to perform a series of operations to change itself into a similar logical state as its predecessor. With the state data loaded and the correct logical state achieved, it transmits a message to the configuration manager to inform it that it is ready to accept incoming method calls.

The configuration manager will not proceed until a signal is received that it is ready. This condition cannot be safe guarded with a timeout operation as it can take a variable amount of time to establish a logical state similar to its predecessor. With the acknowledgment that the state data has been loaded, the configuration manager sends the method call to *enable* the component and at the same time enables its own public interface which it disabled at the start of the reconfiguration process. Once the public interface is enabled, the configuration manager processes any method calls that were blocked whilst waiting for the component to become operational and routes the corresponding method calls on to the correct component. The configuration manager then returns to operating in 'transparent mode' while at the same time listening to all communications and events which may trigger a subsequent switch back to the 'configuration mode'.

5.2.4 Sequencing within the Model

An important aspect in handling reconfiguration activities is ensuring the inherent logical sequencing of information and operations flowing between the client and component. This is certainly of some concern when coordinating communications between the configuration manager and component.

5.2.4.1 Ensuring Logical Sequence of Interface Calls

A factor when dealing with distributed systems, especially those dispersed over large areas, is the use of expensive[5] communication links. In dealing with such environments, it is important to ensure the economical use of interfaces located on remote components. Ensuring the logical sequencing of interface calls and operations assists in minimising the use of expensive links for method calls which are irrelevant or used out of sequence.

CORES addresses this issue by incorporating additional functionality into the configuration manager using the approach proposed in Watkins and Thompson (1998) which introduced the concept of a proxy where method calls pass through (locally if possible) before being dispatched to the remote server. The proxy is used in such situations to detect incorrectly sequenced operations with the help of cached data variables which examine the current method call against the expected[6] state of the server. After validating the state of the component and checking to see if the method call is expected, it is passed through to the component and the proxy adjusts its cached data variables to reflect the new state.

In a case where a method call is improperly sequenced, the internal state within the configuration manager traps the method call and returns it to the client. This approach ensures CORES does not knowingly commit resources to a method call which will fail as a result of being incorrectly sequenced.

5.2.4.2 Ensuring Correct Sequence within the Configuration Manager

Based on the architecture of the configuration manager outlined in section 5.2.2, it is evident that the configuration manager makes use of a number of operations to perform a reconfiguration. These operations are required to be performed in a sequential order so as to be able to facilitate the successful management of configuration information and to coordinate the exchange of data between the configuration manager and component.

To ensure the level of sequencing required to manage reconfiguration operations, each individual operation performed within the configuration manager is designed to ensure that one operation completes before another is started. This ensures that the sequencing of the various reconfiguration operations is correct.

In addition to the logical sequencing of configuration commands, the internal configuration state of the component must also be strictly controlled. To provide this level of control and to ensure that it will not be interrupted by any method calls during its reconfiguration, it is placed into a quiescent state. This state is engineered through a series of routines built into the configuration manager

[5]Expensive in the terms of operating costs and/or the time required for data to traverse the link.
[6]The term 'expected' refers to the component being in a suitable logical state to receive the method call.

and component. These routines, discussed in Kramer and Magee (1990) along with the sequencing operations, are important when a consistent component and model is required.

5.2.5 Exceptions within the Model

The handling of exceptions within the distributed environment has always been a challenge. The problems associated with exceptions traditionally center upon trying to get an exception raised on a remote server located anywhere on the network and running any combination of operating systems and programming languages and architectures and then getting it back to the client. An additional process complication is that not all programming languages used for either the client or server are natively capable of supporting exceptions.

The two major organisations behind the distributed object framework, Microsoft initially with COM/DCOM and now with the .NET framework and the Object Management Group (OMG) with CORBA have approached the problem in two distinct ways. The introduction of the new .NET framework however does see the two organisations unifying their approach.

The DCOM architecture approach to dealing with errors centres on a 32-bit data structure known as a HRESULT. Every time a method call is made within DCOM, the HRESULT returned from the method call needs to be examined to determine if an error has occurred. Unfortunately, this approach is totally reliant on the client application checking the return value, rather than having the architecture force the client to deal with the problem when it arises.

CORBA takes a more direct approach to the handling of exceptions. Under CORBA, if an exception occurs, an exception object is instantiated within the server object and is returned to the client. Once the exception is transferred into the clients address space it will continue to progress up the stack of the client program in search of a handler. If the exception can reach the top of the client's stack, the stack in which the client program is executing is collapsed and the program terminates. If a handler does exist then the exception is handled by the corresponding handler and the client program continues to operate.

Additionally, CORBA provides an alternative approach to handling exceptions for those languages that do not directly support exceptions but do have a language mapping to the CORBA framework. When making a remote call the CORBA system passes a `SystemEnvironment` data structure which can be modified by the server to reflect that there has been a problem with the execution of a method call. Upon the return of the method call, the client can check the `SystemEnvironment` data structure to identify whether an error occurred and if so what additional information is available. In this regard, CORBA's `SystemEnvironment` data structure and COM/DCOM's HRESULT error handling system are similar as both require the client to be vigilant when checking the return results from method calls.

The new architecture from Microsoft known as .NET addresses this issue by adopting the same termination exception mode used in CORBA. The .NET framework now forces developers to respond to exceptions as they happen during the execution of the method call as opposed to deferring the problem to a later date. Additionally, most languages which support the .NET framework now have provisions for native exception handling.

Although CORES is capable of handling exceptions, it is unable to directly facilitate the transfer of exception information from server to client. This inability is the result of the underlying configuration manager architecture. As the configuration manager is responsible for relaying operations from the client to component, it is the configuration manager which receives the initial exception from the component. If an exception arrives at the configuration manager which is not expected, it will attempt to interpret the exception as normal. Failing to interpret the exception results in the configuration manager issuing a `rethrow(...)` call which sends the exception onto the client in an attempt for it to handle the error condition.

5.3 Dynamic Controls

One unique feature of the CORES model is the ability to provide end-users, developers and operators alike with an ability to specify and control the actions that should be taken if a client tries to send a method call to a component which is not available or is in a quiescent state as a result of a reconfiguration operation. CORES provides three dynamic controls that can be used to dictate the behaviour of method calls in these circumstances. These controls are:

- WillWait
- NoWait
- QoSWait

With the introduction of these controls, CORES provides the unique ability to exercise more control over how to deal with method calls to components and servers. Based on the configuration management systems reviewed in chapter 3, most other models and implementations do not provide flexibility for end-users or operators to exercise any level of control. In these situations, the detection of a component not being available results in the blocking of all incoming method calls destined for it. The actual processing of the method calls is only performed when the component returns back into service. CORES prevents this occurrence by use of its dynamic controls. These controls implemented by the model provide a pivotal role in those systems operating under real-time constraints and where there is an instant need to know if a method call can or can not be performed by a component.

5.3.1 Dynamic Control Options

This section addresses each of the dynamic control options available within the CORES model and explains situations in which it is envisaged that using the control would provide end-users or operators with additional flexibility.

5.3.1.1 WillWait

The 'WillWait' dynamic control operation provides end-users and operators with the traditional functionality found in many configuration managers. The 'WillWait' option allows a method call to be blocked at the configuration manager and to remain blocked indefinitely until the configuration manager is informed by the component that it is ready to receive requests.

As a consequence of its blocking characteristics, the 'WillWait' directive is not recommended for real-time systems as method calls sent to servers which are not available could block indefinitely and may result in certain methods missing their time sensitive cues. Additionally, this control does not provide the end-user or operator with any relevant feedback as to why a method call can not be processed.

5.3.1.2 NoWait

The 'NoWait' dynamic control operation is suited to those systems, end-users and operators who require accurate and timely information about the status of their components. As the 'NoWait' name suggests, the directive does not wait for a component to become available. If the component is unavailable at the time the method call is sent the calling process is notified through the exception mechanism. This situation is useful for those environments which operate in real-time environments and where the status of the component and method call is required in a timely fashion so that alternative actions can be performed if necessary.

The exception thrown to indicate that the component is unavailable may contain additional information such as the reason for a component being unavailable and possibly offer a time at which it may become available again. It is important to note that the CORES model places no compulsion on the configuration manager handling the unavailable component to provide any additional information. If the information is available, the appropriate fields within the exception will be populated, otherwise a standard exception is returned. It is up to the component developer to decide how much information is returned to end-users or operators.

5.3.1.3 QoSWait

The 'QoSWait' directive provides end-users and operators with the combined flexibility of the 'Will-Wait' and 'NoWait' directives. In addition the 'QoSWait' control provides additional options that can be used to handle what happens when a method call is sent to a component not ready to process requests.

Unlike other dynamic controls, the 'QoSWait' directive in some cases allows the method call to wait until a certain Quality of Service (QoS) condition is met. This condition may include waiting for another resource to become available or relate to a load value having to reduce throughout the implementation of the system before the method call can be processed. Additionally, the 'QoSWait' directive can be used in situations where if the component is not available within a certain period of time, the method call will give up in its attempt to have the component process the action. In this event, an exception will be thrown indicating the reason. The exception may contain additional information about the method call being rejected but the CORES model places no compulsion on returning such information. The flexibility of the 'QoSWait' control can potentially allow method calls to be tied into other environments which have their own strict QoS servicing requirements or constraints.

5.4 Limitations

Although the CORES model brings with it new functionality providing additional control to end-users and operators, it does not provide a complete reconfiguration management solution. CORES was principally designed to provide a new level of control over method calls in real-time systems and to allow the implementation of the CORES model to be incorporated into other reconfiguration management systems which have been developed. By adopting this design approach, CORES as an individual entity does have some limitations.

5.4.1 Configuration Manager and Resources

One such limitation of the model is visible when you examine figure 5.12. It can be seen in the figure that the CORES model introduces an extra level of abstraction between client and the component. The introduction of the configuration manager which provides this abstraction brings with it the resource burdens required to maintain an additional component. In large environments where the model is deployed this may result in hundreds of configuration managers looking after their own specific component. An ideal solution to reducing the burden of the configuration manager on the implementation of the model could be to incorporate the role of the configuration manager into the client, the distributed object framework or the component itself.

If the idea of incorporating the configuration manager into the role of the component was to be adopted, a new or modified architecture would have to be developed to enable separation of the implementation from the configuration manager. Such an architecture would then allow the implementation of the component to 'detach' itself from the configuration management section and be replaced with a new implementation. A similar architecture to the one required can be found in the SOFA/DCUP environment (Plasil, Balek, and Janecek 1998) where the component implementation and the configuration management section are in the same object.

5.4.2 Task Ordering

As part of its role within CORES, the scheduler is responsible for looking at a series of tasks which belong to a job and rearranging those tasks until an optimised sequence is found. However when dealing with jobs that contain no more than one path to the destination, the CORES scheduler will examine each task as a separate entity and hence provide no support for dependency arrangements. With no dependency information being processed, each task is treated as a self contained action. This behaviour is not exhibited when there is more than one path to the destination as an ordered sequence is implied.

The scheduler does however provide support for the end user to provide a sequence of tasks in a predefined order and have them inserted into the master schedule as given. This process is normally used when the end-user or operator has already sequenced the tasks external to the model. Using this same approach would allow operators or end-users to load a series of tasks dependent upon one another into the model. The only disadvantage of this process is that no sequencing would be performed.

5.4.3 Elimination of Invalid Task Combinations/Jobs

After the initial identification of tasks within a job, CORES is responsible for generating a list which represents every possible permutation on how the job can be executed. From the list the model identifies which job sequences or scenarios are deemed to be valid. It is the application of this validity rule which highlights the limitation of CORES when it comes to the elimination of invalid job scenarios.

While processing the permutation list, CORES makes use of a simplistic validity rule which lays out the tasks to be performed in memory and determines whether they can be completed successfully. If the sequence of tasks is determined not to be valid, CORES rejects the sequence and moves onto the next permutation. This process continues until all the permutations have been examined. When examined further, the job scenario may have been deemed invalid as a result of one or more tasks being located in the wrong sequence. CORES could improve its validation routine with regard to its workload and efficiency by determining at what point in the sequence the scenario was ruled

invalid. With this information, the validation routine could eliminate similar scenarios resulting in a significant drop in the total amount of processing time required to identify valid scenarios.

5.4.4 Scheduling of Jobs into Master Schedule

Once a job has been sequenced and optimised, it is passed to the scheduler so that the tasks within it can be scheduled to execute at specific time points. To provide this scheduling service, CORES makes use of an opportunistic scheduling algorithm. Using this algorithm allows the scheduler to identify the first available location within the schedule where the time is correct, the job can be completed successfully and the resources available. As explained in Blazewicz, Ecker, Pesch, Schmidt, and Weglarz (1996) and Brucker (1998) such scheduling algorithms are used within environments where there is no guarantee of contiguous free space being available in the schedule. These algorithms are also used when there are periods within the schedule which contain operations that can effect the availability (ie. scheduled maintenance). The consequences of using such an algorithm can potentially lead to a schedule which is not totally optimised from an overall job perspective, but where each job within the schedule contains a list of tasks which have been individually optimised.

5.4.5 Vulnerability of Configuration Manager

As the configuration manager is a separate entity from the client and the server, additional precautions need to be taken to preserve the configuration data of the system. One such precaution would be to have the configuration manager replicated throughout the distributed object framework. This replication would ensure that a client could always communicate with a server through a configuration manager. In the case of a fault, the distributed object framework, which CORES uses, could activate a replicated configuration manager and handle the redirection of method calls sent from the client to the configuration manager.

As designed, the CORES model provides no replication or heartbeat services to ensure the longevity of the configuration manager. Such support would need to be incorporated into the CORES model or distributed object framework before CORES could be integrated into any mission-critical environment.

5.5 Chapter Summary

This chapter has focused upon the introduction of the Component Oriented Reconfiguration Environment & Scheduling (CORES) model and has identified those areas that form a pivotal role in its definition. These areas include the formal definition of algorithms used by the model to represent tasks, jobs and schedules as well as introducing the concept of a configuration manager to perform

various actions during periods where components are unavailable. The CORES model also introduces a set of controls which end-users or operators can use to govern the actions of method calls when they encounter components that are unable to accept their request.

To begin the definition of the CORES model, the chapter formally defined those algorithms responsible for sequencing and scheduling tasks and jobs while at the same time providing support to ensure that real-time constraints operating within the model are met. The algorithms which make up the CORES model can be broken into a number of groups. These groups include sequencing, traversing and processing jobs which contain only one path from beginning to end, sequencing and traversing jobs which contain multiple paths from beginning to end and an algorithm responsible for placing these sequenced jobs into a schedule.

The first algorithm formally defined shows how tasks within observation schedules are arranged in such a manner as to allow the preceding task to be completed before the subsequent task can begin. This re-arrangement of tasks ensures that real-time scheduling constraints are met and that tasks are correctly sequenced. The algorithm also details how by varying the sequence of tasks and through the use of permutations, it is possible to identify the task sequence which requires the least amount of time to complete all of the specified tasks. Once the 'optimal' task sequence has been identified it is passed to the scheduler for further processing.

The second algorithm presented within the CORES model addresses the shortcomings of the first by providing an algorithm capable of traversing its way through a job scenario where there exists more than one way to traverse from the beginning through to the end. The algorithm presented is based on the concept of traversing a Direct Acylic Graph (DAG) and is defined with the assistance of mathematical induction rules and provides a means for sub-paths to be identified and processed accordingly within a job scenario. The combination of this algorithm and the re-sequencing of tasks allows for a number of paths and scenarios to be assessed and for the 'optimal' sequence of tasks to be identified. Like the first algorithm, once the 'optimal' sequence has been established, it is passed to the scheduler for scheduling.

The final algorithm formally detailed within the CORES model is that of the scheduling process. As mentioned earlier, the CORES model makes use of a demand-scheduling algorithm which allows jobs to be placed into a schedule as soon as a slot is available. This is achieved by having the algorithm identify the earliest point in the schedule where a job can be completed successfully based on its requirements and the schedulers' availability. A closer examination of the algorithm reveals that it is capable of regulating the time at which jobs can be scheduled. This support enables the scheduler to enforce its own policy restrictions on what time jobs can be placed into the schedule.

In addition to providing a formal definition of the algorithms used within the CORES model, the chapter introduced the concept of a configuration manager and the role that it plays within the model. Specifically, the CORES model showed how a configuration manager is capable of coordinating the interactions which take place between a client and a component. These interactions extend

to the passing of method calls and exceptions and the managing of method calls which are directed towards components that are unable to process the incoming request. The chapter also details how the architecture of the configuration manager has been designed to allow it to be integrated into already existing distributed object frameworks. Other areas relating to the design of the configuration manager which were discussed included the interfaces provided to clients so that they can control their connection and issue reconfiguration operations to their corresponding component and the interfaces required by the manager so that it can control the reconfiguration of components. Issues such as the sequencing of operations and the exceptions used within the configuration manager were also covered.

The CORES model also introduced the concept of *dynamic controls* which can be used by end-users or operators to control the way in which method calls are handled when components are unavailable. This marks a departure from other reconfiguration environments as CORES provides a level of 'control' over how method calls are handled while others would block method requests until the component returns to service. The functionality to control how a method call is handled is critical and a welcome feature to those environments in which real-time responses are required and where end-users or operators are empowered to control the destiny of their method calls. As detailed in the chapter, the CORES model presents three types of controls which can be used to control the actions of a method call which can not be performed. These controls include not waiting for a component to become available, waiting indefinitely for a component to become available or to link a components unavailability to other Quality of Service characteristics which exist within the system. The introduction of this support certainly empowers end-users or operators when it comes to handling method calls.

The next chapter builds on the CORES model and concentrates on translating it from a conceptual model to an implementation which can be deployed into existing distributed object frameworks. The results of this implementation model can be seen in the case study presented in chapter 7.

Chapter 6

Implementing CORES

The previous chapter introduced the Component Oriented Reconfiguration Environment & Scheduling (CORES) model which assists software developers and end-users with the ability to manage reconfiguring components in both a dynamic and real-time environment. As with any conceptual model, it is important to realise that they are expressed using a number of definitions and formalisms which may not have been tested. It is the use of such untested definitions and formalisms which results in having to verify the model. To address this and to ensure that the model is relevant and correct, it is necessary to implement it so that a number of solid experiments can be performed and their results analysed. These experiments aim to examine the many facets of the model and to ensure its feasibility and verify its design objectives. The remaining chapters present an implementation of the CORES model and discuss the various results from the experiments performed upon it.

This chapter focuses on the CORES conceptual model presented in chapter 5 and details how with further refinements the conceptual model can be implemented. Specifically, the areas covered are the architecture used to provide the implementation, evolution of a job throughout the CORES system (including an explanation of the algorithms implemented), handling exceptions within the system, implementation of a client using CORES, user-interface provided to software developers and end-users to manage the system, configuration manager, construction and implementation of a component within the system and the real-time execution engine (job dispatcher).

6.1 Architecture

The underlying architecture of the CORES system can be broken into two distinct sections.

6.1.1 Hardware

CORES has been implemented on a variety of machines within the SUN-SPARC product range. Hardware variations were experienced and were mainly due to the numerous CPU architectures used by the machines. For the case study in chapter 7, a number of machines ranging from a SPARCStation-10 through to a SUNUltra-1 and SUNUltra-10 were used, each machine having a different type of processor.

Processor speeds ranged from 70MHz on the SPARCStation-10 through to 440MHz on the SUNUltra-10. In addition to the varying CPU processor speeds, each machine contained a differing amount of memory, ranging from 128Mb through to 512Mb. During the initial trials, the CORES system was capable of operating on a SPARCStation-5 with 32Mb of memory. All of the nodes used throughout the implementation of the system were inter-connected over a 10Mb/s ethernet network with each network port being individually switched.

