University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Consistency Preserving Dynamic Reconfiguration of Distributed Systems

Kaveh Moazami-Goudarzi

March 1999

A thesis submitted in partial fulfilment of the requirements for the degree of *Doctor of Philosophy* in the Faculty of Engineering of the University of London, and for the Diploma of Imperial College of Science, Technology and Medicine.

BIBL
LONDON
UNIV.

Abstract

Application change and evolution often requires the modification of running computer systems. However, increased reliance on such systems has meant that total shutdown is undesirable; ideally change should be carried out on running systems without causing major interruption to the service provided. This process of *dynamic reconfiguration* is possible for distributed systems in which low coupling between system components permits unaffected sections of the system under modification to continue, thereby providing a partial service.

This thesis identifies the preservation of system consistency to be a major requirement of dynamic change and proposes a framework for the provision of a consistency preserving dynamic reconfiguration service. The framework identifies the basic elements necessary for the provision of such a service and provides a model for interaction among these and existing system services. The interaction model ensures that in the absence of failures an up-to-date view of the system configuration can be obtained. It also maintains a clear separation of concerns between the configuration and the application behaviour.

In addition, a novel approach to the specification of change descriptions is proposed which enables the definition of complex hierarchic and reusable reconfiguration scripts. This is achieved by making the reconfiguration behaviour of application components and subsystems an integral part of their configuration descriptions and using configuration queries to locate reconfiguration targets.

One of the ways in which dynamic reconfiguration can compromise consistency is by invalidating application invariants which govern synchronisation or semantic aspects of a distributed system. This thesis proposes a new approach to re-establishing invalidated application invariants by using component state-access methods from within change scripts. This is in contrast to existing schemes which embed particular state adjustments in the application code and are hence inflexible and difficult to reason about.

Reconfigurations can also compromise consistency by removing components and altering communication paths which are involved in ongoing interactions. To resolve this issue a scheme is proposed whereby components affected by a planned change are identified and moved automatically into a state amenable to reconfiguration. The advantages of this scheme are that it is largely transparent to the application programmer, does not impose a heavy overhead on the runtime system and takes advantage of the runtime behaviour of the application to minimise the disruption to the system.

The final contribution of the thesis is an implementation of the reconfiguration environment as described and a demonstration of its utility using a case study.

Acknowledgements

I gratefully acknowledge the guidance and encouragement of my supervisor, Jeff Kramer, whose experience, insight and patience have proved invaluable during my research. I would also like to thank my second supervisor Naranker Dulay, Anne O'Neill, Tracy Banton, Paul Dias and other members of the DSE section at Imperial for their help and support throughout my studies.

I have been very fortunate to be in the company of an exceptional group of people who, each in their own special way, have helped make this period of my life extremely rewarding. I would therefore like to thank : Andrea who was and always will be *very nice*, Fox who is even more cunning and clever than his name suggests, Dimitra for "ouse ouse schnell gdgdgd" and other mad reasons, Fafar for sharing my awful taste (in foodz, not clothez), Nikos for his strong silent support, Damian for converting me to the religion that is Emacs, Narmin for call-waiting me, Berr for always, always laughing, Keng for being ever so agreeable, Douglas for Haggis, Robert Burns and 3D-tetris, Ranjit for Vikram Seth and other less literary diversions, Roberto for saving my desk, Paris (P.B.) for the many Greek pearls of wisdom, Andrei for his colourful descriptions, Clea for Curry, and Pomme because she is nice. Special thanks go to Christos (who is from Kavala) and Celso (better known as "Papa") for frequent coffee breaks, lengthy discussions, squash, volley and many many reasons besides.

This thesis would not have been possible without the love and support of my parents Ali and Manijé who are quite simply the best. This acknowledgements page would not be the way it is without the influence of my wonderfully eccentric uncle Aziz. I am also eternally grateful to the following people who I'm both proud and lucky to be able to call my friends : Mazout who is to blame for making me do a Ph.D., Amir Dampaee for his unique brand of hospitality at the house of Diplomat, and Mamad (Rajab) Afshar for motivating me, and accompanying me on the same hellish path.

I owe what is left of my sanity and good cheer to my long suffering girlfriend, Nasrin, who is probably the only other person just as happy as I am, now that it is finally over !

Financial support for this research has been provided by the following organisations: Department of Trade and Industry (DTI), Department of Computing Imperial College, The Swiss Bank, ARKA Technologies (Europe) Ltd. and Active Media Solutions Ltd.

In memory of my aunt *Badié*

Contents

1 Introduction **19**

1.1 Motivation 19

1.2 Dynamic reconfiguration 20

1.2.1 Specification and management of change 20

1.2.2 Preservation of consistency 21

1.2.3 Minimum disruption 23

1.3 Thesis contributions 24

1.4 Thesis outline 25

2 Related Work **27**

2.1 Background 27

2.2 Mutually consistent states 29

2.2.1 Consistency through recovery 29

2.2.2 Consistency through avoidance 30

2.2.3 Recovery-avoidance hybrid 47

2.2.4 A comparison 50

2.3 Re-Establishing application invariants 51

2.3.1 Embedded actions 53

2.3.2 Possible non-embedded methods 55

2.3.3 Discussion 55

2.4 Specification of change 56

2.4.1 Darwin - Dulay et. al. 59

2.4.2 Etzkorn 61

2.4.3 Zimmermann et. al. 61

2.4.4 Gerel 62

2.4.5 Discussion 62

2.5 Chapter summary 63

3 The Change Management Service **65**
3.1 A general framework . . . 65
3.2 Reconfiguration manager (RM) . . . 67
3.3 Configuration Database (CDB) . . . 68
3.3.1 Concurrency . . . 70
3.4 Consistency manager (CM) . . . 71
3.5 Event composition service (ECS) . . . 72
3.6 Scalability . . . 74
3.7 Chapter summary . . . 75

4 Specification of Change **77**
4.1 Evaluation of Configuration programming . . . 78
4.2 Reconfiguration programming . . . 81
4.2.1 Overview . . . 81
4.2.2 Primitive state-access methods . . . 82
4.2.3 Augmented configuration specifications . . . 83
4.2.4 Hierarchic configuration information & actions . . . 85
4.3 An extended Example . . . 86
4.3.1 The UK National Lottery . . . 87
4.4 Lottery reconfigurations . . . 92
4.4.1 Adding a new Mach component . . . 93
4.4.2 Adding a new Selector . . . 94
4.4.3 Distributed/Partitioned Database configuration . . . 96
4.4.4 Evolutionary change . . . 99
4.5 Lottery reconfigurations (revisited) . . . 102
4.5.1 RandNumGen . . . 102
4.5.2 Controller . . . 103
4.5.3 Selector . . . 104
4.5.4 WinningNumbers . . . 107
4.5.5 Region component (Tickets Database) . . . 111
4.5.6 Evolutionary change . . . 114
4.6 Benefits of Object Orientation . . . 116
4.7 Discussion . . . 117
4.8 Chapter summary . . . 118

5 Guaranteeing mutual consistency **121**
5.1 Considerations . . . 122
5.2 A generalised safe-state for reconfiguration . . . 123
5.3 Achieving the safe state . . . 125
5.3.1 Assumptions . . . 125
5.3.2 System properties . . . 126
5.4 Pessimistic Blocking Algorithm . . . 128
5.5 Optimistic Blocking Algorithm . . . 130
5.5.1 Concurrency . . . 133
5.5.2 Composite components . . . 133
5.5.3 Compliance with assumptions . . . 134
5.6 Automating compliance checks . . . 136
5.6.1 LTS extensions . . . 136
5.6.2 Detecting interleaved transactions . . . 138
5.6.3 Detecting cyclic dependencies . . . 139
5.6.4 Detecting waiting dependencies . . . 142
5.6.5 Summary . . . 146
5.7 Consistency algorithm . . . 148
5.7.1 The consistency object - CO (figure 5.23) . . . 148
5.7.2 Component participation . . . 150
5.7.3 Programmer participation . . . 156
5.7.4 Transient bindings . . . 156
5.7.5 Open systems . . . 157
5.7.6 Overhead and performance . . . 158
5.8 Discussion . . . 158
5.8.1 Timeliness guarantees . . . 160
5.8.2 More fine grain blocking ? . . . 161
5.9 Chapter Summary . . . 161

6 Implementation Issues and Prototype **163**
6.1 Implementation Issues . . . 163
6.1.1 Flattening reconfiguration scripts . . . 163
6.1.2 Concurrency and reconfigurations . . . 168
6.2 Prototype . . . 170
6.2.1 Regis . . . 172
6.2.2 Runtime hooks . . . 172
6.2.3 Regis and Java connectivity . . . 179

6.2.4 RM . . . 179
6.2.5 CM . . . 185
6.2.6 CDB . . . 187
6.3 Chapter summary . . . 197

7 Case study - The Evolving Philosophers **199**
7.1 The evolving philosophers . . . 199
7.2 Implementation . . . 200
7.2.1 Phil primitive component . . . 201
7.2.2 Fork_server primitive component . . . 204
7.2.3 Function registrations . . . 209
7.2.4 Philosopher composite component . . . 210
7.2.5 Ring composite component . . . 213
7.3 Chapter Summary . . . 219

8 Conclusion **221**
8.1 Summary and comparison . . . 221
8.2 Further Work . . . 226

A Screen Grabs: Philosophers **229**

List of Figures

2.1 A system with dependent transactions 34
2.2 Change Script Code Segment 40
2.3 PCL declaration for producer 43
2.4 PCL for buffer 44
2.5 Bounded buffer 44
2.6 Lazy pipeline component 60
2.7 Lazypipe 60

3.1 The Framework 66

4.1 National Lottery System 88
4.2 WinningNumbers component 88
4.3 Sample database/mach configuration 89
4.4 Region - grouping component 90
4.5 ticket-database - detailed configuration 90
4.6 Selector composite component 91
4.7 newMach script 93
4.8 newSelector script 94
4.9 Distributed Database 96
4.10 Database Component 97
4.11 Actual system configuration 98
4.12 Number monitor and new Selector configuration 100
4.13 NumberMonitor & new Selector script 101
4.14 RandNumGen reconfiguration specification 103
4.15 Controller reconfiguration specification 104
4.16 Selector reconfiguration specification 105
4.17 Selector composite component 106
4.18 Selector.getSeed 106
4.19 Selector.setSeed 106

4.20 WinningNumbers reconfiguration specification 108
4.21 newSelector reconfiguration script 109
4.22 getUniqueSeedAndPrd method definition 110
4.23 Region reconfiguration skeleton 111
4.24 newMach method definition 112
4.25 distributeDBs method definition 113
4.26 NumberMonitor & new Selector script 115
4.27 Inheritance hierarchy 117

5.1 Normal state transition diagram 126
5.2 Cyclic dependency (*leading to deadlock*) 127
5.3 Waiting dependency (*starvation of* C_1) 128
5.4 Modified state transition diagram 129
5.5 Non-BSet member requests 132
5.6 BSet expansion phase 133
5.7 BSet shrinking phase 133
5.8 Sample annotated LTS 137
5.9 Processed LTS 138
5.10 LTS for Node1 139
5.11 LTS for Node2 139
5.12 LTS Node1Sync 140
5.13 LTS Node2Sync 140
5.14 Composed LTS 141
5.15 LTSA - detected deadlock 141
5.16 Two clients and a flawed server 142
5.17 LTS C_1S_1Sync 142
5.18 LTS C_2S_1Sync 142
5.19 LTS C_3 143
5.20 Composed System 145
5.21 Corrected server component LTS 146
5.22 Composed LTS with corrected server 147
5.23 Pseudo code for the consistency object (CO) 149
5.24 Algorithm phase transition diagram 151
5.25 Hook pseudo code for the "normal" phase 152
5.26 Hook pseudo code for the "primed" phase 153
5.27 Hook pseudo code for the "blocking" phase 154
5.28 Hook pseudo code for the "blocked" phase 155

6.1 removeEveryNthElmnts change script . . . 165
6.2 Component chains . . . 165
6.3 A sample reconfiguration . . . 167
6.4 State access methods . . . 168
6.5 Flawed use of access methods . . . 169
6.6 The Framework . . . 171
6.7 rcprocess class . . . 173
6.8 cdb_stub class . . . 175
6.9 cm_stub class . . . 178
6.10 rcprocessT class . . . 179
6.11 Controller constructor . . . 180
6.12 Controller.packParams method . . . 181
6.13 Controller.getReseedPrd state-access method . . . 182
6.14 Controller.setReseedPrd state-access method . . . 182
6.15 State access methods . . . 184
6.16 CM interface . . . 186
6.17 Component class hierarchy . . . 188
6.18 Internal and external bindings . . . 190
6.19 Change script skeleton . . . 195
6.20 NewSelector script . . . 195

7.1 p1 hungry . . . 200
7.2 p1 eating . . . 200
7.3 Ring of Dining Philosophers . . . 201
7.4 Phil C++ header file . . . 201
7.5 Phil configuration description . . . 202
7.6 Phil constructor . . . 203
7.7 Phil create function . . . 204
7.8 Fork_server C++ header file . . . 205
7.9 Fork_server configuration specification . . . 205
7.10 Fork_server constructor . . . 206
7.11 Java `packParams` . . . 206
7.12 C++ `create_fn` . . . 206
7.13 Java `set_fork` . . . 207
7.14 C++ `set_fork` . . . 207
7.15 Java `get_fork` . . . 208
7.16 C++ `get_fork` . . . 208

7.17 `Phil` transaction 208
7.18 L'neighbour trans'n 209
7.19 L'neighbour trans'n 209
7.20 function registration 210
7.21 Philosopher composite reconfiguration description 210
7.22 Philosopher constructor 211
7.23 run method 212
7.24 set_fork access-method 212
7.25 get_fork access-method 212
7.26 Ring composite component reconfiguration description 213
7.27 Ring constructor 214
7.28 Shrink reconfiguration method 215
7.29 Grow reconfiguration method 216
7.30 GrowBy reconfiguration method 217
7.31 Ring run method 218
7.32 Generating triggering messages 218

A.1 Screen shot of four dining Philosophers 229
A.2 System messages window 230
A.3 CDB Viewer showing top-level Ring component 231
A.4 The dining Philosophers subsequent to a shrink reconfiguration 232
A.5 CM messages subsequent to a shrink reconfiguration 233
A.6 System messages subsequent to a shrink reconfiguration 234
A.7 CDB view - Ring all child types 235
A.8 CDB view - Philosopher 7 all child types 235
A.9 CDB view - ForkServer 8 neighbours 236

List of Tables

2.1 Comparison of approaches to preserving mutually consistent component states during Reconfiguration 52

6.1 Event activation methods 176
6.2 PNode methods 190
6.3 CNode Methods 191
6.4 PPort methods 192
6.5 CPort methods 192
6.6 PPortref methods 193
6.7 CPortref methods 194

Chapter 1

Introduction

1.1 Motivation

The increased reliance on long running computer systems has meant that it is undesirable to stop and restart applications in order to make changes to them. For software systems to remain useful, they must survive alterations to their operating environments, keep up with rapid advances in technology, and respond to the ever changing needs of the human user. Some even argue that the introduction of software is in itself a stimulus for change [Leh82]. All these factors attest to the inevitability of change in the software life cycle and illustrate a pressing need for applications which can undergo change at runtime without causing major interruptions to the services they provide.

Increased awareness of the power and utility of the Internet and communication networks in general, coupled with the emergence of commercial distributed programming environments and standardisation work such as CORBA [Obj95] has allowed distributed systems to supplant centralised systems in many areas, becoming to a large extent the dominant force in the software world. Briefly, a distributed system is one where a number of programs (*components*) running on geographically separate machines (*hosts*) cooperate with each other through a communications medium to achieve a common goal. For such systems, useful change can be expressed in terms of modifications to the structure or *configuration* of the system. Dynamic reconfiguration is the processes of changing this configuration at runtime.

Components making up a distributed system exhibit a high level of autonomy and are loosely coupled. These properties permit runtime changes to be performed on a fraction of a distributed system while components in unaffected sections continue to provide a partial service. The two types of change a system may undergo are

anticipated changes foreseen by the system designer henceforth referred to as *programmed change* and unanticipated or *evolutionary* changes which become necessary over the execution lifetime of an application. An example of a programmed change could be the redimensioning of a system by increasing or decreasing the number of servers in order to keep the "client to server" ratio constant. Evolutionary changes could encompass reconfigurations such as the addition of monitoring components to collect statistical data regarding the frequency, size and content of messages transmitted over connections or upgrading sections of the system by introducing new or replacing/migrating old components.

At present there are few distributed programming environments which provide structured support for dynamic reconfiguration. The goal of this thesis is to identify a flexible and structured way of writing distributed applications such that they can undergo programmed and evolutionary change, as well as providing the necessary framework and associated services to facilitate their management throughout their lifetime. Special emphasis is placed on preserving system consistency and minimising the disruption to the system.

1.2 Dynamic reconfiguration

There are three basic issues which must be addressed when considering dynamic reconfiguration. These are the specification and management of the change process, the preservation of system consistency, and minimisation of disruption to the provided service. Let us consider each of these in turn.

1.2.1 Specification and management of change

In order to achieve dynamic reconfiguration many programs developed using existing approaches embed all the extra functionality necessary to manage the synchronisations, creations, deletions, bindings and so on inside the application. Embedding this type of application specific code into the components of a distributed application detracts from their generality and prevents them from being reused when constructing other distributed applications. Even when discounting this and the duplication of effort involved in re-implementing very similar management tasks for every new system, the approach remains unsatisfactory as it cannot cater for evolutionary changes.

One model of change management which has become prevalent in recent years is that put forward by the configuration programming paradigm [Kra90]. In this

approach distributed systems are constructed by linking together context independent components in different configurations. Descriptions of initial and subsequent system configuration are made explicit and expressed in a separate *configuration language*. This separation of concerns between configuration and computation is beneficial as it frees the component programmer from dealing with configuration concerns, while permitting the configuration programmer to plan and reason about the changes to the application without having to delve into the low level component code.

To facilitate dynamic change we believe that the dynamic reconfiguration service must preside over the *specification* and *execution* of change programs, and provide the specialised support structures that each phase requires. However despite possessing a fairly well understood set of requirements, the dynamic reconfiguration service is in need of a framework which establishes a clear separation of concerns amongst its various responsibilities, identifies the runtime entities which are necessary for its operation and elucidates their interfaces and interactions. In the absence of such a framework it is not possible to customise the reconfiguration service to suit the requirements of the target application [Hil98]. Furthermore components of the reconfiguration service cannot be designed in isolation without running the risk of duplicating functionality which rightfully belongs elsewhere, or failing entirely to address a relevant issue.

Also in the case of the specification of change, despite their growing number, there is little consensus on the properties that reconfiguration languages must possess. Existing reconfiguration languages are often extensions of static configuration languages and as such fail to provide the change designer with the constructs to model and specify the dynamic nature of the application configuration.

1.2.2 Preservation of consistency

The goal of dynamic reconfiguration is to take an existing system and through configuration changes improve its functionality. Clearly, the change process must satisfy safety and liveness properties so that the modified system is left in a "correct" state after reconfiguration. All the effort in reconfiguring a system would be in vain if the resulting system did not function properly. The correctness of a distributed system follows if :

1. **the system satisfies its structural integrity requirements** - At the component level, these requirements specify the way in which a component must

be connected to the rest of the system so that it can send and receive necessary communication, and at the system level they constrain the way the application is built. For example a component level structural integrity requirement might specify that each component should have communication paths established between itself and its predecessor and successor components, with the system level integrity requirement constraining configurations such that components are arranged to form a single ring structure.

For the purposes of this thesis we assume that the envisaged reconfigurations will preserve this property and result in a structurally sound system. This property can be guaranteed by the change designer and can even be checked by automated tools. Structural integrity is addressed in [You91, Wer98].

2. **the components in the system are in mutually consistent states** - Components in a distributed system need to be in mutually consistent states if they are to interact successfully with each other. To this end every interaction, on completion, must result in a transition between well defined and consistent states for all the components involved. Consider the case of a shared resource such as a directory-service being replaced in the middle of a query. The client may be left waiting for a reply which will never arrive as the replacement directory-service is unaware of the request made to its predecessor.

3. **the application state invariants hold** - Application state invariants consist of a list of predicates, each expressed over the state of a subset of the components in the system. These predicates must hold if the correct operation of the system is to be guaranteed. For example one such predicate might specify that in a token ring system there must exist exactly one component in the ring which possesses the token.

Dynamic reconfiguration is primarily concerned with altering the configuration of a system and therefore it might seem, as some researchers maintain [You91], that only the first of the above *consistency preserving criteria* is of relevance to the change programmer. However we believe this to be an over-simplification of the issues. Performing configuration changes on a running system is an intrusive process which may interfere with ongoing interaction between components; servers may be removed in midst of serving client requests or messages lost as a result of broken or altered communication paths. Also consider that application invariants are predicates expressed over the state of a subset of the components in the system system.

Configuration changes can alter system state by adding or removing components and may therefore invalidate the invariants e.g. removing a component whilst in possession of the token.

Clearly these eventualities must be addressed by the change management system otherwise there is no guarantee that application invariants will hold, or that the components in the resulting system will be in mutually consistent states after reconfiguration. As we shall see in chapter two, existing approaches to these issues are not ideal and could benefit from further research.

1.2.3 Minimum disruption

The primary attraction of performing changes to a system at runtime is that down times are dramatically reduced because unaffected sections of the system continue to provide a partial service. It follows that minimising the disruption to the system should be a principal concern during dynamic reconfiguration.

An obvious means of reducing system disruption during dynamic reconfiguration is to ensure that the execution of change takes the least amount of time possible. This requires efficient implementations of the change actions along with support for concurrency both within single and across multiple reconfigurations. System disruption however can not be avoided entirely during reconfiguration as there will always be a set of components whose execution will be directly affected by the change process (e.g. components targeted for deletion). The membership of this set is well defined and can be derived by applying a simple set of rules to the change specification.[1]

Now recall that aside from resulting in structurally sound systems, reconfigurations must also ensure that application invariants hold, and that components are left in mutually consistent states. This means that potentially the *region of disruption* will encompass not only the components which are directly involved in the configuration changes but also those which need their state either adjusted or protected from the indirect side-effects of reconfiguration. Few of the existing mechanisms used to deal with the indirect side-effects of dynamic reconfiguration are geared towards minimising disruption. In addition, they often impose substantial overheads on the system, even in the absence of reconfiguration, and require a considerable amount of participation from the application level programmer and the change designer. Support for multiple concurrent reconfigurations is also rare as it complicates the

[1]Different schemes have different rules for deciding on this membership.

consistency preservation mechanisms mentioned above.

1.3 Thesis contributions

The contributions of the thesis are in the areas of consistency, minimum disruption and the specification and management of change.

The thesis proposes a framework for the provision of a consistency preserving dynamic reconfiguration service. The framework identifies the basic elements necessary for the provision of such a service to be - the reconfiguration manager (RM), the configuration database (CDB) and the consistency manager (CM). Together these elements encapsulate much of the functionality necessary for writing and managing reconfigurable applications and thereby minimise the duplication of effort and programmer overhead. A model for interaction among these and existing system services is then set out such that in the absence of failures a consistent view of the system configuration can be obtained. The model maintains a clear separation of concerns between the configuration and the application behaviour.

A novel approach to the specification of change descriptions is proposed which enables the definition of complex hierarchic and reusable reconfiguration scripts. This is achieved by extending the *configuration programming paradigm* such that configuration descriptions encapsulate both the configuration and reconfiguration behaviour of application components and subsystems. Reuse is facilitated by forcing change scripts to select reconfiguration targets by their configuration properties alone. This results in the definition of generic change scripts which are independent of specific configurations and can be reused throughout an application's life-cycle. The paradigm also incorporates a means of re-establishing invalidated application invariants which permits the change programmer to query and adjust component states from within change scripts. This reinforces the separation of concerns between the computation and configuration of a system, as component programmers no longer need to embed extra code in their components to cater for specific programmed reconfigurations. Instead, these concerns are elevated to the reconfiguration language level where the change programmer can manipulate the application's state using general mechanisms provided by the application programmer and thus re-establish invalidated application invariants as part of both programmed and evolutionary changes.

A new approach to preserving mutual-consistency is presented which given a description of the intended changes, automatically identifies and forces compon-

ents which will be affected by the change into a "safe" state. The nature of this "safe" state is such that reconfigurations can go ahead without leaving components in mutually inconsistent states. The approach caters for general reconfigurations, is largely transparent to the application programmer and takes advantage of the runtime behaviour of the application to minimise the disruption to the remainder of the system. It is possible to apply the approach to different regions of a distributed system concurrently, thus permitting the parallel execution of multiple reconfigurations. It was shown that Labeled transition systems in conjunction with automated analysis tools can be used to verify that a target system adheres to the requirements of the proposed consistency scheme. A preliminary version of this work appeared in [MGK96].

The final contribution of the thesis takes form in a prototype implementation of the reconfiguration environment and a demonstration of its utility using a case study. The implementation is based on top of the *Regis* [MDK94b] distributed programming environment and uses *Java* [GM95] as the reconfiguration language.

1.4 Thesis outline

The organisation of the remainder of the thesis is as follows. *Chapter two* presents a survey of the literature in the area and establishes shortcomings of existing research efforts. *Chapter three* presents the interaction framework and elaborates on the roles and requirements of the RM, CDB, CM and the runtime hooks necessary for realising dynamic reconfiguration. *Chapter four* describes the reconfiguration programming paradigm which permits the definition of hierarchic and reusable reconfiguration scripts and sets out the scheme for re-establishing invalidated application invariants. The application of these concepts are illustrated though the use of an extended example, the "UK National Lottery". *Chapter five* presents the consistency preserving algorithm and also gives implementation details in the form of pseudo code. *Chapter six* addresses outstanding implementation issues and goes on to describes the prototype reconfiguration service. As a case-study, *chapter seven* presents an implementation of the "Evolving Dining philosophers" over the implemented reconfiguration environment. *Chapter eight* presents a summary of the goals and contributions of the thesis and concludes by suggesting possible directions for future research.

Chapter 2

Related Work

Chapter one motivated the need for dynamic reconfiguration and highlighted the key issues involved, namely the specification and management of change, consistency and minimum disruption. In this chapter we conduct a survey of existing literature and in evaluating them expose certain areas which could benefit from further research.

Sections §2.2 and §2.3 deal with work related to maintaining mutually consistent states during reconfiguration and re-establishing invalidated application invariants respectively. The issues pertaining to minimum disruption are dealt with throughout these sections as and when they arise. Section §2.4 deals with the issues involved in the specification and management of change. The chapter concludes with a summary of outstanding issues which will be addressed in the remainder of the thesis.

2.1 Background

When addressing dynamic reconfiguration there are of course various granualities of change that one can consider. Systems such as PODUS [FS91] and DYMOS [SF88, CL83] deal with fine grain dynamic change within individual components, permitting replacement of non-active procedures for client server systems. However for distributed systems such fine grain changes are rarely useful. It is often simpler to replace the component entirely and not incur the overhead and associated complexity of replacing individual procedures within it. In our work we consider this type of reconfiguration which primarily involves the creation and deletion of components and alterations to the bindings between them in a running distributed system.

Many of the approaches surveyed here are based on the configuration programming paradigm [Kra90] which asserts that the structure (cf. configuration) of a distributed system should be specified separately to that of its computation. Initial

attempts at reconfiguration were frustrated by the fact that computation, inter-component bindings and the configuration of the system were all specified together as part of the component code of the distributed system. The separation of concerns between configuration and computation enforced by the configuration programming paradigm is highly advantageous as it makes the configuration of the system explicit, frees the component programmer from dealing with configuration concerns, while permitting the configuration programmer to plan and reason about the changes to the application without having to delve into the low level component code.

In this model, a distributed system consists of a finite number of computational components, where each component is an instance of an abstract component type. Creating an instance of a component type involves choosing a concrete implementation of the abstract component type and executing it on a host machine. The abstract component type specifies the component's communication interface. This interface consists of a set of named communication end-points through which typed messages are sent and received. Communication is made possible when a binding is established between two communication end-points creating a link. The properties exhibited by a communication link depend on the type of end-point used (e.g. request/reply (RPC invocations) or complex group communication protocols such as causal and total orderings [BSS91])

Once the necessary component types and communication end-points have been decided upon and suitable implementations coded, constructing a distributed system is reduced to instantiating components on target hosts and establishing links between them. The collection of component connectivity, instance types and host placement information is termed the configuration of the system. In effect the process of constructing a distributed system establishes the system's initial configuration. This initial configuration and the abstract component types are specified in a separate configuration language.

As a means of reducing the design complexity of large distributed systems, the configuration programming paradigm encourages application designers to group logically related components together (e.g. components which cooperate to provide a service). These *composite components* can then be instantiated and bound together to create other more complex composite components and so on. In this way application designers can think in terms of complex objects which perform abstract operations and use them as the building blocks for constructing their applications.

2.2 Mutually consistent states

Existing approaches to this problem fall into two major categories. Those which permit inconsistencies to occur but rely on a *recovery* mechanism to restore it after the reconfiguration has completed and those which attempt to *avoid* inconsistencies from occurring altogether.

2.2.1 Consistency through recovery

The recovery route is the logical approach for systems which already make use of atomic actions. In these systems components are programmed such that they can deal with *aborts* (i.e the failure of an action to commit/complete normally) and thus the mechanism used to restore consistent state after aborts can be used to recover from inconsistencies introduced during dynamic reconfiguration. All that is necessary is for an exception mechanism to be put in place such that components affected by the reconfiguration are forced to abort any uncommitted transactions thereby reverting to a mutually consistent state. The exception mechanism is necessary as under certain circumstances reconfigurations may result in situations which the recovery subsystem was not designed to deal with e.g. when a component is waiting for the result of a query on a communication end-point which has been unlinked from its original communication partner during reconfiguration.

A prime example of such systems is the *Argus* system [Lis88, LS83] , developed at MIT, which uses a strict two phase locking protocol coupled with a versioned roll-back recovery scheme to implement nested atomic actions. In [BD93, Blo83] Bloom[1] exploits the basic recovery scheme such that dynamic module replacements is made possible. The scheme allows *guardians* which are the unit of distribution and logically encapsulate and coordinate access to resources through the invocations of *handler* procedures, to be replaced at runtime. Runtime replacement of guardians allows system maintainers to replace old implementations with newer improved version and add extra handler routines to a guardian's interface for use by other newly created components. The scheme relies on the change programmer to manually direct affected guardians towards a quiescent state by aborting outstanding transactions and connecting handlers from the old guardians to those in the substitutes.

Argus, *Arjuna* [PSWL95], and *Guide* [BBD+91] are all examples of distributed

[1] [Day87] describes an attempt at replacing guardians and handlers without any system level support. A brief discussion of both approaches to reconfiguration in *Argus* is given in [BD92]

runtime systems which incorporate persistent objects and atomic actions and can therefore benefit from this model of maintaining mutually consistent states.

Evaluation: Although the recovery route is an effective way of ensuring "mutually consistent states", it brings with it certain drawbacks. Atomic actions are not suited to all application domains and there are classes of applications whose weak consistency requirements do not warrant incurring the programmer and runtime overhead imposed by these schemes. Also, due to the course grained nature of locking and concurrency control mechanisms employed by the majority of existing distributed runtime systems, using atomic actions can often result in reduced levels of concurrency in the target application. Although this can be alleviated to some extent by making use of nested atomic actions, type specific recovery and concurrency control mechanisms, the added concurrency is bought at the expense of the programmer who must already provide extra code to save and restore data-structures to stable storage. In addition locking of objects, which is a necessary side-effect of the serialisability requirement, can result in dead-lock and starvation situations which in certain circumstances need to be resolved externally through other runtime services.

These drawbacks however are the price one needs to pay for consistency in a distributed environment. Putting aside these "necessary" drawbacks, other criticisms which can be levelled at the use of recovery mechanism for ensuring mutually consistent states during dynamic reconfiguration are that systems which do not use atomic actions are excluded, and that no attempt is made at reducing the disruption to the system. Aborting transactions prior to reconfiguration can prove to be an expensive process, particularly if they results in cascading aborts. A little effort spent in trying to judge when to abort - e.g. by delaying a particular reconfiguration for a few seconds rather than aborting a transaction, which could take 10's of seconds - may result in much smaller disruptions to the system and quicker execution of the reconfiguration.

2.2.2 Consistency through avoidance

In contrast to the recovery approach which assumed that reconfiguration would introduce inconsistency and provided the means for restoring it, the work described here focuses on preventing inconsistencies from occurring in the first place. The proposition is that if the section of the system likely to be affected by the reconfiguration is frozen in a consistent state before reconfiguration, then inconsistencies will not be introduced.

Avoidance methods roughly follow the three stages set out below :

1. identify the region of disruption,
2. begin interactions which will bring about the consistent and frozen state across that region,
3. detect that the frozen state has come about

Once the frozen state has been detected, reconfigurations are carried out and components returned to their normal state of operation. The approach adopted for stage one determines the amount of disruption suffered by the system during reconfiguration and the mechanisms chosen for steps two and three govern the level of required programmer participation and overhead placed on normal system operation. An ideal avoidance mechanism would minimise system overhead, disruption and programmer participation whilst remaining applicable to a large class of distributed applications.

Note that a number of the approaches surveyed here also deal with the preservation of application invariants and will be revisited in section §2.3.

2.2.2.1 Conic - Kramer et. al.

The work described in [KM90] was the first to take the avoidance route to ensuring that reconfigurations did not result in mutually inconsistent component states and forms the basis for much of the work that has followed. The approach uses the description of the planned changes and the current system configuration to identify the set of components whose activities must be restricted if reconfiguration is to go ahead without leaving them in mutually inconsistent states. Once these components have been identified, a configuration manager component instructs these components to restrict their behaviour by becoming *passive* so that the safe state for reconfiguration *quiescence* is brought about over the affected region. The passive abstract component state is a necessary condition for quiescence and is defined in terms of component activities pertaining to initiation and servicing of *transactions*.

Component interactions are characterised as bounded transactions which are assumed to be the only means through which connected components can affect each others' states. It is further required that the initiator of a transaction is always informed of its completion. A transaction t is said to be *dependent* on the consequent transactions t_1, t_2,..t_n (written $t/t_1 t_2$) if it can complete only after they complete. Otherwise the transaction is termed *independent*. Thus, an independent transaction

involves only two components - the initiator and server; while in the case of a dependent transaction multiple components may become involved.

The authors show that the safe state for reconfiguration is reachable in finite time, this is first done for systems which utilise only independent transactions and then for dependent systems. The definition and reachability arguments for the passive and quiescent states follow.

Systems with independent transactions: For systems with independent transactions a component is said to be in the passive state if it :

(a) continues to accept and service transactions, but

(b) does not initiate new transactions, and

(c) any transactions it has already initiated have terminated.

Given that transactions terminate in finite time the passive state is reachable once a component refrains from starting new transactions and waits until all the transactions it has started terminate. Note that the passive state can be reached by a component irrespective of the configuration state (passive or otherwise) of the components to which it is connected.

Passive components however must continue to accept and service transactions and as such reconfigurations which attempt to remove or unlink them may occur during a period when the component is servicing such transactions and result in leaving the components in mutually inconsistent states. It is here that the notion of quiescence becomes relevant. A component is said to be *quiescent* if it is passive but also that :

(a) it is not currently engaged in servicing *any* transactions (self initiated or otherwise), and

(b) that no transactions have been or will be initiated by other components which require service from this component.

The *reconfiguration rules* and justification for where and why quiescence is necessary for each of the different reconfiguration actions are given below :

1. *Component deletion* - **remove**

Rule: The component targeted for removal must be *quiescent* and *isolated*, where isolated means that no connections are directed to it from other components or from it to other components.

Justification: An isolated component cannot affect the system and so can be independently removed.

2. *Connection* - **link** and **unlink**

   Rule: The precondition for either linking or unlinking is that the component N from which the connection is directed must be in the quiescent state.

   Justification: Quiescence of the initiator component ensures that its state is consistent and frozen with respect to that connection and no outstanding transactions are ongoing.

3. *Component Creation* - **create**

   Rule: The precondition is that the component should be quiescent.

   Justification: This is trivially true as a newly created component is initially isolated and consequently can neither respond to nor initiate transactions.

Earlier we saw that reachability of the passive state could be guaranteed once a component altered its behaviour, however quiescence is not a state that a component can bring about unilaterally. The extra stipulations on top of the passive properties require the cooperation of other components in the system. More specifically a component Q will become quiescent only if all components in its passive set PS, denoted PS(Q) are in the passive state. For independent systems the membership of PS(Q) is as follows :

1. the component Q
2. all components which can directly initiate transaction on Q i.e all components with connection arcs directed towards Q.

Given that Q is in PS(Q) the passive properties of quiescence are immediately satisfied once Q moves into the passive state. The other properties of quiescence follow from the passive state of the other components in PS(Q) i.e once all components which can initiate transactions on Q are passive then all transactions involving Q are complete and no new ones will be initiated[2]. It follows then that quiescence is reachable in finite time.

[2]Quiescent requirement (a) & (b) .

Systems with dependent transactions: For systems with dependent transactions the situation is more complicated and the definition of *passive* and *PS(Q)* need to be amended to allow for the initiation and service of consequent transactions. Consider the system depicted in figure 2.1 and suppose that $N3$ is in the passive state and $N1$ has initiated transaction a. In this situation a cannot complete because b depends on a consequent transaction c which $N3$ cannot initiate. This means that neither $N1$ nor $N2$ will be able to move into the passive state in bounded time if requested.

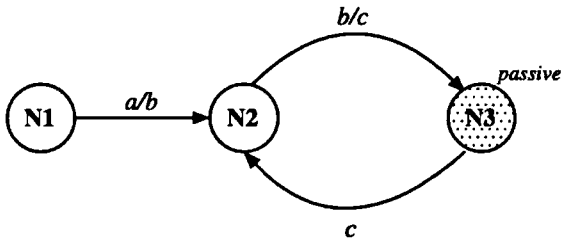


Figure 2.1: A system with dependent transactions

To ensure the reachability of the passive and consequently that of the quiescent state it is proposed to modify the requirements of the passive state as follows. For dependent system a component is said to be in the passive state if it :

(a) continues to accept and service transactions *and initiate consequent transactions*, but

(b) does not initiate new (nonconsequent) transactions, and

(c) any (nonconsequent) transactions it has already initiated have terminated.

The modification to the first requirement of the passive state *(a)*, ensures that passive components are able to initiate consequent transactions and as such enable other components to move into the passive state. However given the existing definition of PS(Q) only components which are directly connected to Q are assumed capable of initiating transactions upon it. Clearly for systems with dependent transactions this is not true. In figure 2.1 *PS(N3)* would include *N3* and *N2* only. This allows *N1* to initiate a and consequently b on *N3*. Therefore if quiescence is to be achieved *all* components which are capable of initiating transactions directly or

indirectly on Q must be included in the passive set. This set is called the *enlarged passive set* (EPS) and for a component Q, the membership of EPS(Q) is defined as follows :

1. all components in PS(Q) are in EPS,
2. all components which can initiate dependent transactions which result in consequent transactions on Q are in EPS.

Together the adjustments of the passive requirements and the expansion of passive set mean that quiescence is once more reachable in finite time.

Management View: Having shown that quiescence is a reachable and sufficient condition for safe reconfigurations the approach goes on to describe the management process. The reconfiguration rules of page 32 are applied to the reconfiguration script and the list of components which need to be in the passive state *CPS* (change passive set) is derived. This is done by first determining the set of connections *CS* which must be unlinked to isolate components which will be removed. From this together with the set of connections *LS* directly specified in link or unlink directives the set of components QS (quiescent set) which must be made quiescent are identified. The CPS is the union of the passive sets PS of each component in QS.

CS = {connections c | c is a connection to/from a component to be removed}

LS = {connections l | l is a connection in a link/unlink directive}

QS = {components n | n is the initiator component on a connection in $(CS \cup LS)$ or n is to be removed}

$CPS = \cup\ PS(i)$ **forall** i in QS

In the event of concurrent reconfigurations the set of components which need to be frozen (LockSet) is a superset of the CPS. In detail the lock set is :

$LockSet = CPS \cup$ {compo'nts n | n is a *recipient* compo'nt on a connection in $(CS \cup LS)$}

The lock set must include components to which connections are directed so that reconfigurations do not attempt to make a connection to components deleted by other concurrent reconfigurations.

The model was illustrated by applying it to the "Dining Philosophers Problem" [CM84] with the solution implemented and tested in the *Conic* environment for distributed programming [MKS89].

Evaluation: The approach is quite general in that all types of reconfigurations are catered for and concurrent reconfigurations can be supported. However it places a heavy burden on the application programmer who must write components such that they will respond correctly to the *passivate* configuration management directive. This increases the complexity of the application code as each component must in effect exhibit two different behaviours.

Another disadvantage is that the approach uses component connectivity information as a means of determining the membership of CPS. The existence of a communication path (direct or indirect) from a component N to a component N' is taken to imply that N may initiate a transaction with N', which inevitably results in pessimistically large CPS'. In particular for systems with high levels of component connectivity, even small reconfigurations involving a few components will result in substantial disruptions to the system.

2.2.2.2 Equus - Kindberg

[Kin93] addresses dynamic reconfiguration in the context of loosely coupled client server software in open distributed systems. In the assumed interaction model, server processes with common interfaces are logically grouped into *server peers* and each client is associated with a particular peer which deals with its transaction requests. A transaction is defined as a series of *bilateral* processing steps which leaves both the client and the server in a consistent state. The goal of reconfiguration in this case is to reassociate a client already bound to a particular server with another server peer. Emphasis is placed on doing this transparently to the client and where possible to the server.

The proposition put forward is that a client c's association with a server s can be severed and re-established with another server peer s' without c's knowledge if s is in a state of *server-quiescence* with respect to c; s is said to be in a state of server-quiescence with respect to c if it is not engaged in a transaction initiated by c, and will not receive any further invocations from c (reminiscent of *passive* in [KM90] above)

The method attempts to bring about the state of server quiescence by interposing a *proxy* process between the client and the server. The role of the proxy is to act as a filter and selectively buffer or deliver messages exchanged between the server and the client such that new transactions from c are buffered while messages belonging to already initiated transactions are delivered.

Assuming that s does not hold any transient state i.e server peer s' will process c's further transactions with exactly the same results as s, then server quiescence is a sufficient condition for safe reassociation of c because s is in a consistent state with respect to c. Furthermore from c's point of view the delay due to the buffering of its new transactions, prior to being relayed to s', will appear as if s had suddenly becoming highly loaded.

For server-quiescence to be reachable client transactions must either not overlap or if they do, concurrent (overlapping) transactions must not synchronise with each other. This is necessary as otherwise the buffering of new transactions may lead to deadlock i.e when an already initiated transaction cannot complete because the client is blocked waiting for the completion of a new but buffered transaction[3].

It is assumed that proxies can distinguish between the first message of a new transaction and messages belonging to ongoing transactions by their content and therefore buffer/deliver messages appropriately. Proxies are also expected to determine when transactions have terminated and signal that the state of server-quiescence has been reached. The suggested mechanisms for this are to either infer from the exchange of messages when the transaction is over or to wait until a period of silence indicates that the transaction has terminated.

The intended platform for the approach is the *Equus* [Kin92] distributed runtime system which can support the initial reconfiguration of interposing the proxy between the client and server through its support for changing bindings between communication end-points and forwarding buffered messages atomically.

Evaluation: The advantages of the outlined proxy method are that the reassociations are transparent to the client and even though proxies are heavy weight processes and messages between the client and server must go through them, the system overhead is only imposed for the duration of the reconfiguration.

However, the approach is not general and only considers the reassociation of loosely coupled client servers in open distributed systems. Furthermore the suggested mechanism by which proxies infer the termination of transactions is not foolproof. Examining the exchanged data is not a reliable means of determining when a transaction has terminated especially when the duration of the session (i.e number of requests) depends on the client behaviour e.g. number of line drawing requests a client makes to a graphics server to draw "a picture" cannot be predetermined or inferred from message contents. The "*wait for a period of silence*" approach is also inadequate as the upper bound on communication delays and transaction processing times are highly variable and rarely known for distributed systems.

Also the transparency afforded to the clients during reconfiguration comes at the cost of writing intelligent proxies which are tailor made to understand the interaction protocol implemented by the clients and server peers they are to reassociate. Finally the assumption made about the underlying distributed runtime system, regarding its rebind and message forwarding capabilities, may have an impact on the general applicability of the approach.

[3]This is reminiscent of the adjustments made to the *passive* state in [KM90] for systems with dependent transactions.

2.2.2.3 Law governed systems - Minsky et. al.

[MUWZ96] describes reconfigurations in the context of distributed systems which operate under *Law-Governed Interactions* (*LGI*). In such systems components belong to groups whose message interactions are controlled by an explicit and strictly enforced set of rules referred to as the *law of the group*. The sending and arrival of messages are regulated events which depending on the type, source and destination of the message, trigger the evaluation of individual laws. Group laws determine whether sent messages are *forwarded* to their destination and whether arrived messages are *delivered* or *rejected*. The precise ruling of the law is a function of the group's *control state* and any temporal constraints specified in its definition. The control state CS_x is the part of the law associated with member x. It is a bag of attributes which can be queried and altered during law evaluation. The law is made global in scope by local agents called *secretaries* which enforce it on all group members. In the *Moses* toolkit [MU97], laws and the control states are expressed as Prolog clauses with a Prolog interpreter (controller) on each component which intercepts messages and acts as the secretary for collocated group members.

The overall aim of LGI's is to move synchronisation aspects of component interactions up to the group-law level so that they are made explicit and can be better controlled and reasoned about. The group control state therefore reflects information pertaining to externally visible component states and synchronisation variables which control the way in which components interact with each other. The addition of configuration information to the control states makes it possible to bring about controlled reconfigurations which do not introduce inconsistency. It is suggested that the laws of the system be augmented with rules which once triggered will affect the desired reconfigurations. In the token ring examples given, the control state reflects both configuration information (successor component) and synchronisation variables (the token). Thus, by examining and modifying component control states, the rules to *insert* and *remove* elements into the ring can determine when to affect the reconfiguration such that inconsistency will not be introduced. Since laws govern the forwarding and delivery of messages it is even possible for rules to buffer messages for later forwarding much in the same way as it was done in [Kin93] §2.2.2.2.

Evaluation: From the change designer's view point, the *controller* in law governed systems acts as a highly flexible and programmable proxy which will allow the change architect to program the safe state and subsequently the reconfiguration. However the approach suffers from the drawback that the "safe" state must constantly be rediscovered for each and every new application and even for every reconfiguration. In [KM90] §2.2.2.1 as long as components were programmed to move into the *passive* state, they could be used in any application and in any configuration without having to be reprogrammed to

bring about the safe state. This is not the case here as the safe state is not described in terms of abstract component states but programmed specifically for each situation. Given that groups cannot be organised hierarchically, this is liable to be a non-trivial task subject to human error, particularly for complex reconfigurations on large systems which require multiple components to be in the safe state. In effect reachability of the safe state is not guaranteed and depends on the programmer. In fact none of the three stages of consistency through avoidance §2.2.2 are automated and the change programmer is left to bear the brunt of the work - identifying the components which need to be in the safe state, ensuring that the safe state is indeed "safe" and reachable etc.

Also it is not made clear what information must be reflected in the control state of a component for it to become reconfigurable. Even if the static limitation on laws is removed so that they can be updated at runtime, the scheme cannot support evolutionary change as it is not possible to update the control state retrospectively such that it reflects needed but absent state and configuration information. Furthermore aside from the assumption that components involved in *LGI* can deal with the rejection of sent messages, the model imposes a large overhead on normal system operation as all regulated messages must be filtered and therefore go through the Prolog evaluator before reaching their destination. Finally no discussion of concurrency or minimising disruptions are made, presumably these aspects of change management are also left for the change programmer to deal with.

2.2.2.4 PCM - Etzkorn

Etzkorn [Etz92] rejects the option of defining the "safe" state for reconfiguration in terms of abstract component states, proposing instead to precede change actions with an exact application-specific definition. Thus the satisfaction of the safety condition becomes a prerequisite for the execution of the contingent change action(s). The argument put forward in favour of this approach is that it avoids increasing the functional complexity of the components[4] and given the tailored nature of the safety conditions and finer grain "port level blocking" it is likely to result in a much smaller disruption to the system. This style of change programming is termed Passive Configuration Management (*PCM*), as components do not need to participate in bringing about the safe state. In contrast Active Configuration Management (*ACM*) is the term applied to systems such as Conic [KM90], where component participation is an integral part of the avoidance mechanism.

In the distributed system model adopted, each component is associated with a communication interface which is made up of a set of communication end-points or *ports*. Ports are *linked* together to establish a reliable and error free channel over which typed messages can be sent (asynchronously) and received. It is assumed that for every port in

[4]This is because the extra code segments needed for reacting to high-level configuration management directives such as *passive* in [KM90] §2.2.2.1 are no longer necessary

their communication interface, components maintain a complete record or *port trace*, of the sequence of send and receive calls made. This port trace information (reminiscent of Hoare's *trace predicates* [Hoa85]) is made accessible to the change programmer via a small set of operators which can be used from within change scripts to specify the application-specific safety conditions mentioned earlier. In addition it is possible to *block* ports. Any component which attempts to make use of a blocked port will have its execution suspended until that port becomes *unblocked* again by a subsequent call to the *unblock* primitive.

Together these tools enable the change programmer to monitor and alter the behaviour of the application such that the safety conditions are brought about. For example consider the code segment dealing with a bounded buffer (see figure 2.2).

$$
\begin{aligned}
M\ : & \#(\Downarrow \{Req\}) = \#(\Downarrow \{Get\}) \\
& \rightarrow \text{block M.Req} \\
& \quad \text{unlink } \ldots, \text{ relink } \ldots \\
& \quad \text{unblock M.Req}
\end{aligned}
$$

Figure 2.2: Change Script Code Segment

It states that when the number # of send calls, in the port trace for port *Req* - $\Downarrow \{Req\}$ - of component M is equal to the number of receive calls in the port trace of its *Get* port - $\#(\Downarrow \{Get\})$ - then the *Req* port should be blocked preventing any new requests from being generated. The *relink* primitive acts in much the same way as the *Equus* rebind, atomically altering the port bindings and transferring buffered messages.

Evaluation: As with law governed systems the major disadvantage here is that the safe state needs to be rediscovered for every application and every new reconfiguration and the imperative nature of the safety conditions means that reachability of the desired configuration states does not automatically follow. Unlike LGI's however, where the control state of a group contained semantic and synchronisation information which helped the change programmer reason about the application and the safe state at an abstract level, in this approach the change programmer only has access to port trace information and must therefore think about the change and safe state in terms of low level message exchange information. Also for applications with complex interactions it may not be possible to determine from port trace information alone when the safe state has come about[5]. Furthermore for non-trivial reconfigurations which involve multiple components with dependent transactions, identifying and freezing the safe state is liable to be a difficult and complicated task

[5]See picture drawing client example in termination argument for *Equus* page 37

subject to human error. In addition the resulting change descriptions resemble large highly nested *if-then-else* statements which are difficult to comprehend. Again none of the three stages of consistency through avoidance have been automated leaving the change programmer to bear the brunt of the work.

Also considering that the safety conditions are expressed over port traces of running components, the evaluation of the safety conditions poses certain problems. Clearly if the outcome of the evaluation is to be meaningful the port traces under scrutiny should be prevented from changing during the evaluation process. This is done by suspending the execution of any components whose port traces appear in a condition currently under evaluation by the change management system. If the safety condition does not hold, e.g. number of request and replies are not equal, then the execution of the suspended components is resumed until the next message interchange which will suspend the components once more and start another evaluation cycle. As well as requiring many message exchanges between the affected components and the change management system, this process will result in long delays before the safe state is reached, particularly if a large number of messages need to be exchanged.

It is also not clear just how much port trace information needs to be kept, given that it is not feasible to maintain a complete history. Finally the assumption made about the underlying distributed runtime system, regarding its relink and message forwarding capabilities, may have an impact on the general applicability of the approach.

2.2.2.5 Warren & Sommerville

In [WS96] Warren and Sommerville propose an approach which is a combination of ACM and PCM. As in ACM a configuration manager attempts to impose the safe state. However to reduce the level of participation required from the component programmer some aspects of PCM are incorporated. The scheme categorises components as either *active* or *passive*. Active components can initiate interactions unilaterally while passive components only react to external stimuli. Components interact through ports (communication end-points) which are linked together to establish directed communication paths. A configuration manager component *(CM)* presides over reconfigurations and also handles message interchanges such that, under normal circumstances, sent messages are buffered and only dispatched after the intended recipient issues a corresponding receive call. During reconfiguration the CM is in a position to *block* ports such that components which attempt to send

messages on blocked output ports become suspended and buffered messages are not delivered to blocked input ports. The supported reconfiguration forms are component addition, replacement, migration and removal. The CM's strategy for bringing about the safe state for these reconfigurations is summarised in the following paragraphs.

For component addition - the CM takes no action aside from creating and initialising the component. A component added in this way is isolated and cannot reach or be reached by other components in the system. Thus, it is incapable of introducing inconsistency into the system.

For replacement and migration - port blocking is used to prevent the reconfiguration target from sending or receiving any messages, thus client messages are buffered but not delivered. The CM then halts the execution of the reconfiguration target, extracts its state and initialises the substitute component[6] before delivering any buffered messages.

A potential problem however is that in halting an active component, it is possible to interrupt its execution at a crucial point e.g. when writing a file to disk. Such an action may jeopardises the consistency of the component being replaced and hence that of the replacement which may now be initialised with invalid data. To avoid this situation, the notion of an *intra-atomic* operation is introduced. An intra-atomic operation is one which once started must not be interrupted and allowed to terminate. The component programmer is required to compile a list of the intra-atomic operations which occur in active components and make this information available to the CM. This is done by augmenting a component's configuration specification, which already consists of its communication interface, with the necessary additional information (see figure 2.3). Configuration specifications are written in the *PCL* [SD96] object oriented configuration language.

Another class of inter-atomic operations which need to be considered are *dependent-inter-atomic* operations. These are operations which must also be allowed to complete once started, but require computation from more than one component. As with inter-atomic operations the component programmer includes them in the component's configuration specification but needs to specify the ports which are required by each such operation as well, so that the CM does not block these and thereby prevent the operation from completing.

[6]The state capture and restore model proposed by Herlihy [HL82] is adopted for extracting and reinstating component states.

```
family producer inherits component
    class
        type ⇒ active
    end
    interface
        requires ⇒ put
    end
    behaviour
        intraAtomicOperation ⇒
            (writeToDisk)
        dependentIntraAtomicOperation ⇒
            produceAndSend, (put)
    end
end
```

Figure 2.3: PCL declaration for producer

Figure 2.3 is the PCL description for an *active* component *producer* which has one output port *put* and two atomic operations; the intra-atomic-operation *WriteToDisk* and the dependent-intra-atomic-operation *produceAndSend* which requires access to the port *put* to succeed.

The strategy employed for removal - is to block ports of the reconfiguration target and the output ports of its clients so that no new interaction can begin allowing the consistent reconfiguration target to be safely removed. However a problem exists with outstanding messages i.e those already sent by the component's clients prior to blocking. In order for the target component and clients which have started dependent-inter-atomic operations with it to be in a consistent state, these outstanding messages must be delivered and processed.

To allow these messages to be processed a policy similar to that of Etzkorn [Etz92] is taken. The component programmer exposes component state information[7] in the form of boolean variables accessible from the *PCL* configuration language. The change programmer will then use this information in conjunction with port trace information, such as the number of unsatisfied receive calls made on a port, to guide the configuration manager (ala PCM) towards a port blocking strategy which will permit outstanding messages to be processed.

[7]Similar to control states in LGI's

```
family buffer inherits component
    structure
        ...
    end
    behaviour
        full ⇒ get (queueSize(put))
        empty ⇒ put (queueSize(get))
    end
end
```

Figure 2.4: PCL for buffer

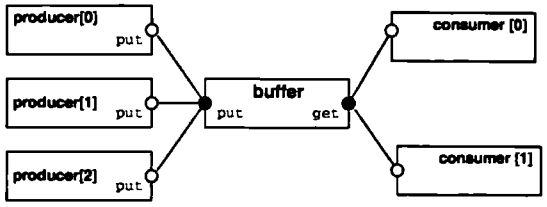


Figure 2.5: Bounded buffer

The PCL code in figure 2.4 shows that *buffer* exports two boolean variables *full* and *empty*. The behaviour section prescribes that for the safe state to come about when the buffer is *full* the CM should delay blocking the *get* ports of the consumers until *queueSize(put)* request have been made. Similarly when the buffer is *empty* the blocking of the *put* port of the producers should be postponed until *queueSize(get)* messages have been produced.

Evaluation: Much of the criticisms directed at PCM also applies to this scheme. Although some attempt has been made towards automating the process of establishing the safe state, particularly for the special cases of migration and replacement, the remove reconfiguration requires the participation of both the component programmer and the change architect. Coding the behaviour section of the PCL scripts requires detailed knowledge of the inter-working of the components in the system and is liable to become quite complicated and thus prone to programmer errors for non-trivial systems. The boolean state variables and the port trace statements are the means through which the CM brings about the safe state and must be evaluated after each message send or receive. Depending on the outcome of these and alterations to boolean state variables, ports need to be blocked/unblocked etc. This process of continuous checking and evaluation is expensive both in terms of communication and processing costs, it is therefore likely that for even moderately sized systems the CM will become a bottleneck.

In addition there is a great deal of detailed information which is exported from the component implementation level to the configuration level. Thus the configuration specification is tied not to the behaviour of a component type but to a particular implementation of that component type (e.g. a different implementation of the producer component figure 2.3 might not backup produced messages to file but send them to another component in a dependent-inter-atomic operation). This means that reconfigurations will invariably entail modifications to the currently static PCL definitions. Finally, although the scheme seems to support the most commonly

used reconfigurations, it is not made clear whether other reconfigurations can be programmed or not and if so what type of support if any is present for concurrency and minimum disruption.

2.2.2.6 Polylith - Hofmeister & Purtilo et. al.

There is a body of work based over the *Polylith* [PH91, Pur94] distributed programming environment which deals with dynamic reconfiguration.

Surgeon [HWP93] deals with reconfiguring systems which have a very loose notion of consistency - it assumes that components do not synchronise with each other, do not require any special initialisation or parameterisation when created, and that as long as messages are not lost consistency will be preserved. The mechanism for preventing message loss during reconfiguration involves *holding* the set of bindings which are to undergo change and transferring buffered messages to new destinations. The nature of the *hold* operation is such that components whose communication links are being *held* can operate "normally" but cannot communicate with other components. The assumptions however rule out the great majority of useful applications, for example even a request-reply interaction (cf. RPC) would invalidate the non-synchronisation assumption.

Later work [HP93] concentrates on providing a mechanism for packaging and restoring program state at runtime. The technique requires the component programmer to define *reconfiguration points* within the component code and indicate the nature of *program state* in the form of variables. A reconfiguration tool will then instrument the component source code with the extra statements necessary to capture this state along with the activation call stack at the reconfiguration point, package it up and reinstate it in the replacement, resuming the execution at the same point. Since instrumentation occurs at the imperative language level the compiler is left to deal with the machine specific details, making such state transfers possible across heterogeneous systems.

Evaluation: *Surgeon* and other work such as [MJ94] sidestep the consistency issue by dealing only with systems which do not synchronise and hold no state between individual message exchanges and as such are not interesting from our perspective.

The state transfer method on the other hand is relevant to our work as it addresses the issues of consistency and state. The call stack and the list of variables specified at the reconfiguration points define the pertinent state of the component at that instance thus facilitating the automation of the packing and unpacking of

state. Unfortunately this process is not entirely automated. Heap data and operating system related structures, e.g. file descriptors, sockets etc. are not transferred automatically. It is stated that transferring this type of data requires the cooperation of the programmer, but the nature of this cooperation and its integration into the state transfer and reinstatement is not expanded upon. In addition due to the way that the function call graph is reconstructed in the replacement copy it is possible for the replacement component to crash.[8]

Even if these problems are solved however, the main limitation here is the fact that the technique only addresses replacement; and even in that special case can only be usefully applied to exact-copy replacement i.e the replacement component must have the exact same implementation as its predecessor (useful for process migration), otherwise the state restoration process will fail.

2.2.2.7 Hauptman et. al.

[HW96] describes a mechanism very similar to that of Hofmeister and Purtilo §2.2.2.6. It is designed with the specific aim of supporting component replacement as a means of facilitating on-line maintenance, but differs from that above in that state transfer is not automated. Not only are component programmers expected to specify which objects constitute state at the reconfiguration points but must also code save and restore operations for said objects to facilitate state transfer. Component source code must also be modified to check for reconfiguration signals etc.

The prototype implementation is based over the *Chorus* [RAA+92] distributed operating system and uses the port migration facility provided by the underlying run-time system to transfer buffered messages from old components to replacements.

Evaluation: The main advantage of this approach over [HP93] is that it supports general replacements i.e not just exact copy replacements. However this flexibility is bought at the expense of the component programmer who must hand-code and preside over much of the state transfer and restore procedures.

Aside from the fact that other reconfigurations such as removals or changes in connectivity are still unsupported, it is worth noting that transferring state between components which are not identical is quite a complex task. Particularly in cases

[8]This arises when the parameters to function calls are expressions containing variables whose values change by the time they are checkpointed at the reconfiguration point. For example the expression $(\frac{z}{x})$ which originally succeeded, may with the new value of x at the reconfiguration point, lead to a divide by zero exception in the replacement.

where there isn't a direct correlation between the data and execution trace at a reconfiguration point in the old component and that of the target reconfiguration point in the replacement. This situation becomes even more complex if not unmanageable, when the old component is to be replaced with more than one component. Finally, although the mechanism is not tied to the underlying *Chorus* system it utilises a number of its features which may make it difficult to port onto other distributed runtime systems.

2.2.3 Recovery-avoidance hybrid

2.2.3.1 Guide - Oueicheck et. al.

[OdP96] proposes a basic reworking of the recovery method based over the *Guide* [BBD+91] object oriented distributed platform. In the scheme *active* (threaded) objects acquire locks on *passive* (non-threaded) objects before invoking their methods. An *activity* initiated by an active object may invoke the methods of any number of passive objects before committing. If the activity is unable to commit then a rollback mechanism is required to rollback the side-effects of the operation.

The recovery process relies on activities periodically checkpointing the state of the objects they have access to. This is referred to as storing a *cut*. The cut of an activity a at time t (written $cut_t(a)$) consists of the set of values at instant t for all the objects in the set $Inv(a)$, where $Inv(a)$ is defined as the set of objects whose methods can be invoked by a. As a means of increasing the level of achievable concurrency in reconfigurable applications, simple (not strict) two phase locking is used. However, the simple two phase locking protocol allows objects to be unlocked prior to a commit and can thus give rise to cascading aborts. To avoid the possibility of cascading aborts, it is assumed that in the case of aborts only objects currently locked by the activity need to be rolled back. The assumption that unlocked objects are consistent and that changes made to them are effectively committed means that activities are not atomic entities and have more in common with *serialising actions* described in [SW90]. In addition to this assumption, activities are required to store a cut on every lock and unlock operation.

The "avoidance" aspect of the scheme operates as follows. Prior to a reconfiguration passive objects which are targeted for removal or replacement, are instructed by a configuration manager process to move into the *flushing* state. In this state new lock requests are refused but method invocations by activities which already hold locks on the object are permitted to go ahead. Since in a dead-lock free system,

method invocations are guaranteed to terminate in finite time, eventually all activities holding locks on flushing objects will release them, thereby allowing the objects to move into the *potentially disconnected* state which is amenable to reconfiguration.

Evaluation: The advantages of this method over the traditional recovery method is only apparent in a few special cases. The flushing state allows activities which have already gained locks on an object to complete, however the completion of an activity is often conditional on gaining locks on more than one object. Flushing objects are not aware of each other and so a flushing object will refuse all lock requests even if they originate from an activity which already holds locks on a number of other flushing objects and is about to commit. In effect the only case were the flushing state will be of any real benefit is when an activity has gained all the locks it needs and is allowed to run to completion, whereas in the recovery method it would have been aborted. Aside from this and the special case of replacement in which components wait for a substitute object instead of aborting, little advantage is gained in minimising system disruption. A further disadvantage is that the approach incurs the combined runtime overheads of both avoidance and recovery schemes. Also due to the way configuration information is maintained, reconfigurations involving component removal become very slow and costly to execute.

From the descriptions given in the paper it is not clear why all invokable objects need to be included in an activities *cut*, and not just those which have been modified. The frequency of storing cuts is also unexplained unless a cut also acts as a commit.

2.2.3.2 Lim

Lim [Lim93, Lim96] describes a reconfiguration environment which relies on having access to finite state machine (FSM) specifications for all the components in the system. The application programmer is expected to provide the FSM's which are then fed to a generator to produce intermediate results for analysis and runtime control. These intermediate results are used to produce plans for failure recovery and dynamic reconfiguration. The computed reconfiguration schedule is supplied to a runtime control server which enforces synchronisation control, implements recovery and manages reconfigurations. Components are obliged to request permission from the control server to move to a destination state, by invoking the `request_permit` method and announce their new state by a call to the `announce` method. Depending on the situation the server may grant or withhold permission. Components can choose to wait for the transition to become acceptable by calling the `block_request`

method and wait until permission is GRANTED.

It is expected that FSM's contain recovery transitions which will allow components to move to a consistent state in case of errors or transitions forced on the component by the runtime system. The scheme supports and encourages the organisation of component groups into hierarchies partly as a software engineering tool but also as a means of reducing the size of the composite state machine generated for the system. By default a product machine is automatically generated by taking the Cartesian product of the individual basic machines to model the behaviour of the system. Composite state machines written by the system designer consist of composite states and constraints which identify only legal states. These are used to prune illegal states and compute a *restricted product machine* for the group.

To bring about change in this scheme the system administrator supplies a set of reconfiguration transitions to the compiler which generates a reconfiguration schedule for the runtime control server. The control server acts in a similar way to the configuration manager in the avoidance approaches by forcing the affected components to a state where the reconfiguration transitions can begin. This is done by first calling the `force_mode` method to move components from autonomous mode to exception mode where they can be steered through a partial recovery path to the desired state via calls to the `mandatory_transition` method.

Evaluation: There are a great number of advantages to using FSM's as the basis for modelling a system. Immediately a number of desirable properties such as reachability of states, absence of deadlock, live-lock and starvation can be checked for. Since dependency information is available recovery can be carried out on a much finer scale with the consistency preservation system working out which machines need to roll back to which states. This allows programmers to avail themselves of richer interaction models than that of transactions which almost all the other models rely on.

There are however an equally large number of disadvantages. In the absence of automated tools, specifying FSM's for every component in the system is a laborious task particularly as FSM's for components of even moderate complexity can become unmanageably large. This is even the case when there is a many to one mapping between program state and machine state. The state explosion problem is an inherent problem of combining FSM's and comes about despite efforts to prune the product machine by deleting illegal states identified by constraints and composite machine specifications. The size of the product machine can impact the time it takes

BIBL LONDON UNIV

to compute the reconfiguration schedule, becoming intractable for large systems.

Another issue is that the recovery path selected by the scheme is not necessarily an optimal one, so although there is no need to wait for a reconfiguration state to come about as is the case with *Conic*, the forced reconfiguration path may result in greater disruption. Calculating the optimal recovery and reconfiguration path is problematic both in terms of the metrics used in selecting among the many available paths and also in terms of the computational overhead.

Furthermore even simple reconfigurations require that a new product machine be constructed for the affected section of the system. Potentially this change may need to be propagated up the hierarchy requiring the suspension of affected sections of the system while the new synchronisation schedules are uploaded to the control servers. The scheme can only be applied to finite state automata, does not deal with concurrent reconfigurations and has a relatively high runtime penalty with components needing to exchange an average of two messages per transition.

2.2.4 A comparison

The mutual consistency approaches surveyed differ in many respects such as the type of reconfigurations they cater for, the amount of programmer and system level support they require, the disruption they cause to the remainder of the system etc. In this section we will use the criteria set out below to draw a comparison between them.

1. **Type of supported reconfiguration** - This ranges from specific cases such as exact copy replacement useful only for migration, to general support for all types of programmed and evolutionary reconfigurations.

2. **Minimum disruption** - This is a measure of how well the mechanism manages to minimise the disruption to the running system during reconfiguration.

3. **Support for Concurrent Reconfigurations** - To minimise system disruption it would be advantageous if the consistency mechanism could be used such that concurrent reconfigurations could be carried out on the system.

4. **Programmer participation** - This is a measure of the extent of participation required from the application and change level programmers by the consistency mechanism.

We believe that it is important for the participation of the programmer, in bringing about the safe state for reconfiguration and reducing system disruption, to be kept to a bare minimum. This is so that the possibility of human errors are reduced and the process of writing reconfigurable applications is simplified.

5. **System Requirements & Overhead** - This is a measure of the overhead and requirements imposed on the system by the consistency mechanism (e.g. system should be transaction based with rollback recovery support). It is important for these requirements to be as few as possible so that the approach can be applied to a broad set of application domains.

   Also, the addition of the consistency mechanism should not have a dramatic impact on system performance, particularly during normal system operation.

From looking at Table 2.1 it is possible to observe that few of the surveyed mechanisms provide support for general reconfigurations. Those which do, either impose large penalties on the runtime system (e.g. Lim [Lim96], Recovery mechanisms [Blo83, Lis88, PSWL95]) or on the Programmer (Conic [KM90]). Where the issue of minimising disruption to unaffected sections of the system is concerned, again few mechanisms accomplish the task without relying heavily on the programmer. This evidence suggests that there is a need for a consistency preservation mechanism which is able to ensure consistency during general reconfigurations and minimise system disruption, but do so without imposing large penalties on the programmer or the runtime system. This issue is addressed in chapter five.

2.3 Re-Establishing application invariants

As mentioned in chapter one application state invariants are predicates expressed over the states of the components in the system such that the safety and liveness properties of the system are conditional on their satisfaction. Although invariants can be global in scope it is more often the case that they are expressed over the state of a subset of the components which together make up a logical part of the system. The members of these subsets (subsystems) are usually characterised by certain properties pertaining to their configuration e.g. their type, topology, the machine they are executing on etc. Since reconfigurations can alter the membership of these sets they can also invalidate the invariants.

	Supported Reconfigurations	**Minimum Disruption**	**Concurrency Support**	**Programmer Participation**	**System Requirements & overhead**
Recovery e.g Argus	Programmed & Evolutionary	Not Addressed. All Active Transactions are Aborted	Supported	Requires Defensive progr-ming + save/restore procs	Small
Conic	Programmed & Evolutionary	Leads to large disruption in interconnected systems	Supported	Large for component programmer	Small
Equus	Not general - specific case only	Not Applicable	Not Applicable	Special proxies needed for each service type	Small - except during reconf requires rebind& msg forwarding
LGI	Programmed	Not Addressed	Not Addressed	Large for change programmer	Large - all msgs filtered by Prolog law evaluator
PCM	Programmed	Fine-grain tailored safety conditions	Left to change programmer	Excessively large for change programmer	Small - except during reconf, requires relink & msg forwarding
Warren	A subset of Programmed	Not Clear	Not Clear	Large - Required from component & change programmer	Large during reconf
Polylith	Not general - Exact Copy Replace only	Not Applicable	Not Applicable	Required if component state includes heap/OS data	Small
Hauptman	Not general - Replacements only	Not Applicable	Not Applicable	Large - must code state transfers	Small - requires port migration
Guide	Programmed & Evolutionary	Reduces aborts & restore operations in some cases	Not Addressed	Needs locks & recovery mechanisms	Large - combination of recovery & avoidance runtime overheads
Lim	Programmed & Evolutionary	Fine grain recovery compared to transactions	Not addressed	Large - must supply FSM's & recovery states	Large - during normal system operation

Table 2.1: Comparison of approaches to preserving mutually consistent component states during Reconfiguration

The mechanisms covered in the previous section prevent the *execution* of reconfigurations from leaving components in mutually inconsistent states but do nothing to preserve or restore application invariants. Thus it is possible for a reconfiguration to result in a system which fails to satisfy its invariants but is nevertheless structurally sound and all its constituent components are in mutually consistent states. For example in a token ring system, the structural integrity requirement is for the components to be arranged in a single ring structure with the application state invariant specifying that there should be exactly one component in the ring which possesses the token. The removal of the component which possesses the token and linking its predecessor component to its successor under one of the consistency preserving mechanisms covered above will result in just such a system. Even though the reconfiguration results in a structurally sound system with all components in mutually consistent states, progress cannot be made since no token exists. Another example of an invariant might be that a replacement component has the same state as that of the component which it replaced.

Although under certain circumstances no action needs to be taken as the invalidated invariants have been phased out or replaced as a result of an evolutionary reconfiguration, it is more often the case that the invariants need to be re-established. The problem therefore is to identify which invariants have been violated as a result of dynamic reconfiguration and to take the necessary measures to restore them. In the following sections we will survey existing approaches to this problem.

2.3.1 Embedded actions

2.3.1.1 Conic

In Conic [KM90] performing the *link* and *unlink* reconfiguration actions results in the execution of preprogrammed *initialisation* and *finalisation* routines in the (*passive*) components involved. These routines are written by the application designers and are expected to carry out the post-reconfiguration state transitions necessary for re-establishing the application invariants.[9] The authors observe that the complexity of the actions which bring about these transitions will depend on the complexity of the application and the autonomy of the components, and may involve communication with other components in the system.

Warren [WS96] adopts a similar but more coarse grained approach where the

[9]The *create* and *remove* reconfiguration actions are supported implicitly, as it is assumed that newly created components need to be *linked* to the rest of the system and components targeted for removal will first be *unlinked*.

initialisation and finalisation routines are called only on component creation and deletion respectively.

Evaluation: Components in a distributed system are rarely in a position to determine, based on local state alone, what course of action needs to be taken in order to re-establish invalidated application invariants. This is especially true when as is often the case, invariants are expressed over the combined state of a number of the components in the system. What this means in practical terms is that application designers are forced to embed information regarding the nature of the invariants and the components on which they depend, into the components of the system. This is so that when the initialisation/finalisation routines are invoked components will be able to determine which other components to communicate with and how to proceed in re-establishing the invariants. Hard wiring this type of application specific information into the components of a distributed system has a number of undesirable side effects.

Components written in this way lose their generality, as the routines make assumptions about the system invariants, the configuration in which the component is used, and the cooperation of the other components in the system. Some or all of these assumption may prove false if the component is used in a different system with different invariants etc. or as a result of an evolutionary reconfiguration. Furthermore these assumptions are not made explicit and remain hidden in the application code, thereby increasing the probability of errors when planning reconfigurations. Finally the task of the application level programmer is made more difficult. In the case of *Conic* the mutual consistency mechanism already requires components to be programmed such that they can react to high-level configuration management directives. In addition to this the programmer will now have to write the initialisation/finalisation routines and coordinate the exchange of information between components necessary for restoring the invariants.

2.3.1.2 Clipper

The *Clipper* system [AHP94] deals with this issue by requiring that components provide a *dump-state* method which once invoked will transmit "necessary" state information to a specified component. The *dump-state* methods of any number of components can be triggered externally and it is assumed that from this exchange of information components will be able to work out how to re-establish application invariants.

An almost identical approach is taken by Zimmermann in [ZD94], where a *copy* operation is invoked instead of a *dump-state* method.

Evaluation: This is a slight variation on the basic "embedded actions" approach. The difference is that the change programmer and not the components themselves, coordinate the exchange of information. This is advantageous as it removes the necessity of embedding that information into the component of the system. However components are still assumed to know what information needs to be transmitted and on receipt how to deal with that information in order to re-establish the application invariants. As before these assumptions detract from the generality of a component and restrict the range of reconfigurations it may safely undergo.