6.1.2 Software

Due to a wide number of computing architectures and associated resources being used, it has been impossible to maintain a consistent operating system across all of the platforms. This resulted in CORES being tested on a number of different Solaris-SPARC operating systems. In particular:

- Solaris 2.5.1
- Solaris 2.6
- Solaris 7
- Solaris 8

The underlying components of CORES such as the client, the user-interface, the configuration manager and the components themselves were all implemented in the C++ language. In total, the CORES system contains approximately 30,000 lines of C++ code of which 79.5% represents actual source code responsible for performing operations while the other remaining 20.5% is made up from the respective header files (19.0%) and interface definition language statements (1.5%). The code behind CORES requires the use of SUN's SPARCworks version 4.0 compiler which runs under a Solaris/SPARC architecture and includes the C++ language bindings. This compiler was used so that third party libraries required by the distributed object framework which are included as part of the compilation and linking process could be accommodated.

CORES has the capability of being able to communicate with a number of components scattered throughout a distributed network by using a CORBA distributed object framework implemented by Iona known as ORBIX. ORBIX version 2.0, with its multi-threading options enabled allows for

the C/C++ language bindings to be used to implement servers (instantiation of components), clients and configuration managers. Once implemented, these servers, clients and configuration managers can be incorporated into CORES.

6.2 Job Evolution through CORES

When a job is introduced for the first time into the CORES system, it must progress through a number of stages which are responsible for loading the tasks contained within the job and transforming them into an optimised job suitable for scheduling.

6.2.1 Sequencing of Tasks

One of the most important actions that CORES performs is the sequencing of tasks in the overall system as the tasks are used to provide a list of all the possible permutations available for a job. The sequencing process begins by validating all of the information contained within a job definition file (refer to appendix B.1.1). Once validated, details contained within the file such as the number of tasks, the resources required and the number of times that a job has to be repeated in the schedule are recorded. This information is used to provision the amount of memory required to establish a data structure capable of holding details on all of the tasks within the job.

With the tasks identified, the sequencer allocates two memory locations in the data structure for each task found within a job. One memory location is allocated to the actual task to be performed while the other is used to store a command that readies the infrastructure so that the task can be performed.

In addition to allocating memory for the tasks and their corresponding infrastructure setup tasks, the sequencer also allocates additional space for tasks indicating the start and finish of the job. Depending on the circumstances, the sequencer may allocate extra memory within the data structure to support any additional tasks required to prepare resources specified by the job.

With memory allocation complete, the job is revisited and each task identified and examined. Task identification involves counting the number of predecessors and successors associated with each task. The count is used to determine whether the task to be performed is standard in nature and only has to be performed, a fan-out task where there are more paths leaving the task than entering or a fan-in task where there are more paths entering the task than leaving.

Based on the links between tasks and the number of predecessors and successors in each task it is possible to allocate a 'node type' for each task. This 'node type' can be either a type 'AND' (used for normal tasks or situations where tasks have to run in parallel) or a type 'OR' (used for those tasks which indicate a decision flow).

The 'AND' and 'OR' node relationship is shown in figure 6.1.

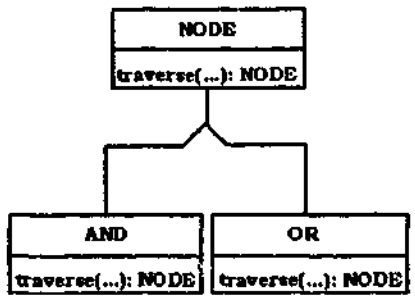


Figure 6.1: OMT Diagram representing the NODE, AND, OR Relationship

Figure 6.2 illustrates the layout of tasks in memory for a sample job after the nodes have been examined, node types assigned and dependencies identified. The job pictured also contains a multiple flow path stemming from an 'OR' node which can alter the path taken by the job to achieve its goal. Note that figure 6.2 does not use the Petri Net notation but rather represents the references that each node has to one another.

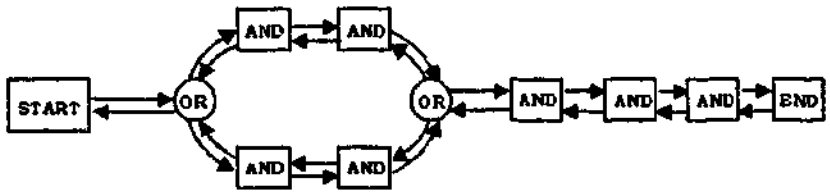


Figure 6.2: Data Structure representation of a Job combining Multiple Paths

Whilst the sequencer can handle jobs which include flow paths, the typical job definition file used for an astronomical observation in the case study involves having observations sequentially sequenced. The only exception to this linear sequencing arrangement is when the infrastructure tasks need to be performed at the beginning of the job.

Figure 6.3 illustrates a typical observation schedule (job) where the initial infrastructure tasks are performed in parallel and the remaining tasks follow one another sequentially. A task layout such as that illustrated assists the sequencing algorithm by allowing tasks to form one to one mapping with an element representing the task within an array.

The same sequencing approach is not possible for a job containing multiple flow paths as each flow path within the job must be identified, extracted and sequenced. Once the various sequences within

the sub-flow path have been identified, they then have to be integrated back into the original job. This results in the construction of additional scenarios as each permutation of the sub-flow path has to be added to the original job.

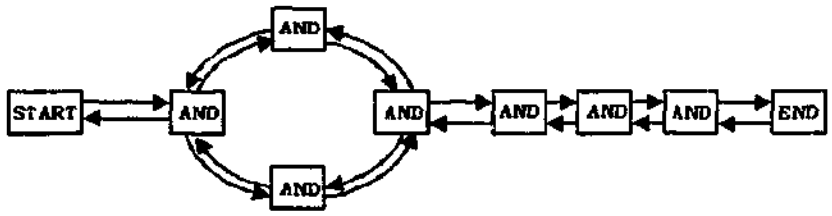


Figure 6.3: NODE representation of an Astronomical Observation after Sequencing

The task sequencer which is invoked after the tasks have been loaded works by establishing an array of pointers corresponding to the various tasks. Using the pointer array allows the task sequencer to manipulate the logical ordering of tasks without the manipulation of the initial data structure which represents the job scenario.

Upon the establishment of the pointer array, a permutation algorithm is employed to manipulate the required indexes to allow various permutations to be identified.

The permutation algorithm shown in figure 6.4 demonstrates how the algorithm uses recursion to manipulate the indexes to generate the various sequences. When the permutation algorithm is first instantiated, the initial ordering of the job scenario is recorded.

With the initial permutation recorded the algorithm can start to transpose and promote digits from the far right to the left of the index array. The following example outlines how with three tasks it is possible to generate six permutations representing all of the possible ways that the three tasks can be performed by the transposition of digits within the indexing array.

The initial array of indexes pointing to the tasks can be represented as:

1,2,3

Performing the transposition gives the following permutations:

1,3,2
2,1,3
2,3,1
3,1,2
3,2,1

These results allow the algorithm to represent each permutation of a job and allows each possible sequence to be registered in order that it can be traversed at a later date.

```
int calculatePermutations
    (int *permutation,int currentNumber,
     int numberOfElements,PermutationCombination *permutationTable)
[...]
     if  (currentNumber == numberOfElements) // Permutated Entry complete
      return (permutationTable->addNewCombination(permutation));

     for (digit = 1; digit <= numberOfElements; digit++)
     {  // Ensure that we use every digit only once
      if (isDigitUsed(permutation,currentNumber,digit))
       continue; // Digit is used, get another one

      permutation[currentNumber] = digit;
      calculatePermutations(permutation,currentNumber+1,
                            numberOfElements,permutationTable);
     }
[...]
```

Figure 6.4: Permutation Algorithm

6.2.2 Traversing Permutated Jobs

Once the permutations have been registered, each permutated job path is traversed in an attempt to determine which paths are viable (eg. star is above the horizon) and which path requires the least amount of time to traverse.

As explained in section 5.1.2, the traversal routine uses a simple algorithm to navigate the job scenario. Using recursion, the algorithm starts at the initial node and processes each successor while at the same time ensuring that the task can be completed at that point in time. The algorithm also checks that the predecessors of a task have been completed before the task is executed. If the predecessors for a task have not been completed the algorithm recurses back through the job scenario until it can find another path in which to proceed.

The traversal routine plays an important part in the calculation of the total time required to navigate a job scenario. As the routine examines each task, it extracts both the minimum and maximum amount of time required to perform it. Given these two numbers, the traversal algorithm will select the appropriate time based on whether it is calculating a best case or worst case time scenario adding the corresponding time value to the total time spent traversing the path. In most cases the traversal routine will be searching for the shortest path.

The challenging issue of processing multiple flow paths within a job is handled by the generic `traverse(...)` function. This function uses the object oriented programming paradigm concept known as 'inheritance' to provide a specific version of the function depending on the node type.

Using this concept allows the `traverse(...)` function to meet the specific needs of calculating the time spent on a series of tasks flowing from either an 'AND' node or an 'OR' node.

```
// First Successor Being Processed
 forwardTraversalList.successor1->
   traverse(this,timeCalculated,...);

// Subsequent Successors Being Processed
 forwardTraversalList.successor2->
   traverse(this,
            timeCalculated-getTaskTime(),...);
```

Figure 6.5: Processing Successors in an AND Node

```
// All successors processed in the following manner
 forwardTraversalList.successor1->
   traverse(this,timeCalculated,...);
```

Figure 6.6: Processing Successors in an OR Node

The code segments in figures 6.5 and 6.6 illustrate the differences between the `traverse(...)` functions. Figure 6.5 demonstrates how subsequent successors of the 'AND' node require the accumulated time for the current task to be removed. Performing this subtraction prevents the double counting of the current task's time when at a future point all paths which originated from the current node are combined. Figure 6.6 demonstrates how all the 'OR' node traversals are passed in the same time. The elimination of the current node's time for 'OR' paths is not required because when the paths are converged, only the optimal path is selected.

Once a job scenario has been traversed, the path taken and the total time spent traversing the scenario is recorded. When all of possible permutations have been examined, the system selects the scenario that satisfies the criteria the operator has specified. In most cases, the scenario with the shortest traversal time is sought.

After the desired scenario has been identified, it is passed to the scheduler for scheduling.

6.2.3 Scheduling of Tasks

The scheduling process is responsible for taking a job and its required resources and locating a position in the overall schedule where the job will complete successfully. The scheduler also has to be aware of other jobs and resource allocations which have already been committed to the schedule to avoid any clashes of time slots or resources.

The design of the scheduler from an implementation perspective represents a different approach to the way that scheduled tasks are stored within memory. The initial scheduler design called for the establishment of a data structure capable of tracking every second within the scheduling period. The memory requirements for such a data structure (even for a small scheduling period of two weeks) is quite significant, particularly when combined with a number of resource requirements. In addition to the memory usage, the scheduling approach results in a large number of resources being used to support its on-going management.

To overcome a problem of an unmanageable data structure, a new approach to storing the scheduling information has been developed involving the restructing of how scheduled jobs are placed into and represented within a schedule. This new approach has led to redeveloping the way in which the dispatcher and other supporting tasks interact with the scheduled jobs.

Rather than tracking each second within the scheduling period, the new approach only records the deltas or changes made to the schedule. This approach also simplifies the role of the dispatcher as it has only to move through the schedule and execute tasks as it finds them. This is remarkably different to the dispatcher which was going to use an alternative scheduling approach involving it in examining each second to identify whether a task had changed and executing it.

The technique of recording deltas in the schedule as opposed to accounting for every second leads to a significant decrease in memory requirements to establish the scheduler's data structure. This reduction can best be demonstrated by comparing how both scheduling techniques are used to store one job consisting of ten tasks operating over six resources for the period of one hundred minutes where in each ten minute period the task will change.

Using the first scheduling technique (accounting for every second) the scheduler allocates an array the size of 100 (minutes) × 60 (seconds per minute) × 6 (resources) = 36,000 elements. If each element contains 512 bytes of information, then the total schedule data structure size would be 36,000 (elements) × 512 (bytes per element) = 18,432,000 bytes or 17.58Mb to record 100 minutes of scheduling information.

Using the second scheduling technique (only the deltas are recorded) requires the scheduler to allocate an array the size of 10 (100 minutes / 10 tasks) + 6 (1 initial setup × 6 resources) × 6 (resources) = 96 elements. If each element contains 512 bytes of information, then the total data structure size would be 96 (elements) × 512 (bytes per element) = 49,152 bytes or .04Mb. The result is a massive 99.73% reduction in the memory requirements when utilising the second scheduling approach.

The process of scheduling jobs, mentioned in section 5.1.3, commences with the scheduler obtaining the number of jobs awaiting scheduling together with the total number of job placements (jobs to be inserted into the schedule multiple times). After the calculation of the job numbers, the scheduler takes all job instances and establishes a loop. This loop known as the 'outer loop' continues to perform until there are no more job instances awaiting scheduling. Nested within the 'outer loop' is a 'inner loop' which iterates through the various jobs awaiting scheduling.

Once a job is identified as requiring scheduling, the scheduler commences the process of finding a time at which all the tasks within the job can be performed successfully while ensuring that the resources required are available and that the job will not clash with any others already scheduled. The scheduler calculates a starting time for the job by testing times from the start of the scheduling period through to the end. Once a time has been selected, the system checks to see whether the job can be performed.

Where a starting time has been calculated for a job, the scheduler will perform a number of checks to ensure that the calculated time frame will fit into the schedule. The checks performed include:

1. Checking to ensure that the calculated finishing time of the job fits within the scheduling period
2. Checking the schedule and confirming that a window of idle time from the calculated start time to the finish time
3. Checking to ensure that all the resources which have been requested by the job are available for the calculated time period

All of these checks are implemented by searching the schedule for the nearest event to the proposed starting time. Once located all subsequent entries within the schedule are examined to ensure that the above conditions are met.

All conditions being met, the scheduler will indicate that the calculated start time is valid and will continue with the processing required to schedule the job. In the event that one or more of the conditions is not satisfied, the scheduler will continue to try and find a start time where all conditions can be met. Where an appropriate start time can not be calculated within the scheduling period, it is assumed that the job can not be scheduled and the scheduler moves onto the next while notifying the end-user that the previous job is no longer schedulable. When this occurs, the associated job instances are set to zero to prevent the scheduler from trying to re-schedule it at a later date.

With the start time confirmed the scheduler performs one final assessment to ensure that the proposed start and finish times of the job satisfy the business domain rules enforced by the scheduler before scheduling the job. As introduced in chapter 7, the Australian Telescope National Facility imposes a set of business domain rules which ensure that jobs must start and finish on the closest hour. In certain jobs, it may be impossible to adjust the starting or finishing time of a job to match the rule and therefore the scheduler allocates 'idle time' to the job.

Periods of 'idle time' are represented within the schedule as special tasks so as to prevent the scheduler from overwriting the period with a different job.

Once allocated 'idle time' has been placed at the beginning and end of a job, the actual placement of the job within the schedule can begin. Job placement involves going through each task within

the job and registering the resources and time block required within the schedule at the appropriate time specified by the task. Once registered, no other job or task can be allocated to the same time period using the same resources. It is possible however to allow different jobs to be executed at the same time providing that they do not use the same resources.

The job being scheduled results in the scheduler decrementing the job instances count associated with it and moves onto the next job which has one or more job instances awaiting scheduling. This process continues until there are no more job instances awaiting scheduling.

6.3 Client / User Interface

Using a slightly modified client/server architecture as the underlying CORES communication mechanism, allows clients wishing to make use of a particular service to connect to the configuration manager responsible for managing the component providing the service. The modification to the client/server architecture is evident when the relationships between the client, configuration manager and the component providing the service are identified. This new architecture gives CORES clients more control over the way they interact with the service. Figure 5.12 identifies the relationships between the client, configuration manager and component providing the service. As the figure illustrates, rather than forming a connection from the client directly to the service provider, CORES requires all clients to communicate with the configuration manager. With the initial connection formed, it is the responsibility of the configuration manager to relay the requests to the component providing the service.

In addition to providing support for clients, the CORES system provides a user-interface allowing end-users or operators to specify instructions to the system as well as having the system provide real-time information relating to its status. An example of an interface based on the case study is presented in chapter 7 and illustrated in figure 6.7.

Figure 6.7 identifies that the user interface consists of three window panes. The panes have been specifically designed to provide operators or end-users with the opportunity to monitor the various parts of the CORES system while at the same time being able to provide commands.

The first pane provides the operator with real-time information about the components within the system while at the same time displaying the progress of tasks. The second pane provides information to the operator or end-user including details about events happening within the CORES system. The scheduler also provides messages to this window including whether a job can be scheduled, what tasks are currently running and when a scheduled task is to finish. This panel is also used to report any errors which may have occurred while performing tasks such as sending method calls to unavailable components or system errors which the component may have experienced.

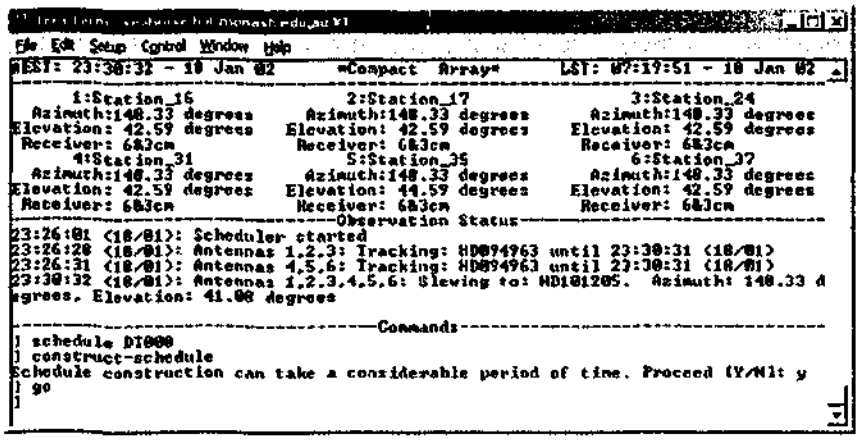


Figure 6.7: Sample of the User Interface for the CORES System

The final pane presented on the user interface is the command line. The command line denoted by the prompt ']' is used to provide a range of commands (refer to appendix C.1 for a complete list of commands which can be specified at the command line). These commands control both the underlying system which executes tasks and the CORES system which is responsible for reconfiguration operations or determining actions to be performed when a component is no longer available.

A command of importance in CORES is the `reconfigure-mode`. This command allows CORES to change the behaviour of method calls originating from the client and handled by the configuration manager. As detailed in section 5.3, the CORES system provides three reconfiguration modes: `WillWait`, `NoWait` and `QoSWait` which control how the client and the configuration manager respond to components which are not available. These dynamic controls also effect the selection of the appropriate exception harnesses which are used in the client and the configuration manager to enable a response to any exceptions received. These conditions may reflect an error in the system or a reconfiguration event.

Both the client and user interface are able to provide a level of control by manipulating the `ModuleUnAvailableInstruction` data structure located within the `ModuleAvailability` definition shown in figure 6.8. For each method call that the client, user-interface and real-time execution engine dispatch, a corresponding `ModuleUnAvailableInstruction` is sent to ensure that the configuration manager knows the operations to perform if the component is unable to process the method call.