2.3.2 Possible non-embedded methods

2.3.2.1 LGI

As was mentioned in §2.2.2.3 the aim of LGI's is to specify and control the semantic and synchronisation aspects of component interactions at the group law level. To do this component control-states include much of the state and synchronisation variables which would in other systems be a part of the component's program state. So although no mention of this issue is made in LGI literature one can envisage a scheme where the control-states of the components and the Laws of the groups can be amended after reconfigurations such that application invariants are re-established. Of course this would require that application invariants are expressed not over the state of the components but over their control-states instead.

Evaluation: LGI schemes are inherently inefficient as all messages need to be filtered by the law evaluators which will become bottle necks. Furthermore in current implementations group laws are static and cannot be changed at runtime.

2.3.3 Discussion

It is a fact that despite continuing research in the area of formal specification, application designers rarely invest the time and effort necessary for specifying application invariants in a format which would lend itself to automated processing. Thus when dealing with the issue of invalidated invariants it is the programmer who must be relied upon to identify and rectify the situation.

However, as we have seen existing approaches by and large rely on embedding the extra functionality necessary for dealing with invalidated invariants, into the code of the components. This is unsatisfactory as it detracts from the generality of the components, increases their complexity and narrows the range of reconfigurations which can be applied to them. The alternative LGI approach and those similar to it are also inadequate as the impact on normal system operation is excessively large.

In chapter four we propose a scheme whereby invalidated invariants can be identified and re-established by the *change designer* from within the reconfiguration language, with only marginal assistance from the application programmer. The scheme has a minimal impact on the complexity of the components, and actually enhances their generality and potential for reconfiguration.

2.4 Specification of change

As we mentioned in chapter one the main aim of a configuration management system is to facilitate structured support for consistency preserving, programmed and evolutionary reconfiguration of running distributed systems. To do this it is necessary for the change management system to provide the means through which change can be expressed. Expressing change to a dynamic system introduces certain problems which need to be resolved. These and those of consistency and minimum disruption must be taken into account when selecting a suitable specification mechanism.

At its most primitive, dynamic reconfiguration is brought about through the parameterised execution of a sequence of basic change actions. The basic reconfiguration actions needed to bring about structural change at the component level are *link*, *unlink*, *create* and *delete*. These establish/break a communication link between components, remove an existing component and create a new component, respectively. The particular sequence of these actions necessary to bring about a desired reconfiguration is specified by the change designer, either textually in a change script written in a reconfiguration language, or interactively through a graphical interface.

Although convenient for performing simple reconfigurations, interactive graphical reconfiguration tools such as the *Architect's Assistant* [KMNS93], *Schooner* [HS94] or *Beagle* [FS96], are unsuitable for describing complex, or programmed changes. This is largely due to the lack of selection, looping and structured programming constructs which make expressing complicated or even simple but repetitive change slow, difficult and subject to human error. These disadvantages stem from the fact that human interactions are by nature slow and error prone,

in addition as reconfiguration actions are dispatched a few at a time, the issues of consistency, minimising disruption and concurrency are further exacerbated. Reconfiguration languages provide a much sounder basis for supporting programmed and evolutionary changes and it is these and their associated runtime systems which we concentrate on here.

There are a relatively large number of reconfiguration languages and when conducting a survey of them, it quickly becomes evident that there is little consensus on what properties they must possess. In the following sections we will identify a number of central issues which we believe must be dealt with by reconfiguration languages. We will then survey a representative sample of existing reconfiguration languages to illustrate the way in which these issues are addressed in real systems.

The access problem - This problem refers to the way in which components targeted for reconfiguration are selected by the change designer. Clearly when planning a particular reconfiguration, change designers must be able to gain *handles* to *specific* components/links in the system and supply these handles as parameters to the change actions. In reconfiguration languages this refers to the mechanism through which the association between a label in the reconfiguration script and an actual running component is established.

Using names or "Roles" to identify components uniquely is problematic as to fully support evolutionary change *every* component in the system must be given a unique and meaningful string name or role. Not only is this difficult and laborious but it can also serve to confuse if evolutionary change alters the actual roles of components but does not change their corresponding role names. An alternative mechanism allows change designers to select components based on their configuration. This requires access to a selection mechanism which can query the current system configuration and return handles to running components which match the configuration properties specified in the query. This is advantageous as the structural preconditions/assumptions which are made by the script are embedded in the queries and thus made explicit. This is not the case when the component selected is specified by its "role".

Programming abstractions - An obvious goal in the design of programming languages is to make the jump from the design phase to actual implementation as small as possible by providing structures in the language which mirror the abstractions used in the design methodology of choice (e.g. Object Oriented Design and

Object Oriented Languages [Boo94]). In this regard reconfiguration languages rate poorly as the great majority fail to provide support structures which would permit the change designer to organise change scripts such that they reflect the abstractions used when designing the system.

As we mentioned earlier distributed systems are often designed by establishing a hierarchy using composite components. However since composite components are design abstractions, performance considerations often force them out of the final system elaboration stage. Thus the design time hierarchy is collapsed and a flat component structure is created. Despite this fact it is unwise to expect change designers to reason about and specify changes to large and complex distributed system in terms of a flat seemingly structure-less sea of interconnected components. Therefore the change management system must maintain the configuration information in such a way that the design time hierarchy of the system is not lost, but made available to the change designer through appropriate constructs in the reconfiguration language. Composite components and primitive components must be treated uniformly so as to enable the change designers to reason and program changes at different levels of the composition hierarchy.

Concurrency - Since one of the principal goals of performing changes at runtime is to minimise the disruption experienced by the clients of the application any advantage which can be leveraged from concurrency must be considered. The two possible scenarios are concurrency within the execution of a single reconfiguration, and that possible among multiple reconfigurations.

Assuming that change actions within a single reconfiguration do not conflict with each other, it is relatively easy to exploit naturally occurring synchronisation points, evident from their semantics, to parallelise them e.g. a component must first be created before it can be bound, components must only be removed after any other commands involving them have terminated etc.

Concurrent execution of multiple reconfigurations must be subject to the serialisability property. This means that access to the system configuration should be subject to a concurrency control mechanism which can prevent or resolve conflicts among ongoing reconfigurations.

Integration with consistency mechanisms - As was set out in sections §2.2 and §2.3 in order for dynamic reconfiguration to preserve the consistency of the system it is necessary to prevent change actions from leaving components in mutually

inconsistent states, and to re-establish application invariants invalidated as a result of the reconfiguration. Depending on the mechanism chosen to deal with these issues, certain information will need to be exchanged between the language runtime system and the consistency mechanisms. For example in the case of Etzkorn's (see Figure 2.2 page 39) mutual consistency approach the reconfiguration language had to be augmented with the facility of querying port trace information maintained by the runtime system.

Thus it is important for reconfiguration languages to provide the constructs which will permit change designer to cooperate with the mutually-consistent-states mechanism, and the restoration of invalidated invariants.

Triggering of programmed reconfigurations - Programmed changes are reconfiguration which were foreseen at system design time. A simple example of such change is the reconfiguration script to grow a server pool at runtime in response to increasing client requests. In general programmed reconfigurations are associated with a condition which when satisfied make their execution desirable. In the example given, the grow reconfiguration becomes desirable when the number of clients per server exceeds a certain limit.

The way in which a configuration management system deals with the specification, and the binding of the triggering conditions to their associated change scripts has a serious impact on the type and flexibility of programmed changes and the runtime overheads of the system. Triggering conditions can be quite complex, containing temporal expressions in combination with expressions using logical, ordering and comparative operators applied to component state values and events generated by the application and its environment (e.g if a server is busy for more than five minutes and it receives more than 3 client requests, new servers should be created)

Using the configuration language to specify both the change script and the triggering conditions dramatically increase the complexity of the language and its associated runtime system. Embedding the binding of "triggering condition to script" in the reconfiguration language is also inadvisable as future changes to the triggering condition requires changing the reconfiguration script.

2.4.1 Darwin - Dulay et. al.

Darwin [MDK94a] is the latest in a series of configuration languages following research carried out on the *Conic* [MKS89] and *REX* [KMSD92] projects. Although Darwin's primary function is to establish the *initial* configuration of distributed ap-

plication, it has some facilities for expressing certain types of programmed change. Programmed change is supported through *lazy instantiation* of *recursively* defined components. Dynamically created components are indistinguishable from their "normal" counterparts in every respect other than the use of the **dyn** keyword before their instantiation. The instantiation and bindings to these components however are not carried out immediately upon system elaboration, but delayed until a message is actually sent to them. Figure 2.6 defines a lazily instantiated recursive component **lazypipe** which can be used to implement the primes sieve of Eratosthenes.

```
component lazypipe {
    provide input ;
    require output ;
    inst
        head: filter ;
        tail: dyn lazypipe ;
    bind
        input - head.left ;
        head.right - tail.input ;
        head.output - output ;
        tail.output - output ;
}
```

Figure 2.6: Lazy pipeline component

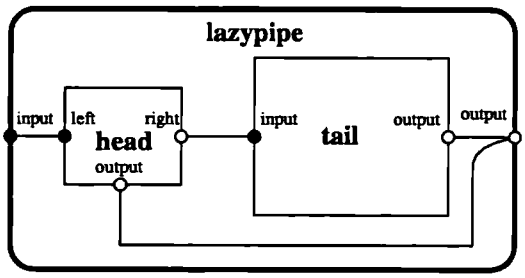


Figure 2.7: Lazypipe

Lazypipe is a composite component with two subcomponents; *head* of type filter and *tail* of type Lazypipe (recursive). Its two communication end-points *input* and *output* are bound to filter and tail as depicted in Figure 2.7. In the **inst**antiation section, head is instantiated normally while tail is specified to be a **dyn**amic component. Thus, whenever a head component sends a message to its neighbouring tail component the recursive structure is elaborated and the chain of lazypipes is lengthened. The Darwin runtime system takes care of the instantiation and binding necessary to do this. Darwin also supports a service which allows composite and primitive components to be created dynamically from within the computational components of the system. This however is undesirable as the separation of concerns between configuration and computation is lost.

Although in its current form Darwin can be used to express many useful reconfigurations, support for evolutionary, and programmed changes which require remove and rebind operations is nonexistent. Due to the restricted set of supported reconfigurations Darwin does not need to deal fully with the access problem, concurrent reconfigurations, and integration with consistency mechanisms. It does however provide support for composite components. Reconfigurations are triggered

implicitly when messages are sent to dynamic components.

2.4.2 Etzkorn

The language proposed by Etzkorn [Etz92] is similar to Darwin in that process types are defined in terms of their communication interfaces and an initial configuration script realises the system by instantiating and linking components. Etzkorn's language includes *remove* and *unlink* primitives lacking in Darwin, which permit all types of reconfigurations to be expressed.

However as described in §2.2.2.4, the language and the mutually-consistent-states algorithm are very closely integrated, so much so that the consistency mechanism determines the structure of the change program. Furthermore, the language lacks support for composite components, does not exploit the potential for concurrency in executing individual or multiple reconfigurations, nor does it address the issue of re-establishing invalidated invariants.

The access problem is resolved through a naming scheme which requires components to have unique names within the system. It is not made clear how the change programmer gains access to these names. The triggering scheme is not described.

2.4.3 Zimmermann et. al.

In [ZD94] Zimmermann and Drobnik propose a language for the specification and evolution of distributed systems. The section of the language which deals with specifying reconfigurations is based on Darwin with the difference that it permits unlink and remove operations.

The access problem is resolved through the concept of "Roles" mentioned earlier. Upon creation components are associated with certain *roles* within the distributed system. The change programmer gains handles to running components by asking the system for components which embody specific roles in the system. Roles can be delegated at runtime and are therefore superior to the unique naming schemes which result in change scripts that name the components they act upon directly and thus cannot be reused easily. The *Conic* mutual consistency mechanism (see §2.2.2.1) is adopted, although it would seem that the change programmer has to manually select which components need to be *stopped (passivated).*

Triggering conditions for programmed changes are specified within the reconfiguration language and are predicates expressed over configuration information and predefined network/application events. The conditions themselves and the binding

to particular reconfigurations once defined cannot be changed.

Composite components, concurrency and mechanisms for re-establishing application invariants are not supported.

2.4.4 Gerel

Gerel [End92, EW92] follows on closely from Darwin but again with the distinction that it supports unlink and remove operations. It addresses the access problem by defining a separate language called *Gerel-SL* which allows change designers to query the system and gain handles to components based on their configuration. It supports composite components although reconfigurations cannot span multiple levels in the hierarchy.

There is no integration with a mutually-consistent-states mechanism, or support for re-establishing application invariants. Triggering of change and concurrency are alluded to but not explained.

2.4.5 Discussion

Clipper [AHP94], PCL [SD96], Durra [BDW+92], The Raven Configuration language [CN92, CN94], CL [JC92], Vigil [Buz94] and those of Young [You91], and Vestal [Ves94] are some of the other proposed reconfiguration languages. These languages are in general tightly coupled with their underlying distributed runtime systems and only address a subset of the issues set out at the beginning of this section (see page 57).

This stems from the fact that there isn't a clear separation of concerns between the different issues which a change management system must address. The runtime objects which should implement these operations and the interactions between them have not been elucidated. Thus each language comes with its own particular runtime support system, which both duplicates and hides operations which should have been implemented in a separate layer. Concurrency control, the maintenance of configuration information, and even consistency preservation mechanisms are configuration management concerns which should be independent of the reconfiguration language. In this way different languages which better suit a particular application domain can be built over the same set of basic services. This approach also leaves room for experimentation with different means of accessing configuration information, concurrency control mechanisms and consistency preservation mechanisms without the need for changing the entire languages runtime system.

2.5 Chapter summary

In this chapter we surveyed existing literature dealing with consistency, minimum disruption and the specification of dynamic change.

We presented existing mechanisms for maintaining mutually consistent states during reconfiguration and found that the approaches either did not deal with general reconfigurations or relied excessively on the application and or change programmer for their operation. Furthermore these approaches did not perform well on the issue of minimising system disruption.

The methods for re-establishing invalidated application invariants where also found lacking, as they relied on embedding application specific details in generic components. This restricts both the types of reconfiguration an application could undergo, as well as the application environments in which components written in this way could successfully participate in.

Given the requirements of the dynamic reconfiguration service we formulated a set of necessary properties which reconfiguration languages needed to exhibit if they are to be used successfully in specifying dynamic change. Our survey revealed that none of the existing reconfiguration languages possess the full complement of properties and that their runtime systems duplicated much of the functionality which should rightfully have been implemented elsewhere. This led us to the conclusion that the dynamic reconfiguration service was in need of a framework which establishes a clear separation of concerns among its various responsibilities and identifies the runtime entities which are necessary for its operation; thereby providing a stable platform over which reconfiguration languages and other reconfiguration services can be built.

Chapter 3

The Change Management Service

Chapter two motivated the need for a framework which identifies the basic element of the dynamic reconfiguration service and establishes a clear separation of concerns between the responsibilities of the components which implement it. In this chapter we describe such a framework and in doing so keep view of fact that the change management system is part of a wider set of system services which comprise the distributed runtime system. Thus wherever appropriate we assume the existence, and become clients of system services which are necessary to the reconfiguration service but do not logically belong within it.

3.1 A general framework

The presented framework is based on the connection oriented distributed runtime model. This choice was guided by a desire to make the discussions here as widely applicable as possible. The communication oriented model affords the most generality since it forms the basis over which other model abstractions such as the process oriented and distributed shared memory models are built.

Figure 3.1 depicts our view of the reconfiguration service. In it we identify the reconfiguration manager (RM), the configuration database (CDB), the consistency manager (CM) and a number of runtime hooks as the fundamental elements necessary for the provision of a dynamic reconfiguration service. In addition to these we identify the requirement for an event composition service (ECS) of which the reconfiguration service is a client. The reconfiguration service is provided on a per-application basis and thus runtime objects which implement the logical tasks attributed to these components exist alongside application components and are in fact a part of the application configuration.

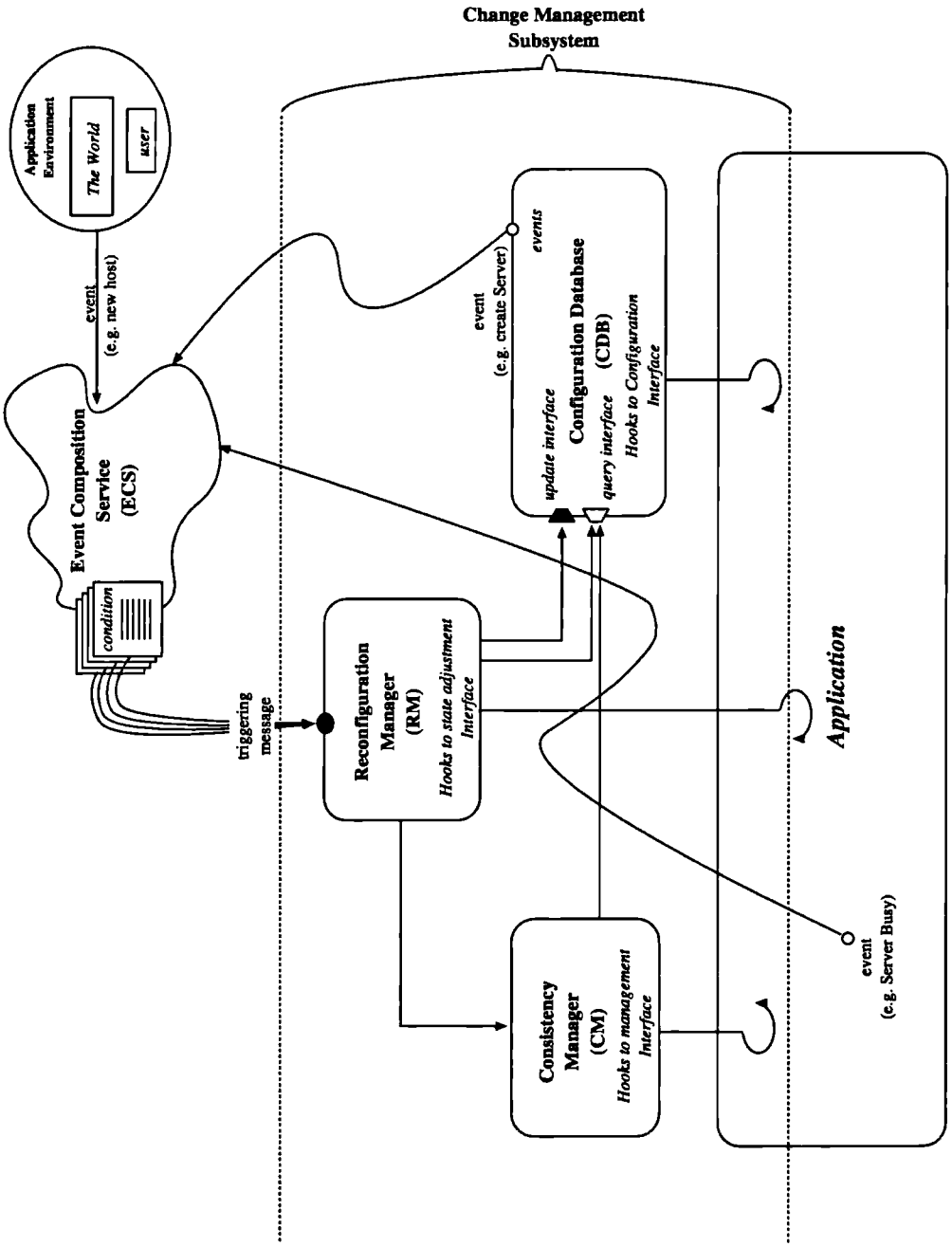


Figure 3.1: The Framework

The reconfiguration manager (RM) is the entity which upon the arrival of triggering messages selects and executes reconfiguration scripts. The RM coordinates the execution of the reconfiguration scripts with the consistency manager (CM) and configuration database (CDB) such that reconfigurations do not interfere with each other and leave components in mutually consistent states.

The consistency manager (CM) encapsulates the safety mechanism used to ensure that components are left in mutually consistent states after reconfiguration. Thus reconfigurations only proceed after the CM has been consulted and has signalled that they can go ahead safely.

The configuration database (CDB), maintains and affects changes to the system configuration. It exports an interface which can be used to query and modify the system configuration. Interactions with the CDB are transaction based and go through an internal concurrency control module which coordinates concurrent access to the system configuration.

In the framework change scripts are written in a reconfiguration language and remain dormant pending the generation of messages which trigger their execution. The event composition service (ECS) evaluates the triggering conditions written by the change designers and generates the messages which trigger the execution of the reconfiguration scripts.

The re-establishment of the application invariants is controlled from within the reconfiguration program with the aid of specialised runtime hooks. The mechanisms for re-establishing application invariants and ensuring mutually consistent states are elaborated upon in chapters four and five respectively.

The following sections give a general view of the roles these components fulfil in the proposed framework and justify the particular separation of concerns we have chosen.

3.2 Reconfiguration manager (RM)

The function of the reconfiguration manager (RM) is to use triggering messages generated by the ECS to select and execute reconfiguration scripts[1]. As part of the script selection process any parameters included in the triggering message are unmarshaled and propagated to the script prior to execution. However before the reconfiguration manager can proceed with the execution it must ensure that running

[1]Strictly speaking these are two separate tasks, but they are quite closely integrated and more easily accomplished by a single entity.

the change script will not interfere with that of other ongoing reconfigurations, or leave components in mutually inconsistent states.

To this end, it communicates with the CDB to ascertain whether the script's execution can go ahead without interfering with other running scripts. When this is the case, it instructs the CM to carry out the actions necessary to ensure that components will not be left in mutually inconsistent states. Given these assurances the RM executes the script and upon completion signals the CM to reverse any actions taken as part of the mutual consistency mechanism, thus returning the system to its normal state.

As mentioned in chapter one §1.2.2, performing consistency preserving dynamic reconfiguration requires that the new system satisfies application state-invariants. In chapter two §2.3.3 we observed that the approach most systems adopted towards this issue was to embed all the necessary interactions for preserving/reinstating application invariants, in the code of the components making up the application. This approach was unsatisfactory as it increased the functional complexity of the components, detracted from their generality, and restricted the range of reconfigurations they could safely undergo. In our framework we propose to coordinate these interactions from the configuration language level. The immediate consequence of this decision is that the execution of change scripts must now have *two* concrete side-effects.

(a) to bring about the envisaged configuration changes, and

(b) to coordinate the state adjustments necessary to identify and reinstate invalidated application invariants.

As figure 3.1 depicts, the reconfiguration manager has access to specialised management hooks in the distributed application which allow the reconfiguration programmer to instigate the state adjustments necessary for re-establishing invalidated application invariants. Configuration changes are accomplished in the normal way by performing CDB updates, as described in section §3.3.

We defer the discussion of coordinating state adjustments from the configuration language level to chapter four.

3.3 Configuration Database (CDB)

Before we go on to describe the properties of the CDB and its interactions with the other system entities, it is important to realise that we are not so much introducing

a new system component but bringing an already existing one out into the open. Regardless of the approach a distributed programming environment adopts, be it process oriented RPC, shared memory, tuple space or connection oriented message passing, all distributed run-time systems are faced with basically the same problem. Namely that of creating objects on remote sites and establishing communication between them. To do this they must maintain configuration information. What we require in addition to this is that this information not be discarded after initialisation or embedded in the application state, but made externally visible and kept up-to-date as changes occur.

In our framework the CDB not only maintains the configuration of the system but facilitates modifications to it by accepting reconfiguration requests in the form of the basic change actions *link, unlink, create* and *remove*. Updates to the database are reflected onto the application configuration using hooks into the underlying distributed runtime system. All modifications to the system configuration are carried out through the configuration database and thus in the absence of failures the information held in the CDB will provide an accurate image of the system configuration.

Access to accurate configuration information is essential for planning reconfigurations and thus the CDB provides an interface which allows the system's configuration to be queried. Although these queries can originate from any source, the most notable client is the reconfiguration language which will use this information to resolve the access problem introduced earlier in section §2.4 (Page 57). The configuration information stored per component in the CDB consists of the component's type, connectivity (i.e bindings), and the host it is executing on. In this context a component's type information specifies its communication interface and a number of intransient values initialised at component creation time.[2]

Modern distributed runtime systems treat communication end-points as first class objects which can be passed around in messages. Thus components may establish communication paths with each other simply by exchanging communication end-points in messages. The CDB takes the stance that these bindings are short term in nature and as such not of interest to the change designer who is concerned with altering the long term configuration of components. As a result the binding information stored in the CDB does not include transient communication paths established in this way.

[2]As in the case of a ring, configuration information alone may not be enough to uniquely identify a component. The inclusion of a constant (intransient value) e.g. component-name can resolve such situations.

In addition to the query and update interface the CDB also supports an event generation interface (cf. active databases). Clients such as the triggering conditions in the ECS can subscribe to this interface and receive notification of interesting reconfiguration events such as component creations or removals.

3.3.1 Concurrency

The need for concurrency was motivated in chapter one mainly as a means of reducing the disruption suffered by the system during reconfiguration. This is certainly the main advantage of supporting concurrency within a single script. Supporting concurrency across multiple reconfigurations is also advantageous as it makes the system more responsive to corrective change, allowing multiple upgrades or repairs to be carried out simultaneously.

The CDB is the entity which handles updates to the system configuration and therefore it is also the component which implements the necessary structures and control algorithms to facilitate concurrency within single and across multiple reconfiguration scripts.

Concurrency within single scripts - Since all interactions are transaction based, the CDB is capable of identifying change requests which belong to the same transaction and can therefore parallelise their execution by exploiting the naturally occurring synchronisation points in the basic reconfiguration actions. Briefly these synchronisation points are as follows - *bind* and *unlink* actions which act on different communication end-points can occur in parallel but must operate on existing objects and thus where appropriate will need to follow *create* requests; *remove* operations are only carried out if the target component is fully unlinked. This topic will be dealt with in more detail in section §6.1.2.

Concurrency across multiple reconfigurations - In order to maximise concurrency, traditional concurrency control mechanism allow updates to go ahead and rely on locks, versions, deadlock detection and roll-back schemes to resolve conflicts as and when they arise. The CDB is an active database and as such updates to it have concrete side-effects on the underlying system. This property poses problems when rollbacks become necessary. Consider that even though the execution of reconfiguration actions are not slow or costly in themselves, when coupled with the price of employing consistency preserving actions which adjust application states, abort

application level transactions and or passivate regions of the system, the overall penalty for rolling back their results becomes quite large.

The strategy adopted by the CDB is to require that database updates are only reflected onto the application configuration on commits. Section §6.1.2 describes this scheme in detail and examines its ramifications when used in conjunction with existing concurrency control mechanisms.

3.4 Consistency manager (CM)

The consistency manager is the entity responsible for ensuring that components are not left in mutually inconsistent states as a result of the execution of change actions. The CM receives a description of the intended change from the reconfiguration manager (RM) and when ready, signals the go ahead for execution of the reconfiguration. These interactions remain unchanged regardless of the actual approach taken by the CM towards the mutual consistency problem. This is important as depending on the nature of the application domain, it might be sensible to adopt different approaches to resolving the mutually-consistent-states problem. In the framework this is easily accomplished by selecting a CM component which implements the appropriate strategy for the target application. Naturally the operation of the CM is largely dependent on the mutual-consistency approach it adopts (e.g *recovery* vs *avoidance*).

Avoidance schemes have a number of distinct advantages over recovery schemes particularly with respect to minimising the disruption to the system. Chapter five gives a detailed description of the particular avoidance scheme proposed by the thesis. However a general description of the CM behaviour for both *avoidance* and *recovery* approaches is set out below.

Recovery approaches - as mentioned in section §2.2.1, require an exception mechanism which can identify and abort ongoing transactions which will be affected by the configuration changes. For such schemes, the CM implements this exception mechanism and will abort affected transactions.

Note that if the underlying distributed runtime system comes equipped with an exception model which incorporates reconfigurations and has hooks into the application level transaction system, the CM component becomes redundant and can therefore be excluded from the final application configuration.

Avoidance methods - need to identify and move affected components into the "safe state" for reconfiguration, and thus require communication with application components and access to the system configuration. For such schemes, the CM implements the avoidance algorithm and channels any necessary component interactions through the management hooks depicted in figure 3.1. Access to the system configuration is gained via the CDB query interface.

The extent of interactions necessary between the CM and the components depend on the particular avoidance scheme chosen and are the determining factor in the nature of the management hooks.

3.5 Event composition service (ECS)

Change management is in essence a reactive system. A number of possibly independent events occur which when considered together make the execution of a particular reconfiguration desirable. The events that go on to make up the triggering condition are varied and can be temporal in nature. Therefore the evaluation of the condition is not a trivial task, especially in view of communication delays. Also it is quite common for combinations of the event attributes used in the evaluation of the triggering condition to appear as parameters of the reconfiguration script it triggers. Consequently some form of communication must be established between the component which evaluates the triggering condition and the entity which executes the triggered reconfiguration script. For example a triggering condition may specify that, to release underutilised system resources, *if the average load of a server group is less than a threshold value over a period of an hour, then a member of the server group should be destroyed.* Obviously the server group that the rule needs to modify is the same one whose load average was calculated.

As we saw in chapter two §2.4 many of the reconfiguration languages/systems which dealt with the issue of triggering programmed changes chose to incorporate the specification and evaluation of the triggering condition into the reconfiguration language. This increases the complexity of reconfiguration language and its runtime system, and also binds the triggering condition and the reconfiguration scripts together in such a way that a change to the condition will require changing the entire script.

In keeping with these observations, the framework proposed in chapter three confined the responsibilities of the reconfiguration language to the specification of configuration changes only, and relegated those of specifying triggering conditions

to the event composition service (ECS). Communication between the triggering condition and the reconfiguration script is established via the triggering message generated by the ECS which contains the necessary script parameters within it.

Although the provision of an event composition service can easily fit within our framework, the need for an event composition service is not unique to the change management subsystem. Monitoring systems are a major client of such a service and there are a number of languages such as GEM (Generalised Event Monitoring Language) [MS95] which support this type of event composition. Notable features of such systems are, support for dynamic registration of new event types, dynamic registration of new clients and the ability to generate new events composed of incoming ones. Therefore as figure 3.1 shows we have chosen not to include the event composition service as part of the change management subsystem, but have instead assumed its existence and become its client.

So far we have discussed how events are used by the ECS to generate the messages which trigger reconfiguration scripts, however we have not discussed where these events come from, nor how they are bound to the conditions in the ECS, and from the ECS to the scripts.

Event generation - Events types occurring in a distributed application environment can be categorised as follows :

1. **Application related events -** These pertain to the application state of the components. Examples of these are events such as the load of a server, size of its input queue etc.

2. **User and system generated events -** This category matches system exception events such as host-up, time related events like clock-ticks or user generated events e.g. shutdown.

3. **Configuration related events -** Examples of these are events such as the creation, destruction or modifications to bindings of the components making up the application.

Category one events are application dependent and the programmer must therefore either program the components to generate them explicitly or direct a monitoring tool[3] such as that used in the ISIS and Meta toolkit [MWCB90, BCM90] to

[3]Statistical analysis tools, Performance monitors, debuggers etc. all require the use of a monitoring service and thus we have not attempted to incorporate such a service as part of the change management system but have assumed its existence.

instrument the code as necessary. Similarly events falling into the second category are generated by the system and entities external to the application and fall outside the jurisdiction of the change management service.

Events falling into category three however are the responsibility of the change management service and the provisions in the CDB for generating events (§3.3 page 69), where put in place for this specific purpose.

Binding - The mechanism used to bind generated events by the CDB and other event sources to the triggering conditions, and the messages generated by the conditions to the reconfiguration scripts, require that the change management system expose the event sources and sinks so that the necessary communication paths can be established. Although the binding issue falls outside the scope of the reconfiguration service and thus is not dealt with in the thesis, the problem has all the hallmarks which would make it an ideal application area for the configuration paradigm.

One can imagine the elaboration of the final distributed application being the result of the realisation of two sets of configuration descriptions. The first would establish the component hierarchy and communication structure of the system and the second would create the bindings between the triggering conditions and the reconfiguration scripts they invoke. This approach to binding would also pave the way for applying reconfigurations to the bindings between the event sources and the triggering conditions, and those from the triggering conditions to the reconfiguration scripts they invoke.

3.6 Scalability

Although in the framework we describe the logical tasks and interactions of the sub-components of the reconfiguration service we do not give any indication of how these components should be implemented. This is because such implementation decisions are closely tied to the behaviour of the client application. For example when considering a large distributed system, it is quite conceivable that a centralised implementation of the CDB will not scale well and thus may become a bottle-neck. However a distributed implementation will be equally inappropriate for a moderately sized system which only reconfigures occasionally.

Although distribution is often seen as a solution to scalability problems, recent research [BDLM96] casts doubt over the viability of standard distributed solutions. It is suggested that the additional communication and synchronisation delays due

to existing distributed solutions can dramatically reduce any perceived advantage. A possible avenue for research investigated by the paper is the use of *mobile-agents* which embody parts of the management software and move close to where the code needs to be executed. This area is still under investigation and is a subject for further research.

3.7 Chapter summary

Chapter one specified that the task of the change management system was to allow change designer to specify and bring about configuration changes on a running application whilst preserving consistency, and minimising disruption to the remainder of the system. In this chapter we proposed a framework of interacting components which collectively perform the tasks necessary for managing dynamic reconfiguration. The framework came about as a result of breaking these tasks into related sub-tasks and is novel in that it identifies the responsibilities of the reconfiguration service and assigns each logical task to a sub-component. The interactions between these sub-components are specified and lead to a clear separation of concerns between their individual responsibilities and consequently that of the reconfiguration service as a whole. This is useful as it allows researchers to categorise their work based on the particular aspect of the change management system addressed, and leads to a clearer understanding of the context of each work.

The framework also provides a design template for the implementation of flexible reconfiguration services whose components can be easily modified and or reused. This is a byproduct of separating the responsibilities of the various components and limiting communication between them to predefined interactions.

Another strength of the model is due to the fact that the change management system is itself part of the application configuration. This allows change designers to customise the reconfiguration service for the specific needs of the application it is helping to manage. For example when defining the configuration of the configuration management system, CM implementations can be used which implement the appropriate strategies for the consistency requirements of the application.

Chapter 4

Specification of Change

One of the conclusions of the survey in chapter two (section §2.4) was that existing reconfiguration languages did not provide an adequate platform for the specification of evolutionary and programmed changes.

A common problem was due to the fact that the responsibilities of the reconfiguration language were not well-defined. This resulted in misguided reconfiguration languages with over-burdened syntax and semantics, which in addition to configuration changes attempted to facilitate many other aspects of change management as well. A large proportion of these languages were also quite poor at providing constructs which help the change programmer bridge the gap between the abstractions used when designing the system and those available when programming reconfigurations. Other major flaws included the lack of a general solution to the invalidated invariant problem (see §2.3), and the adoption of access mechanisms which do not scale well and are unsuited to the task of describing programmed and evolutionary change.

Chapter three presented a framework which divided the responsibilities of the reconfiguration service amongst a number of logically separate runtime entities and restricted the responsibilities of the reconfiguration language to facilitating the specification of change and coordinating the re-establishment of invalidated application invariants. In this chapter we will describe a new approach to designing and writing reconfigurable distributed systems which relies on this framework and extends the ideas put forward in the configuration programming paradigm to the specification of dynamic reconfiguration. The approach, which we have termed *reconfiguration programming*, incorporates a new solution to the invalidated invariant problem and facilitates the expression of evolutionary and reusable programmed changes.

4.1 Evaluation of Configuration programming

One of the main reasons that dynamic reconfiguration has become a feasible possibility for modern distributed system is due to the adoption of the configuration programming paradigm as a design methodology. In view of this fact it is not entirely surprising that the great majority of current reconfiguration languages are in some way or form descendents of an existing configuration language. It is important to remember however, that the configuration programming paradigm itself was devised primarily for expressing the static initial structure of large distributed applications, and was not aimed at supporting dynamic change. As a result, although these reconfiguration languages/environments have benefited from the considerable strengths of the configuration programming paradigm, they have also unwittingly taken on board its weaknesses.

In this section we will evaluate the strengths and weaknesses of the configuration programming paradigm with a view to extending the paradigm such that it caters for dynamic reconfiguration. Other issues pertinent to the specification of dynamic change such as language features and access mechanisms are also considered.

Separation of Computation and Configuration concerns The main thesis of the configuration programming paradigm was to arrive at a clear separation of concerns between the description of system configuration and computation through their independent specification. Computational components were programmed in a computational language while the description of system structure was defined in a configuration language. To facilitate this approach, each computational component had a corresponding configuration specification which encapsulated its communication interface and instantiation parameters. These configuration descriptions were then manipulated, from the configuration language level by the system architect to coordinate the create and bind actions which would establish the application's initial configuration.

This separation of concerns promotes the development of context independent computational components that can operate in a variety of different configurations and can thus cope with reconfiguration. It is therefore vital that any proposed reconfiguration scheme maintains and if possible reinforces this separation of concerns.

Note however that the separation of computation and configuration concerns which is clearly present when defining the initial elaboration of the system, becomes blurred when describing dynamic reconfiguration. This is because modifying the configuration of a running system has a direct impact on the operation of its

computational elements and can invalidate or alter the application invariants.[1]

Application Invariant mechanisms The configuration programming paradigm was not designed to cater for reconfigurations and as such does not provide support for reinstating application invariants which have been invalidated as a result of reconfigurations.

As we saw in chapter two the great majority of existing environments which actually do deal with this issue, rely on the component programmer to embed the necessary actions for reinstating application invariants in the component code. Components written in this way however, lose their generality as the embedded code segments make assumptions about the system invariants, the configuration in which the component is used, and the cooperation of the other components in the system. Some or all of these assumption may prove false (e.g when the component is used in a different application with different invariants). Additionally, this solution does not cater for evolutionary reconfigurations which become necessary after the component code has been compiled and is already running. More importantly however, the separation of concerns between the computation and configuration aspects of the application is lost since the computational components must now be aware of the results of reconfigurations.

The solution, we proposed in chapter three, was to coordinate these interactions from the reconfiguration language level instead of embedding them in the code of the application components. For this solution to work, application components need to provide general purpose state access-methods which can be invoked by a third party to query or adjust the state of the components. These *state-access* methods would then be invoked from within the reconfiguration language by the change programmer to identify and re-establish invalidated application invariants.

However the component descriptions in the configuration programming paradigm are inadequate for this purpose as they do not encapsulate sufficient information about the computational elements of the application to facilitate this type of manipulation. In order to facilitate such manipulation configuration descriptions must be extended to encapsulate the *state-access* interface of computational components as well as their communication interface and instantiation signatures.

Composition and Hierarchy A cornerstone of the configuration programming paradigm was its advocacy of composition as a design abstraction for constructing

[1] Another side-effect of reconfigurations is that components can be left in mutually inconsistent states. This issue is dealt with in chapter five.

distributed systems. At its most basic, composition is a convenient mechanism for grouping components together such that the collective exhibits certain useful computational behaviour.

The configuration programming paradigm however took this concept a step further by introducing composite components, which in addition to grouping components together also encapsulated their configuration behaviour. This was achieved by allowing composite components to have configuration descriptions which could be manipulated in the same way as those of actual computational components. Composite components encapsulated the exported communication interface of their internal components and could be instantiated and linked in the same way as primitive components. A major benefit of composite configuration descriptions is that they enable the system architect to think in terms of complex objects which accomplished high-level tasks, and to design applications by establishing a composition hierarchy of progressively more complex components.

It would follow, that the reconfiguration programmer should also be allowed to reason about and express changes to the application configuration in terms of these abstract composite components. Most configuration environments however discard this information subsequent to system elaboration. This places the reconfiguration programmer at a considerable disadvantage as in the absence of the composition hierarchy it is necessary to reason about and describe reconfigurations in terms of the flat view of the application.

Therefore it is essential for reconfiguration approaches to maintain the design-time composition hierarchy of the system and allow the change designer to express reconfigurations to it. Furthermore, the composite configuration descriptions should be extended such that they incorporate the definition of programmed reconfiguration scripts which perform useful reconfiguration on the internal components of the composite. In this way composite components would encapsulate both the configuration and reconfiguration behaviour of their internal components. This feature promotes reuse as complex change scripts can now be expressed simply by combining existing reconfiguration methods.

Access mechanism Another issue which is pertinent to reconfiguration programming is that of *access*. Change designers need to gain handles to reconfiguration targets before they can express changes to their configuration. In chapter two section §2.4 we observed that query based approaches which gained handles to their reconfiguration targets by querying the application configuration at runtime enjoyed

considerable advantages over direct naming and role based schemes.

This suggests that reconfiguration approaches should adopt the query based route to resolving the access problem.

Language support One of the aims of the configuration programming paradigm was to make the application configuration as explicit as possible. To this end configuration languages were made as bare as possible, consisting of little more than simple looping and conditional constructs and the basic *bind* and *create* configuration actions. This approach was advantageous as the configuration of the application could be easily deduced from these descriptions and thereby made configuration descriptions amenable to automated analysis tools.

Although sufficient for the expression of the initial configuration of the application, these languages are unequal to the task of expressing complex reconfigurations. In particular they lack the expressive power and support for structured programming techniques which specifying complex reconfigurations requires. Additionally, language features such as inheritance can be used to establish another level of grouping which is in addition to that created by the composition hierarchy

4.2 Reconfiguration programming

The principal concern of the configuration programming paradigm was to facilitate the description of the initial configuration of distributed applications. In this section we propose the *reconfiguration programming paradigm*, which aims to support both the initial configuration and subsequent reconfigurations of the application.

The requirements of the reconfiguration programming paradigm are set out below. These have been chosen with the aim of addressing the issues raised in the previous section. Where appropriate we have indicated which of the logical entities in the framework presented in chapter three implements the necessary functionality.

4.2.1 Overview

In the reconfiguration programming paradigm, each computational component type has a corresponding configuration type-description specified in the reconfiguration language. The initial and subsequent configurations of the application are realised by manipulating *instances* of the configuration type-descriptions. Thus, changes made to an instance of a configuration type in the *reconfiguration program*, are

reflected[Mae87] onto the configuration of that instance's counterpart component in the *distributed application*.

Thus at any one time, there are two programs executing. One is the distributed application and the other is the configuration program. For every component in the distributed application, there is an instance of a corresponding configuration type in the reconfiguration program. Once begun, interactions with the configuration program are done through an event based interface. Triggering messages are sent to predefined communication end-points and can invoke programmed reconfigurations or result in the dynamic loading and execution of an evolutionary change specification.

The following sections will describe the requirements placed on computational components, and elaborate on the nature of the configuration type-descriptions and the reconfiguration language.

4.2.2 Primitive state-access methods

Computational components in the reconfiguration programming paradigm implement and export a set of state access-methods. These access-methods allow an external agent to query and alter a selected subset of a component's internal state at runtime. The particular subset of the components state which is exposed by these access-methods is decided upon by the component designer. In general primitive components should provide "get" and "set" methods for state variables which control synchronisation and computational behaviour of the component.

These methods are used by the change designer to identify and reinstate application invariants which have been invalidated as a result of reconfigurations. To aid concurrency, it is forbidden for "get" methods to have any side-effects which may be observed through the invocation of other access-methods (i.e. read only). Access-methods are further constrained to only return and update information which relate to the internal state of the component they are invoked upon.

The invocation mechanism for this process is provided by the reconfiguration service runtime system; in figure 3.1 (see page 66) this corresponds to the state adjustment hooks extending from the reconfiguration manager (RM) into the application.

4.2.3 Augmented configuration specifications

Configuration descriptions in the reconfiguration programming paradigm are a superset of those in the configuration programming paradigm and contain sufficient information to encapsulate both the configuration and reconfiguration behaviour of primitive and composite components. The additional information included in the augmented configuration descriptions is as follows:

4.2.3.1 Primitive configuration descriptions

In addition to the communication interface and instantiation parameters, primitive configuration specifications now also include the method signatures of the state access-methods exported by the computational component. This is so that change designers can invoke these methods from within their reconfiguration scripts and use them to identify and reinstate application invariants which may be affected by the change.

The reconfiguration language runtime system is responsible for interacting with the reconfiguration manager (RM) and performs the marshaling and unmarshaling of parameters and return values that remote method invocations entail.

4.2.3.2 Composite configuration descriptions

In addition to the communication interface and constructor declaration, composite configuration descriptions in the reconfiguration programming paradigm support the definition of two new method types.

- **Reconfiguration methods -**

  It is possible for a component designer to augment the configuration description of a composite with any number of reconfiguration method declarations. A reconfiguration method is a programmed change script which alters the internal configuration of its associated composite component in some useful way. Reconfiguration methods exhibit the same characteristics as member functions in OO languages and thus promote encapsulation and information-hiding. For example, useful reconfiguration methods for a server-pool composite component would include *grow* and *shrink* methods which increase and decrease the number of servers in the server-pool, respectively.

  Composite components provide both an event based and a procedural interface for the invocation of their reconfiguration methods. The event based interface

allows triggering messages to invoke reconfiguration methods of the composite or to trigger the dynamic loading and subsequent execution of an evolutionary change specification. The procedural interface allows reconfiguration methods to be invoked through an instance of their composite component type.

Reconfiguration methods must be written such that they maintain the composite component's exported communication interface, respect its structural integrity requirements and preserve subsystem invariants.

Reconfiguration methods should be constrained such that they cannot, except through the invocation of other reconfiguration methods, alter the internal configuration of subordinate composite components. This is a natural application of the encapsulation and data-hiding software-engineering principals and results in robust and extensible programs. In this way, the reconfiguration methods of high-level composite components can accomplish complex reconfigurations which span many levels of the hierarchy simply by invoking the reconfiguration methods exported by their child-composite components.

- **Composite state access-methods -**

  Composite configuration descriptions can also include the definition of state access-methods. These serve the same function as those provided by the primitive component programmer (see section §4.2.2 above), with the distinction that they provide state access-methods for "composite state". Composite state access-methods generally combine the invocation of a number of child state access-methods with some computation over the returned values in order to bring about a complex state adjustment or to return a compound value. For example, the designer of the server-pool component mentioned earlier, might provide a `setMaxSimultaneousClients(int)` composite state adjustment method which alters the maximum number of clients a server can serve simultaneously, for all existing servers in the pool.

  Composite state access-methods are not allowed to alter the application configuration and therefore their declarations cannot include reconfiguration method invocations or calls to any of the basic reconfiguration actions such as *create*, *bind* and *unlink*.

The inclusion of composite state access-methods and reconfiguration methods encourage encapsulation and thus reuse. Not only are composite components more likely to be used without alterations but the task of reconfiguration programming

also becomes considerably simplified as the most simple reconfigurations no longer require the change designer to understand the internal workings of all the components in the system.

4.2.4 Hierarchic configuration information & actions

In the reconfiguration programming paradigm, the access problem is resolved by selecting components based on their configuration properties alone. Change programmers therefore, query the system configuration, from within the reconfiguration language, for components which match the configuration properties exhibited by the reconfiguration targets. The component handles returned as the result of the queries, are then used by the reconfiguration scripts.

The reconfiguration programming paradigm also stipulates that the application's composition hierarchy must be maintained and made available throughout its lifetime. Thus depending on the nature of the queries composite or primitive component handles can be returned.

This enables the reconfiguration programmer to think and express changes in terms of the same abstractions used originally during system design.

The configuration database (CDB), as described in the framework of chapter three, maintains the application configuration and also supports an interface for servicing configuration updates and queries. Performance considerations make it undesirable to carry the composition hierarchy over to the actual system elaboration stage and so the CDB will need to maintain the mapping between the design time hierarchic view of the system and the actual configuration. The CDB interface will also need to support queries and updates which operate on composite components.

4.2.4.1 Reconfiguration language

The reconfiguration language and runtime[2] requirements of the reconfiguration programming paradigm are as follows:

- **Object Oriented** : The reconfiguration programming paradigm takes the stance that reconfiguration programming is no different from other programming disciplines and requires that reconfiguration languages be expressive and provide the full complement of constructs and object oriented programming

[2]This refers to the language runtime system.

abstractions which programmers in other fields have found so invaluable. Inheritance is of particular importance as it can be used to specify generic (re)configurations. This feature is illustrated by example in section §4.6.

- **Dynamic Linking capabilities** This feature is needed to support the execution of evolutionary reconfigurations.

  For compiled languages this capability is provided by the operating system. For interpreted languages this is a language runtime feature. In both cases, such support is almost universal.

- **Expression of reconfiguration descriptions** The language must be able to express the configuration type-descriptions. The concept of a class or even a structure is sufficient for this purpose. This is automatically implied by the compliance to the object orientation requirement.

- **State-access interface** The language must interface with the reconfiguration service runtime system in order to facilitate the invocation of state access-methods from within the language.

  Such a general purpose invocation interface and its integration into the Java language is described in section §6.2.2.3.

- **CDB query & update interface** The language must interface with the reconfiguration service runtime system and expose the CDB's query and update interfaces. This enables the change programmer to query the system configuration and also to invoke the *create*, *bind*, *unlink* and *remove* actions from within the language.

  Section §6.2.6 describes such an interface made between the prototype CDB and the Java language.

Note that these requirements are such that most modern object oriented languages can be adapted and used as the reconfiguration language. The prototype implementation used Java as the reconfiguration language.

4.3 An extended Example

Now that we have presented the basic principals of the reconfiguration programming paradigm we will illustrate its utility through the use of an extended example. The example is based around the "UK National Lottery".

Section §4.3.1 gives an overview of the lottery system and describes a possible distributed solution to the problem. Section §4.4 describes a number of useful reconfiguration which the lottery application needs to undergo. These reconfigurations are first expressed in the style of existing reconfiguration environments and are used to point out the shortcomings of such solutions. Section §4.5 expresses the same reconfiguration but this time the reconfigurations are expressed in the style advocated by the *reconfiguration programming paradigm.*

4.3.1 The UK National Lottery

The lottery operates as follows. Lottery machines are placed in shops and other outlets throughout the UK and allow customers to purchase lottery tickets. Each lottery ticket is a record of the six numbers the customer has chosen as his lucky numbers and guarantees the purchaser a single entry into the "lucky draw". At the time of purchase it is possible to associate a repeat period with a ticket such that the same numbers are entered into the following n draws. The cost of such tickets is $n \times u$ where n is the repeat period of the ticket and u is the unit cost of a single entry into the draw. The purchasing of tickets stops only an hour before the draw is scheduled to take place.

On the Wednesday and Saturday of each week, a computerised "Winning Numbers" component selects at random six lucky numbers. Customers which have selected identical numbers to the six chosen at the draw are awarded the lion's share of the jackpot. A percentage of the jackpot is also divided amongst tickets which included five, four and three correct numbers. The draw is televised and presenters need access to information such as the size of the overall jackpot, number of principal winners and the size of their winnings, along with similar information for the other winning categories. Other miscellaneous information such as the most/least popular numbers chosen are also of interest.

A distributed solution Figure 4.1 depicts a possible national lottery system consisting of three main elements - the control-display, winning numbers and ticket database components.

The diagrammatic notation adopted here follows that used by Darwin[MDEK95] in that rectangles are used to represent computational components and small circles on the edges of a rectangle are used to represent the communication end-points of the component. Filled circles are used when the communication end-point is an input port and output ports are represented by empty circles. Established communica-

tion paths are depicted by directed arcs which connect communication end-points together. Round rectangles are used to represent composite components.

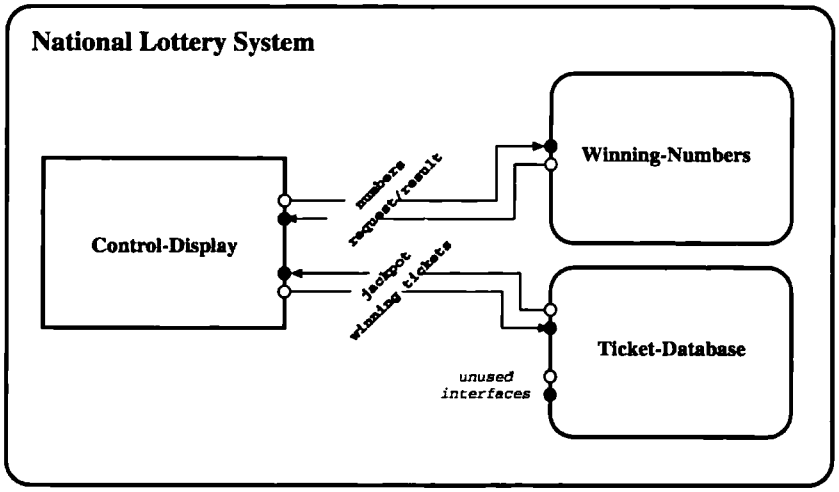


Figure 4.1: National Lottery System

The control-display unit, is used by the presenters of the national lottery to instruct the winning-numbers component to generate the winning lottery numbers, and to query the ticket-database for information such as the size of the jackpot, the number of winners, individual winnings etc.

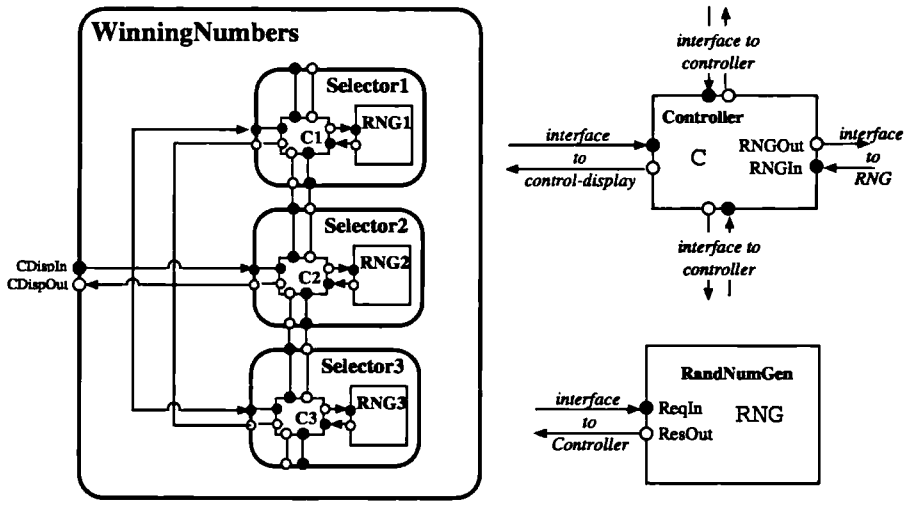


Figure 4.2: WinningNumbers component

The winning-numbers component is itself made up of **Selector** components which receive the number requests from the control-display component and negotiate

with each other before one of them replies with the winning numbers. As figure 4.2 shows each number Selector in turn consist of a **Controller** and a random number generator **RandNumGen**.

The controllers carry out three separate tasks. First, they negotiate with each other in order to decide which of the Selector components will in fact choose the winning numbers. Once this decision has been made, the chosen controller will make successive requests to the random number generator until the six requisite winning numbers have been generated before forwarding them to the control-display component. In addition to this, the controllers periodically reseed the random number generators. It is an invariant of the subsystem that every Selector's random number generator has a unique seed. Thus when reseeding occurs the controllers need to reach a consensus over the seeds they allocate to the random number generators. This strategy is put in place in order to prevent people with knowledge of random number generators to predict the outcome of the lottery.

The ticket-database maintains a record of tickets bought across the UK. Every lottery machine has a corresponding process of type **Mach** associated with it. As tickets are issued by the lottery machine the Mach process transmits details of the transaction such as the ticket identifier, six chosen numbers, repeat period of selected numbers and the machine identifier via one of its communication end-points. These messages are received by a DBase component, **DB**, which can be queried about the number of tickets sold and asked to calculate number of winners etc. Figure 4.3 shows a sample configuration with three Mach processes bound to a single DBase component.

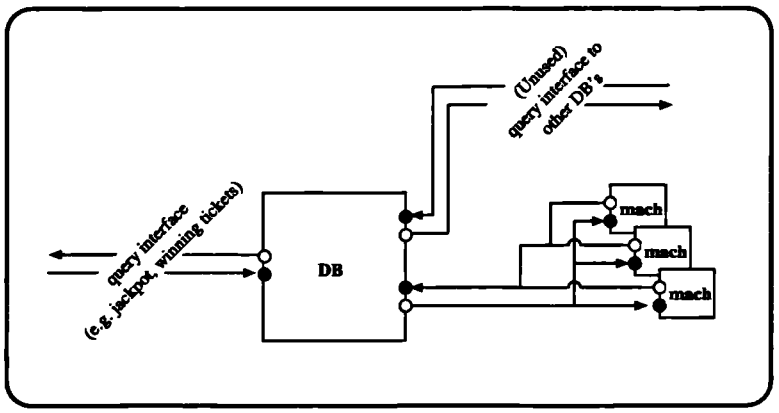


Figure 4.3: Sample database/mach configuration

Of course given that there are thousands of lottery machines nationwide, a pos-

sible way of organising them is to group Mach processes based on their geographical location. In our example this is achieved by their containment in a **Region** component as depicted in figure 4.4.

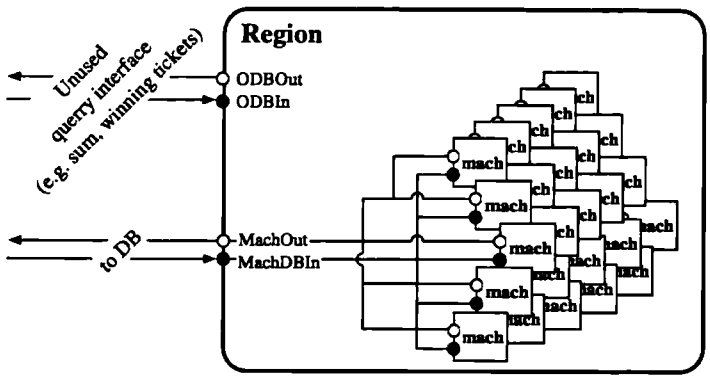


Figure 4.4: Region - grouping component

Thus the overall configuration would be as shown in figure 4.5 where Mach components are grouped in terms of counties, countries and so on.

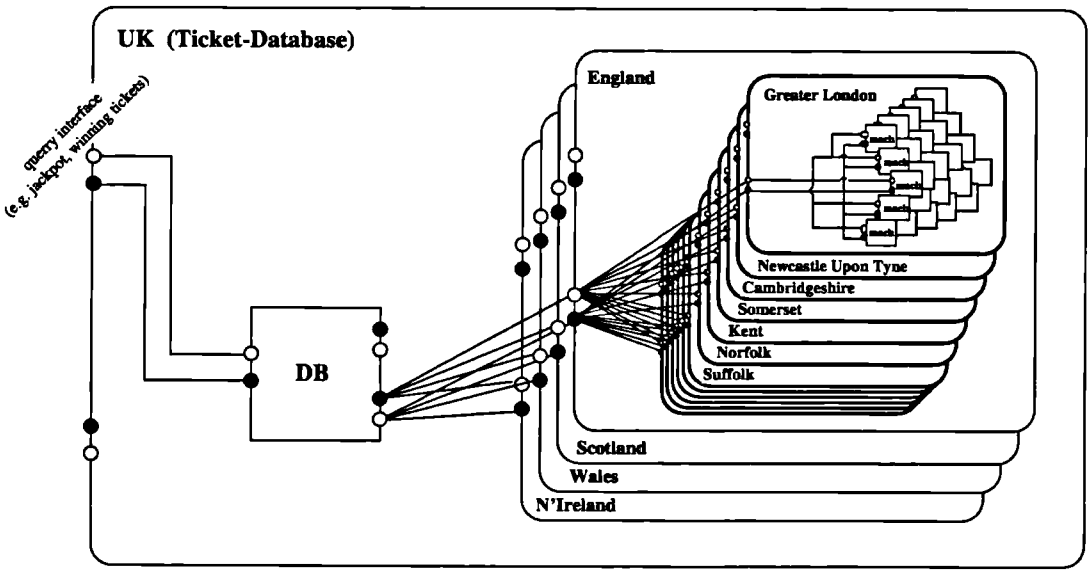


Figure 4.5: ticket-database - detailed configuration

The ticket transaction messages however all end up arriving at one central database component which can be queried by the control-display unit. The DBase and the Region components support communication interfaces which have not been used

in the current configuration. These will be used when we come to consider reconfigurations to the lottery system.

Notice that when building the lottery system, we have made use of composition as a design tool; creating the application by establishing a containment hierarchy of components. Primitive components such as Mach, DBase, RandNumGen and Controller generally form the bottom layer in the hierarchy and are grouped together to form composite components. This allows the system designer to abstract away from the simple tasks carried out by primitive components and think in terms of high-level functions performed by the more complex composite components.

The winning-numbers component (figure 4.2) is a good example of this. First, Selector components were created by composing controller (C) and RandNumGen (RNG) primitive components to create a composite object which combined coordination/synchronisation functionality with random number generation. The winning-numbers components then configured a collection of Selector components to create a higher level component which combines the generation of winning lottery numbers with periodic reseeding and a discretionary selection mechanism.

Modern configuration languages allow system designers to describe composite component types and later use them when describing the initial system configuration. Thus having provided such a description for the **Selector** composite component application designers can create the subsystem depicted in figure 4.6 by simply requesting the creation of a `Selector` component.

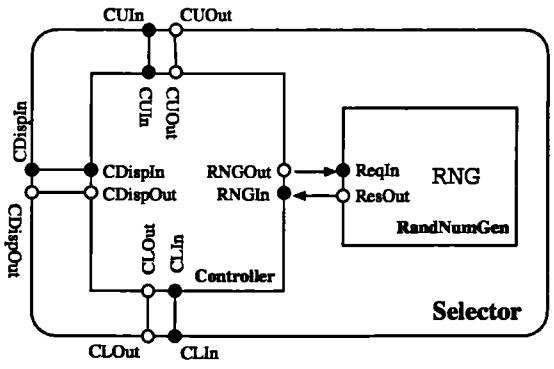


Figure 4.6: Selector composite component

Furthermore, the internal configuration of the WinningNumbers component would be established by creating and binding composite `Selector` components rather than creating and binding individual `Controller` and `RandNumGen` components. An example of a static composite component description is the Selector method in fig-

ure 4.16 (see page 105) described later on in this chapter.

4.4 Lottery reconfigurations

The next section describes a number of useful reconfiguration scenarios for the lottery application. To illustrate the difficulties involved in specifying and bringing about dynamic change, we provide sample reconfiguration scripts for these changes.

To emphasise the importance of hierarchy and the utility of query based access-methods, the sample reconfiguration scripts are based on an example environment which discards the design time composition hierarchy subsequent to system elaboration, and uses a direct naming scheme to specify reconfiguration targets. The reconfiguration environment adopts the *embedded code* approach for preserving application invariants (see section §2.3). It is important to note that since the thesis does not address fault-tolerance issues, the expressed reconfigurations are not resilient to failures and may lead to inconsistencies should failures occur.

The notation used throughout the sample reconfiguration scripts is as follows:

- **Component Creation** is done through the use of the **new** keyword. The example shown below creates a component of type `RandNumGen` with the *unique name* "`RNG1`" at host location `loc`. The parameters following the components unique name are the components runtime instantiation parameters. In this instance only one runtime parameter, `seed` is supplied.

```
new RandNumGen ( @loc , ''RNG1'', seed )
```

- **Bind operations** are done by invoking the bind method of an output port and supplying it with an input port. In the example given below the `bind` method establishes a communication link between the `DBOut` output port of the `ma10005` component and the `MCIn` input port of the `DB` component.

```
ma10005.DBOut.bind ( DB.MCIn )
```

- **unlink operations** follow the syntax of bind operations and are invoked on an output port and are supplied with an input port. The result of the unlink operation is to sever the communication path between the output port and the input port. In the absence of an input port parameter, all previous bindings of the output port are severed.

```
C1.CDispOut.unlink ( )
```

In the example above the `unlink` method is invoked on the `CDispOut` output port of the component `C1`. Since there is no input port parameter the output port is unlinked from *all* its bindings.

- **Code adornments** in the shape of borders around important statements are used to focus the readers attention.

4.4.1 Adding a new Mach component

It is expected that as time goes on the national lottery will increase in popularity and new lottery machines will have to be installed and linked into the existing system. Therefore a basic reconfiguration that the lottery system must support is the addition of new `Mach` components.

```
// create new mach component with
// name ma10005 on host 10005

new Mach( @10005, "ma10005" );

// link into database
ma10005.DBOut.bind ( DB.MCIn ) ;
DB.MCOut.bind ( ma10005.DBIn ) ;
```

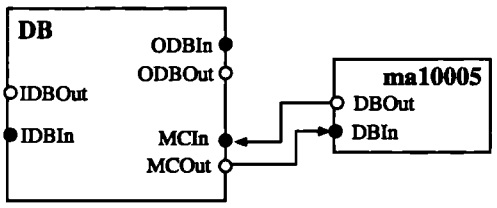


Figure 4.7: newMach script

This reconfiguration requires that a new Mach component is created and bound to the database. In the script, the `new` keyword is used to create a new `Mach` component named *ma10005* on host 10005. Notice that since the design time hierarchy has been collapsed subsequent to system elaboration, no `Region` components exist and therefore the newly created Mach component is bound directly to the database. The bind operation uses the unique name of the component when establishing the communication paths between it and the DB component. This means that should it become necessary to create another Mach component, the script cannot be reused and a new one must be written with the new names.

Also if the DBase component *DB* is replaced, then pre-written script templates such as that above will need to be updated with the new DBase component name. Using *roles* instead of names can make scripts more resilient to change by introducing a level of indirection in the mapping process. Thus the **role** of **DB** would be assigned to a particular DBase component and subsequent references to DB would map to this component. Replacing the actual DBase component will not entail changes to other scripts as long as the role of DB is delegated to the replacement component.

4.4.2 Adding a new Selector

Another reconfiguration which the designers of the system want to consider is the addition of a new Selector to the winning-numbers component. This is to further reduce the possibility of successful attempts at predicting the outcome of the lottery. The reconfiguration script in figure 4.8 will accomplish this task for the given configuration.

```
  // create new Selector component
new Controller (@loc, "C4") ;
new RandNumGen ( @loc, "RNG4", seed) ;

C4.RNGOut.bind ( RNG4.ReqIn ) ;
RNG4.ResOut.bind ( C4.RNGIn ) ;

  // link in to existing chain
C3.CLOut.bind ( C4.CUIn ) ;
C4.CUOut.bind ( C3.CLIn ) ;

  // bind new Selector to CDisp unit
C4.CDispOut.bind ( CDisp.NRes ) ;
CDisp.NReq,bind ( C4.CDIspIn ) ;
```

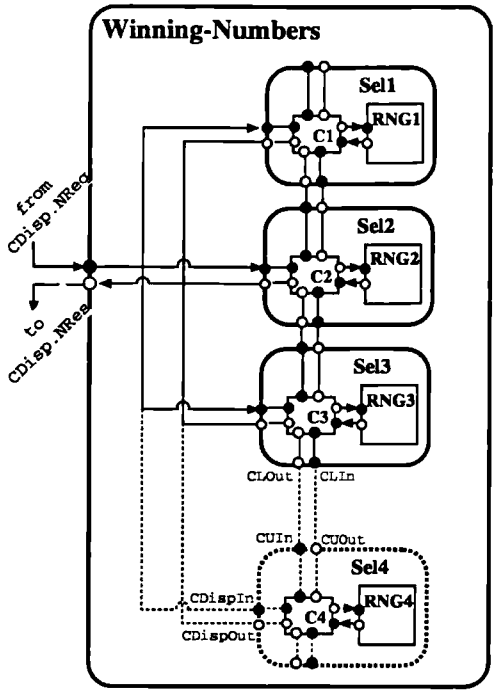


Figure 4.8: newSelector script

However when writing such a reconfiguration script it is important to realise that it is likely to be used more than once during the application's life-cycle. The script in figure 4.8, like the newMach script, names the components in the script directly and thus can only ever be used for the depicted configuration. Clearly, direct naming schemes are inadequate as their use prevents the specification of reusable change scripts. Creating a *role* for the "last-controller" could help matters, as then new Selector component could be created and bound to the controller which fulfils the "last-controller" role (e.g C3). However, the use of role schemes requires careful maintenance. For example if an independent reconfiguration such as removeSelector fails to update the "last-controller" role, subsequent calls to the newSelector script will fail. In effect a role is a primitive means of identifying a component with specific configuration properties. In contrast, access methods which rely on querying the system configuration provide a more general and flexible means of doing this with the added advantage of eliminating the programmer overhead.

4.4.2.1 Re-establishing Application invariants

As we mentioned before, it is crucial that reconfigurations maintain application invariants. The winning-numbers subsystem which the newSelector script reconfigures, has an invariant which specifies that no two Selector components should have the same seed for their random number generator (RNG) component. This means that the new Selector added to the existing chain during reconfiguration must be assigned a unique seed. However since the controllers reseed the number generators periodically it is not possible to know which seeds are currently in use. A possible solution to this problem, favoured by existing reconfiguration systems, is to modify the code for the controller component such that it caters for this specific reconfiguration scenario. Of course these interactions are not made explicit in the script, and in order to learn about them maintainers of the system must either consult the implementation documents or the code itself.

A possible implementation of such a solution might be as follows; a fragment of the extra code is written such that each time reseeding occurs the list of allocated seeds for all random number generators in the subsystem is recorded. This code would run in all the controllers regardless of whether they were undergoing reconfiguration or not. The other code segments are logically divided into the server and client side sections and are activated only during reconfiguration. At link initialisation time the distributed runtime system triggers the client side code in the new controller, C4 in our example, and the server side code in the existing controller, C3. The server side code in C3 would use an existing communication link or a special purpose one to transmit the current list of recorded seeds to the client side code in C4. This information is then used to choose a unique seed for the random number generator of the new Selector (RNG4).

In general the complexity of additional code, such as that above, increases with the complexity of the subsystem invariant it helps preserve, and becomes less and less orthogonal to the operation of the component in which it is embedded. Consider that if a further subsystem invariant stipulated that Controllers had to be assigned unique identifiers which increased in strict order, then the removal of an intermediate Selector component (such as Sel2) would require additional code in the link initialisation/finalisation sections of the controller to reallocate the identifiers along the Selector chain.

Also, since by their nature reconfigurations cannot be entirely predicted, embedded solutions will fail to cater for evolutionary reconfiguration which become

necessary during an applications life-cycle.[3]

4.4.3 Distributed/Partitioned Database configuration

The designers of the lottery system are concerned that if the popularity of the national lottery increases beyond expected levels, the initial configuration which uses a single database to handle all Mach requests, might prove too slow.

In the previous scheme (figure 4.5) a steady increase in the number of mach processes could mean that the top-level database would be unable to deal with the large volume of incoming transmissions. This would result in delays when issuing tickets and reduce overall sales. Also servicing queries over an already large and growing data-set would eventually become too much for a single database and lead to excessive delays.

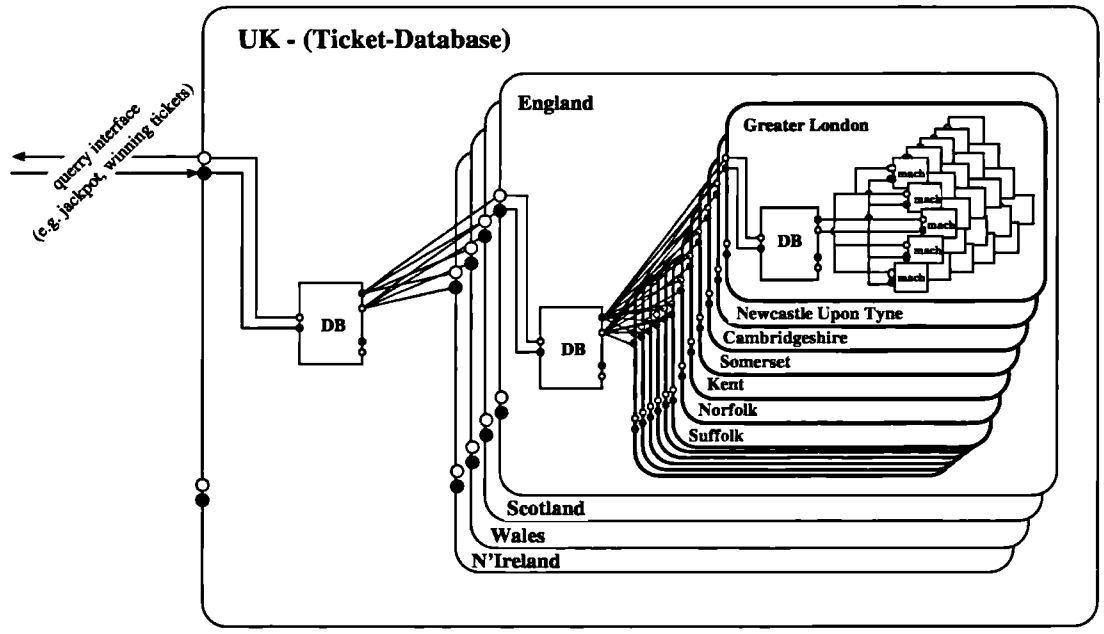


Figure 4.9: Distributed Database

To cater for this possibility, it is intended that a reconfiguration path should be provided which will results in the configuration depicted in figure 4.9. Under the new scheme mach processes interact with a regional database component which is exclusive to their county. In step with the composition hierarchy the county-wide

[3]Section §2.3.1.1 (page 54) details further disadvantages of this approach.

databases report to a country-wide database and in turn the country-wide databases are linked to a single nationwide database.

Partitioning ticket data across county-wide databases is advantageous as even large increases in nationwide ticket sales will be a result of comparatively modest increases at the county level. The regional databases are able to tolerate such increases with ease allowing ticket sales to go ahead unabated. Furthermore, since the calculation of the actual winning tickets can be done simultaneously for the different regions, the queries will be serviced more quickly. The hierarchic organisation of the databases is also leveraged by filtering the data which is transmitted up the hierarchy. Thus the country-level databases would contain data of a more high-level nature to those stored in the county-level databases (e.g tables storing information such as ticket buying patterns, frequency of weekly tickets etc.).

With this change in mind, the database component (figure 4.10) has been written such that it can operate in both configurations. The outgoing database query interface (ODB) will be used to propagate queries which cannot be serviced locally. The results of these when they arrive, are collated and sent on to the query originator via the incoming database query interface (IDB). The mach component interface (MC) functions as described before - receiving mach transmissions and updating ticket data.

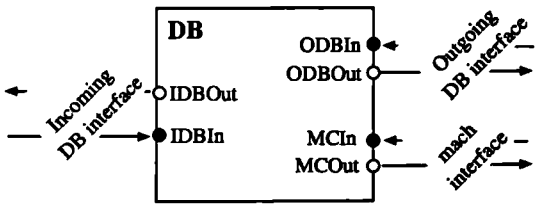


Figure 4.10: Database Component

However as we mentioned before, the example reconfiguration environment discards the design time composition hierarchy of the application once the initial system configuration has been established. This poses serious problems for the change programmer particularly with respect to this specific reconfiguration. The semantics of the new configuration demands that mach components which are in the same county be linked to the same regional database. This is so that data propagated to the higher-level databases can be interpreted correctly. Unfortunately, in the absence of the design time composition hierarchy the change programmer is unable

to identifying which mach components belong to which county. To appreciate this problem consider figure 4.11 which depicts the actual lottery system configuration subsequent to system elaboration.