Figure 6.9 provides a segment of code which the client uses to setup the environment where method calls will not wait for a component if the method call can not be processed. This figure also illustrates that both the client and real-time execution engine located within the scheduler are informed of the new *reconfigurationOptions*.

```
module ModuleAvailability
{
 enum ModuleUnAvailableAction {NoWait,    // Don't wait if object is
                                          //  unavailable
                               WillWait, // Wait until object is available
                               QoSWait}; // Quality of Service Wait
 struct ModuleUnAvailableInstruction
 {
  ModuleUnAvailableAction anInstruction;
  ...
 };
};
```

Figure 6.8: Module Availability Data Structure

```
theScheduler.setReconfigurationOptions(ModuleAvailability::NoWait,0,...);
anInstructionSet.anInstruction = ModuleAvailability::NoWait;
anInstructionSet.someQoSData.timePreparedToWait = 0;
```

Figure 6.9: Client adjusting Reconfiguration Options

The code sample shown in figure 6.10 illustrates how the `ModuleUnAvailableInstruction` data structure is parcelled with a method call being made to a configuration manager. The code illustrates a request destined for a configuration manager to process a series of Quality of Service (QoS) requirements if the component is unable to accept the request when a method call is sent.

```
// Build the request, pass in the parameters and invoke it
configManager << CORBA::insert(QoSProcessing::_tc_QoSAction,
                               &QoSToSatisfy, CORBA::inMode)
              << CORBA::insert(ModuleAvailability::
                               _tc_ModuleUnAvailableInstruction,
                               &anInstructionSet, CORBA::inMode)
              << CORBA::insert(ExceptionHandling::_tc_ManualException,
                               &aManualException, CORBA::outMode);
aRequestForTheObject.invoke();
```

Figure 6.10: Assembly and Transmission of QoS Reconfiguration Information

6.4 Configuration Manager

The configuration manager provided as part of the CORES system and described in section 5.2 is responsible for processing incoming method requests and determining whether or not the intended component is capable of receiving them. If the configuration manager determines that the request can be passed through to the component, then a number of operations are performed. These include having the configuration manager cache any data variables which can be used to speed up future data enquiries as well as adjusting any internal states used to ensure the correct logical sequencing of operations. When the operations have been performed, the configuration manager forwards the request onto the intended component.

Requests which can not be processed by the component result in the configuration manager performing a number of operations to ensure that any instructions contained within the accompanying `ModuleUnAvailableInstruction` data structure are processed accordingly. Typically, components are unavailable as a result of reconfiguration operations.

Figure 6.11 illustrates the `processUnAvailability(...)` function which is responsible for taking the availability instructions (instructions to perform when the component is not available), the time at which the component will be available (if specified) and the object reference to the component itself and determine what action needs to be performed.

The figure demonstrates that if the `NoWait` or `QoSWait` dynamic control with invalid QoSCriteria data is passed into a component that is unavailable, the `processUnAvailablity(...)` function will `throw(...)` an exception resulting in control being passed back to the configuration manager and then to the client.

```
void processUnAvailability
           (const ModuleAvailability::ModuleUnAvailableInstruction
             &availabilityInstructions, ...)
{
 if (availabilityInstructions.anInstruction == ModuleAvailability::NoWait)
   throw ObjectBeingReconfigured("Object is currently being reconfigured.");
 if ((availabilityInstructions.anInstruction == ModuleAvailability::WillWait)&&
     (availabilityInstructions.someQoSData.timePreparedToWait == 0))
   throw QoSObjectInvalid
    ("The time value specified in the QoS structure has not been specified.");
[...]
}
```

Figure 6.11: Processing of Reconfiguration Information

Table 6.1 lists those exceptions which the client and the configuration manager must be prepared to handle for each method call sent to the configuration manager and component. The `processUnAvailability(...)` function is responsible for raising those exceptions which match the requirements of the instructions passed in with the method call. These exceptions are initially caught by the configuration manager and then the corresponding value in the `SystemException` data structure is set. This structure is then sent to the client to indicate that an exception has been raised and needs to be handled.

CORBA::SystemException	ModuleAvailability::ObjectBeingReconfigured
ModuleAvailability::QosObjectBeingReconfigured	ModuleAvailability::QoSObjectInvalid

Table 6.1: Configuration Manager Exceptions

NOTE: CORBA::SystemException is a base class which resembles a number of exceptions.

The `SystemException` data structure is used to signify exceptions rather than using the `throw(...)` mechanism for two reasons. The first is to ensure compatibility across those systems that have no native support for exceptions. The second relates to the underlying CORBA framework which CORES uses. The CORBA framework within CORES was developed using Iona's ORBIX product which at the time of development and implementation was unable to support the throwing of any user defined exceptions between the component and client. This led to the development of the `SystemException` data structure being used as a means for passing user defined exceptions between the CORBA framework, the component and client[1].

In addition to being responsible for handling method requests destined for components which are unable to accept them due to their reconfiguration commitments, the configuration manager is charged with the responsibility of establishing the initial connection between itself and the component as well as acting as the choreographer for reconfiguration operations.

The process of establishing the initial connection to a component involves the configuration manager taking an object reference which it has received and requesting a connection to be established. Upon establishment[2], the object reference is committed and use for all subsequent communications until the component is no longer used or a reconfiguration operation has resulted in the reference changing.

Object references which are textual representations of the location and interface of a component, are used within the configuration manager to allow the flexibility of being able to dynamically update the location of the component. As the component location changes, so does the object reference recorded in the configuration manager.

[1]The user defined exceptions being referred to are customised exceptions which operate between the configuration manager and the client to reflect reconfiguration information. They are not Iona's system exceptions.

[2]The underlying CORBA framework in the ORBIX package takes care of the establishment phase.

The decision to place the configuration manager between the client and the component was engineered to overcome the inherent difficulties in making use of other approaches such as Iona's implementation of filters. Filters as implemented in the ORBIX product allow developers to attach two additional classes to the component they are implementing. One of these classes acts as a *pre-filter* while the other a *post-filter*. These filters provide developers with the ability to intercept method calls before they arrive at the component and after they leave the component.

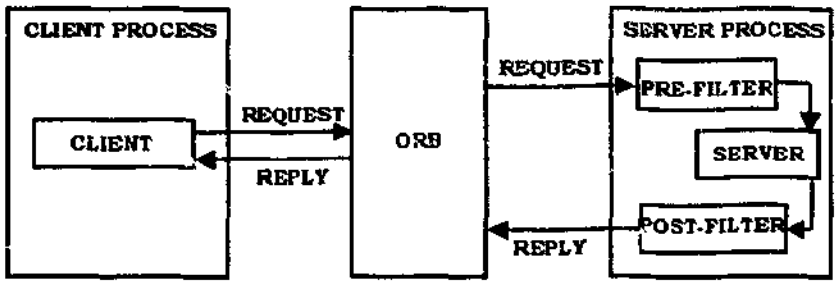


Figure 6.12: Pre and Post Filtering in Iona's ORBIX

Figure 6.12 illustrates the relationship between the client, request, reply, filters and server (instantiation of the component). At first glance, the filter approach seems ideal for the role of the configuration manager, however the implementation provided by ORBIX results in both the *pre* and *post* filters being destroyed when the server process providing the implementation of the component is deactivated or replaced. This destruction makes the filter approach unusable as it is not possible to provide a consistent 'well-known' location for the configuration manager. Additionally, the filter implementation developed by Iona is proprietary in nature and would make porting CORES system to another system very difficult.

The solution to the problem led to the introduction of the configuration manager and placing it between the client and the component. Having the configuration manager between these two entities allows the client to deal with the configuration manager at a well known address while allowing it to manage the configuration and reconfiguration of a component.

Steps involved in reconfiguring a component:

- Ensuring that the component being replaced is in a consistent state[3]
- Ensuring that the data from the component to be replaced can be extracted in a safe and consistent manner

[3]The component may never reach a totally consistent state as a result of internal processing, but the reconfiguration process attempts to place the component into the best possible state to support reconfiguration.

- Ensuring that the new component which is to replace the old has been initialised
- Ensuring that the object reference of the new component has been recorded within the configuration manager
- Ensuring that the state data from the old component can be transferred to the new component

To meet these goals, the component reconfiguration routine is broken up into a series of segments. Each segment focuses on a particular section of the reconfiguration process. The first segment of code (figure 6.13) is responsible for initialising a semaphore which ensures there is no concurrent access to the component during the reconfiguration process. The use of this semaphore is widespread throughout the configuration manager as each method call which passes through must first check and gain exclusive access to the semaphore before performing its operation. In addition to the establishment of the semaphore, the code demonstrates the configuration manager sending a request to the component to commence the quiescent process.

```
[...]
 mutex_lock(&m_reconfiguringComponent);
 reconfiguringComponent = 1;

 CORBA::Request aRequestForTheObject(objectPtr,"disableAntenna");
 aRequestForTheObject.send_oneway();
[...]
```

Figure 6.13: Commencing the Reconfiguration Process

The remaining segment of code shown in figure 6.14 is responsible for organising and coordinating the extraction of state data from the component which is to be decommissioned and migrating it to the new component. As the reconfiguration function is responsible for migrating data it is necessary to provide a method to allow the transfer of data. A number of methods exist to facilitate this transfer including the use of a persistent file, establishing a direct link between the old and new components or making use of another form of inter-process communication. The migration of data between the various components in the case study is handled with the assistance of a persistent file which both components share through a common file system.

After the state data has been extracted from the component, the configuration manager proceeds to contact the new component to confirm its accessibility and so that the object reference of the new component can be obtained. Once obtained from the component, the object reference is recorded for future use. The actual process of using the new object reference does not take place until the component is *enabled*. Upon the successful loading of the state data, the component initialises its internal state. Figure 6.14 illustrates the code that the configuration manager uses to coordinate

the transfer of state data from the old component to the new. Additionally, this code shows the replacement component being *enabled* with the `enableAntenna()` command.

```
[...]
 aRequestForTheObject.reset(objectPtr,"saveAntennaData");
 aRequestForTheObject << CORBA::insert(_tc_stringStructure, &antennaDataFile,
                                       CORBA::outMode);
 aRequestForTheObject.invoke();  // Perform the operation
 ...
 aRequestForTheObject.reset(newObjectPtr,"restoreAntennaData");
 aRequestForTheObject << CORBA::insert(_tc_stringStructure, &antennaDataFile,
                                       CORBA::inMode);
 aRequestForTheObject.invoke();  // Perform the operation
 ...
 aRequestForTheObject.reset(newObjectPtr,"enableAntenna");
 aRequestForTheObject.send_oneway();  // Perform the operation
[...]
```

Figure 6.14: Finalising the Reconfiguration Process

The remaining part of the reconfiguration process involves manipulating the object reference which is used by the configuration manager to transmit method calls to the components. The process of exchanging the object references involves placing an exclusive lock on the object reference identifier (to ensure no concurrent accesses), releasing the memory allocated to the old reference, allocating memory for the new reference, assigning the new reference to the configuration manager and releasing the exclusive lock so that requests can be forwarded onto the new component.

6.5 Server / Component

The role of the component or server as it is known within the CORBA architecture is pivotal to the CORES system as it is responsible for providing the implementation of the service. The only role that CORES provides is the medium (implemented via the configuration manager) to link requests from the client through to the implementation provided within the component.

Although CORES places little restriction on how a component is developed, it does if they desire, to have it integrated into the CORES system or have the configuration manager exercise some level of control over the component. In order to make a component manageable by CORES, the developer must implement at the very minimum the interfaces shown in figure 6.15.

```
interface Antenna
[...]
 boolean restoreAntennaData(...); // Restore state data
 boolean saveAntennaData(...);    // Save state data
 oneway void enableAntenna();  // Commence Quiescent mode
 oneway void disableAntenna(); // Leave Quiescent mode
[...]
```

Figure 6.15: Interfaces required by a CORES Component

These interfaces are used by the configuration manager to exert a level of control over the component. Specifically, the component must provide a routine which allows the configuration manager to extract the internal state data and to keep the data in a persistent form while awaiting transfer to another implementation or component. In addition to providing a routine which allows the configuration manager to extract the data, the component must provide a routine capable of receiving data and being able to initialise its internal state based on the data received. In some cases this may result in the newer implementation having to contain conversion routines so that old data streams are interpretable.

The remaining two interfaces are used to manage the quiescent state of the component so that the configuration manager can extract the state data held within the component. The use of the interfaces assists the configuration manager in ensuring that the component is consistent when managing its internal state or while it is performing reconfiguration activities.

As the implementation and requirements of a component differ with a situation and implementation, it is impossible for the CORES system to prescribe a way of being able to control the consistency and quiescence within a component. As an example, the components presented in chapter 7 are designed to make use of a semaphore and a boolean value to indicate that the component is enabled and operating. Each routine performed within the component continually checks the boolean variable which is guarded by a semaphore to ensure that the component is still enabled. When the variable is toggled to reflect that the component has been disabled and is entering a period of quiescence, all of the internal routines where possible[4] return to a state of quiescence. Each routine waits until the boolean flag is reset and semaphore released before resuming an 'active' state within the component.

When the state of quiescence is achieved, or the timeout value for quiescence has expired, the component is ready to support reconfiguration activities including the extraction of the internal state data.

[4]In some cases it may not be possible to halt all of the internal processing being conducted by a component. Every attempt is made to put the component into a quiescent state.

6.6 Exceptions

Exceptions and the `ManualException` data structure within the CORES system provides a means for various communications. These communications can be categorised into the following:

- Client receiving exceptions from the configuration manager
- Configuration manager receiving exceptions from the component
- Client receiving exceptions from component after being relayed from configuration manager

In addition to the identification of those communication categories for exceptions, the exceptions used within CORES can be divided into two groups. One group refers to those exceptions used by the configuration manager to indicate that a method call can not be forwarded to a component due to reconfiguration operations. The exceptions commonly associated with reconfiguration operations are contained in table 6.2.

ModuleAvailability::ObjectBeingReconfigured	ModuleAvailability::QosObjectBeingReconfigured
ModuleAvailability::QosObjectInvalid	

Table 6.2: Reconfiguration Event Exceptions

The `ModuleAvailability::ObjectBeingReconfigured` exception is the most commonly encountered exception by the client and results from performing reconfiguration operations. This exception is raised when the system is operating under the dynamic control `NoWait` and a method call attempts to access a component that is not available. The remaining exceptions in table 6.2 are generated when the system is operating within the `QoSWait` dynamic control and encounters a component not available. The exception (`ModuleAvailability::QoSObjectBeingReconfigured`) is raised when QoS characteristics passed in with a method call via the QoSCriteria parameter can no longer be met as a result of the component being unavailable. The `ModuleAvailability::QoSObjectInvalid` exception is raised in similar circumstances except that rather than being raised when the QoSCriteria can not be met, it is raised due to the QoSCriteria data structure carrying invalid data or not being properly formatted.

The remaining group of exceptions dealt within CORES is slightly harder to identify. The difficulty stems from the actual implementation of the component where a component designer may cause any number of pre-defined system exceptions. An example of such exceptions include 'Divide By Zero' through to user defined exceptions which are created by the developer and embedded within the implementation of the component. As CORES is able to handle both user defined and system exceptions it is not possible to produce a table of every expected exception. However the most common exception which fits this category is the `CORBA::SystemException` exception classes.

Given the identification of the exception communication paths it is possible to associate exception groups with the various communication paths provided in CORES. Table 6.3 highlights the exception groups which can be found communicating within the different parts of the system.

Communications / Exceptions	Client <-> CM	CM <-> Component	Client <-> CM <-> Component
Reconfiguration Exception	✓	✓	✗
System Exception	✓	✓	✓

Table 6.3: Communication Paths and Exception Groups

Note: CM refers to Configuration Manager.

Note: Reconfiguration exceptions are delivered to the client through the `ManualException` data structure.

Table 6.3 confirms the special programming relationship between the client and configuration manager as both need to be programmed in such a manner so as to support the reconfiguration exceptions group. From the perspective of the client, specific attention must be given to the various types of exceptions which may be returned by the configuration manager. Each method call made by a client must be encapsulated within a harness to trap any exceptions from the configuration manager and when detected dealt with by the client. From the other side the configuration manager must ensure that it correctly identifies the exceptions it receives when processing the components availability. Any exception information obtained is used to populate the `ManualException` data structure which is sent back to the client. Additionally, each interface within the interface definition file for a configuration manager must include a list of expected reconfiguration exceptions (refer to table 6.2). This list is required by the underlying ORBIX architecture so that it can compile the code associated with the configuration manager and is not a restriction placed upon the configuration manager by the CORES system.

Exceptions originating at the component and destined for the configuration manager can fit into one of two categories. The first category to be considered are those exceptions which belong to the reconfiguration exception group. Exceptions fitting into this category allow the configuration manager to receive information from the component regarding its configuration status. The most specific instance of this is the communication which takes place in the form of exceptions between the configuration manager and the `processUnAvailability(...)` function. Information passed between the availability function and the configuration manager is used to construct the `ManualException` data structure which is sent to the client. Additionally, any information contained within the reconfiguration exceptions group can be used by the configuration manager to reactivate any method calls which have had their conditions met. The information contained within these exception structures

can be vitally important for any method calls blocked as a result of data held within a QoSCriteria structure, especially any information concerning the time at which the component is to be reactivated.

The second exception category relates to those exceptions belonging to the system exception group. These are normally the result of an internal error occurring within the component. Wherever possible the configuration manager attempts to handle these errors locally, but in certain circumstances it may trap the relevant exceptions and `throw(...)` them back to the client for handling. It should be noted that system exceptions are normally generated in the event of the component incurring a fatal error in which case the client and the configuration manager may not be able to provide much support for the component.

The final communication path which CORES offers is the ability to relay an exception from the component through the configuration manager and back to the client. As discussed, this communication path has to be taken because the client has no direct link with the implementation of the component, but only with the implementation of the configuration manager.

CORES provides the unique ability of being able to pass exception information from the component to the client via the configuration manager by sending the configuration manager an exception which it does not know how to process. The entire process centres around the way in which the configuration manager responds to exceptions from the component it is managing. When the configuration manager catches an exception from the component that it is managing, it will start processing the exception and trying to identify the type of exception thrown. As the configuration manager is only suited to capturing and dealing with exceptions of a certain type, it will reach the end of its expected exception list where it will then match the exception to the 'catch all' clause. Exceptions which make it to the 'catch all' clause are re-packaged and 'thrown' back to the client to be handled.

The limitation to this process is that the ORBIX implementation used for the CORES system only allows `CORBA::SystemException` types to be thrown back and not user defined exceptions. In the event that a user defined exception is raised, the Iona implementation will present an ORBIX system error message informing the operator that no support exists for throwing any user defined exceptions between the component and the client.

6.7 Real-Time Execution Engine

The Real-Time Execution Engine or dispatcher as it is commonly known, is located within the scheduler and is responsible for identifying when tasks are to be performed and controlling the dispatching of those tasks to the corresponding resources. Due to the real-time nature of the dispatcher, it is important that it operates in a timely fashion so as to avoid any delays resulting in tasks not being dispatched at the appropriate time. Timing constraints such as these are important in hard real-time operating environments.

The process of identifying and executing tasks at a specific time begins with the dispatcher establishing a thread for each resource that the scheduler is scheduling. Once the threads have been created, each thread obtains a time or a reference point (the case study uses a central timing source known as the `ObservatoryClock`) and enters a loop which continually evaluates the time or reference point until such a point that the loop reaches the end of the scheduling period.

While in the loop the dispatcher constantly checks the schedule to determine whether or not there are any tasks waiting to be processed at a specific time point. When a task execution point is achieved, the dispatcher records the task number allocated by the scheduler and passes it onto the routine responsible for transmitting the task to the resource.

With the task transmitted, the dispatcher returns to checking tasks located within the schedule. After transmission of the task to the resource, the dispatcher has to be careful not to subsequently resend the same task back to the resource. In order to prevent this, the dispatcher before transmitting a task to the resource, will compare the task number with the last task sent. If the number is different the dispatcher will transmit the task to the resource otherwise the dispatcher will ignore the task and not transmit it.

The dispatcher developed for the case study has a provision to *sleep* for the period of one second so as to reduce the requirements on the scheduler's processor. In a true real-time environment, the dispatcher would either be constantly checking the time or be notified via a hardware interrupt that the time has changed. This sleeping provision can result in some instances where the scheduler will miss a second. To avoid missing any tasks, the dispatcher calculates the number of seconds that have elapsed whilst being in the sleep mode. If the lapsed period is less than a second, the dispatcher will continue to search for tasks. If the period is greater than one more seconds the dispatcher enters a 'catch-up' mode and goes through each time point it has missed in an effort to identify those tasks that it may have missed. Upon the discovery of any tasks during this time period, it will transmit them onto the corresponding resource. This 'catch-up' process continues until all of the missing seconds have been accounted for and processed.

6.8 Limitations

On implementation of the CORES system, it was noted that the conceptual design presented in chapter 5 did not map transparently to the implementation model. Specifically, problems were experienced with the construction of the configuration manager and parts of the CORES scheduler. The following sections detail the implementation limitations.

6.8.1 CPU/Memory Intensive Operations

Like other systems responsible for calculating permutations, the CORES task sequencer requires a considerable amount of CPU time. The requirements on the CPU and memory are even more than usual when CORES is performing its sequencing operations as the algorithm used to calculate the permutations is not optimised.

This lack of optimisation results in CORE's having to use large amounts of CPU time to calculate all of the possible permutations and store them in a large table. Ideally, it would be beneficial to use a more optimised permutation algorithm and to assess each permutation as it is generated. Doing this would allow the sequencer to ignore invalid permutations before having the CORES system commit memory resources and CPU time.

6.8.2 Construction of Configuration Manager

The issue of constructing the configuration manager refers more to the implementation of the CORES system than the design as discussed in chapter 5.

The implementation of the CORES configuration manager is predominately built using a manual process. It involves using the interface definitions of the component which the configuration manager is to manage and is made up from a considerable amount of code which is repeated throughout. Due to the large amount of repeated code and the information available within the interface definition files it should be possible to automatically generate a considerable amount of code which could be used as a stub for the configuration manager.

The stub could then assist developers in introducing new functionality to the configuration manager or speed up the time the developer needs to integrate components into CORES. This would be beneficial to developers as they currently require an understanding of the CORES system before being able to modify the appropriate code that provides the increased flexibility and functionality of managing method calls in real-time.

6.9 Chapter Summary

The Component Oriented Reconfiguration Environment & Scheduling (CORES) system from an implementation perspective has been examined. This included examining the underlying implementation and the architecture that it relies upon to communicate and manage the various components distributed over a computer network.

A further area considered was the evolution of a job as it passed through the system. This involved the tracing of tasks and jobs from the point where a group of tasks are associated with a job through to the calculation of the various permutations representing every possible order in which a job could

be performed. The investigation continued by examining the permutation list and calculating the optimum permutation. The optimum path was calculated and the process of actually scheduling the job into the schedule examined.

Additional to those areas responsible for processing jobs, the chapter focused on the client implementation which is responsible for sending method requests to the component via the configuration manager and the user interface providing a front end to the CORES system. Both the client and the user interface provided the flexibility for the end-user to manipulate the various reconfiguration modes offered by the system.

Examined was also how various elements of the CORES system were transformed from a conceptual model to an implementation. The elements included examining the role and implementation of the configuration manager including the additional control that dynamic controls provide, investigating how exceptions are handled within CORES and the extent of support that the configuration manager can provide to integrated components.

Implementation and the role that the Real-Time Execution Engine plays in identifying when and what tasks should be executed and where tasks should be sent were also examined.

A case study is presented in the following chapter involving the CORES system and the implementations discussed here. These will be applied in a practical context using the Australian Telescope National Facilities Compact Array telescope at Narrabri.

Chapter 7

Case Study: Radio Telescope Array

Over the last two chapters the thesis has introduced and focused upon the design principles which make up the Component Oriented Reconfiguration Environment & Scheduling (CORES) model and its associated implementation. The CORES model was introduced to provide more control in the way method calls are dealt with when components are unable to accept or process a method call. This is especially important in real-time systems. This chapter examines an implementation of the CORES model which has been incorporated into a real-time practical scenario. The selection of a real-time system provides the underlying architecture of CORES with the ability to verify its design and to ensure that it is capable of providing additional *control* to end-users[1] with regard to the way that method calls are managed.

The scenario chosen incorporates CORES into a simulated radio telescope[2] array and is based upon the radio telescopes found at the Australian Telescope Compact Array (ATCA) located at Culgoora near the township of Narrabri in north-western New South Wales, Australia.

7.1 Background

The case study presented will demonstrate the architecture of CORES on a series of radio telescopes linked to one another and linked back to a central control room which coordinates the activities of the telescopes. The abstracted telescope system presented simulates the functionality of the Australian Telescope Compact Array (ATCA) operated by the Australian Telescope National Facility

[1] In the context of this chapter, end-users of the system will be referred to as observers.

[2] The simulated telescope operates in the same manner as a normal radio telescope, however the simulation does not provide support for the signal processing aspects nor the data capturing capabilities.

(ATNF). The following section provides a brief background on the ATNF which is charged with the responsibility of managing the compact array, while the following section provides details about the compact array itself.

7.1.1 Australia Telescope National Facility

The Australia Telescope National Facility (ATNF) located in Marsfield, NSW is an organisation contained within the radio astronomy division of Australia's Commonwealth Scientific and Industrial Research Organisation (CSIRO). The ATNF has a number of roles. The most important role the ATNF has, is to coordinate the continuing research across Australia into the field of radio astronomy which is viewed in science as one of the modern and emerging fields in astronomy. The organisational structure of the ATNF also provides it with the ability to contribute to government policy regarding the future of radio astronomy.

Besides these organisational roles, the ATNF is also responsible for providing support and managing the operation of eight telescopes located in three different observatories. These observatories include Parkes where there is a sixty-four metre antenna, Mopra with a twenty-two metre antenna and the compact array located in Culgoora which has six radio telescopes each measuring twenty-two metres in diameter. All of these telescopes can be used individually or tied in with one another to provide part of what is known as the Australian Telescope or the Australian Long Base Array. When connected together the telescope spans from Parkes to Mopra encompassing the ATCA near Narrabri. Each of these observatories are located within New South Wales, Australia.

7.1.2 Australian Telescope Compact Array

The Australian Telescope Compact Array shown in figure 7.1 consists of six antennas, five of which are mounted to a rail track measuring three kilometres in length running in a east-west direction. The sixth antenna is fixed to a sixty-one metre rail segment running in a east-west direction and is spaced approximately three kilometres from the end of the other east-west rail track. Recent expansion work conducted at the array has led to the installation of a new rail track segment running in a north-south direction. By having the antennas mounted on rail tracks, the ATNF is able to modify the various compact array configurations. Each configuration allows for an antenna to be located in a different position along the rail track and allows for the distance between the antennas to vary and hence provides observers with a different view of the various radio point sources.

Figure 7.1: Radio Telescopes at Australian Telescope Compact Array

Photo courtesy of Australian Telescope National Facility

Each telescope within the array has a reflector dish measuring twenty-two metres in diameter and is capable of listening to and processing multiple frequencies at the same time. Within the antenna a PDP-11 computer[3] is used to coordinate the antennas activities such as processing the inputs received from the control room. It manages various systems within the antenna including the cryogenics and antenna drive systems as well as passing those signals that the antenna receives from the radio point source and atmosphere back to the control room for further processing. Both the local PDP-11 antenna computer and the DEC equipment located within the control room use a number of tightly coupled programs written in FORTRAN to coordinate the schedule and the operations sent to the various antennas. The ATNF is currently undertaking an on-going long term upgrade of the observatory which will result in the replacement of the PDP-11 computers with a new hardware architecture (most likely an embedded operating system). In addition to upgrading the observatory hardware, the software is also being upgraded to take advantage of the object oriented programming paradigm so that additional functionality can be added to the array in terms of objects. At the time of writing, consideration was being given to deploying the objects throughout the system with the aid of a CORBA distributed object framework.

The telescope array is commonly used for the investigation of pulsar and quasar emissions as well as studying dead stars and the process of galaxy creation and destruction. It is important to clarify that the radio telescopes are not optical as they measure radio waves from distant galaxies and hence

[3]The antenna computer is enclosed within a faraday cage to reduce the electromagnetic radiation being emitted from it.

are capable of operating at any time of the day (providing that the radio point source is above the horizon).

7.1.3 Radio Telescope

A radio telescope is a complex piece of scientific equipment which requires the use of a number of components so as to be able to successfully observe a radio point source. The number and sophistication of components varies between telescope models. From a generic point of view, a radio telescope in some shape or form consists of the following components:

- Receiver / Frontend
- Signal Processor / Backend
- Control System

One of the most distinctive features of more powerful radio telescopes is the large antenna dish which is used to collect the radio waves as they arrive on earth. The reason for the large dish is to maximise the amount of faint radio waves and focus them toward the focal point (a point where all the radio waves are concentrated). Additionally, the large dishes are required due to the actual size of the radio waves being observed. At the focal point, the radio signals are transferred to a receiver which allows the observer to filter out wavelengths that they are not interested in capturing. While operating, the receiver is cooled with the aid of cryogenics to a temperature of approximately minus two hundred and sixty degrees centigrade or around thirteen degrees kelvin in an effort to eliminate any additional radio interference from being detected. In some radio telescope configurations, the receiver is actually located on a rotatable turret which allows the observer to change the receiver frequency and hence be able to detect varying wavelengths. Changes to the receiving frequency are not made during an observation as a significant amount of calibration work needs to be performed before any observation at a new frequency can commence.

As radio waves travel further away from their source, the intensity, power or strength of the wave varies inversely to the square of the distance from the source. This is known as the inverse-square law of propagation. The implication is that by the time the radio wave reaches a radio telescope on earth from a distant source, the signal is very faint. To overcome this problem, radio telescopes use a signal processor consisting of an amplifier which increases the signal being fed from the receiver as well as introducing other frequencies used to calibrate the signal. Once the signal has been amplified, it is then sent onto a computer known as a correlator which is responsible for cleaning up the signal. Once the signal has reached the correlator and other related computers it is here that the processing of the signal is performed. In most cases the actual processing is performed offline after the data from the antenna has been captured and stored on some removable media type.

The control system has an important role to play in the operation of the radio telescope. It is responsible for processing all aspects of the antenna including such activities as pointing the antenna in the appropriate position for the observation and coordinating the data being passed from the receiver to the signal processor. The control system is also responsible in some cases for performing the scheduling and running of observations and reporting the status of the antenna back to a central control room. In addition to processing the observations, the antenna control system processes timing signals from the control room and integrates them with the signal processor as well as passing data back to the correlator for correlation. Other tasks involve running various system checks to ensure that the antenna is operating in a correct manner.

Figure 7.2 provides an illustration of a radio telescope processing incoming radio waves and passing them onto the signal processor unit for amplification and then to the computer for further processing.

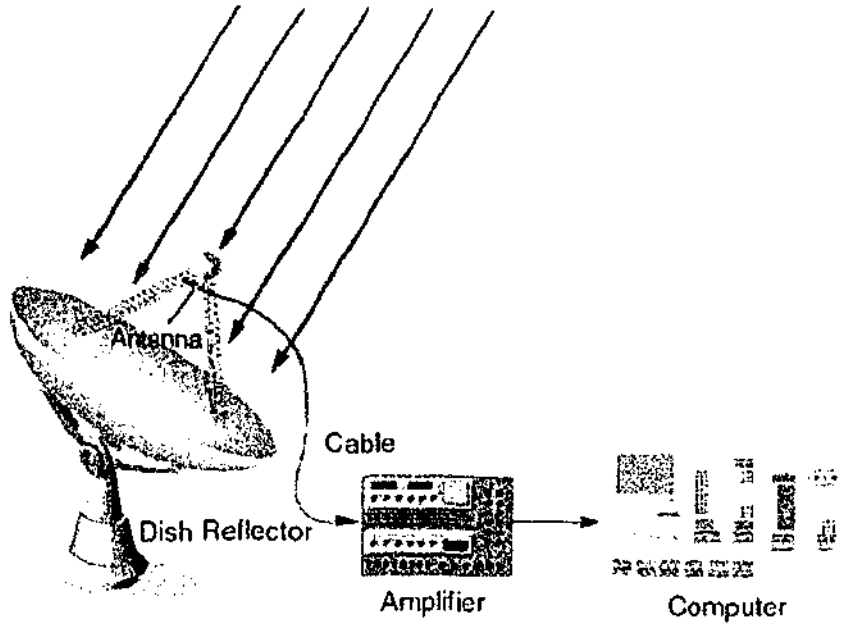


Figure 7.2: Overview of a Radio Telescope

Reproduced with permission from University of Washington

7.2 Introduction to Radio Astronomy

Just like optical astronomy, radio astronomy is the study of distant celestial objects including planets, galaxies and stars. The difference between the two areas of astronomy is shown in figure 7.3 which illustrates the electromagnetic spectrum. The spectrum shows that both radio and optical astronomers perform their examinations at differing wavelengths.

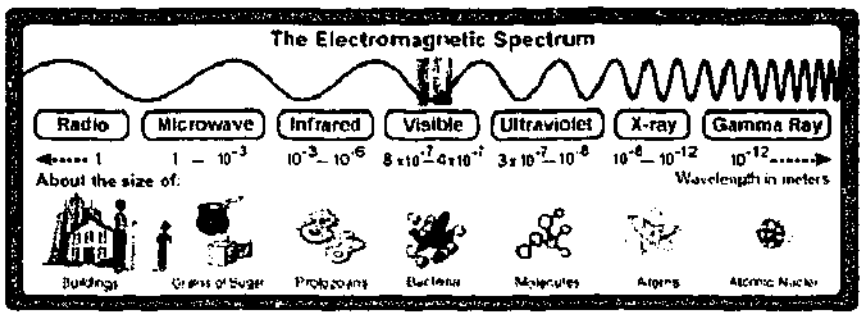


Figure 7.3: Electromagnetic Spectrum

Reproduced with permission from the Space Telescope Science Institute

The advantage that radio astronomers have over their optical counterparts is the ability to be able to perform observations around the clock. This provides radio astronomers with a much larger observation window. However even though there is a larger observation window, radio observations can take a considerable period of time to complete and the observations can only proceed if the point source to be observed is above the horizon. During the daylight hours, radio telescopes normally perform observations on point sources which are close by due to the fact that the sun emits a considerable number of radio waves which can interfere and introduce 'noise' to an observation.

Radio astronomers perform their observations by using radio telescopes (refer to section 7.1.3) to detect and analyse the radio waves emitted from the various celestial bodies being observed. As mentioned, the radio telescope receives these signals, filters out the 'noise' and then passes the results onto a computer for processing.

A problem radio astronomers face is the actual length of the wave itself as is illustrated in figure 7.3. The wavelengths can vary from a couple of millimetres through to a couple of metres in length compared to wavelengths in optical astronomy which are only a fraction of this size. To provide a clear and sharp picture, the receiving area of the telescope must be many times larger than the wavelength being receiving (regardless as to whether it is optical or radio). Depending on the wavelength size being observed, it may be that no single telescope is capable of providing a clear picture.

To address this problem, a number of radio observatories use a process called 'interferometry' which allows a number of telescopes scattered over a region to be linked to one another and take a sky sample which is the total diameter of the distance between the telescopes. The process of 'interferometry' involves having all of the radio telescopes pointing toward the same observation point source. All the data from the various telescopes is fed into a special electronic device known as a correlator.

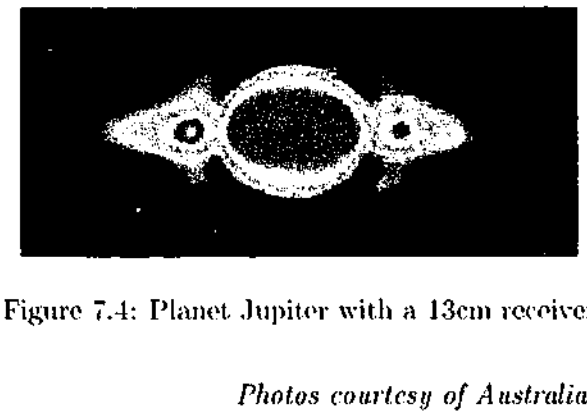


Figure 7.4: Planet Jupiter with a 13cm receiver

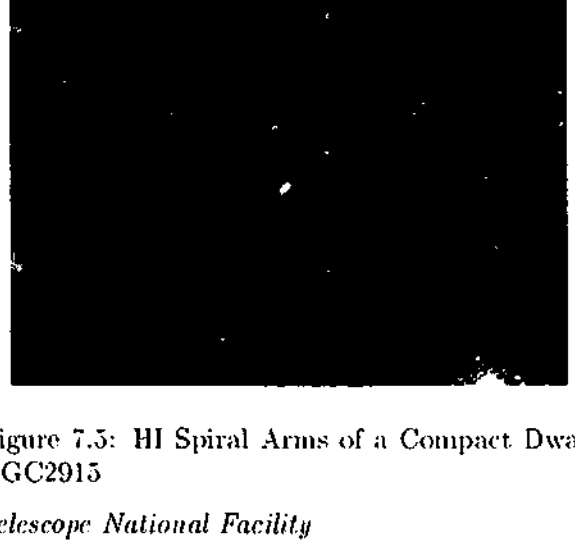


Figure 7.5: HI Spiral Arms of a Compact Dwarf NGC2915

Photos courtesy of Australian Telescope National Facility

The role of the correlator is to take a pair of inputs and compare them with one another. For example, in a case where there are three antennas operating, the correlator would compare the data feeds from antennas 1 and 2, 1 and 3 and then the remaining combination of 2 and 3. Put simply, the correlator identifies those artefacts which appear in only one of the data streams and not in the corresponding pair and removes the offending artefact. At the end of the observation, the data stream reflects all that the antennas observed. Figure 7.4 represents an observation after the incoming data stream has been processed offline while figure 7.5 represents the HI (hydrogen) spiral arms surrounding a blue compact dwarf known as NGC2915.

7.3 Implementation

Before examining the case study architecture of the Component Oriented Reconfiguration Environment & Scheduling (CORES) model in a practical context, it is necessary to first understand the implementation of the modified radio astronomy scenario into which CORES was incorporated. As previously mentioned the implementation developed for this thesis is based upon the radio telescope

system operating at the Australian Telescope Compact Array located at Culgoora. It should be noted however that the implementation which makes the CORES system is an abstracted version of the one at Culgoora and does not perform all of the functions that occur within the radio telescope array.

The case study used for this thesis is an abstracted simulation of a radio telescope observatory and provides for coordinating support, controlling and processing the flow of instructions to and from six radio telescopes simultaneously with the aid of an abbreviated antenna interface shown in figure 7.6. The complete antenna interface can be found in appendix A.1.1. The antenna interface used throughout this implementation is a subset of the operations that an actual radio telescope can perform.

```
interface Antenna
{
 // Provide a method call for the antenna to slew into position
 oneway void track(in double rightAscension, in double declination,
                   in double trackUntil, ...);
 // Bring the complete antenna to a stop
 oneway void stopAntenna(in stringStructure projectLine,
                         in stringStructure informationLine);
 // Stow the antenna
 oneway void stowAntenna(in stringStructure projectLine,
                         in stringStructure informationLine);
[...]
}
```

Figure 7.6: Simplified Antenna Interface

To ensure that the CORES system is capable of operating in a distributed framework, each simulated telescope used is located on an individual SPARC workstation so as to represent each of the individual antenna control computers responsible for processing the antenna instructions. Additionally, other elements used in the implementation are spread out across the distributed network including the ObservatoryClock which synchronises the time with the various antennas and the client/user interface.

Apart from providing complete control over each individual telescope, the observation system also allows the observer to import observation files (an example is shown in appendix B.1.1) into the system for sequencing. As described in sections 5.2.4 and 6.2.1 the radio point sources contained within each observation job can be sequenced in a particular order. This order is defined and specified by the observer when lodging the job. The sequencing process is capable of operating in three different modes. These include:

- Best case scenario: Calculate the observation sequence which requires the least amount of time to complete
- Worst case scenario: Calculate the observation sequence which requires the most amount of time to complete
- No sequencing required: No sequencing is required as the observation has already been sequenced by a third party

During the sequencing process, the sequencer pays careful attention to the placement of the various radio point sources and ensures that only those observations which are viable (ie. jobs which can be performed to completion without any errors) are considered. After the sequencing process has been completed, the 'selected' observation sequence is written to a file which can then be imported into the scheduler at a later date during schedule construction.