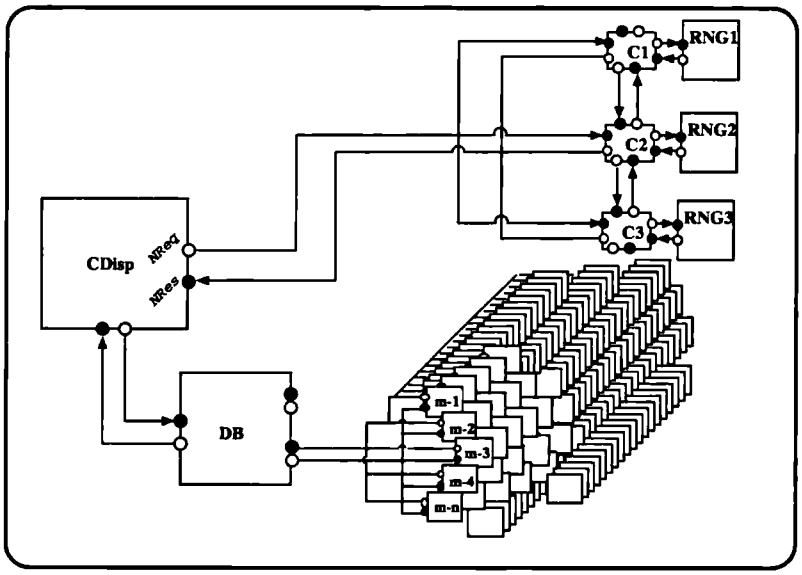


Figure 4.11: Actual system configuration

A possible solution would be to record the name and location of the mach components and name them explicitly in the script. However such a script is not resilient to change and needs to be altered if for example a new mach component is added to the system. Also, since the maintenance of the mach name-directory is done manually it has the potential of becoming out of step with the actual system configuration. In addition the reconfiguration script for adding a new Mach component (see figure 4.7) needs to be modified or at best two different versions provided.

As figure 4.11 illustrated, in the absence of the logical grouping information provided by composite components, the specification of even simple changes can require change programmers to work their way through multitudes of unrelated components and their associated web of interconnections. This makes reasoning about even moderately sized distributed systems extremely difficult and renders certain reconfiguration such as that above virtually impossible. Using a roles scheme instead of unique naming would not solve any of the problems here either as essentially there needs to be a more sophisticated mechanism for maintaining and querying information about the logical grouping and configuration of components within the distributed system cf. the CDB.

4.4.4 Evolutionary change

In chapter one we distinguished between two types of dynamic change, namely programmed and evolutionary change. The distinction was that programmed changes could be foreseen and catered for by the system architect at system design time, while evolutionary changes could not be predicted and had to be dealt with as and when the need for them arose.

Unfortunately, the support mechanisms favoured by existing reconfiguration environments are not suited to evolutionary change. The embedded approach to preserving application invariants operates by adding extra code to the executable image of running computational components. Since the individual requirements of evolutionary reconfigurations cannot be predicted in advance, the necessary extra code will need to be written and added to the component as it is executing.[4] While most modern runtime systems provide a means for loading code dynamically, taking this route is fundamentally flawed. To preserve an application invariants which is going to be affected by an evolutionary reconfiguration, the change designers must perforce understand the inner-workings of all the computational components governed by the invariant, before writing the extra code to be embedded in each of them. In doing so it will also be necessary to understand all the additional code segments which have been added for the sake of previous programmed and evolutionary reconfigurations. This is necessary, as the additional code segments must not interfere with the workings of these other code snipets. Such a process does not scale and will in the long run lead to errors.

Another issue is that although evolutionary changes are often slight variations or amalgamations of already programmed reconfigurations, the evolutionary change designer is rarely able to reuse them. This is due to the fact that the majority of existing reconfiguration environments lack the necessary support structures which would grant access to programmed reconfiguration scripts. The format of programmed reconfigurations also becomes an issue when direct naming of components in reconfigurations lead to disposable scripts. To illustrate this latter problem consider the evolutionary reconfiguration described below.

4.4.4.1 Add Selector and NumberMonitor

Following concern over the predictability of the winning numbers generated, it has been decided to increase the number of Selector components yet again and also

[4]This is reconfiguration at the procedural level and should not be confused with the addition of a completely new computational component to a parallel or distributed system

to monitor the output of the winning-numbers component in order to see whether it is possible to find patterns in the way winning numbers are generated. To do this, in addition to increasing the number of Selectors, a new NumberMonitor component will need to be inserted between the control-display and winning-numbers components.

To accomplish this task the reconfiguration script of the newSelector needs to be duplicated such that a new fifth Selector is created and added after the existing fourth Selector. The links between the control-display and winning-numbers will also need to be severed and re-established through the number-monitor component. The new configuration is shown in figure 4.12.

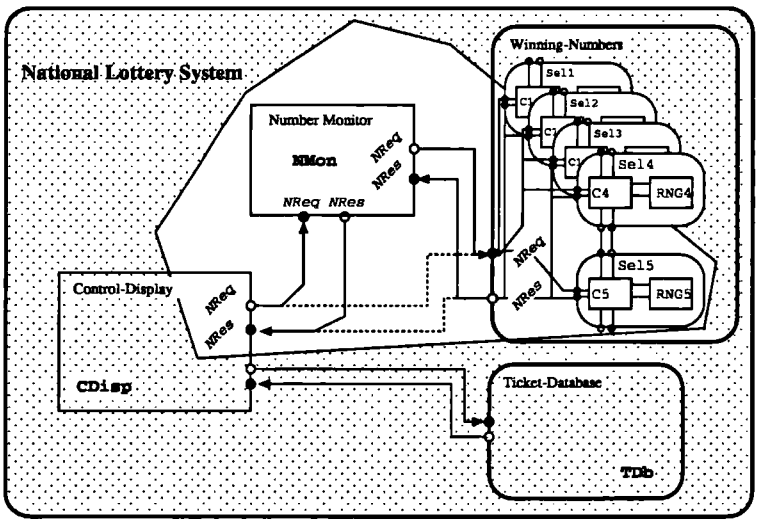


Figure 4.12: Number monitor and new Selector configuration

The evolutionary reconfiguration script which accomplishes this change is given in figure 4.13. As can be seen, even though the newSelector reconfiguration was planned and coded at system design time, it could not be reused in the evolutionary change. The relevant sections have therefore been copied here and modified to bring about the addition of a fifth Selector. As with the newSelector script the actions which preserve the subsystem invariant are not made explicit in the reconfiguration script but lie hidden in the application code.

The disposal of the composition hierarchy has also meant that the benefits of encapsulation and information hiding which helped reduce the complexity of designing large distributed applications has been lost. The change programmer is reduced to thinking and specifying change in terms of primitive component creations and

bindings.

```
  // create new Selector component
new Controller (@loc, "C5") ;
new RandNumGen (@loc, "RNG5", seed) ;
C5.RNGOut.bind ( RNG5.ReqIn ) ;
RNG5.ResOut.bind ( C5.RNGIn ) ;
  // link in to existing chain
C4.CLOut.bind ( C5.CUIn ) ;
C5.CUOut.bind ( C4.CLIn ) ;
  // break old connections
CDisp.Nreq.unlink ( ) ;
C1.CDispOut.unlink ( ) ;
    ... same for C2,C3 & C4 ...
  // create NumberMonitor component
new NumberMonitor (@loc, "NMon") ;
  // bind CDisp and NMon together
CDisp.NReq.bind ( NMon.DispReqIn ) ;
NMon.DispNumsOut.bind ( CDisp.NRes ) ;
  // bind NMon and WinningNumbers together
NMon.NumsReqOut.bind ( C1.CDispIn ) ;
    ... same for C2, C3 & C4 ...
NMon.NumsReqOut.bind ( C5.CDispIn ) ;
C1.CDispOut.bind ( NMon.NumsResIn ) ;
    ... same for C2, C3 & C4 ...
C5.CDispOut.bind ( NMon.NumsResIn ) ;
```

Figure 4.13: NumberMonitor & new Selector script

An issue which was side-stepped in the presented script descriptions has been the way in which change designers gain access to component names. Throughout the examples, we have used direct naming, which relies on change designers using the same labels in the scripts as those used by the runtime system to identify running components. Thus the label `C4` used in the newSelector script (see figure 4.8 is also the identifying tag used by the underlying runtime system for that component. However given that components are being added and removed to the system dynamically, change designers must somehow consult the current system configuration and then select the labels which match the components they need to reconfigure. This was evident in the newSelector script which required the change designer to identify the last existing Selector in the chain so that the new Selector could be linked to it. Although feasible for small system this process is quite laborious and becomes

difficult when dealing with large systems (e.g the ticket-database).

4.5 Lottery reconfigurations (revisited)

The previous section illustrated some of the problems of expressing dynamic change using existing reconfiguration solutions. We saw that direct naming and role based access schemes lead to reconfiguration scripts which could not be reused easily. Additionally the loss of the design time hierarchy meant that the change designer had to think and express reconfigurations in terms of the flat structure of the system. Change scripts could not be called from within each other and certain reconfigurations which relied on the logical grouping of the components in the design time hierarchy could not be expressed. The lack of a general purpose way of reinstating application invariants meant that all the necessary actions had to be embedded inside the applications code. This was unsatisfactory as the impact of these actions on other planned reconfigurations and future unplanned reconfiguration was not made explicit. Additionally these problems combined to make the expression of evolutionary changes particularly difficult in such environments.

In the following sections we will express the same lottery reconfigurations but this time expressed in the style advocated by the *reconfiguration programming paradigm*. This will help illustrate the way in which the reconfiguration programming paradigm overcomes the shortcomings highlighted in the previous sections. Note that in the reconfiguration programming paradigm programmed changes are expressed as reconfiguration methods of composite components (see section §4.2.3.2 on page 83). Thus the addition of a new Mach component, section §4.4.1, and the distributed database reconfiguration, section §4.4.3, are realised as a reconfiguration methods of the Region component in section §4.5.5.1 and section §4.5.5.2 respectively. The addition of a new Selector described in section §4.4.2 is expressed as a reconfiguration method of the Selector component in section §4.5.4.1

4.5.1 RandNumGen

Figure 4.14 shows the reconfiguration specification of the RandNumGen primitive component type. Notice that in addition to its constructor signature and communication interface which are fixtures of configuration descriptions, the specification also includes a number of state query and update routines which have been exposed at the configuration language level. These methods have been implemented by the component programmer as a means of facilitating dynamic reconfiguration and can

be invoked externally through the state adjustment interface alluded to in chapter three.

The declaration of the RandNumGen reconfiguration specification class also indicates that it **extends** a superclass named `PNode`.[5] We will elaborate on the advantages of the reconfiguration class hierarchy in section §4.6.

```
class RandNumGen
      extends PNode
{
    // communication interface
  PPort ReqIn ;
  PPortref ResOut ;
    // constructor
  RandNumGen ( int loc, int seed )
    // exported state access-methods
  int getSeed ( )
  void setSeed ( int seed )
}
```

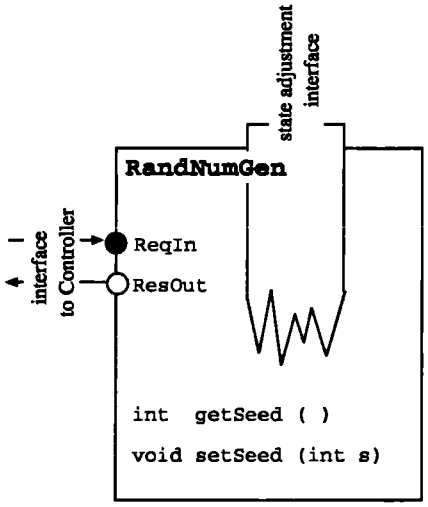


Figure 4.14: RandNumGen reconfiguration specification

4.5.2 Controller

The reconfiguration specification for the controller component is given in figure 4.15 and shows that `Controller` also extends `PNode`. In addition the reconfiguration specification exposes access-methods which allow the reseeding period to be queried and altered. As it happens these methods are not necessary for the planned lottery reconfigurations, however they have been provided by the component programmer in order to facilitate evolutionary reconfigurations. For example a system which uses the Controller component might have an invariant which links the reseeding period to the length of the `Controller/RandNumGen` chain. In such a system the addition or removal of a `Selector` component would require changes to the reseed period of the controllers in order to preserve the application invariant.

[5]We have adopted Java syntax - the **extends** keyword implies inheritance

```
class Controller
        extends PNode
{
    // communication interface
  PPort CUIn, CLIn, CDispIn, RNGIn ;
  PPortref CUOut, CLOut, CDispOut,
            RNGOut ;
    // constructor
  Controller ( int loc, int reseedPrd )
    // exported state query/update methods
  int getReseedPrd ( )
  void setReseedPrd ( int Prd )
}
```

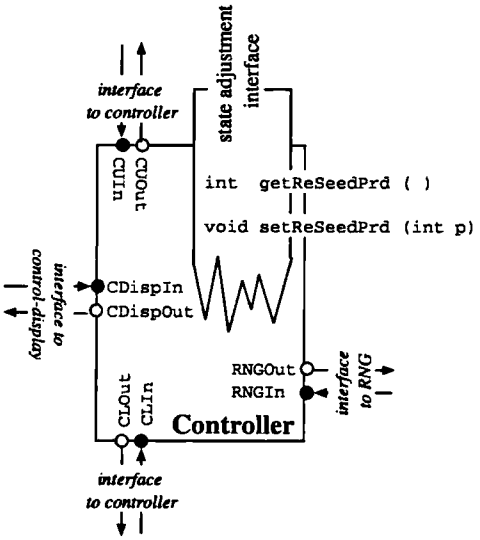


Figure 4.15: Controller reconfiguration specification

4.5.3 Selector

The reconfiguration specification of the `RandNumGen` and `Controller` primitive components captured a one-to-one relationship between the reconfiguration representation and the actual physical component implementation. In the case of composite components however, this is often a one-to-many relationship, where the logical behaviour of a composite is provided by more than one physical component implementation. Thus, as in the traditional configuration programming paradigm, the `Selector` configuration specification in figure 4.17 includes the definition of a constructor method bearing the same name, which gives an imperative description of the create and bind actions which establish the configuration of the composite's subordinate components. It also details the mapping between the composite's exported communication interface and that of the actual underlying communication end-points of the enclosed components.

The reconfiguration description of the Selector composite component is given in figure 4.16. Taking a closer look at the Selector constructor method we see that it is parameterised with three integer parameters. The first parameter, `loc`, indicates the host location where the component is to be created. The `seedPrd` and `seed` parameters are used to instantiate the `Selector`'s controller and random number generator subcomponents, `c` and `rng`, with the correct parameters prior to binding them together. The communication interface of the composite component is

embodied by the `CUIn`, `CDispIn`, `CLIn`, `CUOut`, `CDispOut` and `CLOut` communication objects. These virtual composite communication end-points are grounded in the constructor by binding them to identically named actual ones on the controller component `c`.

```
public class Selector extends CNode
{
      // communication interface
  CPort CUIn, CDispIn, CLIn ;
  CPortref CUOut, CDispOut, CLOut ;
  public Selector ( int loc, int seedPrd, int seed )
    {
       // instantiate sub-components
      Controller c = new Controller ( loc, seedPrd ) ;
      RandNumGen rng = new RandNumGen ( loc , seed ) ;
       // bind subcomponents
      c.RNGOut.bind ( rng.ReqIn ) ; rng.ResOut.bind ( c.RNGIn ) ;
       // establish composite communication interface
      CDispIn.bind ( c.CDispIn ) ;    CDispOut.bind ( c.CDispOut ) ;
      CUIn.bind ( c.CUIn ) ;          CUOut.bind ( c.CUOut ) ;
      CLIn.bind ( c.CLIn ) ;          CLOut.bind ( c.CLOut ) ;
    }
      // composite state access-methods
  public int getSeed ( )
  public void setSeed ( int seed )
  public int getReseedPrd ( ) { ... }
  public void setReseedPrd ( int prd ) { ... }
}
```

Figure 4.16: Selector reconfiguration specification

In the configuration programming paradigm the configuration representation of the Selector component would have ended with the definition of its constructor. In the reconfiguration programming paradigm however a composite component exists for as long as the application, and encapsulates reconfiguration behaviour as well as configuration properties. Thus the configuration specification of the Selector component goes on to define a number of composite state access-methods which can be used to control the external behaviour of the composite. Providing these methods at the composite level is extremely important as then the composite component can be treated as a black box which has specific behaviour and whose state can be queried and adjusted without the need to delve into its internal configuration. This

"black-box" view of the composite component is depicted in figure 4.17.

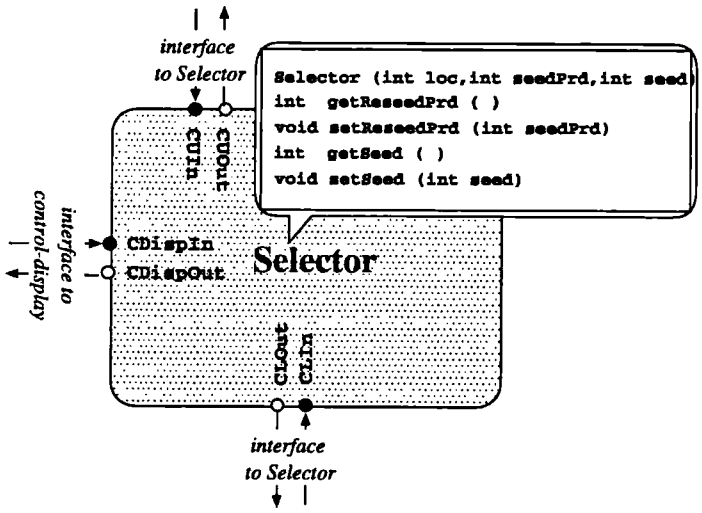


Figure 4.17: Selector composite component

The state access-methods implemented by the Selector component allow the change designer to get and set the seed and reseed period of the Selector. Of course the invocation of these composite state methods will need to be channelled to the actual component(s) which implement that aspect of the composite's behaviour. To understand how this is done let us take a closer look at the implementation of the `getSeed` and `setSeed` methods depicted in figures 4.18 and 4.19.

```
public int getSeed ( )
 {
   Enumeration rngs ;
   rngs =
     ClassInstances("RandNumGen");
   RandNumGen rng ;
   rng = rngs.nextElement () ;
   return rng.getSeed () ;
 }
```

Figure 4.18: Selector.getSeed

```
public void setSeed ( int seed )
 {
   Enumeration rngs ;
   rng =
     ClassInstances("RandNumGen");
   RandNumGen rng ;
   rng = rngs.nextElement () ;
   rng.setSeed ( seed ) ;
 }
```

Figure 4.19: Selector.setSeed

Recall that the logical behaviour of the Selector composite component is provided by the RandNumGen and Controller primitive components it encapsulates. The RandNumGen component (figure 4.14) exports a getSeed state query method which returns the generators current seed. The `Selector.getSeed` method therefore needs to invoke the getSeed method of the RandNumGen component. To do this however

it must first gain a handle to that component.

This can be done through one of the exported CDB access-methods which are exposed to the change programmer from within the reconfiguration language. The method in question is the `ClassInstances` methods which given a component type, returns a list of all instances of that component type.[6] The method operates in the scope of the composite component in which it is invoked, thus only instances of the matching type which are enclosed by the composite are returned.

Since the Selector component contains only one instance of the RandNumGen component, the getSeed method selects the first (and only) element in the list returned by the call to ClassInstances, invokes its `getSeed` method and returns the result. The process by which methods of primitive components are invoked from within the reconfiguration language and the parameters and results marshaled, are implemented by the reconfiguration manager (RM) and the management interface of the primitive components themselves (see figure 3.1 on page 66). The implementation of the `getReseedPrd` and `setReseedPrd` follow the same pattern as that of `getSeed` and `setSeed`, only they invoke methods of the controller primitive component instead.

Although unnecessary in the case of the relatively simple Selector component, composite state adjustment and query methods can be defined that are more than just mirrors of those exported by child components (see the `getUniqueSeedAndPrd` method of the WinningNumbers composite component in figure 4.22).

A final point to notice is that unlike RandNumGen and Controller which extended the PNode reconfiguration class, Selector extends `CNode`. CNode is the superclass of all composite components and supplies default reconfiguration behaviour for composite components.

4.5.4 WinningNumbers

One of the intended reconfigurations for the lottery system was the addition of a Selector to the existing chain in the WinningNumbers composite component. As figure 4.20 shows, in the reconfiguration programming paradigm this reconfiguration would be a part of the WinningNumbers' reconfiguration specification.

[6] The *Gandiva* toolkit [Lit96, LW98] provides similar functionality for Java applications.

```
public class WinningNumbers extends CNode
{
  CPort CDispIn ;
  CPortref CDispOut ;
  public WinningNumbers ( int loc, int numSlctrs, int seedPrd ) { ... }
  public void newSelector ( int loc ) { ... }
  void getUniqueSeedAndPrd ( int & seed, int & seedPrd ) { ... }
}
```

Figure 4.20: WinningNumbers reconfiguration specification

4.5.4.1 Adding a new Selector

The code for the newSelector reconfiguration method is given in figure 4.21. It operates by creating and linking the new Selector to the bottom most Selector component in the existing chain. The reconfiguration script identifies the bottom most Selector by its unique configuration properties - namely that it is a Selector component and that its lower control interface, CLOut, is unbound. Notice that since the CDB maintains the composition hierarchy, the change designer does not need to think in terms of the flat configuration of Controller and RandNumGen components but can instead write change scripts in terms of the creation and binding of composite components. This is evident in the newSelector reconfiguration script where the call to the `ClassInstances` method requests the list of Selector components, and the subsequent call to the `isBound`[7] method within the `while` loop queries a composite's communication interface.

Having identified the bottom most Selector component, the script goes on to create a new Selector component and link it into the existing chain of Selectors in the WinningNumbers component. However remember that an invariant of the WinningNumbers component stipulated that no two Selector components should have the same seed. The difficulty in selecting a unique seed for the new addition was that the Selector components periodically reseeded themselves. The newSelector reconfiguration script overcomes this problem by invoking the `getUniqueSeedAndPrd` method to supply it with a unique seed for the new Selector and also to get the reseed period.

[7]This is an exported CDB method which returns true if the communication interface is bound, false otherwise.

```
public void newSelector ( int loc )
{
  Enumeration slctrs = ClassInstances ( "Selector") ;
  Selector sel, tmp ;
  int seedPrd, seed ;
    // find bottom most in ring - i.e. one whose CLOut isn't bound
  while ( slctrs.hasMoreElements () )
    {
    tmp = slctrs.nextElement () ;
    if ( ! tmp.CLOut.isBound () )
      {
      sel = tmp ;
      break ;
      }
    }
  getUniqueSeedAndPrd ( seed, seedPrd ) ;
    // create new Selector with seed and period at location loc
  Selector newSel = new Selector ( loc, seedPrd, seed ) ;
    // link newSel into Selector chain
  newSel.CUOut.bind ( sel.CLIn ) ;
  sel.CLOut.bind ( newSel.CUIn ) ;
    // bind newSel to WinningNumbers interface
  CDispOut.bind ( newSel.CDispOut ) ;
  CDispIn.bind ( newSel.CDispIn ) ;
}
```

Figure 4.21: newSelector reconfiguration script

4.5.4.2 Re-establishing Application invariants

getUniqueSeedAndPrd - A composite state access-method The getUniqueSeedAndPrd is a good example of a composite state query method which in this case was used by an internal reconfiguration method. Composite state access-methods are necessary in order to hide the internal complexities of composites and to facilitate a higher-level interface to the combined component state.

The definition of the getUniqueSeedAndPrd is given in figure 4.22. The unique seed value is decided upon by invoking the getSeed method of all Selector components in WinningNumbers and choosing a seed value which is bigger than all seed values currently in use.[8] The seedPrd is the same for all Selector components so any one of the Selectors can be queried via the getReseedPrd to supply the current value.

[8]This strategy is not foolproof and was chosen for its simplicity.

```
void getUniqueSeedAndPrd ( int & seed, int & seedPrd )
{
  // select appropriate seed by picking a number bigger than all
  // the other seeds

  Enumeration slctrs = ClassInstances ("Selector") ;
  int sVal = 0 ;
  Selector slctr = null ;
  while ( slctrs.hasMoreElements () )
    {
    slctr = slctrs.nextElement () ;
    sVal = slctr.getSeed () ;
    if ( seed <= sVal )
      seed = sVal + 1 ;
    }

    // get period - same for all so query last one
  seedPrd = slctr.getReseedPrd () ;
}
```

Figure 4.22: getUniqueSeedAndPrd method definition

The newSelector reconfiguration description given in figure 4.21 has a number of advantages over that of section §4.4.2. Firstly by using the state access-methods exported by the primitive components the script preserves the subsystem invariant without embedding extraneous code into the primitive components. Furthermore the necessary interactions are controlled from the reconfiguration language level and are thus made explicit in the script. Also since the script does not name any components directly but instead relies on the configuration properties to target components, it can be used again and again without the need for any modifications i.e. the same newSelector script can be invoked any number of times to add Selector components to the chain. Finally, thanks to the stipulation that reconfigurations defined at each level in the hierarchy respect the boundaries of their subordinate components, and the requirement that internal reconfigurations maintain the exported component interface and preserve subsystem invariants, the integrity of these scripts will not be affected by reconfigurations at different levels in the composition hierarchy.

4.5.5 Region component (Tickets Database)

The ticket database is perhaps the most complicated component in the lottery system and reconfigurations aside, even expressing the initial component description (constructor) would prove difficult in many existing configuration languages.

The reconfigurations which we considered for this section of the lottery system were the additions of new Mach processes and also the reorganisation of the subsystem such that the management of the ticket information would map from a centralised scheme to a distributed one.

```
public class Region extends CNode
{
  final String name ;
  CPort ODBIn, MachDBIn ;
  CPortref ODBOut, MachOut ;
  public Region ( RegionRec prms ) { ...}
  public void newMach ( int loc ) { ...}
  public void distributeDBs ( ) { ...}
}
```

Figure 4.23: Region reconfiguration skeleton

The skeleton of the Region composite component's reconfiguration specification is given in figure 4.23. It consists of the Region component's constructor, exported communication interface and two reconfigurations methods, `newMach` and `distributeDBs`.

Before we go on to describe these reconfiguration methods, recall that a number of Region components in the lottery application are used in identical configurations and thus cannot be distinguished from another based on their configuration properties alone. To this end the Region reconfiguration specification also includes a string variable, `name`, which is initialised at component instantiation time with the label of the Region (e.g England, Scotland or Surrey). This variable is used to augment the configuration information and make such components uniquely identifiable. The CDB makes provisions for such situations by allowing certain intransient values to be included as part of a components configuration specification (see §3.3 page 69).

The `newMach` and `distributeDBs` reconfiguration methods of the Region component are described next. They do not make use of the `name` variable which was included for completeness.

4.5.5.1 Adding a new Mach component

Recall that the newMach script needed to work in both the centralised and distributed/partitioned database configurations. The definition of the newMach reconfiguration is given in figure 4.24. It works in both configurations of the Region component because it first searches for a local DBase component to bind to (the distributed case), and if that fails the new Mach process is bound to the centralised DBase via its Region's communication interface.

```
public void newMach ( int loc )
  {
    // check for DBase component and bind to it.
    // Otherwise assume centralised configuration
  Enumeration DBs = ClassInstances ("DBase") ;
  DBase DB = (DBase) DBs.nextElement () ;
  Mach mach = new Mach ( loc, "ma"+loc ) ;
  if ( DB != null )
    {
    mach.DBOut.bind ( DB.MCIn ) ;
    DB.MCOut.bind ( mach.DBIn ) ;
    }
  else
    {
    MachDBIn.bind ( mach.DBIn ) ;
    MachOut.bind ( mach.DBOut ) ;
    }
  }
```

Figure 4.24: newMach method definition

4.5.5.2 Distributed/Partitioned Database configuration

The definition of the distributeDBs reconfiguration method is given in figure 4.25. Despite its length it is a relatively simple script which considers two different internal configurations for the Region component. Either it is dealing with a Region component which contains other Region components such as the top-level Ticket-database component or the country components such as England, or it is dealing with a county component which only contains Mach processes (see figure 4.5).

In both these cases it is necessary to create a new local DBase component and bind its ODB interface to that of the region components. The only exception to this rule is the top-level Ticket-database which already has a DBase component. This

```
public void distributeDBs ( )
{
    // check for existing DB. otherwise create one and bind to
    // region interface
  Enumeration DBs = ClassInstances ("DBase") ;
  DBase DB = (DBase) DBs.nextElement () ;
  if ( DB == null )
    {
    DB = new DBase ( loc () ) ;
    ODBIn.bind ( DB.IDBIn ) ;
    ODBOut.bind ( DB.IDBOut ) ;
    }
  else
    DB.MCOut.unlink ( ) ;
    // for each Region component bind it to DB, invoke their
    // distributeDBs reconfiguration method + unlink old binding.
  Enumeration regions = ClassInstances ("Region") ;
  Region region = null ;
  while ( regions.hasMoreElements () )
    {
    region = regions.nextElement () ;
      // call distributeDBs on sub-region component
    region.distributeDBs ( ) ;
    DB.ODBOut.bind ( region.ODBIn ) ;
    region.ODBOut.bind ( DB.ODBIn ) ;
    MachOut.unlink ( region.MachOut ) ;
    MachDBIn.unlink ( region.MachDBIn )
    }
    // Bind each Mach component to the database. The old binding
    // between Mach components and the region.Mach interface will be
    // replaced
  Enumeration machs = ClassInstances ("Mach") ;
  Mach mach = null ;
  while ( machs.hasMoreElements () )
    {
    mach = machs.nextElement () ;
    MachOut.unlink ( mach.DBOut ) ;
    mach.DBOut.bind ( DB.MCIn ) ;
    MachDBIn.unlink ( mach.DBIn ) ;
    DB.MCOut.bind ( mach.DBIn ) ;
    }
}
```

Figure 4.25: distributeDBs method definition

task is accomplished by the first code block which ends with the first `if` statement. In it the CDB is queried for a DBase component and if none exist then a new one is created and linked in. If one does exists however the DB's Mach interface is unlinked.

The next block of code deals with Region components which contain other Region components. All that needs to be done in this case is to invoke the distributeDBs reconfiguration method of all sub-regions, link their ODB interface to that of the DBase's, and unlink[9] the Mach interface used in the previous configuration.

The final code segment deals with the county Regions each of which, at this stage, contains a DBase component created earlier, and the Mach processes. The action taken in this case is to iterate over the Mach processes and bind them to the newly created database.

Thus, when the distributeDBs reconfiguration method is invoked on the top-level UK ticket-database component, it changes the system configuration from that given in figure 4.5 to that in figure 4.9. However, since the change script is a part of the Region component's configuration description and is written in a general way, it can be used by a range of reconfigurations and not just for the original change which prompted it to be written.

This combination of composite state access-methods with reconfiguration behaviour promotes encapsulation and code reuse such that composite components become more than just convenient ways of describing complex structures. Composite components become descriptions of reusable subsystems which have predefined behaviour in the spheres of computation and reconfiguration.

4.5.6 Evolutionary change

In the reconfiguration programming paradigm, support for evolutionary change is on par with that provided for programmed changes. Since the composition hierarchy is maintained by the CDB throughout the application life-cycle, evolutionary change designers can continue to think about and express changes in those terms. The CDB will handle the mapping between this virtual view of the system and the actual flat elaboration of the running distributed application transparently.

The inclusion of state access-methods in the configuration specification of primitive and composite components grants the evolutionary change designer easy access to system state. This allows the change designer to contemplate evolutionary changes

[9]The unlink action succeeds when invoked on an unbound communication end-point

which have an impact on application invariants without the need to delve into the inner workings and interactions of primitive components.

The incorporation of reconfiguration methods in the configuration specification of composite components allows evolutionary changes to treat composite components as black boxes which encapsulate the state and reconfiguration behaviour of their child components. Thus, wherever possible, evolutionary changes affect changes to the internal configuration of composite components through exported reconfiguration methods. To illustrate these points consider the number monitor reconfiguration of section §4.4.4.1 but as programmed in the reconfiguration paradigm.

4.5.6.1 Add Selector and NumberMonitor

```
public class NumberMonSel implements EvolutionaryChange
{
  public void execute ( CNode cn ) throws ReconfigurationException,
    {
      // get a handle to the Winning numbers, and Control-display
      Enumeration WNs = cn.ClassInstances ("WinningNumbers") ;
      Enumeration CDisps = cn.ClassInstances ("ControlDisplay") ;

      WinningNumbers WNums = WNs.nextElement () ;
      ControlDisplay CDisp = CDisps.nextElement () ;
        // create a NumberMonitor
      NumberMonitor NMon = new NumberMonitor ( WNums.loc() );
        // unlink CDisp and WNums
      CDisp.NReq.unlink ( WNums.CDispIn ) ;
      WNums.CDispOut.unlink ( CDisp.NRes ) ;
        // insert NMon between CDisp and WNums
      NMon.NumsReqOut.bind ( WNums.CDispIn ) ;
      WNums.CDispOut.bind ( NMon.NumsResIn ) ;

      CDisp.NReq.bind ( NMon.DispReqIn ) ;
      NMon.DispNumsOut.bind ( CDisp.NRes ) ;
        // invoke newSelector reconfiguration method of WNums
      WNums.newSelector ( WNums.loc () ) ;
    }
}
```

Figure 4.26: NumberMonitor & new Selector script

Figure 4.26 depicts the evolutionary change script which increases the number of Selector components by one and inserts a NumberMonitor component in between the WinningNumbers and ControlDisplay composite components. The evolution-

ary reconfiguration script is contained in the body of the `execute` method which is invoked with a handle to the top-level lottery composite component as its parameter `cn`.

As with programmed changes, the first step is to gain handles to the reconfiguration targets by querying the system configuration. In this case these consist of the WinningNumbers and ControlDisplay instances WNums and CDisp. Next, an instance of the NumberMonitor component NMon is created and after unlinking existing bindings, it is inserted in between CDisp and WNums. The final step is to invoke the newSelector reconfiguration method of the WinningNumbers component WNums.

Comparing this reconfiguration script with that in figure 4.13 it is clear that expressing reconfigurations in terms of composite components which group logically related components together, makes for more concise and understandable scripts. In addition access to programmed reconfiguration promotes code reuse and reinforces the encapsulation and data-hiding principals which have been proven to be essential software engineering tools.

4.6 Benefits of Object Orientation

One of the distinguishing features of object oriented languages is their support for inheritance. In the context of this thesis, inheritance allows change designers to establish another level of grouping which is in addition to that created by the composition hierarchy. This feature facilitates the expression of generic configuration and reconfiguration descriptions.

Consider that as things stand, configuration descriptions establish component topologies for specific component types. For example, a configuration description which establishes a chain of `Server` components cannot be used to create a chain of `Filter` components. Typing rules prevent this, reasoning that configuration descriptions written for `Server` objects rely on characteristics which are not necessarily present in instances of the `Filter` class. However despite any functional differences which `Filter` and `Server` objects may have it is possible for both to possess the requisite communication end-points necessary for participating in a chain topology. Inheritance allows such commonalities to be made explicit and thereby facilitates the expression of generic configuration descriptions.

Figure 4.27 depicts the inheritance hierarchy which needs to be established for the chains example used in the previous paragraph. The `Server` and `Filter` component

types inherit from the Link class which declares the cin and cout communication end-points necessary for participating in a chain topology.

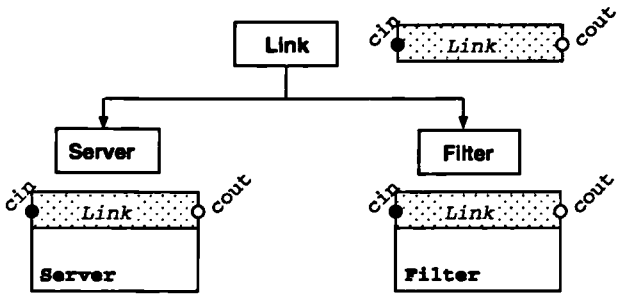


Figure 4.27: Inheritance hierarchy

Thus the chains configuration description is written in terms of the Link class and can therefore be used to create chains of any Link subclass.

Reconfiguration descriptions can also be written in this way. For example a generic migrate reconfiguration methods could be provided which would move isolated components from one host computer to another. A possible implementation strategy could require that all migratable components inherit from a migratable class. Descendents of this class would have to provide component specific exportState and importState state access-methods which the generic migrate method would call to transfer runtime state between the migration target and its replacement.

4.7 Discussion

One of the requirements of the reconfiguration program paradigm concerned the provision of externally invocable state access-methods. It was through the invocation of these methods that the reconfiguration programmer could identify and reinstate invalidated application invariants.

However, querying the state of a running distributed application poses a number of problems. An obvious difficulty is that any returned state information is liable to become out of date. Furthermore in the context of distributed applications, component state-information is rarely useful in isolation. More often than not, it is necessary to correlate returned state-information from a number of different components before a course of action can be decided upon. The state of a distributed system however is more than just the combination of its component-states and reflects the state contained in "en route" messages as well. For example consider a

system where components exchange credits for services. Client $C1$ decrements the number of credits it has by X and sends a message to Server $S1$ requesting a service worth X credits. A state-query which wishes to calculate the total amount of credits in the system will arrive at an incorrect value if it queries $C1$ and $S1$ before the message arrives at its destination. Invoking state-update methods can also be problematic as they can easily lead to the classic "lost update" [Dat95] problem.

The next chapter deals with this issue by ensuring that state-access methods are only invoked on the components of a distributed application when they are in a predefined "safe" state. The nature of this "safe" state is such that state access-methods can be invoked on these components without running into the difficulties set out above.

4.8 Chapter summary

This chapter described the *reconfiguration programming paradigm* which proposed a new way of designing and describing reconfigurable distributed applications. The approach is novel in the way it encapsulates both the static (*configuration*) and dynamic (*reconfiguration*) behaviour of distributed applications. Reconfiguration specifications written in this paradigm are reusable by virtue of the fact that they gain handles to their reconfiguration targets by querying the system configuration. Reconfiguration descriptions written in this way are independent of specific application configurations and can be used throughout the application's life-cycle.

In order to facilitate the re-establishment of invalidated application invariants the paradigm requires that computational components implement a set of general state-access methods which allow an external agent to query and alter a selected subset of their internal state at runtime. The reconfiguration descriptions of components expose this state-access interface and thereby allow the change designer to identify and re-establish invalidated application invariants by invoking state-access methods from within reconfiguration scripts. This is an important property as it offers a practical alternative to embedding extra code into computation components and provides both programmed and evolutionary changes with a general mechanism for re-establishing invalidated application invariants.

The paradigm incorporates both inheritance and composition as part of its design methodology and in so doing facilitates the specification of hierarchic and generic reconfiguration specifications. Additionally in the proposed paradigm the application's composition hierarchy is maintained throughout the application's life-cycle.

This permits the change programmer to think and express changes in terms of the same abstractions used originally during system design.

An extended example, the "UK National Lottery", was used to illustrate the advantages of the proposed paradigm over existing approaches.

Chapter 5

Guaranteeing mutual consistency

The survey in chapter two (section §2.2.4) motivated the need for a general solution to the mutual consistency problem which was geared towards reducing system disruption, as well as minimising the overhead imposed on the system and the application programmer. As we have seen, existing recovery approaches often require heavyweight runtime support systems and can only be applied to systems which are programmed to use them. These factors tend to suggest that an avoidance scheme may be a more suitable choice.

Devising a successful avoidance scheme however is difficult as a delicate compromise must be reached between the general applicability of the approach and the degree of required programmer involvement. Consider that, reducing programmer involvement beneath a certain level requires excessive restrictions to be placed on system behaviour and in so doing effectively rules out most useful applications. Conversely, relaxing these assumptions beyond a certain degree places the burden of bringing about the safe-state almost entirely on the programmer. This is equally bad as it reduces the likelihood of programmers investing the time and effort necessary to write reconfigurable applications.

This chapter presents a new avoidance solution which we believe strikes an acceptable balance between the level of programmer involvement and restrictive assumptions made about the system behaviour. The following sections define the nature of the safe-state chosen by the approach and describe the way it is imposed over the application. The properties of the safe-state are such that state access methods, as described in the previous chapter, can be invoked on the components of the distributed application with meaningful results. In devising the approach special emphasis was placed on minimising the disruption to unaffected sections of the application and reducing the system overhead and programmer involvement.

5.1 Considerations

Many of the avoidance schemes surveyed in chapter two relied on the change programmer to identify and bring about the overall safe-state on a per-application basis [Kin93, MUWZ96, Etz92]. The great advantage of these schemes was that they yielded the best results in terms of minimising system disruption. This however, was achieved at the cost of placing a heavy burden on the change programmer. Aside from the difficulty involved in identifying a suitable safe-state for the affected subsection of the application and specifying the sequence of actions which would bring it about, the imperative nature of this process meant that it was subject to errors and thus the reachability of the reconfiguration state could not be guaranteed.

An alternative approach and one we have chosen to follow was given in [KM90]. It operated by defining a generalised safe-state for individual components in a particular class of distributed applications. The generalised component safe-state possessed certain properties which allowed it to be used as a means of ensuring the mutually-consistent-states property during dynamic reconfiguration. The approach defined a set of *reconfiguration rules* (see page 32) which specified the components which needed to be in the safe-state for each of the basic reconfiguration actions. These rules were then applied to the planned changes and the system configuration to reveal the set of all components which needed to be in the safe-state before the reconfiguration could be executed safely. A configuration manager process instructed these components to move into the safe-state prior to the execution of the change actions. Individual components were coded by the application programmer such that they could, when instructed, move into the component safe-state in finite time.[1]

An important advantage of this approach is that the reconfigurable-state does not need to be rediscovered for every reconfiguration. As long as individual components are able to move into the generalised safe-state, the reachability of the reconfigurable-state is guaranteed. This is an important property which was absent from the "imperative" approaches. Furthermore since the configuration manager process coordinates the acquisition of the safe-state change designers are able to concentrate on the task of expressing reconfigurations.

However, due to the highly abstract nature of the generalised safe-state chosen in [KM90], it was necessary for the component programmer to invest a great deal of effort in writing components which could move from their normal mode of execution

[1] A detailed description and critique of the mentioned schemes was given in section §2.2.2.

into the safe-state. Furthermore, since the actions necessary to move a component to the generalised safe-state are embedded into the code of the component, the components functional behaviour is intermixed with the requirements of the consistency mechanism. In addition the approach was not very good at minimising system disruption.

In the following sections we will present a new approach which does not suffer from these drawbacks. We will begin by describing the generalised safe-state and the corresponding reconfiguration rules.

5.2 A generalised safe-state for reconfiguration

The particular class of distributed applications which we will be considering are composed of an identifiable number of components. Directed communication paths exist between these components, which permit the exchange of information from initiator to recipient. Components are loosely-coupled processes and except for certain periods their state is consistent and self-contained. We characterise this period as a *transaction*; where the state of the component is consistent and self-contained before and after transactions. Transactions encapsulate component interactions, although a component may engage in a transaction which does not involve any external communication (e.g. when a component is performing internal housekeeping which temporarily invalidate its state). The definition of a transaction as used here, is much looser than that in most computing literature. For our purposes, a transaction simply marks a period during a component's execution when its state may not be consistent.

As mentioned before, we are interested in identifying a state where dynamic change can proceed without endangering the mutually-consistent-states property of the components involved in the reconfiguration. To do this we need to derive the reconfiguration rules for each of the reconfiguration actions. The allowed reconfiguration actions are (as in [KM90]) : *create*, *bind*, *unlink*, and *remove*. These create a new instance of a process type, establish a communication path, close a communication path, and remove an existing instance of a process type, respectively. However, since we also wish to permit the invocation of component state access-methods, as proposed in chapter four, we should consider their requirements as well.

The *reconfiguration rules* for deriving the set of components which must not engage in transactions during dynamic change are as follows[2]:

[2]These are comparable to the rules for deriving the members of the *quiescent set* in [KM90].

- *Component creation* - **create**

  Rule: None.

  Justification: A newly created component is initially isolated (i.e. has no communication arcs extended to or from it) and consequently can not respond to nor initiate external transactions.

- *Connection* - **link** and **unlink**

  Rule: The precondition for either binding or unlinking is that the component from which the connection is directed must not be involved in any transactions.

  Justification: The configuration actions *bind* and *unlink* alter the communication paths to and from software components. Since in our definition, a transaction encapsulates all external interactions, a component which is not involved in a transaction is also a component which is not engaged in any external interactions. Such components have consistent and self-contained state and therefore any modification to their external bindings can not compromise their state. Thus it is safe to apply these actions as long as the component from which the communication path is directed (i.e. the sender) does not become involved in a transaction for the duration of the change.

- *Component deletion* - **remove**

  Rule: The Precondition for removing a component is that it is isolated and it is not involved in any transactions.

  Justification: By definition an isolated component can not be involved in external interactions and can therefore be removed safely without leaving other components in an inconsistent state. Furthermore, since the component itself is not involved in any transactions (internal or external) then it is also in a consistent state and can be removed without any adverse side-effects (e.g. removed whilst in the process of writing a file).

- *State access method invocation*

  Rule: The requirement of state access methods is that the component whose methods are being invoked is not involved in any transactions.

  Justification: A component that is not involved in any transactions has state which is consistent and self-contained. Thus its state can be queried and adjusted without error.

In view of the above reconfiguration rules, given a list of reconfiguration actions and state access method invocations we can derive the set of components which should not be involved in transactions for the duration of the change. What remains, is to find a mechanism which forces these components into a state where :

(i) they are not involved in a transaction, and

(ii) they will not participate in any transactions until after the reconfiguration actions have terminated.

In addition, this mechanism must be able to :

(iii) move **all** of the targeted components into this state, and

(iv) do so in finite time.

5.3 Achieving the safe state

Suppose that it is made evident from the program code when a component is about to begin a transaction and when a transaction terminates. Under these circumstances if the component's execution is blocked when it is not involved in a transaction and unblocked after the reconfiguration it will satisfy requirements **(i)** and **(ii)** above. Essentially the acquisition of the safe-state is then reduced to a blocking problem. As long as we can show that we can impose the blocked state over **all** of the affected components **(iii)** and do so in finite time **(iv)**, we can use "blocking" to ensure mutual consistency during reconfiguration.

In the following section we present two algorithms for the enforcement of the safe-state. The first considers the problem of imposing the safe-state across *all of the components* in the system, while the second algorithm improves on the first by attempting to impose the safe state only on those components which are identified by the reconfiguration rules in section 5.2.

However, in order to be able to satisfy requirement **(iii)** and **(iv)** without imposing too much of an overhead on the component programmer, we need to place a number of restrictions on the behaviour of the application. These restrictions along with other assumptions about the application, are set out below.

5.3.1 Assumptions

We consider a system with `Nmax` components which in the absence of dynamic reconfiguration will not deadlock or livelock. Given that transactions encapsulate

inter-component interactions, from an external point of view, components which are participating in a transaction are *active*, while those which do not are *idle* (figure 5.1). More precisely a transaction is a period during which the state of the participating component may not be self-contained or consistent. During the lifetime of a transaction, an active component may make any number of requests to its immediate neighbours. These requests are henceforth referred to as subtransactions. A transaction is guaranteed to terminate and the initiator is always informed of its completion. The initiator component then becomes idle. A component may become engaged in a transaction unilaterally or as a result of receiving a request. Components do not interleave transactions; that is to say that they will not service any non-related requests until the transaction they are currently engaged in terminates, nor will they attempt to start a new transaction if they are already servicing a request. Components, which are in the idle state will respond to the arrival of transaction requests by servicing them. Active components may initiate any number of subtransactions in order to complete a transaction.

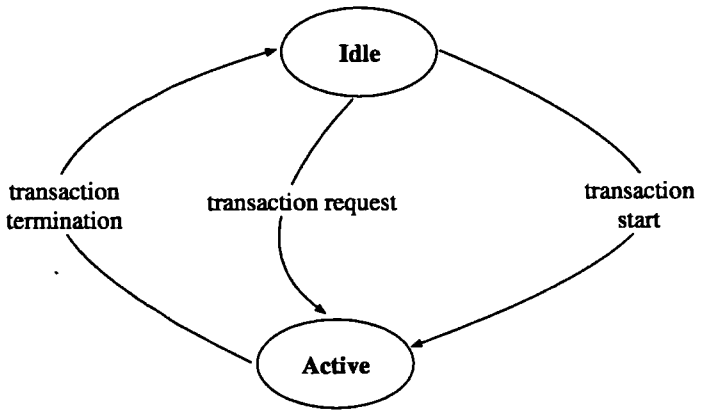


Figure 5.1: Normal state transition diagram

5.3.2 System properties

In a deadlock free system with non-interleaved transactions as we have described, transactions must proceed without interfering with each other. Consider the system in figure 5.2. In the diagram, component N_1 has initiated transaction t_1 which requires component N_2 to deal with one of its subtransactions. However, N_2 has itself initiated a transaction t_7 and due to the non-interleaving requirement, cannot initiate t_3 which would deal with the incoming request from N_1. Thus $N_2(t_3)$ will have to wait until after $N_2(t_7)$ has terminated. But, in order for $N_2(t_7)$ to terminate, its subtransaction must first be serviced by $N_1(t_4)$. As with N_2 however, N_1 is unable

to do this as it is already involved in an independent transaction t_1. It is easy to see that in such a situation deadlock will ensue.

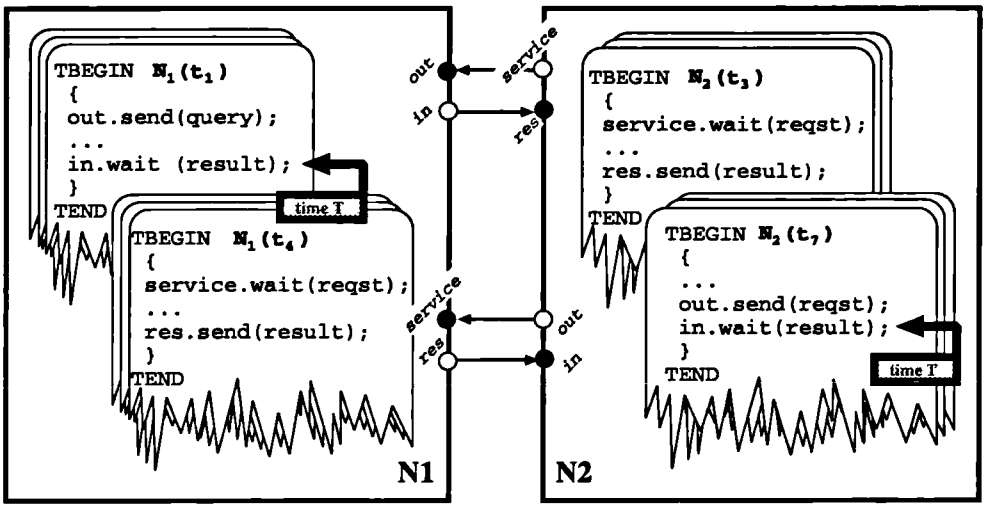


Figure 5.2: Cyclic dependency (*leading to deadlock*)

Two ongoing transactions t_1 and t_2 have a cyclic dependency if t_1 can only succeed after t_2 has succeeded and vice versa. Figure 5.2 illustrated the simplest possible scenario; in practice cycles can span many transactions and components. Clearly, if deadlock situations are to be prevented ongoing transactions must not have any cyclic dependencies.

The requirement that idle components service transaction requests as and when they arrive also means that waiting dependencies are disallowed. A waiting dependency exists when an already initiated transaction cannot complete until after a another non-consequent transaction has terminated. Figure 5.3 illustrates what is meant by a waiting dependency. The two client components C_1 and C_2 are bound to a server component S_1. However, because of the way S_1 has been programmed it will always service a t_1 transaction followed by a t_2 transaction. Thus at time T, even though S_1 is idle and a transaction request from C_1 has arrived, S_1 will not process the request. In effect there is a waiting dependency between transactions t_1 and t_2 - Two t_1 transaction requests will not be serviced unless a t_2 request has been serviced in between and vice versa. Such dependencies can be problematic, particularly in a dynamic environments where for example C_2 can be removed and lead to the starvation of C_1.

An interesting and, as we shall see later, very useful side-effect of a deadlock free, non-cyclic and non-interleaving system with no waiting dependencies is that each

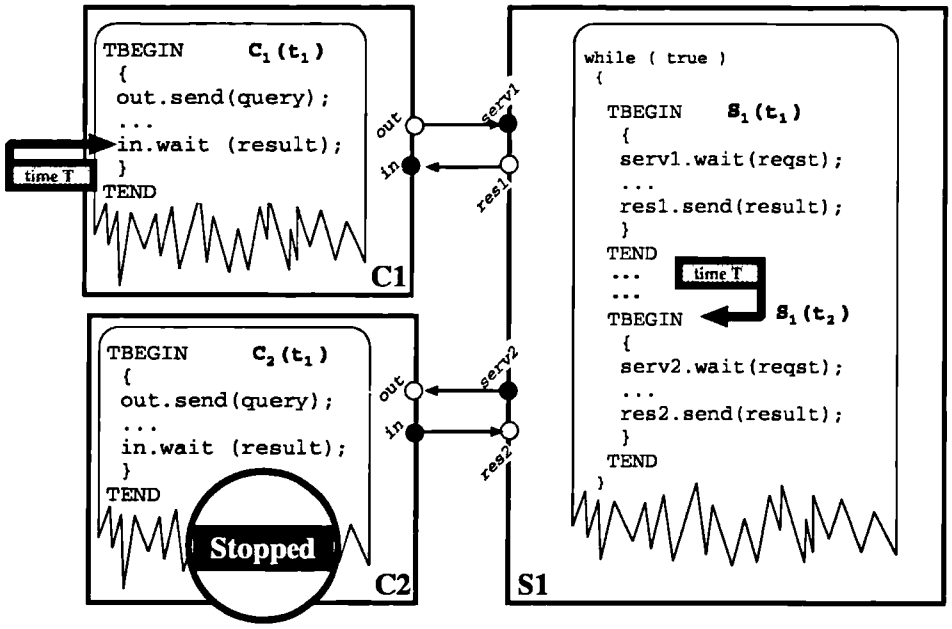


Figure 5.3: Waiting dependency (*starvation of* C_1)

transaction executes as if it is the only ongoing transaction in the system. Transactions which affect non-overlapping sections of the system proceed in parallel and multiple transactions which use the same set of components for their subtransaction requests are processed sequentially.

5.4 Pessimistic Blocking Algorithm

We wish to show that given a system satisfying the above criteria it is *always* possible to externally enforce a state where *none* of the components in the system are engaged in a transaction. We aim to accomplish this by preventing new transactions from starting while allowing ongoing transactions to terminate. The algorithm is as follows.

A coordinator component sends a *block* message to all the components in the system. As components block they send an acknowledge message back to the coordinator. Once all the components in the system have sent the acknowledge message we can be sure that all the components in the system are blocked. As figure 5.4 shows, once the *block* command has been sent to all the components in the system any components which are in the *idle* state move directly into the *blocked* state. Components in the *active* state will continue their execution until they complete

their transactions and become *idle* before moving into the *blocked* state.

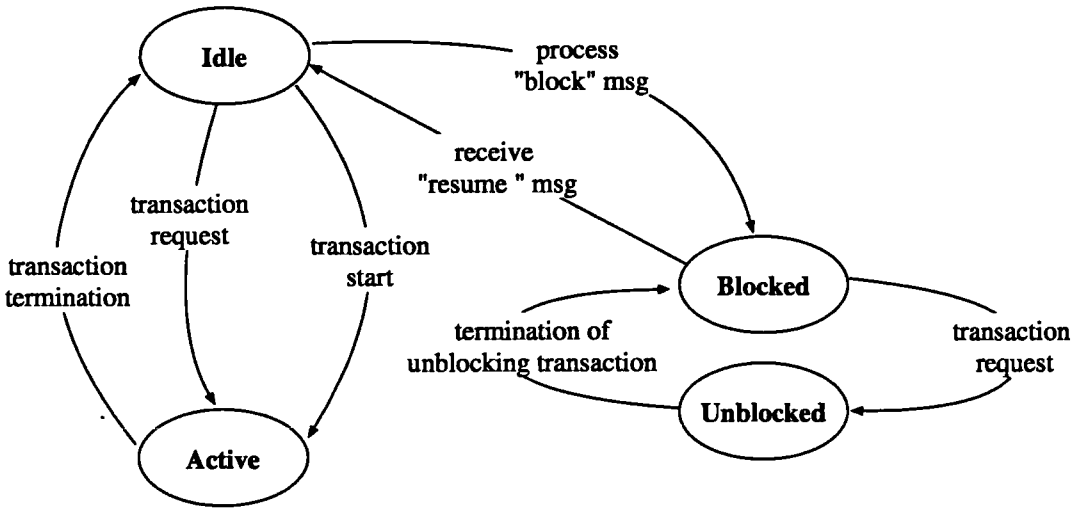


Figure 5.4: Modified state transition diagram

The aim of the algorithm is to prevent new transactions from starting, and also to allow already started transactions to terminate. Therefore requests made by active components must be serviced if their respective transactions are to terminate. It is to this end that the *unblocked* state is introduced. Components which moved into the blocked state but receive a subtransaction request will move into the unblocked state. In this state, components can continue their execution until they have serviced the request which unblocked them. At that time they return to the blocked state.

Note that there is no priority attached to servicing subtransaction requests first, before initiating a new transaction. Hence unblocking a blocked component may well result in a new transaction before the unblocking subtransaction request is serviced. It is therefore necessary to show that unblocking of a component will eventually lead to a fully blocked system.

Justification: We previously argued that each transaction executes as if it were the only one in the system. Therefore if we can show that our algorithm works for a system with only one ongoing transaction it will follow that it will also work with more than one ongoing transaction.

Consider a system where all the components are blocked except one, **N**, which has initiated a subtransaction with one of its blocked neighbours **S**. Since component **S** has received a request it must unblock. Unblocking **S** can have two possible results. **S** can :

1. **Service N's subtransaction immediately.**

   In this case N's subtransaction will be serviced and upon completing any other necessary subtransactions, N will go into the blocked state.

2. **Initiate a transaction with one of its neighbours Q but not N.**

   In this case Q faces the same situation as S and must unblock in order to service the request it has received.

   The case were Q services the request immediately is trivial. However suppose that the chain of transactions is extended with each new partner needing to initiate a new transaction of its own before or as part of servicing the request it received. As in the case with S, with each new transaction there is at least one less *free* component which can be involved in future interactions.[3] In the worst case there will finally be (Nmax-1) components waiting for the result of their subtransactions with the last component forming the final link in the non-cyclic chain of transaction requests.

It is not important whether the transaction initiated by S is consequent to N's transaction. We know that the unblocking of components will allow such transactions to terminate and therefore in time S will process N's request and return the result.

Concurrent subtransaction requests pose no additional problem as they too must conform to the non-interference criterion. Just as with independent transactions any contention between them will be resolved through their sequential execution by the servicing component(s). □

So far we have shown that by selectively blocking and unblocking components it is possible to externally enforce a "transaction-free" state on the system. The algorithm presented enforced the blocked state on **all** the components in the system. However, configuration changes often affect only small sections of the system. It would be desirable therefore to only disrupt the execution of those components which are going to be affected by the change.

5.5 Optimistic Blocking Algorithm

Our new goal is to enforce the safe state on components directly affected by the change. Therefore it makes sense for the controller to send the block message to only

[3]The actual number of *free* components left, depends on the number of concurrent subtransactions a component initiates with its neighbours.

those components which are identified by the reconfiguration rules of section §5.2. Henceforth we shall refer to this set of components as the Block-Set or *BSet* for short.

The difficulty here is that even if all the members of the BSet become blocked, there is no guarantee that the blocked state will stay intact for the duration of the reconfiguration. Looking at the state transitions of the pessimistic algorithm, as given in figure 5.4, we see that *any* subtransaction received by a blocked component causes it to unblock and attempt to service the request. In the new scenario unconstrained components which do not belong to the BSet are capable of unblocking BSet members simply by making subtransaction requests upon them. However if consistency to be preserved during dynamic reconfiguration BSet members must remain blocked for the duration of the change. Applying the old unblocking strategy to this new situation can not guarantee this.

A workable solution is to temporarily ignore requests which originate from components outside the BSet and postpone servicing them until after the completion of the reconfiguration. However, these requests can not all be completely ignored as some of them may need to be processed before the active BSet members can become blocked.

As figure 5.5 shows such a situation could arise if :

(a) The request is made by a non-BSet member as a result of attempting to service a BSet member request, or

(b) Requests made by BSet members are pending execution until the successful processing of the request made by a non-member component to a BSet member.

One possibility is then to only unblock and process requests which fall into the two categories above. However determining whether a request from a non-set member belongs to category (a) or (b) is non-trivial and has substantial run-time overheads.

Under this scenario for case (a) each transaction request would need to carry with it the list of component id's which it had traversed. This list would then be examined each time a request was received to determine whether the transaction had been originally initiated by a BSet member or not. This list would have to be sent as part of every message which would be sent to start a transaction or a subtransaction and further more the size of this list would grow with each step compounding the problem. Case (b) is even more problematic as requests originating from components

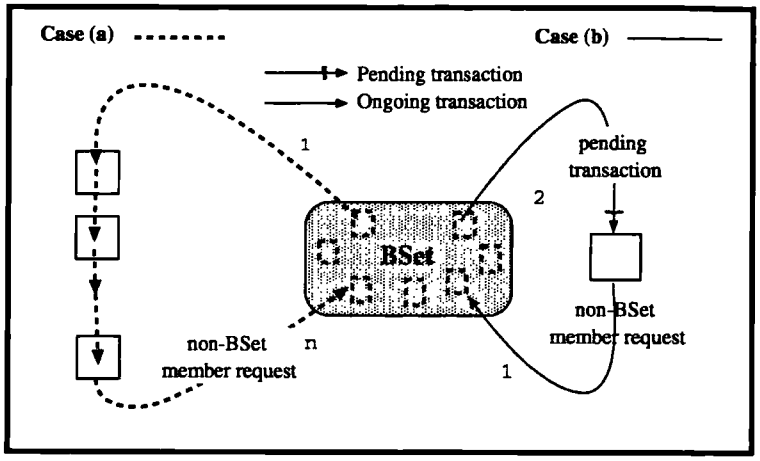


Figure 5.5: Non-BSet member requests

with pending requests from BSet members, which are nevertheless not consequent to that request must somehow be identified.

Instead, we propose to dynamically expand the BSet in step with the outgoing transaction flow. In this way components which receive requests from the *original BSet* members (i.e. those identified by the reconfiguration rules) become members themselves. Components which were not in the original BSet but joined the set as it expanded are referred to as members of the *extended BSet*. By ignoring incoming requests from non-BSet members and expanding the BSet dynamically, the new problem is reduced to the old one. The difference is that instead of assuming from the outset that the behaviour of *all* the components in the system need to be constrained before original BSet members become blocked, we begin with the minimal set and allow the extended BSet to grow as necessary. The state transition diagram for this algorithm is identical to that in figure 5.4 with the exception that the *Blocked* to *Unblocked* transition occurs only if a request was received from a BSet member as opposed to just any component.

It is important to note that the dynamic growth of the BSet is only to facilitate the blocking of the original BSet members. This means that as soon as the blocked state is achieved over the Original BSet, we can unblock all components which were in the *extended-BSet*. The net result is that although the disruption to the system grows during the first phase of the algorithm it shrinks in the second phase leaving only those components which need to be in the safe-state necessary for reconfiguration in the blocked state. This is depicted in figures 5.6 and 5.7 which show the growing and shrinking phases of the algorithm respectively.

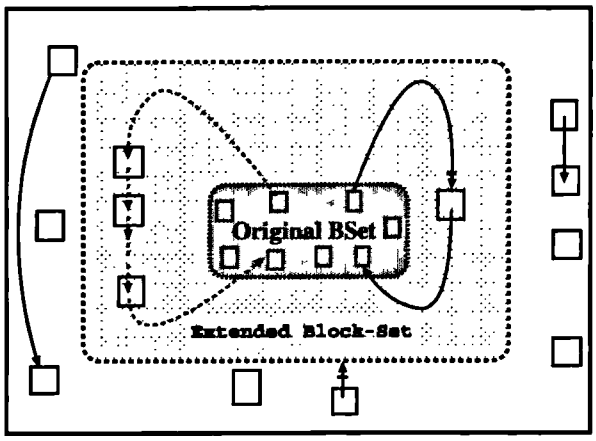


Figure 5.6: BSet expansion phase

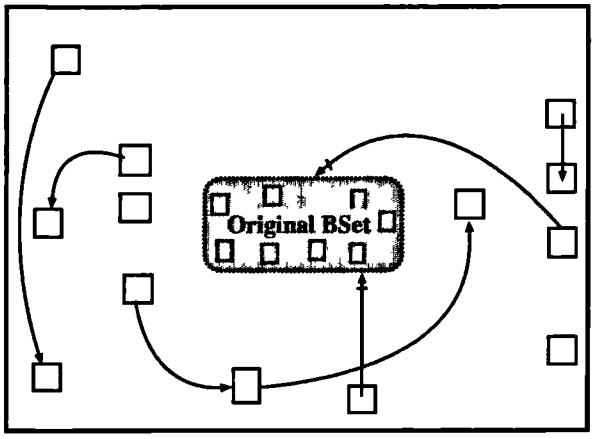


Figure 5.7: BSet shrinking phase

Of course this still leaves those components whose requests are pending the completion of the reconfiguration actions. However, since reconfiguration actions are not expected to be time consuming in comparison with transactions, disruption suffered by these components once the safe state has come about, need not be large.

5.5.1 Concurrency

As we mentioned in section §2.2.4 another means of reducing system disruption during dynamic reconfiguration was to parallelise the execution of multiple reconfigurations. In order to support this type of parallelism however, it is necessary to impose the safe-state for dynamic change on the combined BSets of all the scheduled reconfigurations.

This does not pose any problems for our algorithm as in fact it is capable of imposing the reconfigurable state on the whole system.[4]

5.5.2 Composite components

In the previous chapter we saw that composite components played an important part in aiding the description and management of dynamic reconfiguration. Therefore it is important to address issues which supporting composite components may raise.

Remember that composite components serve mainly as a complexity management tool which group logically related components together by encapsulating their configuration and reconfiguration behaviour in a composite configuration description. However, even though reconfigurations are expressed in terms of creations,

[4]The optimistic algorithm reduces to the pessimistic one, when the BSet encompasses all the components in the system

removals and alterations to the bindings of composite components, all such change description eventually map to the execution of basic reconfiguration actions on primitive components. Our mutual-consistency approach works at the level of these primitive actions and thus as long as it can obtain the BSet membership, its operation will not be affected.

Before dismissing composite components entirely though, recall that many of the behavioral restrictions imposed on the system in section §5.3.1 where put in place so that the mutual-consistency algorithm could guarantee the reachability of the safe-state. It is conceivable that in certain situations these requirements will prove to be quite restrictive.

A possible solution is to use *indivisible composite components* to group logically related components which contravene the behavioral assumptions of the algorithm. Indivisible composite components are given special treatment by the reconfiguration rules in that any reconfiguration action on the composite results in all enclosed primitive components being added to the BSet. The behavioral restrictions required by the algorithm still apply, however they apply to the collective and not to the individual subcomponents. Thus waiting and cyclic dependencies may exist between individual components contained by the composite as long as it is guaranteed that in the absence of externally invoked transactions the group as a whole will move into a transaction free state. A coordinator component will have to be designated such that its state reflects that of the collective i.e if the coordinator is in the safe-state then it can be assumed that so are the other components in the composite. The standard blocking strategy will then be applied to the coordinator component alone and its transition into the safe-state signals the establishment of the safe-state over the entire composite component. Of course by grouping components together in indivisible composite components some of the benefits of the approach, particularly with regards to its minimum disruption properties will be lost. This is a compromise which the application programmer must make.

5.5.3 Compliance with assumptions

So far we have presented a new avoidance scheme which by placing certain constraints on system behaviour, is able to impose the safe-state for reconfiguration whilst minimising system disruption and reducing the overhead placed on the programmer and the runtime system. However, the correct functioning of the algorithm is conditional on the system adhering to the behavioral assumptions set out in section §5.3.1. It would therefore be very useful if one could check that a target

application complies with the required behavioral restrictions before attempting to reconfigure it. The aim of dynamic reconfiguration is to make runtime modifications such that there is little or no disruption to the service provided by the application. This exercise would be futile if due to an unnoticed contravention of the assumptions, an attempted reconfiguration resulted in deadlock.

The difficulty here is that existing tools capable of proving such properties work on abstract specifications of application synchronisation/behaviour as described by Finite State Machines (FSM), Unfortunately, these abstract specifications are far removed from the imperative application descriptions which are actually used to implement the application functionality. Consequently, inferring the application's abstract behavioural descriptions from the application source code is non-trivial and not amenable to automated techniques. This means that in order for an application to be checked for conformance with behavioural restrictions imposed by the presented method (or that of any other), component programmers need to provide these behavioural descriptions.

A major obstacle, which has so far prevented application developers from adopting these schemes is that modelling even relatively simple systems is quite difficult. However, the proposed avoidance scheme takes a very high-level view of transactions and in so doing simplifies the specification of state machines for computational components. Section §5.6 describes how Labeled Transition Systems can be used in conjunction with automated analysis tools such as LTSA [KM98] to check for contraventions of the behavioural restrictions imposed by the proposed avoidance scheme.

Of course a one time static check for conformance is not sufficient as the behaviour of the systems under consideration will be altered by every reconfiguration. In effect this process must be repeated on each target configuration, prior to realising the changes, in order to ensure that the new system will also comply with the behavioral restrictions. The practical benefits of performing such checks is thrown into further doubt when one considers that due to the non-linear complexity of checking algorithms, proving useful properties about the combined specification of an application may take an unacceptably long time.

Recent research efforts have concentrated on reducing the search space for the checking algorithms by mirroring the composition approach to system design and pruning the state space at each level of the composition hierarchy [GKC98, CK95]. The state-explosion problem which fuels the non-linear complexity of the checking algorithms however still persists.

5.6 Automating compliance checks

In the previous section we stated that it was possible to use labeled transition systems (LTS) to verify that the system targeted for reconfiguration adhered to the behavioural restrictions imposed by the proposed mutual-consistency algorithm. In the following sections we will outline how this may be accomplished. More information regarding labeled transition systems and the automated analyser tool LTSA used here can be found in [KM98].

Behavioral restrictions - The consistency mechanism placed three restrictions on the application behaviour. These were as follows :

1. Components were not allowed to interleave the processing of independent transactions,

2. The system was prohibited from having transactions which exhibited cyclic dependencies. Such dependencies would lead to deadlock in an environment where components did not interleave transactions (see above).

3. Waiting dependencies were forbidden. Thus the progress of a transaction depended only on the completion of the consequent transactions its requests have initiated.

5.6.1 LTS extensions

Given the specific nature of the checks that we wish to perform, the basic LTS notation needs to be extended with a number of simple features which will provide hints to the automated analyser. These take form in special attributes, which are selectively assigned by the system designer to certain states and transitions in the LTS.

Transition attributes - Transitions which begin and end transactions are annotated with special attributes. These attributes describe the type of transaction which is started or terminated by the transition. Thus, transitions which begin and end *self-initiated* transactions are assigned the attribute "si-begin" and "si-end" respectively. Similarly the attributes "cons-begin" and "cons-end" annotate transitions which, respectively, begin and end *consequent* transactions. In general LTS' are made up of ordinary and annotated transitions.

State attributes - Aside from the start-state attribute, LTS states can now be designated as "safe" states also. A "safe" state signifies that the modelled component is in a transaction-free state and hence its state is consistent and self-contained. In general LTS' which describe distributed components have *safe* start-states. This signifies that components begin their execution when they are not involved in any transactions and their state is consistent and self-contained.

Transition rules - Ordinary un-annotated transitions carry over the annotation of their source state to their destination state. Thus if an ordinary transition t_1 has a safe state s_0 as its source and d_1 as its destination, then d_1 will also be assigned as a safe state. Alternatively if s_0 isn't safe then unless otherwise assigned, d_1 will not be safe either.

Transitions which are annotated by the "si-begin" and "cons-begin" annotations start from "safe" source states and move the LTS to an unsafe destination state. Conversely, "si-end" and "cons-end" transitions have unsafe source states and "safe" destination states.

Figure 5.8 depicts a simple client server system. The client component C_1 engages in a self-initiated transaction c_1t_1. This transaction is serviced by a consequent transaction in the server component S_1. The filled circles in the start-states of both machines, labeled 0, indicate that the start-state is also a designated safe-state.

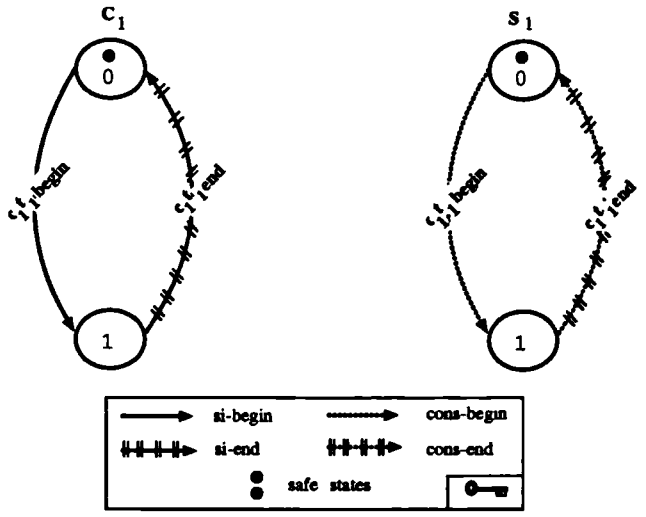


Figure 5.8: Sample annotated LTS

Note that in this instance the consequent transaction in S_1 has the same labels as those in the C_1 LTS. This implies that the interactions between the two machines

is synchronous.

5.6.2 Detecting interleaved transactions

As mentioned before, application components are forbidden to interleave transactions. To verify that an LTS satisfies this property the automated analyser needs to check the LTS against the transitions rules set out in the previous section. Any contravention of these rules point to interleaving transactions. For example consider the LTS depicted in figure 5.9. It shows an LTS for a client component as supplied by the programmer on the left. In it the programmer has annotated the transitions and also identified the start-state of the LTS as a safe-state. As is evident, the component C_1 engages in two self-initiated transactions c_1t_1 and c_1t_2.

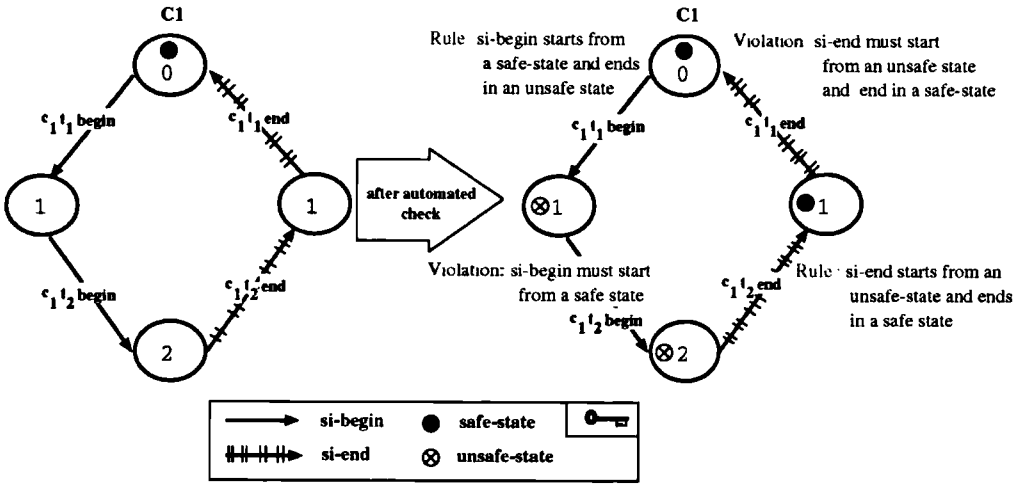


Figure 5.9: Processed LTS

The figure on the right depicts the LTS as it would appear subsequent to checks performed on it by an automated analyser. The analyser has applied the transition rules and identified two violations. These violations result from the fact that the component is attempting to interleave the two self-initiated transactions, by starting c_1t_2 before c_1t_1 has terminated.