Upon the completion of the sequencing process, the observation system is then ready to either accept more observation files for sequencing (the scheduler only accepts sequenced jobs) or is ready to build the overall schedule which controls the radio telescope resources. The scheduling period used by the radio astronomy system is defined as being thirty days from the point at which the system was started. With the scheduling period defined, the system is ready to start processing observations which have been sequenced. The process which the scheduler undertakes to determine a satisfactory starting time for an observation is referred to in section 6.2.3. It is important to note that the scheduler in this astronomy system will not actually start to construct the schedule until the observer specifies the `construct-schedule` directive. Until then, all the tasks passed into the schedule are added to a jobs list which the scheduler uses to determine which jobs actually require placement into the schedule.

Once the schedule is constructed and executed, the radio astronomy system is able to simulate the management, coordination and tracking of each radio telescope in a manner similar to that of the system based at Culgoora. The implementation only simulates the antenna drive control and does not address other technical issues such as the processing or managing of the signal being received by each of the telescopes.

A complete list of commands which are available via the user interface for the radio astronomy system including those responsible for managing the antennas actions (ie. start, stop, stow), the scheduler and the observatory clock can be found in appendix C.1.

The incorporation of the CORES system into the underlying radio astronomy system brings with it functionality which would not ordinarily be available. By utilitising CORES, the radio astronomy system is able to provide the observer with additional functionality to manipulate the way in which the system is working and how reconfigurations are handled and performed. Some of the additional benefits that CORES brings to the system include:

- Reconfiguring a resource while the system is operating
- Specifying what action should be taken when a resource is unable to process a method call
- Specifying Quality of Service arguments

Figure 5.12 illustrates how the architecture of the CORES system was incorporated into the system. As the diagram illustrates, the CORES configuration manager acts as a middleware between those clients wanting to communicate with the resource and the resource communicating back to the client. The introduction of this middleware is fundamental to the success of the CORES system.

7.3.1 Demonstrating CORES

Having completed an overview of the scenario and the implementation of the CORES system used throughout the thesis (refer to section 7.3), it is now possible to demonstrate a practical implementation of the system. In the next couple of sections several scenarios will be put to the implementation of the CORES system to validate the design goals of CORES and to confirm that it is capable of being incorporated into real-time and real-life situations:

- Sequencing tasks contained within a radio astronomy observation
- Scheduling various radio astronomy observations which have been sequenced
- Demonstration of normal operations
- Specifying various reconfiguration modes for the system
- Using Quality of Service parameters
- Reconfiguring a component within the system while operating

7.3.2 Sequencing Tasks with CORES

Figure 7.7 illustrates the process that an observer must undertake within the CORES system to be able to sequence a number of radio point sources (tasks) within the radio astronomy observation (job). By making use of the sequence command the observer is able to directly interact with the CORES sequencer and specify the order in which the tasks are to be scheduled. The default order is to sequence the tasks in such a manner as to minimise the amount of infrastructure setup time required for the job while still achieving all the objectives of the job.

Appendix B.1.1 and B.1.2 illustrates how an observation program looks before and after the sequencer has been through and examined each task. Comparing the two job files with one another illustrates how the sequencer re-orders the tasks to optimise the overall job.

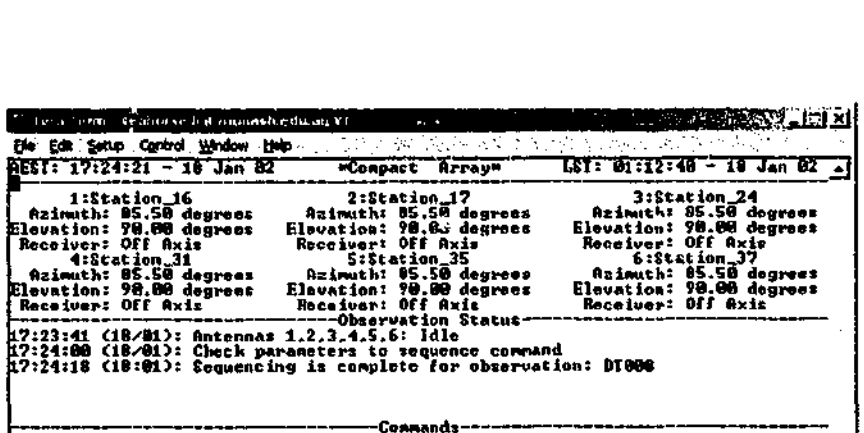


Figure 7.7: Demonstration of Sequencing Observation

7.3.3 Scheduling Jobs with CORES

The process of scheduling an observation (job) into the CORES system is achieved by the use of the `schedule` command. As each `schedule` command is issued, the scheduler reads in the contents of the sequenced file and adds the relevant data to a list containing details about all the observations awaiting scheduling. The actual scheduling process does not commence until the `construct-schedule` command is issued. Figure 7.8 illustrates the loading of three observations into the scheduler while figure 7.9 confirms that the observations were actually loaded into the system. The go command instructs the real-time execution engine to start processing the various tasks loaded into the schedule.

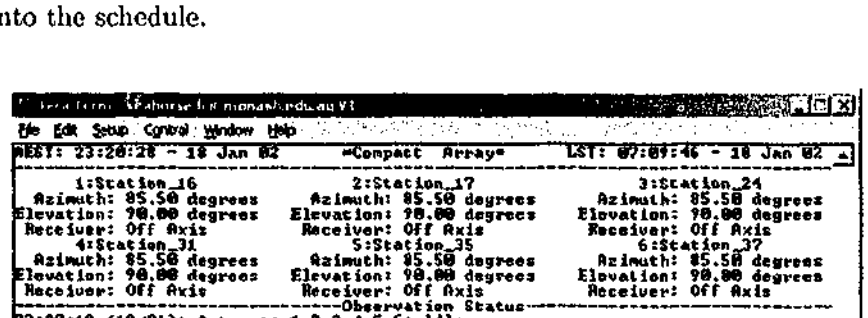


Figure 7.8: Demonstration of Loading and Scheduling Observations

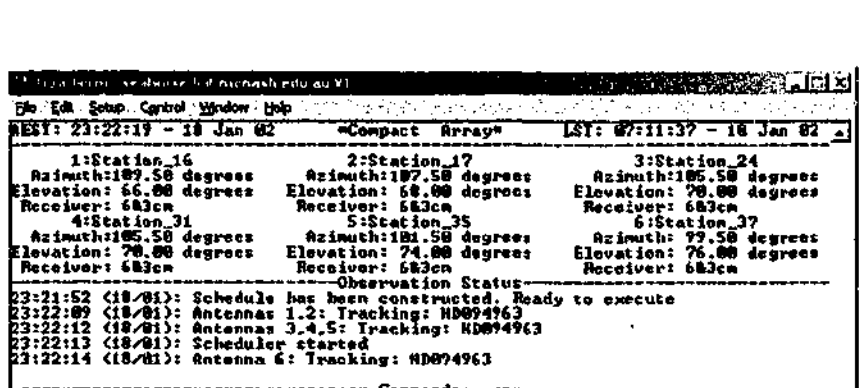


Figure 7.9: Demonstration of Executing Scheduled Observations

Once a schedule has been constructed and the scheduler executing, subsequent entries needing to be inserted into the schedule must be installed with the `insert-into-schedule` directive. This directive informs the underlying CORES scheduling process that the observation which has been passed in, must be placed into the schedule at a point which is greater than the current time reference point.

7.3.4 Normal Operations

During normal operations, the radio astronomy system is responsible for processing each task located within the schedule and coordinating the execution of those tasks to the appropriate resources. In addition to the management of the schedule, the user interface component of the system is responsible for providing real-time feedback from the various components so that their state can be monitored. The user interface allows the observer to interact and specify various commands to the CORES reconfiguration system to adjust the way in which reconfiguration events are treated and/or performed. It also provides the ability for the observer to see the alerts and messages which have been received or raised by the CORES system. Figure 7.10 shows the CORES system operating in a normal mode where tasks are being dispatched to their respective antennas and where the information pane details the components actions within the system.

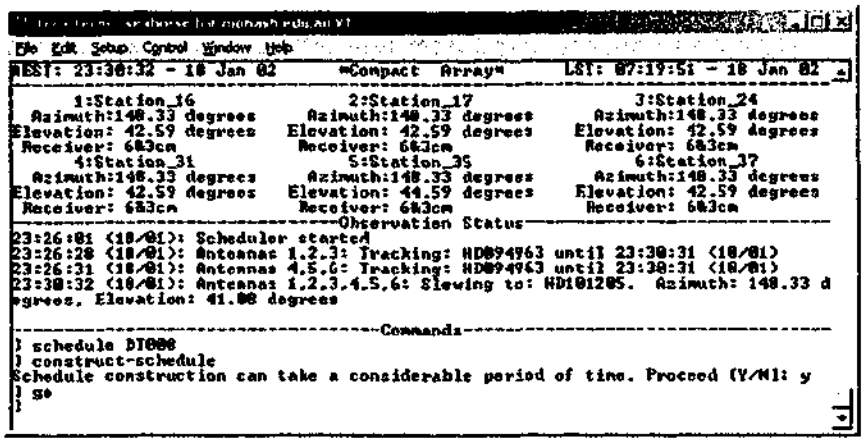


Figure 7.10: Demonstration of Normal CORES Operations

7.3.5 Specifying Reconfiguration Modes

With the inclusion of the CORES system into the simulated radio astronomy system comes the ability for the observer to be able to control how reconfiguration events within the system are processed. As mentioned in chapter 6, the CORES system is capable of providing three modes for reconfiguration activities. The modes include:

- NoWait
- WillWait
- QoSWait

Each mode introduces a new behaviour to the implementation of the system. The `NoWait` directive when specified instructs the underlying CORES implementation not to block those method calls made on components which are unavailable. An attempt to make such a call in these circumstances results in an error condition being returned to the system allowing the observer and/or system to handle the situation. The `NoWait` directive is the default reconfiguration mode for the CORES system. Another directive, `WillWait`, allows a method call to be blocked by the CORES configuration manager until either the desired component is ready to process the method call or until a certain time period specified within the `WillWait` directive elapses. In the case of the time period elapsing, an exception is raised within CORES alerting both the observer and system that the method request has not been performed and that subsequent actions may need to take place. The final reconfiguration mode provided by CORES is the `QoSWait` directive. This directive allows the observer to tie into a method call a series of Quality of Service (QoS) characteristics. Examples of these characteristics include additional resources that a method call requires or a list of constraints that need to

be satisfied before it can proceed. Using the `QoSWait` directive allows CORES to incorporate these QoS requirements with the components availability.

Figure 7.11 illustrates how the reconfiguration mode which effects all interactions within the system can be modified by the observer using the user interface. The figure also demonstrates the additional parameters which can be specified with the `reconfigure-mode` directive. Providing this command introduces greater flexibility to the CORES system and provides the observer with an element of control over method calls and how they are handled.

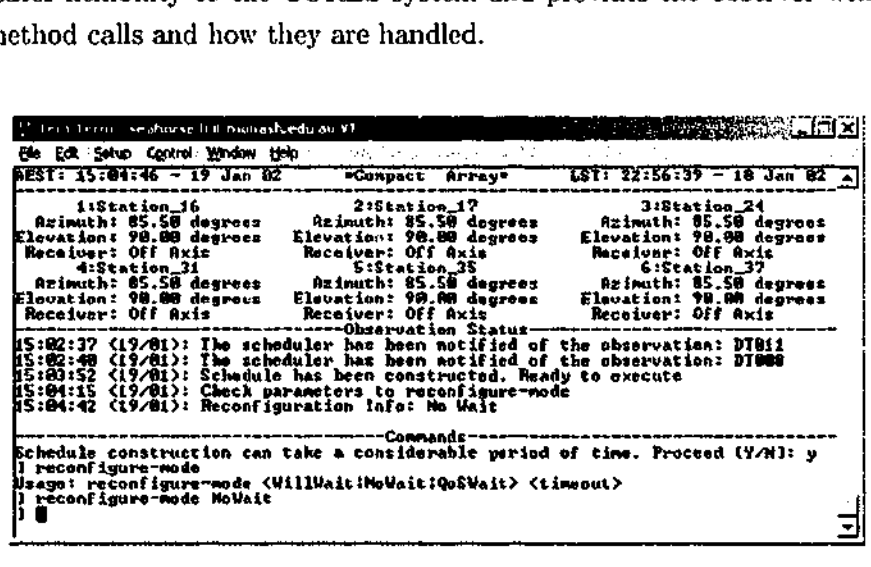


Figure 7.11: Demonstration of Modifying Reconfiguration Mode within the CORES System

7.3.6 Demonstration of Reconfiguration

As discussed in the previous section, the CORES system allows the observer to specify what actions should be performed when a component is unable to honour a method call. Figures 7.12, 7.13 and 7.14 illustrate a situation where a component running in a reconfiguration mode of `NoWait` is going through a reconfiguration process requiring it to move from one resource to another in the system. During this reconfiguration process, which can take a considerable period of time, the component being replaced receives a series of method calls requesting a particular service from the radio telescope antenna.

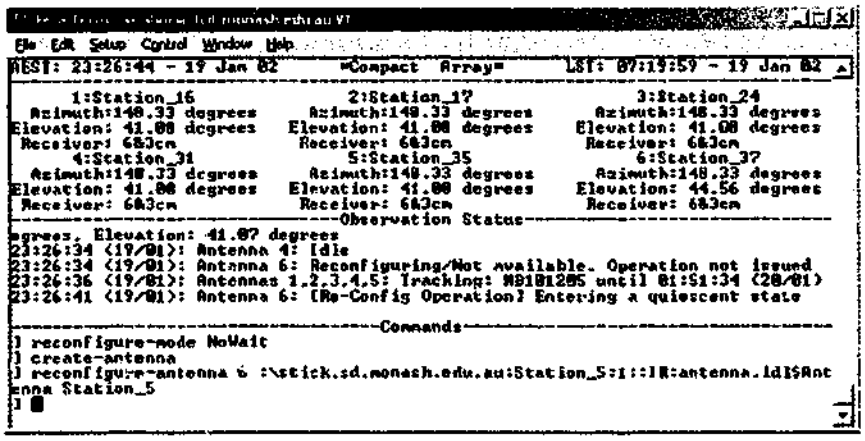


Figure 7.12: Start of the Quiescent State for Radio Telescope Antenna 6

The figures indicate that the configuration manager associated with the resource known as antenna six has intercepted the incoming method calls and has dealt with them in the manner which is reflective of the `NoWait` reconfiguration mode. With every method call arriving at the configuration manager during the reconfiguration process, the observer is notified with the help of the user interface that the component is unavailable and that the instruction has not been processed. This scenario also demonstrates the ability of the `NoWait` directive to allow the system to continue processing rather than blocking method calls directed towards the component reconfiguring itself. At the end of the reconfiguration process, the antenna is left in a 'stopped' state allowing the observer to perform any additional operations which may be necessary before the resource is committed back to the scheduler.

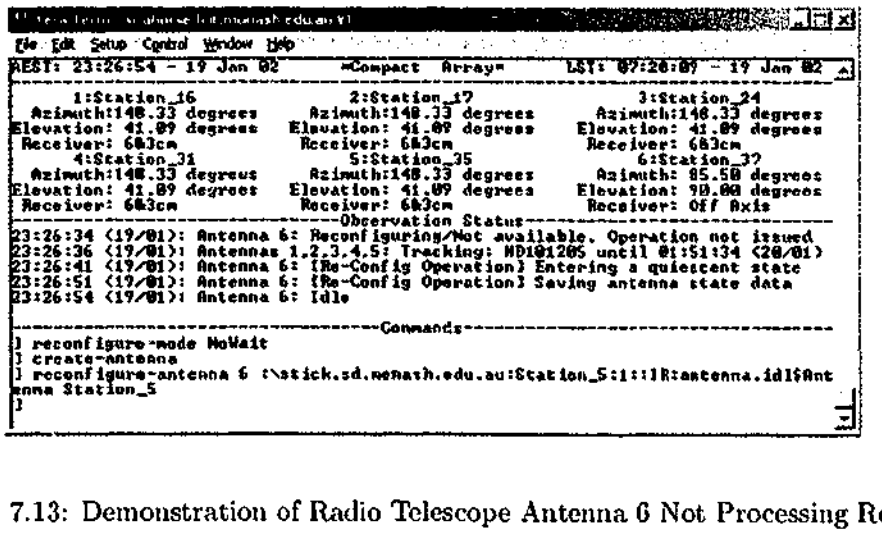


Figure 7.13: Demonstration of Radio Telescope Antenna 6 Not Processing Requests

It is important to note that although the scenario demonstrates a component being reconfigured, the configuration manager responsible for managing it will still assess its availability each time a method call arrives regardless of the actions which need to be performed. In the event that a component is unavailable, the corresponding reconfiguration mode and operations specified within the method call are executed. This may involve waiting for the component to become available, noting that the component is not available and continuing on with other tasks or waiting for a Quality of Service (QoS) level to improve before continuing on with the execution of the task which has been scheduled.

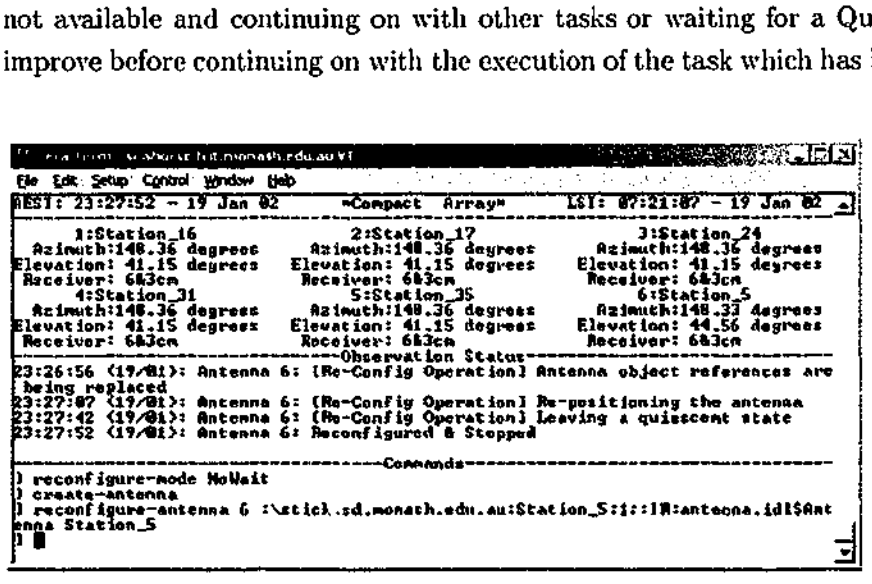


Figure 7.14: Conclusion of the Quiescent State for Radio Telescope Antenna 6

7.3.7 Controlling/Demonstration of Quality of Service

In addition to being able to control the way in which method calls are dealt with when components are not available, CORES provides the opportunity to allow the observer to interact with the Quality of Service (QoS) parameters for the various components and to adjust how they react to a differing QoS level. The implementation of the radio astronomy system with CORES integrated into it is only capable of handling limited QoS controls as each component tends to have its own individual QoS characteristics and its own way of specifying them. The implementation of the QoS support was further complicated due to each component being unique and hence making it difficult to provide a generic interface capable of supporting existing and future components.

Support for the limited version of QoS controls is restricted to controlling how a radio telescope antenna deals with the introduction of radio interference. In order to introduce QoS conditions to the underlying CORES system, the `specify-QoS` command must be used to identify the component for which the QoS limitations are destined (known as the QoSTarget) and the QoSParameters. At this point in time the only valid `specify-QoS` command which can be passed into CORES is:

`specify-QoS RadioInterference <# re-try attempts> <# delay in seconds>`. As the command suggests, it allows the observer to specify what action should be taken if the quality of the incoming signal from the radio telescope antenna is subject to radio interference. By default, when a radio telescope antenna detects radio interference it will place itself into a 'stopped' mode and alert the observer. This command allows the antenna to continue to retry a number of times at a certain time interval to establish whether or not the radio interference has subsided.

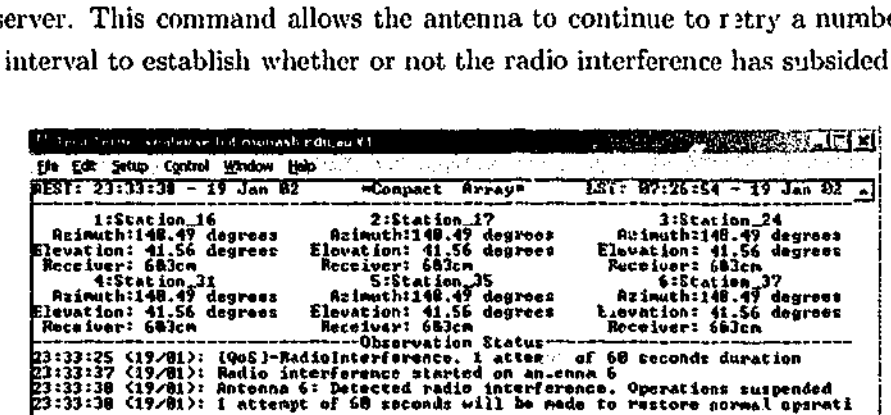


Figure 7.15: Introduction of Radio Interference to Radio Telescope Antenna 6

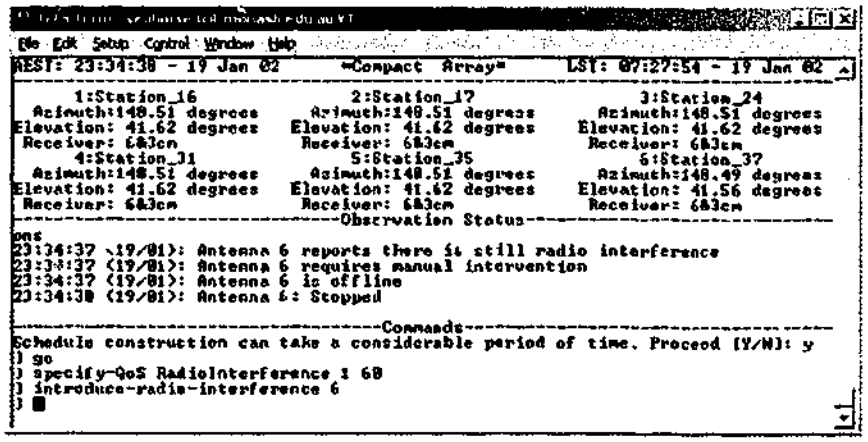


Figure 7.16: Falling Quality of Service Levels 'Stop' Radio Telescope Antenna 6

Figures 7.15 and 7.16 demonstrate how the observer can specify the relevant QoS parameters to the antenna component and introduce 'simulated' radio interference with the aid of the command `introduce-radio-interference` to determine how the radio telescope will react to the interruption. The handling of the radio interference can stop in one of two ways. The first is to issue the command:

`remove-radio-interference` which will remove the radio interference. The second approach is to wait until the radio telescope antenna has tried the appropriate number of times before the process terminates.

7.4 Limitations

The radio astronomy system does suffer one implementation problem as a result of the incorporation of the Component Oriented Reconfiguration Environment & Scheduling (CORES) system. As a result of a memory leak which exhibits itself within the configuration manager, the radio astronomy system is not capable of running for long periods of time even though the scheduler contains a month of scheduling information. The memory leak in question belongs to one of the compiled libraries or executables shipped with version 2.0 of ORBIX from Iona and has been independently confirmed with the aid of a software diagnostic tool called 'Quantify'. The exact location of the problem can be narrowed down to the underlying ORBIX code supporting the configuration managers and the object references that they use to communicate with the actual components being managed. For no apparent reason, the internal Iona code responsible for forming DII requests and transmitting them to the component fails to release the corresponding memory buffers after it has finished with it.

After a period of time, the memory associated with the configuration manager gets to a point where the configuration manager process is unable to obtain any more additional memory. Its inability to allocate any more additional memory is either the result of the process not being able to allocate it for the user process, or that the operating system itself has exhausted all of the available memory. Having reached its critical mass the configuration manager is terminated and the link between the client and the component fails resulting in the system coming to a halt.

It is envisaged that if the radio astronomy system and the underlying CORES system were to be implemented on a newer version of the ORBIX architecture the memory allocation problem would not be reproduced. At the time of development another implementation of the ORBIX architecture which would work with the C++ language bindings and the SPARCworks compiler was unavailable.

7.5 Chapter Summary

The chapter has examined a practical implementation of a radio astronomy system which had the Component Oriented Reconfiguration Environment & Scheduling (CORES) system incorporated into it. From the implementation it is possible to see that the incorporation of CORES provided the observer with the ability to be able to control the way in which method calls are handled when a component they were destined for was unavailable. The chapter also demonstrated how the observer is able to reconfigure a component while the system is operating and to a limited degree be able to

control the way in which components handle situations where their Quality of Service (QoS) levels fluctuate.

A quick overview was also provided on the background behind the practical implementation of a radio telescope observatory in addition to demonstrating the various other aspects of the system. This includes sequencing and scheduling of tasks, demonstrating a normal operating environment and specifying the various reconfiguration modes.

The major points introduced throughout the thesis as well as the addressing of the five research questions introduced in section 1.3 are dealt with in the following chapter.

Chapter 8

Conclusions

The previous chapter provided a comprehensive demonstration of the Component Oriented Reconfiguration and Environment & Scheduling (CORES) system verifying the conceptual model and illustrating how such a system can be incorporated into a real world scenario containing real-time task commitments. Additionally the system demonstrated support for end-users to control the actions performed when a component is unable to process a method call (task). Awareness in dealing with components is necessary in real-time environments, especially those operating under hard real-time constraints where the inability to perform a task at a nominated time can result in serious consequences. Systems operating within soft real-time constraints also operate within strict time frames, but the execution of each task includes a 'slip' time specifying how late a task can be performed or slip. Failure to execute tasks within their nominated slip time results in recovery paths having to be performed.

Reviewed in this chapter are the motivating factors behind the development of the CORES conceptual model and implementation which highlights problems existing in configuration management systems used both academically and commercially. Included is discussion on how the CORES model addresses these issues and a summary of the key points outlined throughout this thesis addressing the research questions proposed in section 1.3. Concluding the chapter is a brief discussion on the issues of future work identified within the thesis.

8.1 Discussion of Findings

Throughout the thesis the issue of performing reconfiguration activities within configuration management systems has been closely examined. Lack of support given to these configuration management systems for managing components operating within real-time environments has been identified. Chapter 2 introduced the objectives of the component based paradigm and discussed the important

considerations which must be undertaken when reconfiguring a component. In addition to examining the various concerns that component reconfigurations raise, the chapter presented a series of steps that software developers can undertake to introduce a level of quiescence throughout components when performing reconfiguration operations. To complement the discussion on the component based paradigm a number of architectures specifically designed to support component based developments are introduced highlighting the varying degrees of support provided to components.

From the architectures reviewed, it is evident that whilst support is provided for the development of components, none is provided to reconfigure components within real-time environments while honouring their real-time commitments. With no reconfiguration support provided at the architecture level, it becomes the responsibility of the various configuration management systems to provide their own level of support. As chapters 2 and 3 explain, each configuration management system has its own perspective on what support should be provided for components and their ability to be reconfigured. To assist with these various interpretations, a criteria was established to identify the levels of support provided for reconfiguring components within configuration management systems[1].

Chapter 3 further investigates the varying level of configuration management support by examining a number of configuration management systems used both academically and industrially and determining the support provided for reconfiguration management (especially dynamic reconfiguration management). The criteria developed in chapter 2 assisted the examination by identifying those systems capable of supporting component reconfiguration and allowing them to be categorised based on the support provided for components and the ability to reconfigure them.

Application of one of the criterion for identifying features in configuration management systems led to the establishment of four categories relative to the level of support provided for components and their dynamic reconfiguration.

The system categories include:

- Systems providing reconfiguration support during component construction
- Static component systems where reconfiguration is only supported when the system has been stopped or is in a static state
- Dynamic component management systems which allow interconnections between components to be modified whilst the system operates
- Real-Time component configuration and consistency systems which offer the same services as dynamic component management systems except that consistency and deadlocking support is performed during the reconfiguration process

From those systems examined in chapter 3 it can be concluded that a number provide support for reconfiguration management (in various forms) but none address the issue of supporting dynamic

[1]Configuration management systems can exist in their own right, or be extensions to an underlying architecture.

reconfiguration in a real-time environment. Those systems that do provide dynamic reconfiguration do so either in an environment which provides no consistency support and dependency analysis or in an environment which does not allow end-users to control the action of method calls when they can not be processed by components. Systems capable of providing dynamic reconfiguration support within a consistent environment handle method calls destined for components that are not available by blocking them indefinitely and waiting until the component becomes available. While this solution works for non real-time systems where there are no timing constraints, it will not work nor is it satisfactory for systems requiring real-time responses from real-time environments.

Chapter 4 continues the examination into the lack of support provided to end-users when dealing with reconfiguring components within real-time systems by examining the configuration management system groupings detailed in chapter 3 which were identified by the criteria. Investigating these groupings revealed that although advancements have been made to assist developers and end-users during system construction, there is still a noticeable gap when it comes to providing dynamic reconfiguration management support to components operating under real-time constraints.

Given this lack of support, the chapter discusses the impact on systems when there are no controls in place to manage real-time reconfiguration requests. Specifically, this lack of support is of concern to hard real-time environments operating in real-time systems which require immediate notification when a task can not be performed as scheduled. Chapter 4 highlights the recurring theme throughout the thesis that configuration management systems whether designed for academic or industrial use, still do not address the issue of providing end-users with control over method calls when components are not available. The chapter also reaffirms that the default behaviour of indefinitely blocking method calls while components are unavailable is unacceptable for real-time environments where the success or failure of a method call to execute should be reported back immediately so that actions can be taken. To address this lack of support for components operating in a real-time environment, the chapter introduces a framework to overcome problems with current configuration management systems.

Chapter 5 develops the framework presented in chapter 4 and the issues raised in chapters 2 and 3 by presenting a conceptual model known as the Component Oriented Reconfiguration Environment & Scheduling (CORES) model. The development of CORES allows end-users to control the actions of method calls when they can not be performed by components and provides an environment where tasks can be sequenced and scheduled accordingly to suit their respective real-time commitments.

CORES is supported by a number of formal definitions which detail the various aspects of the model. Those aspects unique to the CORES model include the optimisation and sequencing of tasks to form a path through a job while ensuring that real-time constraints of the job are met. CORES also provides the ability to schedule jobs to ensure that the overall real-time commitments of the system are met. In addition to providing facilities for tasks and jobs to be sequenced and scheduled, the model introduces an architecture that provides software developers and end-users with the ability

to control method calls. Although the implementation is not discussed in the chapter, the model does introduce the concept of a configuration manager and explains how it is placed between the client and component providing the service. As explained in the chapter, strategically placing the configuration manager in this position allows for the interception and dealing of method calls as they pass from client to component. To provide the desired control over the method calls being sent to components, the CORES model introduces three specific actions which can be performed by the configuration manager. These actions control the behaviour of method calls when components are unavailable.

It is through these controls and the architecture of the CORES model that an environment is provided giving software developers and end-users control over method calls and the ability to ensure that tasks meet their respective real-time deadlines. In addition to providing the required support for software developers and end-users in a real-time environment, the CORES model has been designed so that it can be incorporated into current distributed object frameworks without the need to introduce additional keywords or make any significant changes to the underlying architecture.

Chapter 6 verified the formal definitions within the CORES model by providing an implementation that demonstrates the concepts expressed within the conceptual model. Throughout the chapter a number of the smaller supporting sub-systems specific to the implementation of CORES are detailed including how exceptions are used to implement the controls which influence the behaviour of method calls and how the user-interface can control the various aspects of the system. Verification of the CORES model also allows for the identification of those areas within the conceptual model which were either not formally or poorly defined. As a consequence these areas could not be directly mapped to the implementation. In these situations, the implementation proposes alternative solutions so that the same functionality could be achieved.

The implementation of the CORES model within the CORBA framework confirmed the design objectives stated in the model mentioned in chapter 5 by providing a model which does not require any distributed object framework to be extensively re-designed. The CORES system achieved this by integrating the configuration manager into the distributed object environment between the client requesting the service and the component providing it and by making use of standardised keywords and concepts. This integration allows the configuration manager to be deployed into a range of distributed object frameworks.

Chapter 7 presented the implementation of the CORES model by incorporating it into a real world scenario where a need exists to control and interact with tasks in real-time. To illustrate this scenario, the chapter showed how by using an abstracted implementation of a radio telescope observatory, it is possible to demonstrate the definitions and formalisms used to create the CORES model and implementation. The features most notable in the case study is the ability to reconfigure and isolate components within the system while it is running and allowing end-users to control what actions should be performed when method calls are sent to components that are not available. This is

clearly shown in chapter 7 where a component is being reconfigured and where it receives a number of incoming method calls. From the scenario presented it can be seen that the incoming method calls are rejected and returned to the client which sent them initially while other actions in the system continue to operate as normal. The CORES model also provides limited support for incorporating Quality of Service (QoS) characteristics into components. An example of how Quality of Service (QoS) arguments can influence the availability of a component is presented in the chapter.

8.2 Research Questions

Section 1.3 introduced a number of relevant research questions designed to assist the investigation into the issue of being able to perform dynamic reconfigurations whilst operating under real-time constraints. Each question was tested and answered throughout the thesis and is summarised as follows.

8.2.1 Research Question 1

To what extent do the current reconfiguration management environments provide support for dynamic reconfiguration?

Over the last couple of decades there has been tremendous development in the field of configuration management systems. The developments have included the provision of reconfiguration support for the construction of components through to the development of architectures allowing software developers or end-users to inspect and in some cases modify the bindings and inter-connections between components. Such advances have led to the creation of a number of Configuration Definition Languages (CDL)'s such as DARWIN which allows developers to explicitly define bindings between components.

Current reconfiguration management systems are capable of supporting a range of activities and can be divided into two distinct groups. The first group includes systems not capable of reconfiguring components while the system is operating and those which are more sophisticated allowing software developers and end-users to manipulate and control the environment that components operate within while the system continues to operate. The second group embraces those systems capable of managing the component environment through a number of techniques. One technique involves implementing a series of routines designed to introduce a state of quiescence within the immediate area of the system where the reconfiguration is to take place. In addition to manipulating the environment, these systems normally perform a series of consistency checks throughout the environment as part of the reconfiguration process in order to detect any inconsistencies which may have been introduced as a result of the reconfiguration process or a new component being integrated.

8.2.2 Research Question 2

What support exists for software developers or end-users to deal with a situation where a method call is made on a component which is in the state of reconfiguring itself or otherwise not available whilst operating under real-time constraints?

Significant developments have been made within the software engineering community to provide better support for the component based paradigm and reconfiguration management environments. These developments have centered upon the development of configuration definition languages such as DARWIN and proposed extensions to architectures such as REGIS. Although a significant amount of work has been undertaken to improve the reconfiguration environment, very little has been done on managing method calls and their real-time commitments when sent to components unable to honour the requests due to reconfiguration activities or other obligations.

Configuration management systems which do provide dynamic reconfiguration component services address the issue by blocking all method calls destined for the component which is unavailable. These incoming method calls remain blocked until such time that the component becomes available and is ready to process them. Such an approach to method calls which can not be performed is not acceptable in real-time systems. In such circumstances the system should be notified immediately and in the case of soft real-time environments a notification should be sent to allow the execution of consequential actions to proceed so that the issue of the operation not being performed on time can be addressed.

8.2.3 Research Question 3

What are the requirements of a conceptual model capable of addressing real-time reconfiguration and what additional support is needed when deploying it within a non-trivial environment where software developers and end-users can specify what actions should be taken if a method call is made upon a component reconfiguring itself or is otherwise not available?

To address the lack of support provided for managing method calls made on components that are not available within real-time environments, this thesis presents the Component Oriented Reconfiguration & Environment Scheduling (CORES) model. To provide the functionality required the model sets out a list of requirements, makes use of a number of formal definitions and uses algorithms specifically designed to handle the sequencing of tasks within jobs while at the same time ensuring that real-time constraints are met. In addition to handling the sequencing and scheduling of jobs the model provides support via the configuration manager for software developers and end-users to specify what actions should take place if a component is not available. The configuration manager provides for three actions. These actions include waiting for the component to become available, do not wait for an unavailable component or wait until a certain Quality of Service (QoS) level is achieved.

The application of the CORES model within an abstracted implementation of a radio telescope array confirms that the model is capable of being implemented and used within real world scenarios with real-time commitments. This proves that the CORES system is relevant for use by software developers and end-users and provides the required flexibility. To confirm this flexibility a number of tests[2] were performed on both the CORES conceptual model and its implementation to verify its capability of managing method calls. The management of components within the CORES system also extends to allowing manipulations while it is running. While performing these manipulations the system constantly and consciously monitors the state of all tasks ensuring real-time commitments are met. In the event that these conditions are not met, alternate operations are performed to correct the state of the system.

8.2.4 Research Question 4

Is it possible to integrate into the proposed model the capability to calculate the impact on the overall schedule and to identify what implications may exist when various sequencing algorithms (best case scenario vs. worst case scenario) are used to schedule tasks and jobs?

The sequencing and scheduling engine incorporated into CORES is capable of scheduling tasks and jobs in a number of ways. Techniques provided by CORES include scheduling based on a pre-existing order contained within the job definition file or by using a best case (tasks completed on or before their allotted time) or worst case scenario (tasks require either the total allotted time or additional time) algorithm.

From the analysis of the definitions within the CORES system and experimentation with the scheduler it is conceivable that schedules within a real-time system can be dramatically effected as a result of sequencing tasks in a particular order. This is evident in real-time environments where resources used by tasks have to be tightly controlled and monitored especially when they do not finish within their allotted time. Circumstances such as these can result in significant changes and flow-on effects to the entire schedule and effect the entire system. A fine balance must be reached between those tasks sequenced using the best case scenario and those with the worst case scenario. Achieving this balance provides the scheduler with the flexibility required to deal with situations where tasks do not complete on time.

[2] A systematic testing approach applied to re-occurring reconfiguration requests in a dynamic real-time environment confirmed the flexibility of the CORES system.

8.2.5 Research Question 5

What impact does the incorporation of Quality of Service (QoS) arguments have on the proposed model so that software developers or end-users can associate QoS characteristics with components and then have those characteristics used in conjunction with the respective real-time commitments of the system to calculate a components availability?

The CORES system presented includes the ability for end-users to specify Quality of Service (QoS) `QoSCriteria` data designed to be integrated into the decision making process effecting the availability of a component within a real-time environment. Although an analysis was conducted into the feasibility of incorporating QoS specifications into the CORES model, it concluded that a comprehensive QoS solution could not be implemented due to the lack of standards in place for communicating QoS information with third party components. Further complicating the integration of the QoS implementation into CORES is the ability for each component developer to have their own unique way of specifying QoS requirements making it impossible for total integration.