Note that these checks are applied to individual LTS' and any additional machines which describe the transaction behaviour of an individual distributed component.

5.6.3 Detecting cyclic dependencies

As described in section §5.3.2, in an environment which transactions cannot be interleaved, cyclic transaction dependencies will lead to deadlock. Existing automated analysis tools such as LTSA already identify and warn of potential deadlock scenarios and can therefore be used to identify cyclic dependencies.

Figures 5.10 and 5.11 depict the labeled transition systems for the cyclic dependency example used in figure 5.2 of section §5.3.2 (see page 127). In the example, component N_1 is capable of participating in two different transactions, denoted by t_1 and t_4. The self-initiated transaction t_1 and the consequent transaction t_4 are not interleaved as each transaction-begin transition starts at a safe-state. The LTS for N_2 is identical to that described for N_1 with the exception that it participates in the self-initiated transactions t_7 and the consequent transaction t_3.

State 0 in both diagrams denotes the *safe-state* where the components are idle and not involved in any transactions i.e. their state is both consistent and self-contained. Notice that the transitions systems do not model all the interaction which occur between components, but only those which are instrumental in initiating and terminating transactions. This fact greatly simplifies the modelling process as only the high level transactional view needed by the consistency mechanism is considered and not the full computational/synchronisation behaviour.

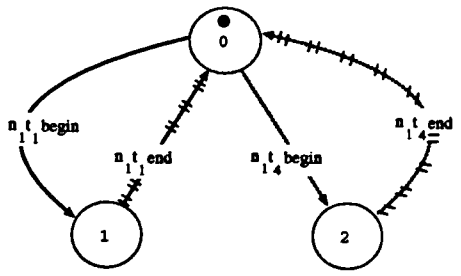


Figure 5.10: LTS for Node1

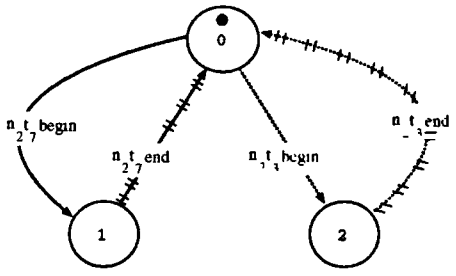


Figure 5.11: LTS for Node2

The LTS' for N_1 and N_2 however do not share any common transitions and are therefore completely independent. The synchronisation machines given in figures 5.12 and 5.13 bind the systems together such that the self-initiated transaction n_1t_1 (transaction t_1 initiated by component n_1) is serviced by a consequent transaction n_2t_3 and similarly the self-initiated transaction n_2t_7 is serviced by the consequent transaction n_1t_4. The synchronisation machines are necessary as the asynchrony inherent in distributed systems needs to be modelled. Binding the n_1t_1 and n_2t_3 transitions together directly, would discount the delay between a transaction request message being sent by the client, and that of it being received by the

server.

The *Node1Sync* machine of figure 5.12 models this asynchrony by specifying the sequence of transitions $n_1 t_1 begin$ followed by $n_2 t_3 begin$, which correspond to the transaction request message being sent by N_1, and the request being received by N_2, respectively. The introduction of the intermediate state, 1, allows other transitions to occur before the message is received by N_2. Transaction termination is handled in a similar manner with $n_2 t_3 end$ occurring before $n_1 t_1 end$. Figure 5.13 describes a similar machine for the interaction between $n_2 t_7$ and $n_1 t_4$.

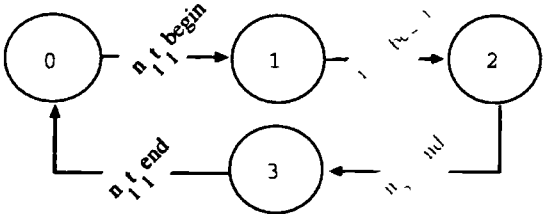


Figure 5.12: LTS Node1Sync

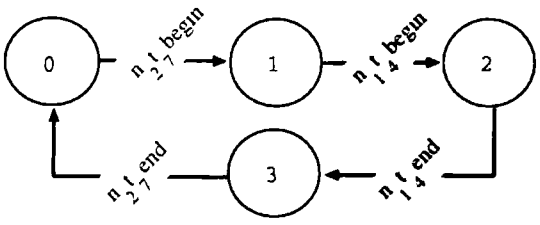


Figure 5.13: LTS Node2Sync

Together these machines describe the subsystem which was depicted in figure 5.2. As mentioned earlier automated LTS analysers combine these machines to create a composite machine which they then use when attempting to prove desirable properties about the system. The composite machine for this system was generated by LTSA and is depicted in figure 5.14.

As one can see, there is a deadlock state, at state 9, which can be reached as a result of both N_1 and N_2 starting transactions $n_1 t_1$ and $n_2 t_7$ together. This can be checked automatically by LTSA as part of the "safety" checks it carries out. Figure 5.15 depicts the LTSA window subsequent to performing "safety" checks. The upper window contains the description of the LTS' for *Node1*, *Node2*, *Node1Sync* and *Node2Sync*. The lower window shows the detected trace to the deadlock state which in the context of the proposed consistency algorithm warns of the existence of a cyclic dependency within the system.

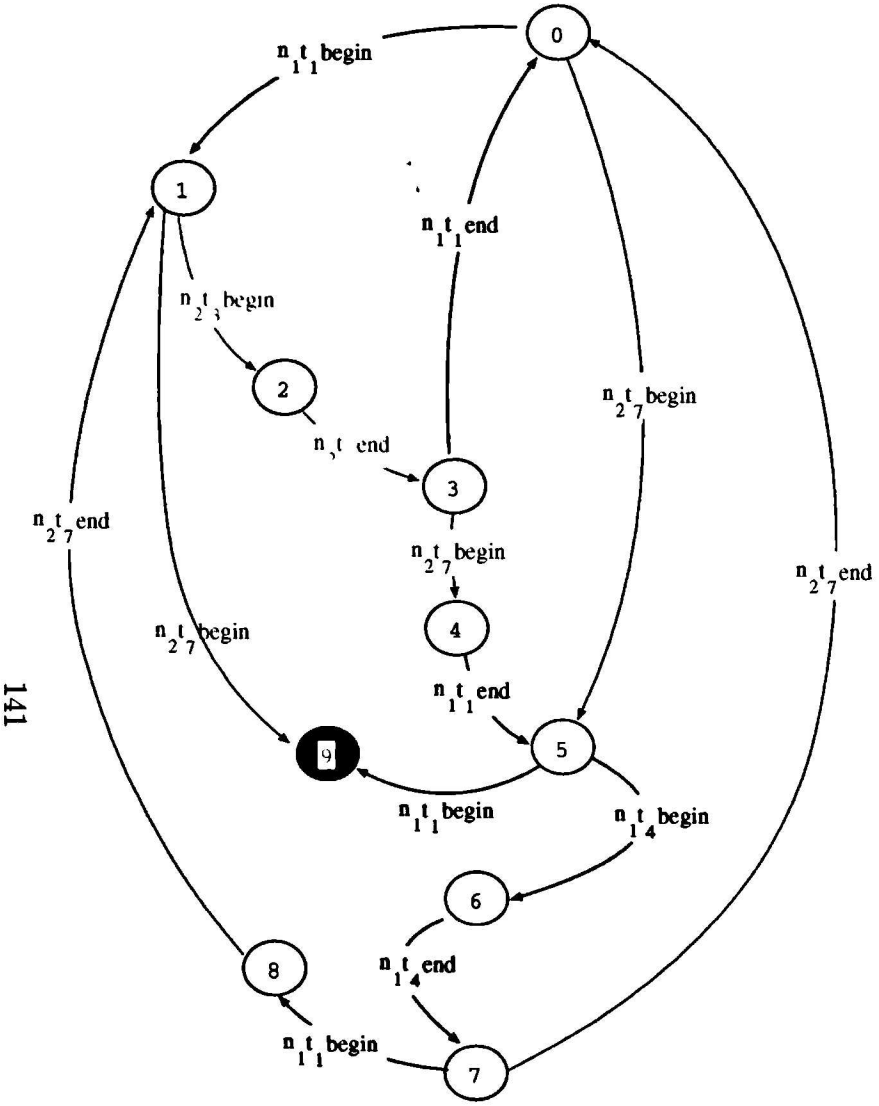


Figure 5.14: Composed LTS

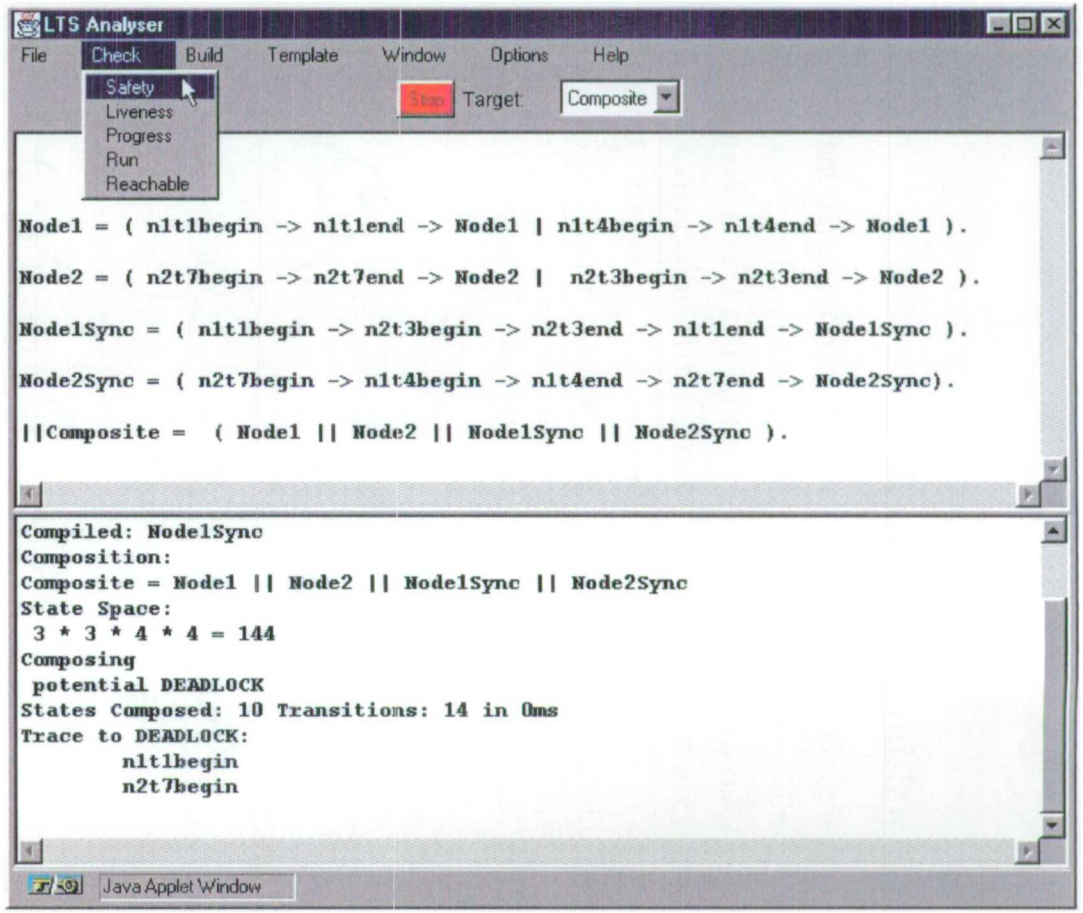


Figure 5.15: LTSA - detected deadlock

5.6.4 Detecting waiting dependencies

The final requirement placed on targets of the proposed consistency mechanism was that they should not exhibit waiting-dependencies. A waiting dependency exists between a self-initiated transaction t_1 and another self-initiated transaction t_2 if t_1 can only succeed after t_2. Figures 5.16 depicts the labeled transition systems for the waiting dependency example used in figure 5.3 of section §5.3.2 (see page 128). We will use this example to illustrate the process by which waiting dependencies can be detected.

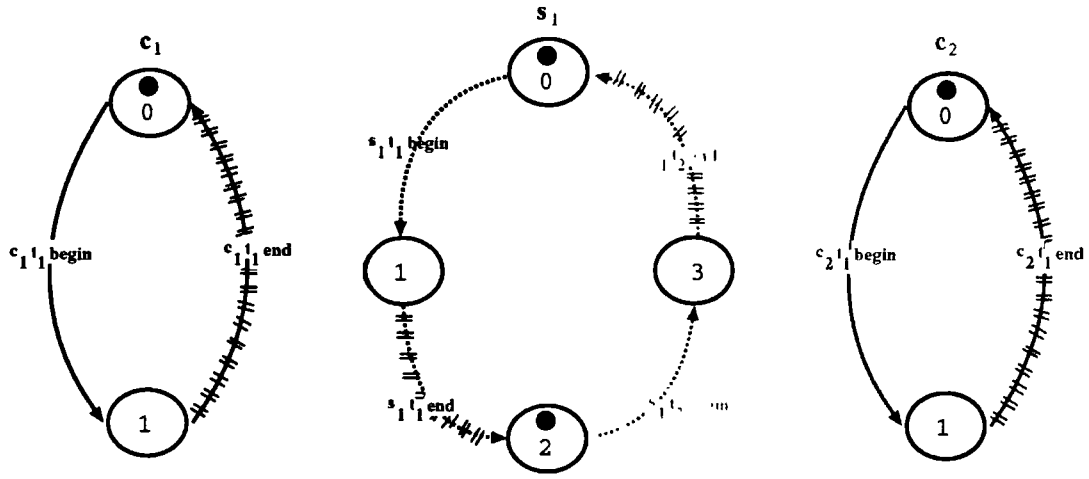


Figure 5.16: Two clients and a flawed server

As the synchronisation machine C_1S_1Sync in figure 5.17 specifies, the transaction t_1 in client c_1 is serviced by the consequent transaction t_1 in the server component s_1. Similarly C_2S_1Sync establishes the association between the self-initiated transaction t_1 in client c_2 and the consequent transaction t_2 in the server s_1.

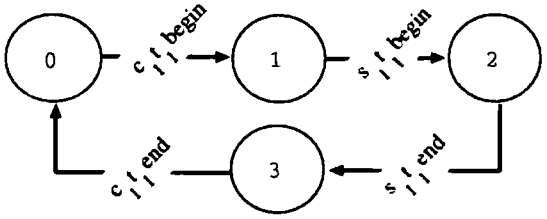


Figure 5.17: LTS C_1S_1Sync

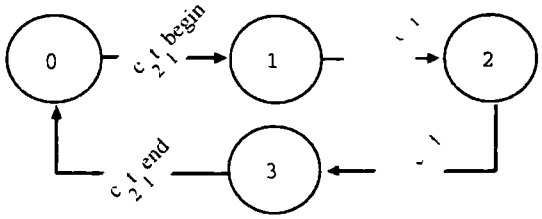


Figure 5.18: LTS C_2S_1Sync

The system described above has a waiting dependency as the transactions c_1t_1

and c_2t_1 cannot succeed independently. This comes about as a result of the way the server component s_1 orders the execution of its two consequent transactions t_1 and t_2. The server will only service a consequent t_2 transaction request after it has accepted a t_1 transaction request. Except for the initial system start-up case, the reverse is also true about t_1 i.e. t_1 requests will only be processed after a t_2 has completed.

It might seem that the existence of the waiting dependency can be deduced from the LTS of the server component alone. This however is not true and in general it is necessary to examine the overall system LTS which is created by composing all the individual machines.

As a counter example, consider the LTS of the client component c_3 given in figure 5.19 below, which if used in conjunction with the server component s_1 would not give rise to waiting dependencies. The LTS for c_3 describes a component which begins a self-initiated transaction that relies on two consequent sub-transactions with the server component s_1. The order in which c_3 invokes these consequent transactions is the same as that in s_1 and therefore there is no waiting-dependency. Of course more complex counter-examples which span multiple components are also possible. Again, as in this case, the ordering imposed by the server component would need to match the order in which clients make requests upon the server.

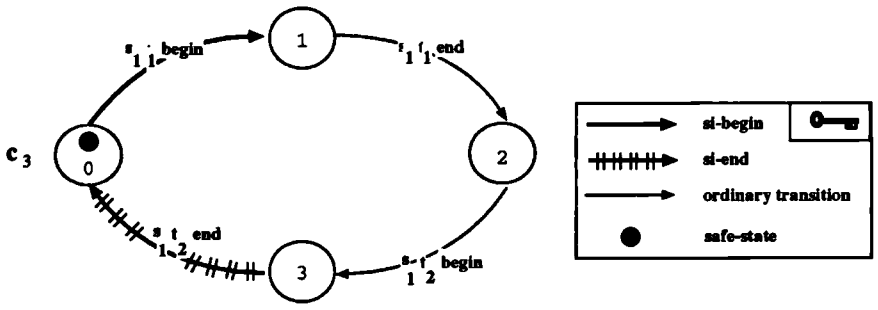


Figure 5.19: LTS C_3

An application does not exhibit waiting-dependencies, if it can be shown that none of the self-initiated transactions within it exhibit waiting-dependencies. To check that a self-initiated transaction does not exhibit a waiting-dependencies it is necessary to examine every safe-state of the transaction in the composed LTS and ensure that progress can be made to another safe-state of the transaction by going through consequent transaction transitions alone.

This in effect restates the basic premise of independent transactions, namely

that a self-initiated transaction should be able to terminate once its consequent transactions have been serviced. Of course before its consequent transactions can be serviced it may be necessary for the server to complete any other consequent transaction requests it may have received earlier. Thus a self-initiated transaction may have to wait for the consequent transaction of other self-initiated transactions to complete before its own are serviced. When checking for this condition, the state and transition annotations described earlier are necessary as otherwise the LTS analyser will not be able to distinguish between self-initiated and consequent transactions.

Figure 5.20 depicts the composed LTS for the "Two clients and a flawed server" system specified in figures 5.16, 5.17 and 5.18. The example exhibited a waiting-dependency since the progress of the self-initiated transaction c_2t_1 depends on that of c_1t_1 and vice versa. This can be verified by performing the above mentioned check on the composed LTS. For clarity the LTS annotations have been presented pictorially as follows.

- Each *self-initiated* transaction has been allocated a colour and its begin and end transitions have been drawn as solid arcs in that colour. Thus $c_1t_1 begin$ and $c_1t_1 end$ transitions are in blue and $c_2t_1 begin$ and $c_2t_1 end$ transitions are in green.

- The begin and end transitions of *consequent* transactions have been drawn as dotted arcs with the same colour as the self-initiated transaction which initiated them. Thus $s_1t_1 begin$ and $s_1t_1 end$ transitions are drawn as dotted blue lines and $s_1t_2 begin$ and $s_1t_2 end$ transitions are drawn as dotted green lines.

- The states of the LTS have been annotated with filled circles. A filled circle marks a state as a "safe-state" of a particular component in the system. The colour of the filled circle matches a unique colour which has been allocated to a component in the system. In our example the colours, blue, green and red have been allocated to the c_1, c_2 and s_1 components respectively.

  Thus a state which is annotated with a blue and a red circle, is marked as a safe-state of the client component c_1 and the server component s_1. The absence of a green filled circle indicates that the client component c_2 is not in a "safe-state" at that point.

The existence of the waiting dependency is evident from the LTS at states 14 and 22 for the c_1t_1 transaction where it cannot make progress until after c_2t_1 terminates.

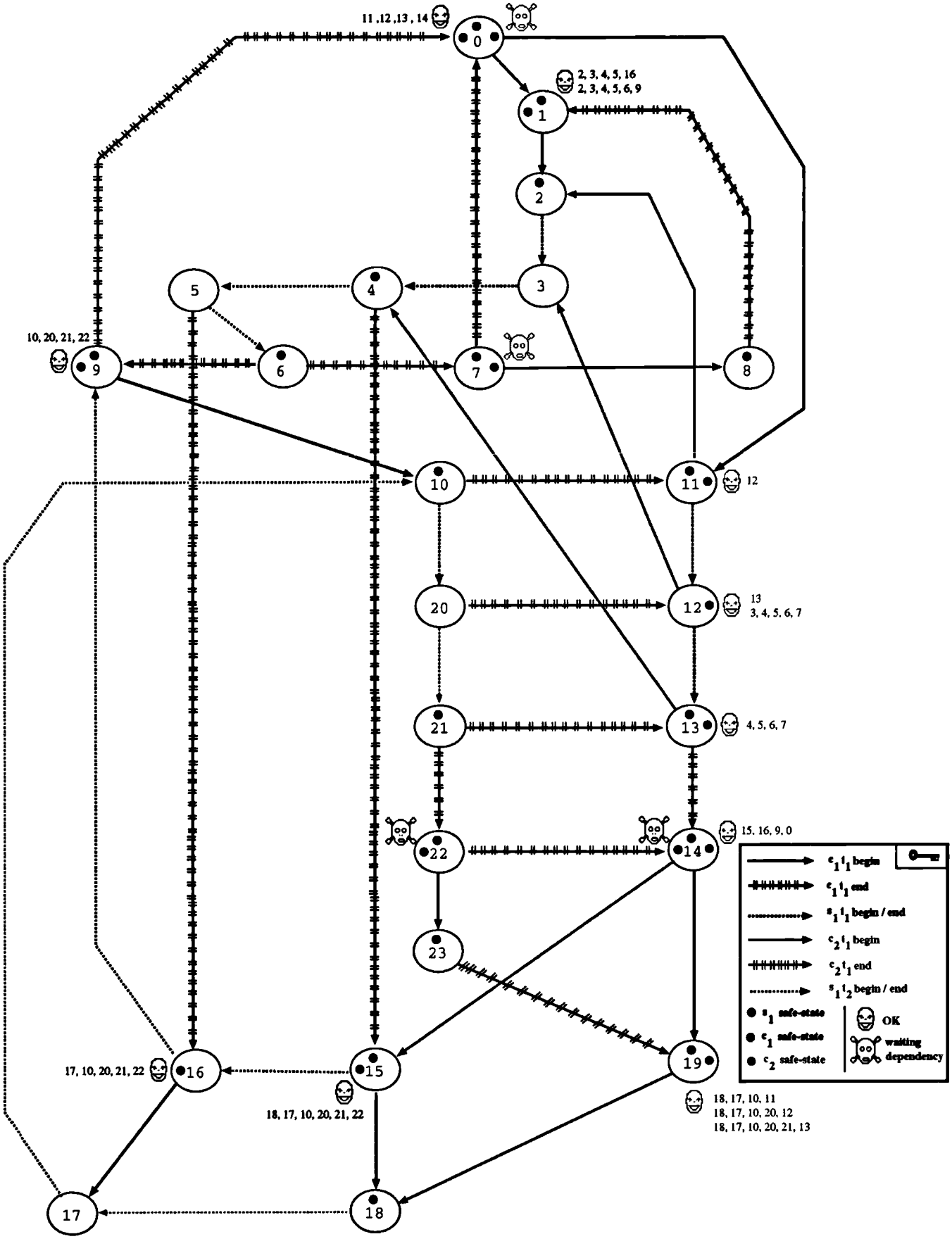


Figure 5.20: Composed System

Waiting dependencies also exists at states 0 and 7 where this time c_2t_1 must wait for c_1t_1 to complete. Other safe-states of c_1 and c_2 have been labeled with the transition paths which take them to another safe-state without going through an unrelated self-initiated transaction.

Figure 5.21 depicts the modified server which does not impose an ordering on the servicing of its consequent transactions. This new definition allows either consequent transaction to begin once the server enters the safe-state. This modification prevents waiting-dependencies from occurring in our example system.

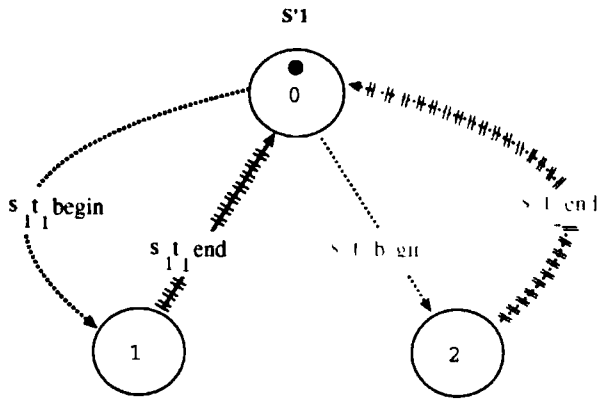


Figure 5.21: Corrected server component LTS

Composing this modified server with the clients c_1 and c_2 and the synchronisation machines C_2S_1Sync and C_2S_1Sync results in the LTS depicted in figure 5.22. The transition and state annotations are as described for figure 5.20.

Examining the safe-state transitions for the self-initiated transactions c_1 and c_2 verifies the absence of waiting-dependencies in the system. This follows from the fact that from every safe-state of the two transactions, it is possible to move to another safe-state by only going through consequent transaction transitions. As with figure 5.20, these transitions have been labeled on the composed LTS of figure 5.22.

5.6.5 Summary

In this section we illustrated how labeled transition systems can be used to verify whether a particular system satisfies the requirements of the consistency algorithm. Due to the specific nature of the performed checks, the basic LTS notation was extended with a number of simple features. These additional features enable an automated analyser to test the conformance of a target system to the requirements of the proposed mutual-consistency mechanism.

The proposed scheme relied on system designers to use the extended LTS notation to model the behaviour of the system. Many programmers however shy away from using notations such as labelled transition systems because of the time and effort that needs to be invested in the modelling process. Fortunately, the assumptions made by the mutual-consistency mechanism relate only to the transactional behaviour of components. This is advantageous as programmers can choose not to model the complete synchronisation and computational behaviour of a component, and only model the more course-grained transactional aspect of the distributed component, needed by the analyser.

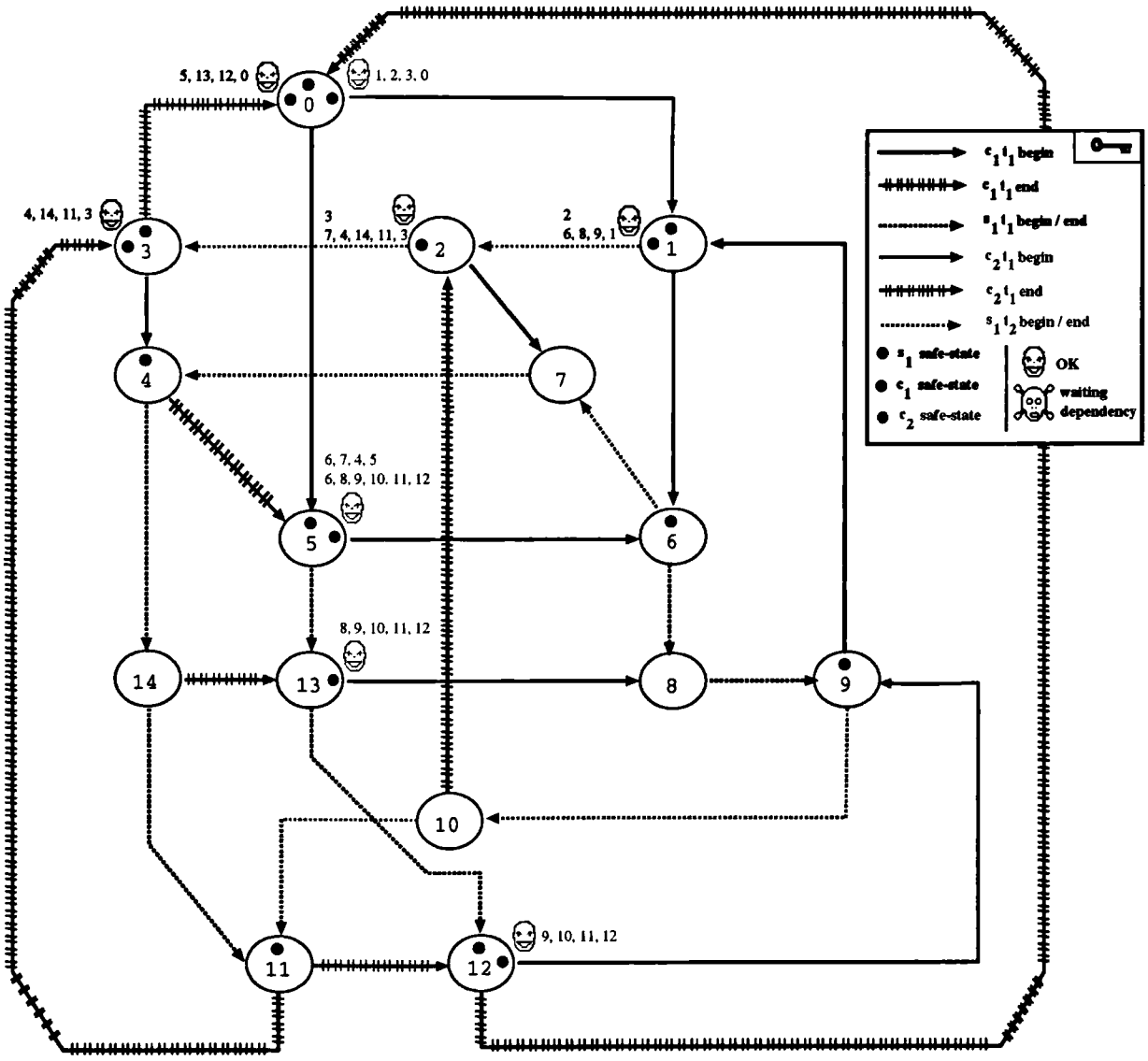


Figure 5.22: Composed LTS with corrected server

5.7 Consistency algorithm

In the framework set out in chapter three (see figure 3.1) we stated that in order for reconfigurations to go ahead without compromising the mutually-consistent-states property of the system, the *Consistency Manager* (CM) and the components in the distributed application needed to cooperate with each other.

This section explores the nature of this interaction for the particular avoidance mechanism we have proposed. These interactions are encapsulated by a consistency object (CO) which the CM uses.

5.7.1 The consistency object - CO (figure 5.23)

The consistency object, figure 5.23, works out the *Original_BSet* members by applying the reconfiguration rules in section §5.2 to the change script and sends them the "block" control message along with the full BSet membership. This is necessary so that BSet members can process requests from other BSet members while queuing requests originating from non-BSet members. Additionally it "primes" components with which BSet members have direct outgoing communication links. This is necessary as we want these primed components to join the BSet if they receive requests from BSet members. In effect these components are potential members of the Extended_BSet.

It then waits until all of the components in the BSet have become blocked. At this point the "freeze" message is sent to Original_BSet members to prevent them from unblocking again, before sending the "resume" and "clear" message to members of the Extended_BSet and Primed_Sets respectively. The reconfiguration script can now be executed and "resume" messages sent to original BSet members.

In more detail - Whenever a BSet member becomes blocked it informs the CO. In this way once the CO has received "Blocked_ack" messages from all of the BSet members it can be sure that the safe-state has come about.

However as we have mentioned, it may become necessary to expand the BSet dynamically. This becomes necessary when BSet members make transaction requests to non-BSet members. In such cases the recipient of the request, which had been primed before, notifies the CO and requests to become part of the extended-BSet. The CO in turn informs old BSet members of the addition and also sends the full BSet membership, which consists of the original BSet members and those in the extended-BSet to the new member. As with the original BSet members the

```
Consistency_Object ( Reconf_Script , Sys_Config )
  {
     Orig_BSet = Get_BSet ( Reconf_Script , Sys_Config )
     Xtndd_BSet = {} /* Extended BSet */
     PSet = {} /* Primed Set */
     Blocked_Count = Size ( Orig_BSet )
     foreach ( N in Orig_Bset )
        N.send ( Block, Orig_Bset )
        foreach ( M in ( linked N → M ) )
           M.send ( Prime, N.n_id )
           PSet = PSet ∪ {M.n_id}
     while ( Blocked_Count > 0 )
        {
        reply_port.receive ( msg )
        switch ( msg.type )
           case Blocked_ack :
              Blocked_Count = Blocked_Count - 1
              if ( Blocked_Count == 0 )
                 foreach ( N in Orig_Bset )
                    N.send@ ( Freeze )
                 foreach ( P in PSet )
                    P.send ( Clear )
                 foreach ( M in Xtndd_BSet )
                    M.send ( Resume )
           case New_BSet_Member :
              Xtndd_BSet = Xtndd_BSet ∪ { msg.n_id }
              PSet = PSet - { msg.n_id }
              Blocked_Count = Blocked_Count + 1
              msg.n_id.send ( BSet, Orig_BSet ∪ Xtndd_BSet )
              foreach ( N in ( Orig_BSet ∪ Xtndd_BSet ) )
                 N.send ( BSet_Add, msg.n_id )
              foreach ( M in ( linked msg.n_id → M ) )
                 M.send ( Prime, msg.n_id )
                 PSet = PSet ∪ {M.n_id}
        } /* while */
     start ( Reconf_Script )
     foreach ( N in Orig_BSet )
        N.send ( Resume )
     exit ()
  }
```

Figure 5.23: Pseudo code for the consistency object (CO)

CO must prime surrounding components which have directed communications paths extended to them from the new member by informing them of its identity.

In keeping with the second phase of the algorithm (as mentioned in section 5.3) once all the components in the BSet have become blocked the CO sends "resume" messages to those in the extended-BSet. Recall that these were only included in the BSet so that the original BSet members could become blocked and since this has occurred there is no longer any need for them to remain blocked. As with all "resume" messages the CO precedes them with a "clear" message to the relevant primed components, informing them that the unblocked components are no longer in the BSet. Primed components are only sent the "clear" message when the blocked state has been achieved over the whole BSet.

Note that there is no reason for sending "prime" or "clear" messages to members of the BSet. These checks have been omitted from the pseudo code. However the operation simply requires the target set for the "prime" and "clear" messages to be checked against the BSet. i.e.

$$\texttt{(linked N} \rightarrow \texttt{P)} - \texttt{(Orig_BSet} \cup \texttt{Xtndd_BSet)}$$

Finally as the Extended_BSet members are going to be unblocked it is necessary to inform Original_BSet members that the blocked condition has been achieved and that they should now ignore *all* incoming requests. In this way any requests originating from the newly unblocked Extended_BSet members will also be ignored. This is done via the "freeze" message. The "freeze" message is unique in that the CO must be sure of its successful delivery to all recipients before sending the "resume" message to members of the Extended_BSet. This property is necessary as otherwise a request from an Extended_BSet member which has been resumed may result in the unblocking of an Original_BSet members, thus compromising the reconfigurable state.

5.7.2 Component participation

The component participation in this scheme is slightly more complex than that of the CO. This is partly due to the fact that it must be performed in an application independent manner and remain transparent to the programmer. This is achieved through a number of hooks which when activated, trigger the execution of the appropriate code fragments which actually implement the algorithm's functionality. Figure 5.24 illustrates a component's transitions through the different phases of the algorithm along with the conditions which trigger the changes.

The points of activation are as follows :

- **Message arrival :** When a message arrives at any of the component's communication ports.
- **Message receipt :** When the component accepts a message.
- **Transaction begin :** The point just before the component is about to become involved in a transaction.
- **Transaction end :** The point immediately after the termination of a transaction.
- **Arrival of control messages :** Messages originating from the CO are processed upon arrival.

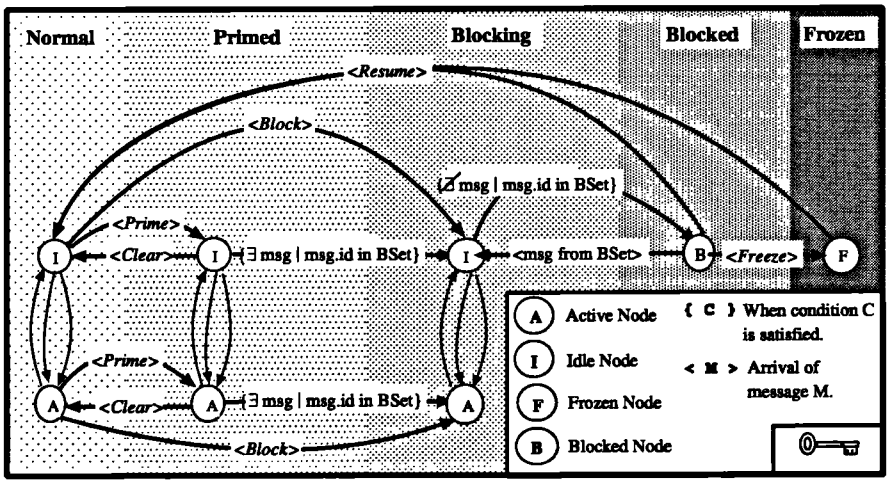


Figure 5.24: Algorithm phase transition diagram

The following sections give the pseudo code for each of the phases of the algorithm. Messages are assumed to carry a unique system wide identifier which can be used to identify the sender.