8.3 Future Work

To date considerable work has been carried out with regard to configuration management. Improvements have been made to various configuration management systems allowing reconfiguration of static objects through to the dynamic and real-time configuration management systems available today. However more work needs to be undertaken with regard to the integration of consistency checking, introduction of versioning control and the overall management and coordination of components operating in real-time environments. The problem of providing support to method calls destined for reconfiguring components in real-time environments is addressed in the thesis by the implementation of the CORES system. It is concluded however that additional work could be undertaken in this area in the future. Areas which could be subject to further investigation or possibly be integrated into CORES include:

- The generation of a software tool capable of examining the interface definition of a component and from that definition automatically generating the necessary code required to build a configuration manager for the component. This tool might aid software developers by including elements and functionality from the CORES system
- The incorporation of the underlying principles in the CORES system being integrated into another configuration management system such as the extensions proposed for REGIS which is capable of performing additional functions such as consistency checking and a more sophisticated engine for component reconfiguration. The ideas presented in CORES could provide the additional support required for such an architecture and enable it to support real-time reconfiguration rather than blocking method calls during the reconfiguration of a component

- Construction of a graphical tool which could be used to illustrate the connection status between the client, configuration manager and the component. Such a tool could be used to control the reconfiguration and quiescence processes as well as controlling the directives which govern method calls sent to components that are unavailable. A graphical tool could also be used to visualise the interconnections existing within a system and possibly provide real-time information about the interconnections and their use

8.4 Concluding Remarks

It is not proposed that the CORES system will solve all the problems related to dynamic reconfiguration management within a real-time environment, however it does address and demonstrate how method calls destined for components being reconfigured or otherwise not available can be handled. The CORES system aims to provide a way in which tasks can be sequenced and scheduled within an environment which is sensitive to real-time constraints. CORES should not be viewed in isolation as a complete dynamic real-time reconfiguration environment but more as a system which provides a solution to the management of components in a real-time context. In an industry where there is a proliferation of real-time components, it will become increasing important to provide additional support for their management and on-going reconfiguration within their own native real-time environment while they continue to operate.

Appendix A

Interface Definition Language

A.1 Interface Definition Files

This appendix contains a series of Interface Definition Language (IDL) files which were used to implement the Component Oriented Reconfiguration Environment & Scheduling system. The IDL files explicitly reproduced here and used throughout the thesis are subject to the following copyright notice:

A.1.1 antenna.idl

The file `antenna.idl` contains the interface definition for the simplified radio telescope antenna.

```
#include "dataStructures.idl"

interface Antenna
{
 // Provide a method call for the antenna to slew into position
 oneway void slew(in double azimuth, in double elevation,
                  in stringStructure projectLine,
                  in stringStructure informationLine);

 // Provide a method call for the antenna to track a celestial target
 oneway void track(in double rightAscension, in double declination,
                   in double trackUntil,
                   in stringStructure projectLine,
                   in stringStructure informationLine);

 // Provide a method call for the rotation of the receiver turret
 oneway void rotateReceiverTurret(in antennaReceiver desiredReceiver,
                                  in stringStructure projectLine,
                                  in stringStructure informationLine);

 // Provide a method call to allow the allocated time period to be specified
 oneway void allocateIdle(in double trackUntil,
                          in stringStructure projectLine,
                          in stringStructure informationLine);

 // Provide a method call to get the antennas current configuration
 boolean getAntennaConfig(out double azimuth, out double elevation,
                          out double receiverTurret,
                          out short radioInterference);
// out antennaReceiver receiverTurret);

 // Provide a method call to get the antennas current status
 boolean getAntennaStatus(out stringStructure projectLine,
                          out stringStructure informationLine);

 // Bring the complete antenna to a stop
 oneway void stopAntenna(in stringStructure projectLine,
                         in stringStructure informationLine);

 // Start thr antenna up again
 oneway void startAntenna(in stringStructure projectLine,
                          in stringStructure informationLine);

 // Stow the antenna
```

```
oneway void stowAntenna(in stringStructure projectLine,
                        in stringStructure informationLine);

// Start the simulation of radio interference
boolean startRadioInterference();

// Stop the simulation of radio interference
boolean stopRadioInterference();

// Check for radio interference
boolean checkForRadioInterference(out short radioInterference);

// Restore antenna data
boolean restoreAntennaData(in stringStructure antennaInformation);

// Store antenna data
boolean saveAntennaData(out stringStructure antennaInformation);

// Update antenna information line
boolean updateAntennaInformation(in stringStructure informationLine);

 // Enable antenna
 oneway void enableAntenna();

 // Disable antenna
 oneway void disableAntenna();
};

interface AntennaFactory
{
 Antenna newAntenna();
 void freeAntenna(in Antenna aAntenna);
};
```

A.1.2 antennaManager.idl

The file `antennaManager.idl` contains the interface definition for the configuration manager which is responsible for looking after the antenna component. Note that the configuration manager definition contains the declarations for the reconfiguration exceptions.

```
#include "availability.idl"
#include "dataStructures.idl"

interface AntennaManager
{
 // Provide a method call for the antenna to slew into position
 boolean slew(in double azimuth, in double elevation,
              in stringStructure projectLine,
              in stringStructure informationLine,
              out ExceptionHandling::ManualException aManualException,
              in ModuleAvailability::ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
         raises(ModuleAvailability::ObjectBeingReconfigured,
                ModuleAvailability::QoSObjectBeingReconfigured,
                ModuleAvailability::QoSObjectInvalid);

 // Provide a method call for the antenna to track a celestial target
 boolean track(in double rightAscension, in double declination,
               in double trackUntil, in stringStructure projectLine,
               in stringStructure informationLine,
               out ExceptionHandling::ManualException aManualException,
               in ModuleAvailability::ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
         raises(ModuleAvailability::ObjectBeingReconfigured,
                ModuleAvailability::QoSObjectBeingReconfigured,
                ModuleAvailability::QoSObjectInvalid);

 // Provide a method call for the rotation of the receiver turret
 boolean rotateReceiverTurret(in antennaReceiver desiredReceiver,
                              in stringStructure projectLine,
                              in stringStructure informationLine,
                              out ExceptionHandling::ManualException
                                        aManualException,
                              in ModuleAvailability::
                                   ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
         raises(ModuleAvailability::ObjectBeingReconfigured,
                ModuleAvailability::QoSObjectBeingReconfigured,
                ModuleAvailability::QoSObjectInvalid);

 // Provide a method call to allow the allocated time period to be specified
 boolean allocateIdle(in double trackUntil,
```

```
                    in stringStructure projectLine,
                    in stringStructure informationLine,
                    out ExceptionHandling::ManualException aManualException,
                    in ModuleAvailability::ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Provide a method call to stop the antenna
boolean stopAntenna(in stringStructure projectLine,
                    in stringStructure informationLine,
                    out ExceptionHandling::ManualException aManualException,
                    in ModuleAvailability::ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Provide a method call to stow the antenna
boolean stowAntenna(in stringStructure projectLine,
                    in stringStructure informationLine,
                    out ExceptionHandling::ManualException aManualException,
                    in ModuleAvailability::ModuleUnAvailableInstruction
                                        theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Start the simulation of radio interference
boolean startRadioInterference(out ExceptionHandling::ManualException
                                                    aManualException,
                               in ModuleAvailability::
                                    ModuleUnAvailableInstruction
                                    theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Stop the simulation of radio interference
boolean stopRadioInterference(in ModuleAvailability::
                                    ModuleUnAvailableInstruction
                                    theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Check for radio interference
```

```
boolean checkForRadioInterference(out short radioInterference,
                                  in ModuleAvailability::
                                   ModuleUnAvailableInstruction
                                    theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Provide a method call to get the antennas current configuration
boolean getAntennaConfig(out double azimuth, out double elevation,
// out antennaReceiver receiverTurret,
                         out double receiverTurret,
                         out short radioInterferencePresent,
                         in ModuleAvailability::ModuleUnAvailableInstruction
                                                theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Provide a method call to get the antennas current status
boolean getAntennaStatus(out stringStructure stationID,
                         out stringStructure informationLine,
                         in ModuleAvailability::ModuleUnAvailableInstruction
                            theAvailabilityInstructions)
        raises(ModuleAvailability::ObjectBeingReconfigured,
               ModuleAvailability::QoSObjectBeingReconfigured,
               ModuleAvailability::QoSObjectInvalid);

// Provide a method to process the QoS requirements
boolean processQoSRequirements(in QoSProcessing::QoSAction QoSToSatisfy,
                               in ModuleAvailability::
                                   ModuleUnAvailableInstruction
                                    theAvailabilityInstructions,
                               out ExceptionHandling::ManualException
                                    aManualException)
        raises(QoSProcessing::NoRadioInterference,
               QoSProcessing::RadioInterference,
               QoSProcessing::NotHandlingRadioInterference);

// Configuration Operations
// Provide a method call to establish the antenna
boolean establishComponentConnection(in string antennaObjectReference);

// Provide a method call to allow for the reconfiguration of the antenna
oneway void reconfigureComponentConnection(in stringStructure
                                            antennaObjectReference,
                                           in stringStructure
                                            stationTitle);
```

```
 // Provide a method call to set the time which the component is available
 boolean setComponentAvailabilityTime(in double timeComponentWillBeReady);

 // Provide a method call which allows the manager to be shutdown;
 boolean shutdownManager(in boolean shutdownFlag);

 // Provide a method call which allows the antenna to be taken off-line
 boolean antennaOffline(in boolean offlineFlag);

 // Provide a method call which brings the antenna back on-line
 boolean antennaOnline(in boolean onlineFlag);

 // Provide a method call to get the station ID
 boolean getStationID(out stringStructure aStationID);

 // Provide a method call to pass the station ID
 boolean setStationID(in stringStructure aStationID);
};

interface AntennaManagerFactory
{
 AntennaManager newAntennaManager();
 void freeAntennaManager(in AntennaManager aAntennaManager);
};
```

A.1.3 availability.idl

The file `availability.idl` contains a number of definitions which CORES uses to provide additional flexibility. These definitions include the `ExceptionHandling`, `QoSProcessing`, `ModuleAvailability` types.

The `Exception Handling` object defines those exceptions that CORES transmits to the client using the `ManualException` data structure to indicate an event. The `QoSProcessing` object defines the Quality of Service characteristics examined when the system is operating in a QoS mode. The `ModuleAvailability` object defines those actions that take place when a component is unavailable. This object is included within every method call made from the client to the configuration manager. The IDL file also defines those exceptions raised between the configuration manager and the routine responsible for processing the `ModuleAvailability` object data.

```
module ExceptionHandling
{
 enum ExceptionTypes {NoException,                    // No exception
                      ObjectBeingReconfigured,        // Object is being
                                                      // reconfigured
                      QoSObjectBeingReconfigured,     // Object is being
                                                      // reconfigured but
                                                      // won't be ready by
                                                      // QoS data
                      QoSObjectInvalid,               // Incorrect QoS data
                                                      // supplied
                      NoRadioInterference,            // Radio interference
                                                      // has ceased
                      RadioInterference,              // Radio interference
                                                      // is still present
                      NotHandlingRadioInterference}; // No handler for
                                                      // radio interference

 struct ManualException
 {
  ExceptionTypes exceptionType;
  double timeAtWhichComponentIsAvailable;
 };
};

module QoSProcessing
{
 enum QoSAction {RadioInterference_Check,  // Quality of Service
                                           // characteristics for radio
                                           // interference need to be
                                           // examined
```

```
                NoAction};                    // No action specified

 exception NoRadioInterference { string reason; };
 exception RadioInterference { string reason; };
 exception NotHandlingRadioInterference { string reason; };
};

module ModuleAvailability
{
 enum ModuleUnAvailableAction {NoWait,        // Don't wait if object is
                                              // unavailable
                               WillWait,      // Call will wait until object is
                                              // available
                               QoSWait};      // Quality of Service
                                              // Characteristics will
                                              // determine whether or not the
                                              // call will block

 struct QoSDetail
 {
  QoSProcessing::QoSAction theQoSActions;
  short numberOfAttempts;
  short delayBetweenEvents;
  double loadAverage;
  any extraQoSData;
 };


 struct ModuleAvailabilityData
 {
  long timePreparedToWait;
  long timeComponentIsAvailableAt;
  short numberOfQoSContracts;
  QoSDetail allOfTheQoSParameters[5];
 };

 struct ModuleUnAvailableInstruction
 {
  ModuleUnAvailableAction anInstruction;
  ModuleAvailabilityData someQoSData;
 };

 exception ObjectBeingReconfigured { string reason; };
 exception QoSObjectBeingReconfigured { ModuleAvailabilityData
                                        specifiedQoSRequirements; };
 exception QoSObjectInvalid { string reason; };
};
```

Appendix B

Job Definition Files

B.1 Job Definition Files

This appendix contains a sample of the job definition files which are used to provide the radio telescope antennas with a series of tasks to perform. The underlying astronomical principles used throughout the CORES system and the job definition files can be found in Roth (1975), Kutner (1987), Roy (1988) and Zombeck (1990). Additionally, the radio point sources used for the various observations were selected with the assistance of the Sky Catalogue 2000.0 which was written by Hirshfield and Sinnott (1995).

B.1.1 Job Defintion File

The contents of the file DT008.obs represents a series of tasks awaiting for sequencing and scheduling into the CORES system.

% Observation file for C186b Observation
Title of Project: *High−spatial resolution observations of Eta Carinae*
Project Coordinators: *R.A. Duncan* (ATNF, AU)
Project ID: *C186b*
Receiver Type: 2
Antennas Required: 123456
Number of Celestial Points: 3
Number of Observation runs: 4
−−− Celestial Point Data #1 −−−
Celestial Point Name: *ETA_CAR*
Celestial Observation Time: 540
Right Ascension Hour: 10
Right Ascension Minute: 45
Right Ascension Second: 3.60
Declination Degree: −59
Declination Minute: 41
Declination Second: 3.00
−−− Celestial Point Data #2 −−−
Celestial Point Name: 1045−62
Celestial Observation Time: 45
Right Ascension Hour: 10
Right Ascension Minute: 47
Right Ascension Second: 42.95
Declination Degree: −62
Declination Minute: 17
Declination Second: 14.53
−−− Celestial Point Data #2 −−−
Celestial Point Name: 0823−500
Celestial Observation Time: 56
Right Ascension Hour: 8
Right Ascension Minute: 25
Right Ascension Second: 26.87
Declination Degree: −50
Declination Minute: 10
Declination Second: 38.49

B.1.2 Sequenced Job Defintion File

The contents of the file DT008.psc represents a series of tasks which have been sequenced by the CORES system and is awaiting scheduling.

% Sequenced Observation file for C186b Observation
Title of Project: High−spatial resolution observations of Eta Carinae
Project Coordinators: R.A. Duncan (ATNF, AU)
Project ID: C186b
Receiver Type: 2
Antennas Required: 123456
Number of Celestial Points: 3
Number of Observation runs: 4
Calculated Observation time: 38502.42
Approx start time of Observation: 17:59:00
−−− Celestial Point Data # 1 −−−
Celestial Point Name: 0823−500
Celestial Observation Time: 3360.00
Right Ascension Hour: 8
Right Ascension Minute: 25
Right Ascension Second: 26.87
Declination Degree: −50
Declination Minute: 10
Declination Second: 38.49
−−− Celestial Point Data # 2 −−−
Celestial Point Name: ETA_CAR
Celestial Observation Time: 32400.00
Right Ascension Hour: 10
Right Ascension Minute: 45
Right Ascension Second: 3.60
Declination Degree: −59
Declination Minute: 41
Declination Second: 3.00
−−− Celestial Point Data # 3 −−−
Celestial Point Name: 1045−62
Celestial Observation Time: 2700.00
Right Ascension Hour: 10
Right Ascension Minute: 47
Right Ascension Second: 42.95
Declination Degree: −62
Declination Minute: 17
Declination Second: 14.53

Appendix C

CORES Command Summary

C.1 CORES Command Summary

This appendix contains a list of commands that the user interface to the CORES system will accept and process. The commands listed here allow tasks to be sequenced and scheduled as well as performing a number of other activities (ie. configuration operations) which relate to the operation of the system.

Command: version
Parameters: <none>
Description: Displays the version information associated with the CORES system.

Command: sequence
Parameters: <jobName> [max|min|preserve]
Description: Specify the observation to sequence. The **sequence** command also allows the operator to specify how they want the observation to be sequenced.

Command: schedule
Parameters: <jobName>
Description: Specify the sequenced observation to admit to the schedule.

Command: insert-into-schedule
Parameters: <jobName>
Description: Insert the nominated sequenced observation into the schedule at a point greater than

the current time reference.

Command: construct-schedule
Parameters: <none>
Description: Instruct the CORES system to build the schedule.

Command: go
Parameters: <none>
Description: Execute the observations which have been loaded into the scheduler.

Command: start
Parameters: <none>
Description: Execute the observations which have been loaded into the scheduler.

Command: stop
Parameters: <none>
Description: Stop the execution of the observations.

Command: stow
Parameters: <none>
Description: Stow the execution of the observations and return the radio telescopes to their stow position.

Command: view-current
Parameters: <none>
Description: View the current observation being performed by each of the antenna resources.

Command: view-last
Parameters: <none>
Description: View the last observation which used each of the antenna resources.

Command: view-next
Parameters: <none>
Description: View the next observation to use each of the antenna resources.

Command: info-schedule
Parameters: <none>
Description: Provide an overview of the contents and the status of the schedule and scheduler.

Command: reconfigure-mode
Parameters: <WillWait|NoWait|QosWait> <timeout>
Description: Specifies the configuration mode that the system operates within. It also allows the operator to specify a timeout value when waiting for components to become available (timeout option is only available with the WillWait|QoSWait options).

Command: show-reconfig-info
Parameters: <no parameters>
Description: Shows the systems current reconfiguration setting.

Command: reconfigure-antenna
Parameters: <antenna #> <new antenna reference> <antenna ID>
Description: Performs a reconfiguration action on the specified antenna and moves the state data located within that antenna to the new antenna specified by the antenna reference.

Command: set-antenna-availability
Parameters: <antenna #> <HH:MM:SS> <DD/MM>
Description: Allows the operator to specify at what time the antenna referenced by the antenna number will be available.

Command: antenna-offline
Parameters: <antenna #>
Description: Takes the nominated antenna offline.

Command: antenna-online
Parameters: <antenna #>
Description: Brings the nominated antenna online.

Command: clock-stop
Parameters: <none>
Description: Stops the observatory clock.

Command: clock-start
Parameters: <none>
Description: Starts the observatory clock.

Command: set-clock
Parameters: <HH:MM:SS> <DD/MM/YYYY>
Description: Sets the time on the observatory clock to the value entered.

Command: specify-QoS
Parameters: <RadioInterference> <# re-try attempts> <# delay in seconds>
Description: Specify a Quality of Service (QoS) level and attach it to a particular element or component within the system.

Command: introduce-radio-interference
Parameters: <antenna #>
Description: Introduce radio interference to the nominated antenna.

Command: remove-radio-interference
Parameters: <antenna #>
Description: Remove radio interference to the nominated antenna.

Command: restart-antenna
Parameters: <antenna#>
Description: Brings the nominated antenna online and then has it rejoin the current scheduled observation.

Command: create-antenna
Parameters: <none>
Description: Initialise an instance of an antenna in the system.

Appendix D

Source Code

D.1 Processing Method Calls for Unavailable Components

This appendix provides the function `processUnAvailability(...)` which is executed everytime a method request is made upon a component which is not available. The source code explicitly reproduced here and used throughout the thesis is subject to the following copyright notice:

D.1.1 processAvailability.cc

The file `processAvailability.cc` is responsible for checking the `ModuleAvailability` object and throwing the corresponding exception if a component is not available.

```
void processUnAvailability
            (const ModuleAvailability::ModuleUnAvailableInstruction
              &availabilityInstructions, time_t componentAvailabilityTime,
             char *objectReference)
{
 [...]
 if (availabilityInstructions.anInstruction == ModuleAvailability::NoWait)
   throw ObjectBeingReconfigured("Object is currently being reconfigured.");
 if ((availabilityInstructions.anInstruction == ModuleAvailability::WillWait)&&
     (availabilityInstructions.someQoSData.timePreparedToWait == 0))
   throw QoSObjectInvalid
    ("The time value specified in the QoS structure has not been specified.");
 if ((availabilityInstructions.anInstruction == ModuleAvailability::WillWait)&&
     (componentAvailabilityTime == 0))
   throw QoSObjectInvalid
    ("The time at which the component is to be ready has not been specified.");
 if (availabilityInstructions.anInstruction == ModuleAvailability::WillWait)
 {
  if (time(NULL) > componentAvailabilityTime)
   return;
  if ((time(NULL) + (availabilityInstructions.someQoSData.
                     timePreparedToWait*60)) > componentAvailabilityTime)
  {
   int sleepValue = componentAvailabilityTime - time(NULL);
   if (sleepValue < 0)
    return;
   sleep(componentAvailabilityTime - time(NULL));
   return;
  }
  else
  {
   ModuleAvailabilityData ModuleAvailabilityDataToReturn;
   ModuleAvailabilityDataToReturn.timeComponentIsAvailableAt =
    componentAvailabilityTime;
   ModuleAvailabilityDataToReturn.timePreparedToWait =
    availabilityInstructions.someQoSData.timePreparedToWait;
   ModuleAvailabilityDataToReturn.allOfTheQoSParameters[0].extraQoSData =
    availabilityInstructions.someQoSData.allOfTheQoSParameters[0].extraQoSData;
   throw QoSObjectBeingReconfigured(ModuleAvailabilityDataToReturn);
  }
 }
}
```

Index

Bibliography

Arnold, K. and J. Gosling (1998). *The Java Programming Language: Second Edition*. The Java Series. Addison Wesley.

Ben-Natan, R. (1995). *CORBA: A Guide To Common Object Request Broker Architecture*. McGraw-Hill.

Bennett, S. (1990). *Real-Time Computer Control: An Introduction*. McGraw-Hill.

Berliner, B. (1990). CVS II: Parallelizing Software Development. In *proceedings of 1990 Winter USENIX Conference, Washington D.C.*

Blazewicz, J., K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz (1996). *Scheduling Computer and Manufacturing Process*. Springer-Verlag.

Bloomer, J. (1992). *Power Programming with RPC*. O'Reilly & Associates.

Brockschmidt, K. (1994). *Inside OLE 2: the fast track to building powerful object-oriented applications*. Microsoft Press.

Brown, N. and C. Kindel (1996). *Distributed Component Object Model Protocol - DCOM/1.0*. One Microsoft Way, Redmond, WA: Microsoft Corporation. http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

Brucker, P. (1998). *Scheduling Algorithms*. Springer-Verlag.

Burch, J. G. and G. Grudnitski (1989). *Information Systems: Theory and Practice* (5th ed.). John Wiley and Sons, Ltd.

Cederqvist, P. (1993, November). *Version Management with CVS*.

Chung, P. E., Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C.-Y. Wang, and Y.-M. Wang (1998, January). DCOM and CORBA: Side by Side, Step by Step and Layer by Layer. C++ Report *10*(1), 18–29.

Dahl, O. and K. Nygaard (1966, September). Simula - An ALGOL-based Simulation Language. *Communications of the ACM 9*(9), 671–678.

Damianou, N., N. Dulay, E. Lupu, and M. Sloman (2000, January). Ponder: A Language for Specifying Security and Management Policies for Distributed Systems: The Language Specification. Technical Report DoC 2000/1, Department of Computing, Imperial College of Science, Technology and Medicine. Version 1.11.

Deitel, H. M. and P. Deitel (1997). C++ How To Program (2nd ed.). Prentice Hall.

Diestel, R. (1997). *Graph Theory*. Springer-Verlag.

Duffett-Smith, P. (1988). *Practical Astronomy with your Calculator* (3rd ed.). Cambridge University Press.

Dulay, N. (1992, March). The DARWIN Configuration Language. Technical report, Imperial College Department of Computing Internal Report, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom.

ECMA Standards Organisation (2001a, December). *C# Language Specification - ECMA Standard 334*. 114 Rue du Rhône - CH-1204 Geneva - Switzerland: ECMA Standards Organisation.

ECMA Standards Organisation (2001b, December). *Common Language Infrastructure - ECMA Standard 335*. 114 Rue du Rhône - CH-1204 Geneva - Switzerland: ECMA Standards Organisation.

Ellis, M. A. and B. Stroustrup (1994). *The Annotated* C++ Reference Manual (2nd ed.). Addison-Wesley.

Ernst, W. (1996). *Presenting ActiveX* (1st ed.). Sams.net.

Estublier, J. and R. Casallas (1994). The Adele configuration manager. In W. Tichy (Ed.), *Configuration Management*, pp. 99–133. Baffins Lane, Chichester, West Sussex PO19 1UD, England: John Wiley and Sons, Ltd.

Exton, C., D. Watkins, and D. Thompson (1997). Comparisions between CORBA IDL and COM/DCOM MIDL: Interfaces for Distributed Computing. In C. Mingins, R. Duke, and B. Meyer (Eds.), *In proceedings of TOOLS 25, Technology of Object Oriented Language Systems*, pp. 15–32. IEEE Computer Society Press.

Fitzpatrick, P. M. (1970). *Principles of Celestial Mechanics*. Academic Press, New York.

Gadonna, C. (1999, August). *MISTRAL User Manual V1* (1 ed.). LGI.

Goldman, K. J., M. D. Anderson, and B. Swaminathan (1993, June). The Programmers' Playground: I/O Abstraction for Heterogeneous Distributed Systems. Technical Report WUCS-93-29, Department of Computer Science, Washington University.

Goldman, K. J., J. Hoffert, T. P. McCartney, J. Plun, and T. Rodgers (1997, February). Building Interactive Distributed Applications in C++ with The Programmers Playground. Technical Report WUCS-94-14, Department of Computer Science, Washington University.

Gordon, A. D. and D. Syme (2001, January). Typing a Multi-Language Intermediate Code. In G. C. Necula and S. P. Rahul (Eds.), *Proceedings of 28th ACM Symposium on Principles of Programming Languages*, Volume 36, London, pp. 248–260. ACM Press.

Goudarzi, K. (1999). *Consistency Preserving Dynamic Reconfiguration of Distributed Systems.* Ph. D. thesis, Dept of Computing, Imperial College of Science Technology and Medicine, 180 Queens Gate, London SW7 2BZ, U.K.

Gough, J. (2002). *Compiling for the .NET Common Language Runtime (CLR).* Prentice Hall.

Gross, J. L. (1999). *Graph Theory and its Applications.* CRC Press.

Hall, R. S., D. Heimbigner, A. van der Hoek, and A. L. Wolf (1997, May). An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *proceedings of the 1997 International Conference on Distributed Computing Systems.* Software Engineering Research Laboratory, Department of Computer Science, University of Colorado.

Hirshfield, A. and R. W. Sinnott (1985). *Sky Catalogue 2000.0.* Cambridge University Press.

Hofmeister, C., E. White, and J. Purtilo (1993, March). Surgeon: A Packager for Dynamically Reconfigurable Distributed Applications. *Software Engineering Journal 8*(2), 95–101.

Kindberg, T. (1991). Equus: an Environment for Reconfigurable Distributed Computations. Technical report, Department of Computer Science, Queen Mary and Westfield College, University of London.

Kramer, J. (1990, May). Configuration Programming - A Framework for the Development of Distributable Systems. In *proceedings of the IEEE International Conference on Computer Systems and Software Engineering (COMPEURO 90).* Department of Computing, Imperial College of Science, Technology and Medicine: IEEE Computer Society Press.

Kramer, J. and J. Magee (1990, November). The Evolving Philosophers Problem: Dynamic Change Management. In *IEEE Transactions on Software Engineering*, Volume 16 of *IEEE Transactions on Software Engineering.* IEEE Press.

Kramer, J. and J. Magee (1997). Exposing the Skeleton in the Coordination Closet. In *Coordination '97.* Department of Computing, Imperial College of Science, Technology and Medicine.

Kramer, J., J. Magee, and K. Ng (1989, October). Graphical Configuration Programming. In *IEEE Computer*, IEEE Compuer. IEEE Press.

Kramer, J., J. Magee, M. Sloman, N. Dulay, S. Cheung, S. Crane, and K. Twidle (1991). An Introduction to Distributed Programming in REX. Technical Report ESPRIT Conference 91 - Project Number: 2080, Department of Computing, Imperial College of Science, Technology and Medicine.

Kramer, K., J. Magee, and A. Young (1990). Towards Unifying Fault and Change Management. In *proceedings of the IEEE International Workshop on Distributed Computing Systems in*

the '90s, pp. 57–63. Department of Computing, Imperial College of Science, Technology and Medicine: IEEE Computer Society Press.

Krause, K. L. (1973, December). *Analysis of Computer Scheduling with Memory Constraints*. Ph. D. thesis, Department of Computer Science, Purdue University, Indiana, USA.

Kutner, M. L. (1987). *Astronomy: a physical perspective*. Harper & Row.

Levi, S.-T. and A. K. Agrawala (1990). *Real Time System Design*. McGraw-Hill.

Lim, A. S. S. (1993). *A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems*. Ph. D. thesis, University of Wisconsin-Madison.

Magee, J., N. Dulay, S. Eisenbach, and J. Kramer (1995, September). Specifying Distributed Software Architectures. In *proceedings of the Fifth European Software Engineering Conference*. Department of Computing, Imperial College of Science, Technology and Medicine.

Magee, J., N. Dulay, and J. Kramer (1994, September). A Constructive Development Environment for Parallel and Distributed Programs. *Distributed Systems Engineering Journal 1*(5), 304–312.

Magee, J. and J. Kramer (1999). *Concurrency: State Models and Java Programs*. John Wiley and Sons, Ltd.

Magee, J., J. Kramer, and M. Sloman (1989, June). Constructing Distributed Systems in CONIC. *IEEE Transactions on Software Engineering 15*(6).

Magee, J., J. Kramer, and M. Sloman (1990). An Overview of the REX Software Architecture. In *proceedings of the 2nd IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Volume 10. IEEE Computer Society Press.

Martin, V. C. (1995, February). There can be only one! A summary of the UNIX standardization movement. *ACM CrossRoads 1*(3).

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice Hall.

Meyers, S. D. (1998). *Effective* C++: 50 specific ways to improve your programs and designs (2nd ed.). Addison-Wesley.

Microsoft Corporation (2001). *Microsoft .NET Framework - Technical Overview* (3.0 ed.). One Microsoft Way, Redmond, WA: Microsoft Corporation. http://www.gotdotnet.com/team/framework/DotNet Framework Technical Overview v3.doc.

Microsoft Corporation and Digital Equipment Corporation (1995). *The Component Object Model Specification* (Version 0.9 ed.). One Microsoft Way, Redmond, WA: Microsoft Corporation and Digital Equipment Corporation.

Naur, P., B. Randall, and J. Buxton (1976). *Software Engineering: concepts and techniques: proceedings of the NATO conference*. Petrocelli/Charter.

Object Management Group (1995a, November). *Common Facilties Architecture* (4.0 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (1995b). *The Common Object Request Broker: Architecture and Specification* (2.0 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (1997, January). *A Discussion of the Object Management Architecture* (1.0 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (1998, December). *CORBAServices: Common Object Services Specification*. Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (1999, July). *CORBA Language Mappings*. Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (2001, December). *The Common Object Request Broker: Architecture and Specification* (2.6 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (2002a, November). *The Common Object Request Broker: Architecture and Specification* (3.0.1 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Object Management Group (2002b, June). *CORBA Components* (3.0 ed.). Framingham Corporate Center, 492 Old Connecticut Path, Framingham, MA 01701 U.S.A: Object Management Group.

Petri, C. A. (1962). *Kommunikation mit Automaten*. Ph. D. thesis, Bonn: Inst. f. instrumentelle Math, University of Bonn, Germany.

Plasil, F., D. Balek, and R. Janecek (1998, May). DCUP: Dynamic Component Updating in Java/CORBA Environment. In *proceedings of the 4th International Conference on Configurable Distributed Systems*. IEEE Computer Society Press.

Pryce, N. and S. Crane (1996, January). A Uniform Approach to Configuration and Communication in Distributed Systems. In *proceedings of the Third International Conference on Configurable Distributed Systems*, pp. 1–12. Department of Computing, Imperial College of Science, Technology and Medicine.

Purtilo, J. M. (1994, January). The POLYLITH software bus. In *ACM Transactions on Programming Languages and Systems*, Volume 16, pp. 151–174.

Purtilo, J. M. and C. R. Hofmeister (1991, May). Dynamic Reconfiguration of Distributed Programs. In *proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, Washington, DC, pp. 560–573. IEEE Computer Society.

Reisig, W. (1985). *Petri Nets, An Introduction*. Springer-Verlag.

Román, M., F. Kon, and R. H. Campbell (1999, June). Design and Implementation of Runtime Reflection in Communication Middleware: The *DynamicTAO* Case. In *proceedings of ICDCS'99 Workshop on Middleware*, Austin, TX.

Roth, G. (1975). *Astronomy - A Handbook*. Springer-Verlag.

Roy, A. E. (1988). *Astronomy: principles and practice*. Bristol ; Philadelphia, PA, USA : Institute of Physics Pub.

Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser (1992). Overview of the CHORUS Distributed Operating System. In *workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, pp. 39–70. Berkeley, California, USENIX Association.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.

Schmerl, B. R. and C. D. Marlin (1996, June). Consistency Issues in Partially Bound Dynamically Composed Systems. In *proceedings of the Australian Software Engineering Conference*, pp. 183–191. Department of Computer Science, The Flinders University of South Australia: IEEE Press.

Schmidt, D. C., D. L. Levine, and C. Cleelan (1999). *Advances in Computers*. Academic Press.

Sedgewick, R. (1988). *Algorithms*. Addison-Wesley.

Sedgewick, R. (2002). *Algorithms in C. Part 5, Graph Algorithms* (3rd ed.). Addison-Wesley.

Sethuraman, R. and K. J. Goldman (1995, November). Formal Specification of a Dynamically Configurable Distributed system. Technical Report WUCS-95-17, Department of Computer Science, Washington University.

Siegel, J. (1996). *CORBA fundamentals and programming*. John Wiley and Sons, Ltd.

Simmons, C. and A. Rofail (2002). *The Microsoft .NET Platform and Technologies*. Prentice Hall.

Stroustrup, B. (1994). *The Design and Evolution of* C++. Addison-Wesley.

SUN Microsystems (1987, June). *XDR: External Data Representation*. Released as a Request for Comment. Number: 1014.

SUN Microsystems (1997, July). *JavaBeans Specification* (Version 1.01 ed.). 2550 Garcia Avenue, Mountain View, CA 94043: SUN Microsystems.

SUN Microsystems (2002, August). *Enterprise JavaBeans Proposed Specification* (2.1 ed.). 4150 Network Circle, Santa Clara, California 95054, U.S.A: SUN Microsystems.

Swamy, M. N. S. and K. Thulasiraman (1981). *Graphs, Networks, and Algorithms*. John Wiley and Sons, Ltd.

Szyperski, C. (1997). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

Thompson, D. and D. Watkins (1997). Comparisons between CORBA and DCOM: Architectures for Distributed Computing. In J. Chen, M. Li, C. Mingins, and B. Meyer (Eds.), *TOOLS 24*, Volume 24 of *Technology of Object-Oriented Languages and Systems*, pp. 333 — 338.

Thompson, D., D. Watkins, C. Exton, L. Garrett, and A. Sajeev (1998). *Information Systems Interoperability*, Chapter Distributed Component Object Model, pp. 39—78. Research Studies Press.

Tichy, W. F. (1985, July). RCS, A System for Version Control. *Software-Practice and Experience 15*(7), 637–651.

Transarc Corporation (1996). *DCE Application Development Guide - Directory Services* (1.1 ed.). The Gulf Tower, 707 Grant Street, Pittsburgh, PA 15219: Transarc Corporation. *http://ovpit.ucs.indiana.edu/DCE-DFS/app_gd_ds_1.html*.

Tsai, J. J. P., Y. Bi, S. J. Yang, and R. A. Smith (1996). *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis* (1st ed.). Wiley-InterScience.

van der Hoek, A., A. Carzaniga, D. Heimbigner, and A. L. Wolf (1998, September). A Reusable, Distributed Repository for Configuration Management Policy Programming. Technical Report CU-CS-864-98, Software Engineering Research Laboratory, Dept. of Computer Science, University of Colorado.

van der Hoek, A., D. Heimbigner, and A. L. Wolf (1996, March). A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pp. 308–317. IEEE Computer Society Press.

van Helden, A. (1995). Gregorian Calendar. *http://es.rice.edu/ES/humsoc/Galileo/Things/gregorian_calendar.html*. Summary from: Gregorian Reform of the Calendar: Proceedings of the Vatican Conference to Commemorate its 400th Anniversary, 1582-1992, ed. G. V. Coyne, M. A. Hoskin, and O. Pedersen (Vatican City: Pontifical Academy of Sciences, Specolo Vaticano, 1983). Jean Meeus and Denis Savoie, "The history of the tropical year", Journal of the British Astronomical Association, 102 #1 (1992): 40-42.

Watkins, D. (2000, January). *Adding Contracts to Interface Definition Languages*. Ph. D. thesis, Computer Science and Software Engineering, Monash University, Australia.

Watkins, D. and D. Thompson (1998). Adding Semantics to Interface Definition Languages. In D. Grant (Ed.), *proceedings of ASWEC 98, Australian Software Engineering Conference*, pp. 66–78. IEEE Computer Society Press.

Yang, Z. and K. Duddy (1995, June). *Distributed Object Computing with CORBA*. Technical Report DSTC Technical Report 23, Distributed Systems Technology, University of Queensland.

Zeller, A. (1995a, October). A Unified Version Model for Configuration Management. In G. Kaiser (Ed.), *proceedings of ACM SIGSOFT'95: Symposium on the Foundations of Software Engineering (FSE-3)*, Volume 20 of *Software Engineering Notes*, Washington D.C, pp. 151–160. ACM Press.

Zeller, A. (1995b, March). Smooth Operations with Square Operators - The Version Set Model in ICE. In I. Sommerville (Ed.), *proceedings of 6th International Workshop on Software Configuration Management (SCM-6)*, Volume 1167 of *Lecture Notes in Computer Science*, Berlin, pp. 8–30. Springer-Verlag.

Zlotnick, F. (1991). *The POSIX.1 Standard: A Programmer's Guide*. Benjamin/Cummings Publishing.

Zombeck, M. V. (1990). *Handbook of Space Astronomy and Astrophysics*. Cambridge University Press.

Vita

Publications arising from this thesis include:

Exton, C., Watkins, D., and Thompson, D. (1997), *Comparisons Between CORBA IDL and COM/DCOM MIDL: Interfaces for Distributed Computing.*, In C. Mingins, R. Duke and B. Meyer (Editors), TOOLS 25, Volume 25 of *Technology of Object Oriented Languages and Systems.* IEEE Computer Society, pp. 15-32.

Thompson, D., and Watkins, D. (1997), *Comparisons between CORBA and DCOM: Architectures for Distributed Computing.* In J. Chen, M. Li, C. Mingins, and B. Meyer (Editors). TOOLS 24, Volume 24 of *Technology of Object Oriented Languages and Systems.* IEEE Computer Society, pp. 278-283.

Thompson, D., Watkins, D., Exton, C., Garrett, L., and Sajeev, A. (1998), Distributed Component Object Model (DCOM) In B. Kramer, M. Papazoglou and H. Schmidt (Editors), *Information Systems Interoperability.* Research Studies Press, Chapter 3, pp. 39-78.

Watkins, D., and Thompson, D. (1998), Adding Semantics to Interface Definition Languages, In D. Grant (Editor), ASWEC '98, *Australian Software Engineering Conference.* IEEE Computer Society, pp 66-78.

Watkins, D., Dick, M. and Thompson, D. (1998), *From UML to IDL: A Case Study,* In C. Mingins and B. Meyer (Editors), TOOLS 28, Volume 28 of *Technology of Object Oriented Languages and Systems.* IEEE Computer Society, pp. 141-153.

Permanent Address: School of Computer Science and Software Engineering
Monash University
Australia

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Glenn Maughan of Monash University and are maintained by Dean Thompson of Monash University.