5.7.2.1 The normal phase (figure 5.25)

Components in the normal phase may receive the "block" control message from the CO which is accompanied by the BSet membership (stored in the *BSet* variable for future use). If the component is currently idle and no messages originating from

the BSet members have arrived, it can move directly to the "blocked" phase.[5] As part of this transition it must send a *Blocked_ack* message to the CO and suspend its execution after installing the hooks for the "blocked" phase.

```
Normal_hooks ()
  case C_MSG_Block : /* Block , Orig_Bset */
      C_state = Blocking
      BSet = msg.Orig_Bset
      if (state == Idle & num_bset_msgs (BSet,arivd_msgs) == 0)
          CO.send ( Blocked_ack )
          C_state = Blocked
          install_hooks ( Blocked_hooks )
          Block ()
      else install_hooks ( Blocking_hooks )
  case C_MSG_Prime : /* Prime , node_id */
      if ( arivd_msgs[msg.node_id] > 0 )
          C_state = Blocking
          install_hooks ( Blocking_hooks )
          CO.send ( New_BSet_Member , node_id )
      else
          C_state = Primed
          PSet = { msg.node_id }
          install_hooks ( Primed_hooks )
  case Msg_Ariv : /* msg */
      arivd_msgs[msg.node_id] = arivd_msgs[msg.node_id] + 1
  case Msg_Rcpt : /* msg */
      activ_msgs[msg.node_id] = activ_msgs[msg.node_id] + 1
  case T_Begin : /* Idle → Active transition */
      state = Active
  case T_End : /* Active → Idle transition */
      foreach ( n_id in activ_msgs )
          arivd_msgs[n_id] = arivd_msgs [n_id]-activ_msgs [n_id]
          activ_msgs[n_id] = 0
      state = Idle
```

Figure 5.25: Hook pseudo code for the "normal" phase

The "prime" control message causes the component to add the component identifier accompanying the message to the *PSet* variable and install the hooks for the "primed" phase. The *PSet* variable consists of the set of components, requests from which should cause a primed component to join the BSet (as described in section 5.3). If however such a message already exists then the component can move

[5]This is an optimisation step which is not depicted in figure 5.24.

to the "blocking" phase directly. In doing so it must install the relevant hooks and request to join the BSet (by sending the *New_BSet_Member* message).

The rest of the hooks simply maintain internal data structures. The *Msg_Arrive* hook updates the *arivd_msgs* variable, which for each sending process maintains the number of unprocessed messages queued at the component. The *Msg_Receipt* hook updates the *activ_msgs* variable, which maintains the number of messages currently being processed by the component for each sending process. The *T_Begin* hook simply toggles the component's internal state from idle to active. The *T_End* hook resets the *active_msgs* and also updates the *arivd_msgs* such that it excludes messages processed during the completed transaction.

5.7.2.2 The primed phase (figure 5.26)

A primed component must react to the changing nature of the BSet by updating the set of components, requests from which cause it to join the BSet. As before when this set is updated the component checks to see if such a message has already arrived in which case it will enter the "blocking" phase and request to become a member of the BSet.

```
Primed_hooks ()
  case C_MSG_Prime : /* Prime , n_id */
     PSet = PSet ∪ { msg.n_id }
     if ( arivd_msgs [msg.n_id] > 0 )
        PSet = {}
        C_state = Blocking
        install_hooks ( Blocking_hooks )
        CO.send ( New_BSet_Member , n_id )
  case C_MSG_Clear : /* Clear */
     PSet = {}
     C_state = Normal
     install_hooks ( Normal_hooks )
  case Msg_Ariv : /* msg */
     arivd_msgs [msg.n_id] = arivd_msgs [msg.n_id] + 1
     if ( msg.n_id ∈ PSet )
        PSet = {}
        C_state = Blocking
        install_hooks ( Blocking_hooks )
        CO.send ( New_BSet_Member , n_id )
  case Msg_Rcpt, T_Begin & T_End as in Normal_hooks
```

Figure 5.26: Hook pseudo code for the "primed" phase

The identifier of new messages are checked against those in the *PSet* to see if the component needs to move into the "blocking" phase. The "clear" control message signals the end of the reconfiguration and the component's return to the "normal" phase. The actions carried out for the *Msg_Receipt, T_Begin* and *T_end* hooks are as those described in the normal phase.

5.7.2.3 The blocking phase (figure 5.27)

Components which move into the "blocking" phase of the algorithm after having been primed will be sent the full BSet membership and must therefore update their *BSet* variable and then try to see if they can move into the "blocked" phase.

```
Blocking_hooks ()
  case C_MSG_BSet : /* BSet , BSet */
     BSet = msg.BSet
     if (state==Idle & num_bset_msgs(BSet,arivd_msgs)==0)
        CO.send ( Blocked_ack )
        C_state = Blocked
        install_hooks ( Blocked_hooks )
        Block ()
  case C_MSG_BSet_Add : /* BSet_Add , node_id */
     BSet = BSet ∪ { msg.node_id }
  case T_End : /* msg */
     foreach ( n_id in activ_msgs )
        arivd_msgs[n_id] = arivd_msgs[n_id]-activ_msgs[n_id]
        activ_msgs[n_id] = 0
     state = Idle
     if (BSet≠{} & num_bset_msgs(BSet,arivd_msgs)==0)
        if ( C_state ≠ UnBlocked )
           CO.send ( Blocked_ack )
        C_state = Blocked
        install_hooks ( Blocked_hooks )
        Block ()
  case Msg_Ariv, Msg_Rcpt & T_Begin as in Normal_hooks
```

Figure 5.27: Hook pseudo code for the "blocking" phase

The only reason a component moves into the "blocking" phase and does not become "blocked" is that it has received requests from other BSet members and must process them. Therefore at the termination of each transaction it must check to see if it has serviced all outstanding requests so that it may block.

Due to the dynamic nature of the BSet such component must also be able

to handle new additions. This is done by keeping the *BSet* variable updated as the membership changes. Once again, the actions performed by the *Msg_Arrive, Msg_Receipt* and *T_Begin* hooks are as described in the normal phase.

5.7.2.4 The blocked phase (figure 5.28)

Components which are "blocked" must "unblock" if they receive a request from other BSet members. This is done by checking the identifiers of new messages against those stored in the BSet and "unblocking" if they match. The BSet variable is updated as new components join the set.

```
Blocked_hooks ()
  case C_MSG_BSet_Add : /* BSet_Add , n_id */
     BSet = BSet ∪ { msg.n_id }
     if ( arivd_msgs [msg.n_id] > 0 )
        C_state = UnBlocked
        install_hooks ( Blocking_hooks )
        UnBlock ()
  case C_MSG_Freeze : /* Freeze */
     BSet = {}
  case C_MSG_Resume : /* Resume */
     BSet = {}
     C_state = Normal
     install_hooks ( Normal_hooks )
     UnBlock ()
  case Msg_Ariv : /* msg */
     arivd_msgs [msg.n_id] = arivd_msgs [msg.n_id] + 1
     if ( msg.n_id ∈ BSet )
        C_state = UnBlocked
        install_hooks ( Blocking_hooks )
        UnBlock ()
```

Figure 5.28: Hook pseudo code for the "blocked" phase

"blocked" components need to keep abreast of additions to the BSet, and thus update their BSet variable as the membership changes. However as new components are added to the BSet, "blocked" components must check to see if there are any pending requests from these new members. If there are, then the component must unblock and continue processing.

As we described in section 5.7.1 once all the BSet members have "blocked" the

configuration object sends a "freeze" message to the Original_BSet members whilst directing those in the Extended_BSet to "resume" their normal execution. Therefore upon receiving the "freeze" control message "blocked" components clear their *BSet* variable and thereby ensure that they will not "unblock" until they receive the "resume" control message from the configuration object.

There are no code fragments for the *Msg_Receive, T_Begin* or *T_End* hooks as a "blocked" component's execution is suspended and it can not carry out any of the above actions.

Note that there is no need for the CO to be informed when a "blocked" component "unblocks". This is because it is guaranteed that the component will return to the "blocked" state as soon as it has serviced the unblocking request.

5.7.3 Programmer participation

In section §5.3 we assumed that it was evident from the application code when a component is about to begin a transaction and when a transaction terminates. In practice however this is not possible without help from the component programmer. The mechanism we suggest involves the programmer manually tagging the point of transaction initiation and termination in the programming phase.

Chapter six proposes one possible mechanism for accomplishing this task. This mechanism is used in the implementation of the case study presented in chapter seven.

5.7.4 Transient bindings

Recall that the configuration database (CDB), as described in section §3.3, did not keep track of transient bindings which were created as a result of components exchanging references to communication end-points in their messages. The view taken in chapter three was that these were transient binding created for the duration of short-term interactions between components and that since reconfiguration programmers were interested in the long-term behaviour of the application these bindings did not concern them.

Although this is perfectly valid, it poses a problem for the implementation of our avoidance scheme. Recall that the expansion of the BSet relied on priming all components that BSet members can initiate transactions on directly. As things stand, the CO only has access to the configuration information maintained in the CDB which excludes these bindings. This is unsatisfactory as BSet members may

now initiate transactions with components which, due to the lack of transient binding information, have been excluded from the PSet. Consequently the BSet will not expand to include components which fall into the (a) and (b) categories depicted in figure 5.5, and therefore negate the reachability argument of the reconfigurable state.

Banning the use of transient binding however is not an adequate solution since it would have a detrimental effect on the dynamic behaviour of distributed applications. Attempting to keep track of these bindings in the CDB is also impractical as the performance impact of communicating with the CDB each time a communication link needs to be established or broken is unacceptable.

Our solution is to require that this information is maintained in an accessible form in the components themselves. Thus when a component is sent the "block" message by the CO, as part of the component interactions the transient bindings information is sent back. Keeping track of such bindings is not difficult. Communication objects are almost invariably system provided classes or derived from system provided classes. Extra code can be added to the constructor and destructors of these objects such that their creation and deletion updates the relevant component variables. Some degree of discipline is required however, as for best results these dynamically created communication end-points should be discarded by the component programmer as soon as the interactions which use them have terminated.

5.7.5 Open systems

Closed distributed applications consists of a number of components which cooperate with each other in the performance of some arbitrary task. Communication with these components however is constrained such that components external to the application cannot interact with any of its member components. In contrast, open distributed systems designate and announce services which they willingly provide for external clients.

So far in our discussions we have assumed that the consistency manager has the ability to interact with and to some extent control the behaviour of the components in the distributed application e.g. block and unblock their execution. This however is not true of open distributed applications, as the consistency manager does not have direct control over the behaviour of external clients.

One possible solution is to require that external clients renew their association with the open distributed system at the end of each transaction, where it is understood that a bind request to an exported service may be refused. In this way,

external requests to affected parts of the open distributed system can be refused and thus stem the flow of new externally initiated transactions.

5.7.6 Overhead and performance

To implement our method certain runtime overheads had to be introduced, however the majority of the mechanisms by which the safe state is enforced remain inactive during normal system execution and therefore do not hinder system performance.

The housekeeping operations performed by the management hooks during the normal phase, and subsequent stages of the scheme are quite minimal and computationally non-intensive. Similarly, the actions performed by the CO are not computationally bound as it relies on simple CDB lookups in order to coordinate the necessary communications which direct components in the affected region towards the reconfigurable state. The communication cost of the protocol is also relatively small as all communication initiated by the CO occurs in parallel. For example the sending of the Block message to the Orig_BSet membership which occurs in the initial phase of the algorithm is done concurrently. Thus the delay before all members of the Orig_BSet receive the "block" message is the time taken by the slowest single message to arrive at its destination, and not the cumulative arrival times. This is true about the sending of "prime", "freeze", "clear", "BSet_Add" messages as well. In fact the determinant factor in the performance of the protocol is the time taken by components to react to the configuration management messages and not vice versa. Thus component behaviour drives the protocol i.e. the speed with which a system moves towards the reconfigurable state is a function of the interdependence and completion speeds of its transactions.

5.8 Discussion

The aim of the presented avoidance scheme was to reduce the level of programmer and system overhead whilst reducing the disruption to the system. With regards to reducing programmer involvement, all three stages of avoidance schemes, namely the identification of the region of disruption, the coercion of components into the safe-state, and the detection of the reconfigurable states have been automated (see section §2.2.2 page 31). As explained in section §5.7.6 this has been accomplished by imposing very little overhead on the system.

Even though the systems which we consider are a subset of those catered for in [KM90] the methods are comparable in their generality. Our method however

requires less programmer interaction and, due to its use of the dynamic flow of transactions at run-time, will always perform at least as well as that in [KM90] with respect to minimising the system disruption. At best it can reduce it to the minimum as identified by the reconfiguration rules in section 5.2.

Our method is suited to general reconfigurations and not just for that of reconfiguring loosely-coupled clients between server peers as in [Kin93]. In fact the method presented in [Kin93] could be considered to be the special application of our method to client server systems; where proxies attempt to determine the point of transaction begin and end from the content of the messages interchanged.

Our notion of a transaction is fairly unrestrictive and imposes little overhead on the programmer. This is in contrast to methods which require recovery mechanisms and user defined state restore and capture facilities. As a result of these facilities such systems can to some extent be reconfigured at will, but the overheads imposed are unnecessary for a large class of problems.

It is worth noting however, that tagging the point of transaction initiation and termination is already common practice in systems which support atomic actions and therefore these systems can be used in conjunction with those conforming to our less demanding requirements and reconfigured together.

The major restriction that our method imposes on applications is that they should not interleave transactions.[6] It may seem that by preventing this, we limit the overall achievable concurrency in the system. However we argue that given the general trend towards operating system support for lightweight processes, improved communications facilities and multi-processor machines, heavyweight processes which interleave many tasks are generally unnecessary and bad practice. Not only does the non-interleaving style of programming make debugging and maintaining such software simpler but it also allows the operating system to make more optimal decisions regarding the use of the resources.

An offshoot of the non-interleaving requirement was that cycles in transaction dependencies could lead to deadlock. System designers had to take this into account and design their applications such that they did not exhibit cyclic dependencies. It is a well known fact that writing distributed or parallel programs is a demanding task. In particular designers of any non-trivial distributed or parallel system must constantly be on the look out to avoid situations which lead to deadlock and livelock. To do this it is necessary to trace the often complex and involved interactions between concurrently executing processes and consider all the possible cases which may arise.

[6]This was necessary to ensure that a reconfigurable state would eventually be reached.

This is a difficult enough task when the interaction do not have cyclic dependencies. Together with the arguments in favour of lightweight processes presented above, we are lead to believe that the non-cyclic restriction is not a significant one.

In any case, as mentioned in section 5.5.2 it is possible to group components which need to exhibit behaviour banned in the assumptions in indivisible composite components.

5.8.1 Timeliness guarantees

A general characteristic of avoidance schemes and consequently that of the approach proposed here, is that reconfigurations cannot go ahead until after the safe-state has come about. This becomes a problem when dealing with applications whose reconfigurations have hard deadlines associated with them.

Since avoidance schemes work on the principal of preventing new transactions from beginning, whilst allowing existing ones to terminate, the quicker transactions terminate, the faster the system will move towards a reconfigurable state. Systems with fine grain transactions are therefore likely to move towards a reconfigurable state much more quickly than ones with coarse grain transactions. Thus in order to improve the "reconfigurable behaviour" of their applications, component programmers might consider decreasing the granularity of the transactions used in the system. Aside from this however there is very little one can do to speed up the transition to the reconfigurable state.

Predicting the time taken to reach the reconfigurable state is also quite difficult as it depends on the runtime behaviour of the application and the particular reconfiguration which is being considered. Static analysis of the application will be able to place some theoretical upper and lower bounds on this period, however this information is of questionable value. Recovery approaches suffer from similar difficulties since the number of transactions which need to be rolled back also depend on the runtime behaviour of the component[7] and the reconfiguration which is being considered.

As things stand, neither avoidance schemes nor recovery approaches can give firm guarantees about when reconfigurations will be carried out. However, in general recovery schemes are likely to behave in a more predictable manner than avoidance schemes. Whether this issue will turn out to be a determining factor in the choice of mutual-consistency schemes remains to be seen.

[7] e.g cascading aborts.

5.8.2 More fine grain blocking ?

The granularity of blocking in our scheme is at the component level i.e. whole components are blocked in order to prevent them from initiating certain offending transactions. Given that components can be involved in any of a number of transactions, it would seem that system disruption can be further reduced with the adoption of finer grain blocking.

A candidate strategy would be to block components only when they attempted to initiate transactions which require the use of entities[8] undergoing change. However a major drawback of such a scheme is that it requires access to detailed transaction dependency information which can not be deduced automatically from the application code (except in the case of very simple programs) and must be provided manually by the component programmer. This aside, the implicit assumption which makes such an approach worthwhile is that computational components are complex entities which accomplish a number of different tasks. Thus, where a transaction blocking scheme would impact only a small portion of these tasks, component blocking would stop progress on all of them.

This however is contrary to our intuition that distributed applications are better written as groups of simple lightweight processes, rather than composed of a small number of heavyweight ones. We suspect that in the former case, transaction blocking and component blocking would yield similar results, the only difference being that transaction blocking would require more support.

5.9 Chapter Summary

This chapter proposed a scheme, suited to a particular class of distributed applications, which automatically imposes the safe state necessary for ensuring mutual-consistency in the presence of dynamic reconfiguration. It was shown that labeled transition systems in conjunction with automated analysis tools can be used to verify that a target system adheres to the requirements of the proposed consistency scheme. The approach takes advantage of the runtime behaviour of the application to minimise the disruption to the remainder of the system and compares favourably with existing schemes in this regard. It is largely transparent to the application level programmer and whilst guaranteeing the reachability of the safe state, no assumptions are made about the ability of components to recover from failed transactions.

[8] i.e. communication links, or components.

Furthermore, the safe-state imposed is such that it supports the invariant preservation mechanism proposed in the previous chapter. The scheme can be used to cater for multiple concurrent reconfigurations and can be used in conjunction with composite components.

Chapter six describes an implementation of this scheme. The implementation is based on the Regis distributed programming environment [MDK94a], and embeds the extra functionality required by the approach into the existing communication and process classes, thereby hiding such details from the users.

Chapter 6

Implementation Issues and Prototype

The previous chapters set out a framework for implementing a dynamic reconfiguration service and put forward mechanisms for specifying change, reinstating invalidated application invariants and maintaining the mutually-consistent-states property during reconfiguration. This chapter begins by addressing outstanding issues which implementing such a reconfiguration service entails and goes on to describe a prototype implementation of the proposed reconfiguration service.

6.1 Implementation Issues

Chapter three gave a general description of how the Reconfiguration Manager (RM), Consistency Manager (CM) and the Configuration Database (CDB), cooperated with each other in bringing about dynamic change. The following sections take a closer look at this interactions and resolve issues raised when implementing the proposed mechanisms for the specification of change, preservation of application invariants, and the maintenance of the mutually-consistent-states property. Additionally concurrency within single and across multiple reconfigurations is addressed.

6.1.1 Flattening reconfiguration scripts

Recall that prior to executing a change script the reconfiguration manager needed to supply the consistency manager with the list of components which had to move into the safe-state. This was necessary as the reconfigurable-state and hence the safe execution of the envisaged changes were conditional on these components making

this transition. Chapter five provided the "reconfiguration rules" (§5.2 on page 123) which when applied to a concrete list of parameterised reconfiguration actions and the current system configuration, would yield the minimum set of components (original block-set members) that needed to move into the generalised safe-state.

Unfortunately, the reconfiguration manager component, does not have access to a flat list of parameterised reconfiguration actions. Change descriptions are specified as procedures in an imperative language and thus possess the complex runtime behaviour which a full complement of nested looping and conditional constructs facilitate. The flat list of parameterised reconfiguration actions needed by the reconfiguration rules are generated once the imperative change description has been executed.

To better understand the difficulty involved in deducing the flat list of parameterised reconfiguration actions from an imperative change description, consider the reconfiguration script in figure 6.1. The imperative change script operates over a subsystem consisting of a number of component chains such as that in figure 6.2.a. The purpose of the reconfiguration is to reduce the length of each of the chains by removing every n^{th} element, where n is a runtime script parameter.

Figure 6.2.b shows the subsystem of figure 6.2.a subsequent to the execution of the `removeEveryNthElmnts` script with the runtime script parameter n set to the integer value 2. As can be seen, every second element in the chains has been removed and the chain re-established. Figure 6.2.c depicts the same subsystem subsequent to the execution of the identical reconfiguration script but this time with the runtime script parameter n set to 5.

Comparing the flattened list of reconfiguration actions given for both reconfigurations and the resulting configurations, it is clear that the script parameters have a dramatic impact on the resultant actions. Of course, the system configuration plays an equally important part, as the behaviour of such scripts is also governed by the configuration of the system. For example had the chain C4 been only four or less elements long, the second reconfiguration would not have altered the application configuration at all.

6.1.1.1 Solution: a two pass execution process

The way to resolve this problem is for the reconfiguration manager to execute the change script in two passes. In the first pass the imperative change scripts are executed, but the results of the reconfiguration actions are not committed to the CDB. Instead they are used to generate the flat list of parameterised reconfiguration actions. This list is then forwarded to the consistency manager which uses

```
public void removeEveryNthElmnts ( int n )
{
  // get a handle to the master component
Master master = ClassInstances ( "Master" ) ;
  // for every chain on master do ...
for ( int i = 0 ; i <= master.numChains ; i++ )
  {
    // set up current and previous list references
  Elmnt current = master.out[i].nodeAt () ;
  Elmnt prev = current ;
  int elmntCount = 1 ;
    // search for nth element
  while ( current != null )
    {
    Elmnt next = current.out.nodeAt () ;
        // if nth elmnt - remove
    if ( elmntCount == n )
      prev.out.unlink () ;
      prev.out.bind ( next.in );
      current.out.unlink () ;
      current.delete () ;
      current = next ;
      elmntCount = 1 ;
    else // go to next elmnt in chain
      prev = current ;
      current = next ;
      elmntCount++ ;
    }
  }
}
```

Figure 6.1: removeEveryNthElmnts change script

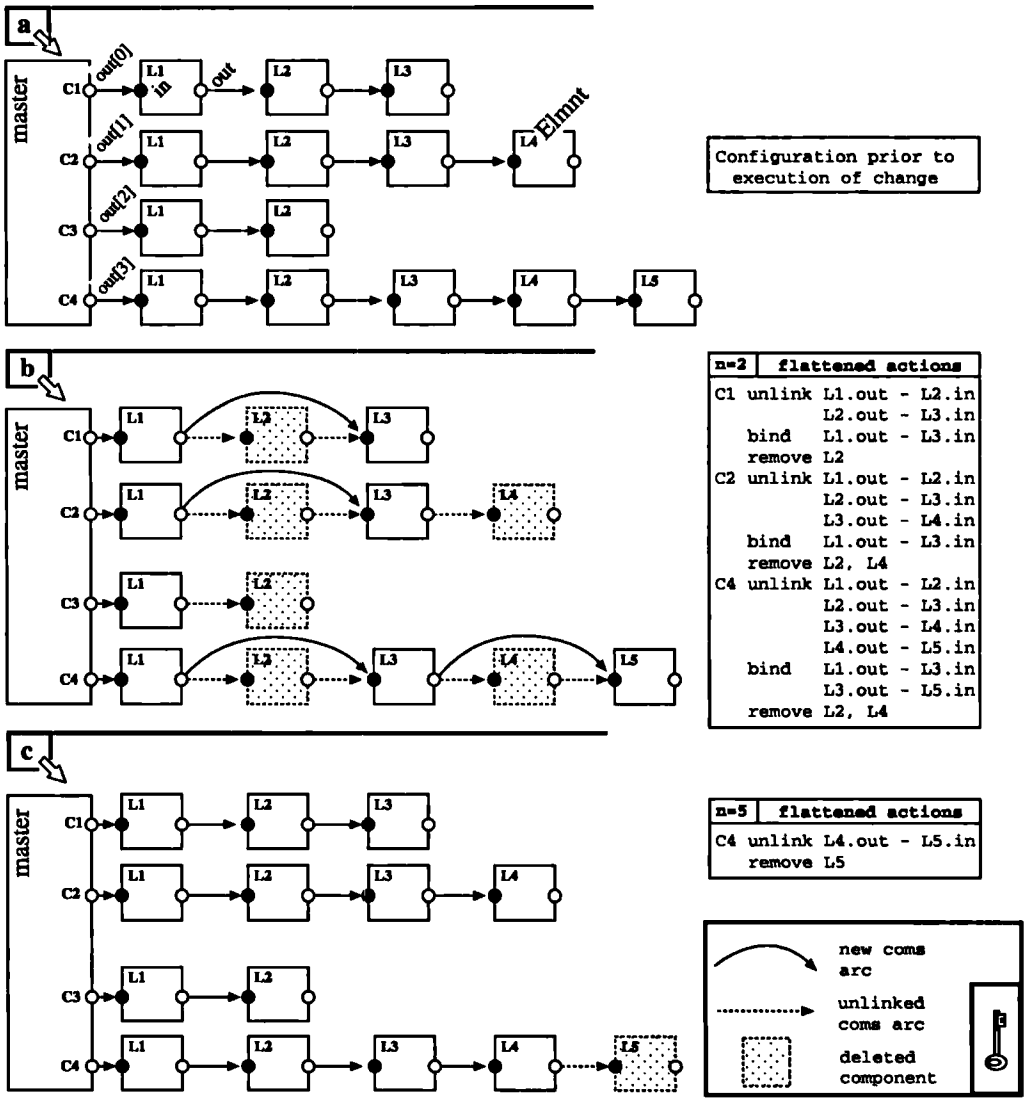


Figure 6.2: Component chains

it in conjunction with the reconfiguration rules to identify the original block-set membership and coordinates the necessary state transitions which bring about the reconfigurable-state. Once this occurs the RM is signalled and the second phase of the script execution begins. This time reconfiguration actions are committed to the CDB and affect configuration changes onto the application. Note that if scripts consisted of reconfiguration actions alone, the second pass would not need to execute the script again and would simply commit the results of the first pass to the database. However reconfiguration scripts also consist of state access method invocations which as we shall see in section §6.1.1.1 pose problems that can only be resolved through a second execution of the reconfiguration script.

Given that two separate executions of the script must occur, the necessary assumption is that reconfiguration scripts behave deterministically between the first and second passes. This is necessary as otherwise the second pass execution may affect components which were not included in the first pass execution and are thus not in the reconfigurable-state. To guarantee this property, the execution path taken by scripts must be governed only by their runtime parameters and the application configuration. The RM certainly has access to the script parameters and will therefore use identical values for both passes. Since reconfigurations are the only way the system configuration can be altered, then it is also possible to keep the system configuration fixed during the execution of the first and second passes of the change script.

The first pass execution of reconfiguration scripts however is not as trivial as it may initially seem. The following sections describe the difficulties which arise and put forward solutions for resolving them.

Reading dirty configuration information Consider that in the imperative programming style adopted for the expression of reconfiguration changes, the task at hand is accomplished through the execution of a sequence of reconfiguration actions. These reconfiguration actions however are not independent in the sense that partial results from previous statements are visible to subsequent ones. This is due to the decision in chapter four to adopt configuration queries as the means through which change scripts select reconfiguration targets. Thus, it is possible for reconfiguration actions to query the CDB for reconfiguration targets which have been modified by a previous statements.

To illustrate this point consider a planned change for the subsystem depicted in figure 6.3.a which involves joining all existing chain elements onto the end of

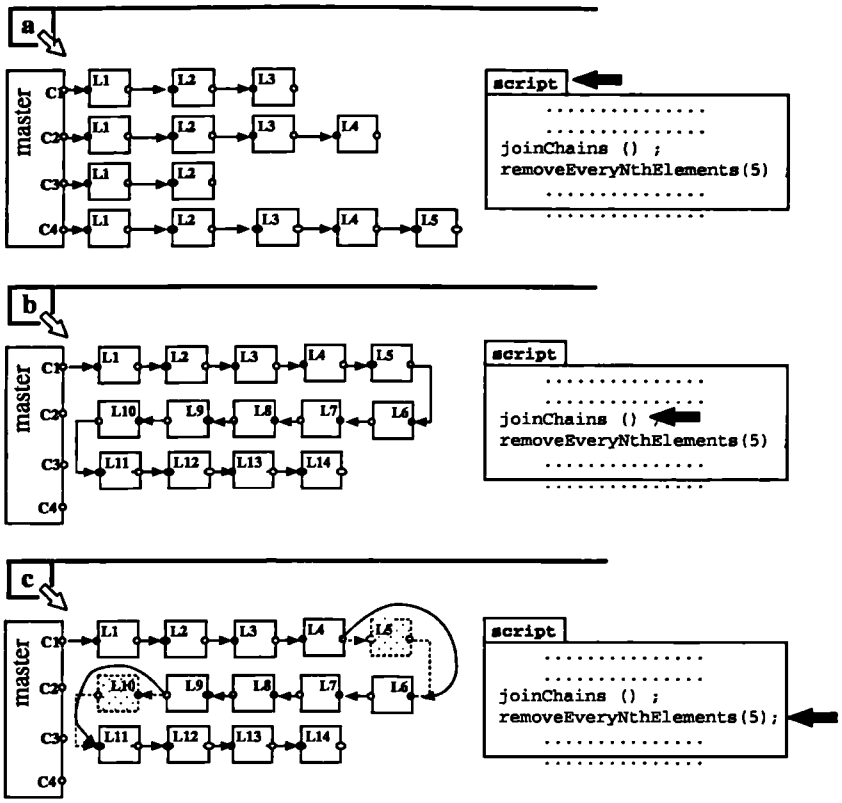


Figure 6.3: A sample reconfiguration

the C1 chain and then deleting every fifth element on the extended chain. To do this the change programmer has made use of two existing reconfiguration methods `joinChains` and the `removeEveryNthElmnts` method. Figures 6.3.b and 6.3.c depict the subsystem configuration after the execution of the `joinChains` and `removeEveryNthElmnts` reconfiguration methods respectively.

The point to notice is that the `removeEveryNthElmnts` reconfiguration method operated on the modified application configuration created by the `joinChains` method and not that of the original application configuration. When the reconfiguration script is actually executed, the CDB queries, such as that in `removeEveryNthElmnts` which enables the `for` loop to cycle through the chain elements (underlined in figure 6.1), report on the modified application configuration established by previous reconfiguration actions.

Therefore during the first pass execution, reconfiguration actions must be committed to a temporary version of the database and subsequent configuration queries directed to this copy. Of course it is not necessary to make a complete copy of the

database and only modified information needs to be duplicated.

State access methods State access methods pose a particularly difficult problem when it comes to the first pass execution of reconfiguration scripts. This is because state query methods return results which can be used to guide the flow of control in the reconfiguration script. Consider the code segment in figure 6.4.

```
if ( a.getStatus () )
  b.setStatus ( val1 ) ;
else
  c.setStatus ( val2 ) ;
```

Figure 6.4: State access methods

Depending on the value returned by the `a.getStatus ()` method either the `setStatus` method of component `b` or that of `c` will be invoked. However during the first pass it is not possible to determine which of these statements will be executed. State access methods query dynamic component state which if queried when the component is not in the generalised safe-state may return either intermediate and hence inconsistent state, or information which will become out of date by the time the second pass is executed. The solution is to take a conservative stance and include both `b` and `c` in the original block-set membership. This complicates the first pass execution cycle as the RM needs to use a backtracking [AHU87] algorithm to traverse all of the possible execution paths of a given reconfiguration script.

When using state access methods, it is important to remember that they were included in the reconfiguration service for the express purpose of reinstating invalidated application invariants. Using them for any other purpose, such as controlling the reconfiguration path taken by a change script is flawed as it results in unnecessarily large disruptions to the application. The reconfiguration script in figure 6.5 has been written in this way and results in components a, b, c, d, and e all to be included in the original block-set. While depending on the configuration path taken, only half those elements will be directly affected by the change.

6.1.2 Concurrency and reconfigurations

In Chapter three §3.3.1 the responsibility of facilitating concurrency both within single and across multiple reconfiguration scripts was assigned to the CDB. This sections details how the CDB accomplishes these tasks.

```
if ( a.getXStatus () == true )
  b.delete () ;
  c.delete () ;
else
  d.delete () ;
  e.delete () ;
```

Figure 6.5: Flawed use of access methods

Concurrency within single scripts - The CDB facilitates concurrency within a single script by allowing reconfiguration actions to succeed immediately by updating the database records. These updates result in the dispatch of asynchronous control messages to the underlying distributed runtime system which actually bring about the configuration changes. Thus even though the RM executes the reconfiguration scripts sequentially, the execution of the reconfiguration actions is done in parallel by the underlying runtime system. There is no need for the execution of the reconfiguration script to await the actual realisation of these change actions as the script operates on the CDB which reflects the changes immediately. Of course the CDB does track the actual configuration changes and once confirmation messages arrive, its internal records are updated with the concrete representations of the reconfiguration targets.

There are times however where reconfiguration actions need to synchronise with each other. The most obvious cases are where reconfiguration actions are invoked on objects which exist in the database but are pending actual creation. In such cases `bind`, `unlink` and `delete` actions must wait until the CDB receives the actual creation message from the underlying runtime system. The invocation of state-access methods are also carried out synchronously partly because the execution path of the change script depends on the returned values but also because these may have side-effects which can be observed by other state access method invocations (e.g. an access method on target A may result in the creation of a file, which is read as the result of another access method on a reconfiguration target B). Change actions which operate on the same reconfiguration targets are executed in the same order as they appear in the script.

Concurrency across multiple reconfigurations - The concurrent execution of multiple reconfigurations can be facilitated by treating them as transactions and relying on slightly modified versions of standard concurrency control mechanisms to

resolve conflicts. In database terms, the basic configuration actions (`bind`, `unlink`, `create` and `remove`) are categorised as "write" operations since they update the CDB. Database queries fall into the "read" operation category. As long as state-access methods are clearly divided between "get" and "set" methods they can be treated as "read" and "write" operations respectively. Of course the requirement is that "get" methods do not have any observable side-effects by other "get" or "set" methods. Strictly speaking state-access methods are not database operations at all as they query and update state on the actual running components. Nevertheless, since they are a part of the reconfiguration transactions their conflicts need to be resolved and the CDB is the most logical entity for performing this task.

Concurrency control mechanisms often rely on rolling back the results of transactions which have started but could not commit due to conflicts with other scheduled transactions. In the context of reconfiguration transactions however it is not viable to rollback actual reconfigurations and therefore it is necessary for the concurrency control mechanism to decide on the serialisable schedule of reconfiguration transactions before any actual changes are made to the application.

Recall that in the two pass execution cycle of reconfiguration scripts, the results of the first pass execution are only temporary and are discarded prior to the second pass execution which actually commits changes to the CDB. This fact can be exploited so that the chosen concurrency control mechanism can select serialisable schedules during the first pass executions and allow them to commit changes to the CDB as and when they are executed. In effect the modifications to existing concurrency control mechanisms need to reflect that reconfiguration transactions posses two distinct checkpoints. The first occurs at the end of the first pass execution which if allowed to succeed, signals to the concurrency control mechanism that the transaction must be allowed to complete. The second occurs at the end of the second pass execution and signals to the concurrency control mechanism that the transaction has completed and internal data structures such as locks etc. can be released.

6.2 Prototype

This section describes a prototype implementation of the proposed reconfiguration service. The implemented reconfiguration environment incorporates the mutual-consistency mechanism of chapter five and the state access invocation support advocated in chapter four. The prototype implementation is based on the Regis

[MDK94b] distributed runtime system with Java [GM95] serving as the reconfiguration language.

The thesis adopted the communication oriented model, as it forms the basis over which other model abstractions are built and thus affords the most generality. Regis was chosen as it is an instance of this runtime model, and also because sources for the system were readily available and could be changed as necessary. Java satisfies chapter four's requirements of object orientation and expressiveness and therefore qualified as a suitable reconfiguration language.

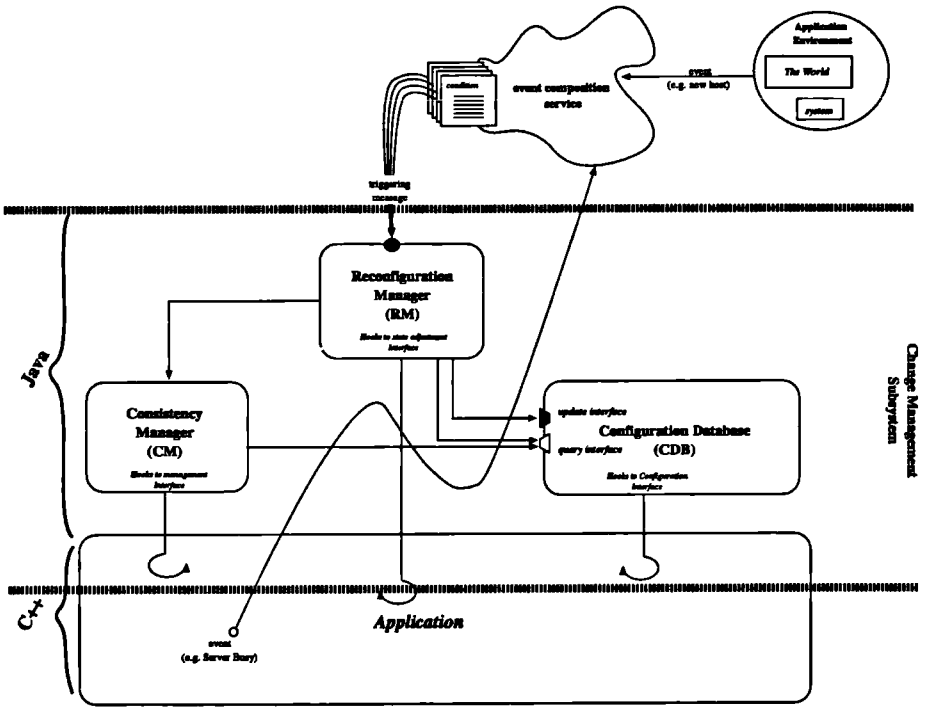


Figure 6.6: The Framework

The implementation follows the logical partitioning set out in figure 6.6. The runtime hooks interact directly with the application components and are thus implemented in C++[Str93] which is the language used by Regis applications. The remainder of the reconfiguration environment which consists of the consistency manager (CM), configuration database (CDB) and the reconfiguration manager (RM) were all implemented in Java. This facilitated better integration with the reconfiguration language and also minimised any future porting costs. The implementation consists of approximately ten thousand lines of code, split evenly between C++ and Java sources.

The following sections describe how each of the elements identified in the framework was implemented. The reconfiguration service which we have proposed exists

on a per-application basis and thus the CM, CDB and RM components are created along side other Regis components at system initialisation time.

6.2.1 Regis

Before we describe the prototype implementation it is necessary to give a brief overview of Regis. Regis is a distributed runtime library written in C++ and provides threading, and communication in a distributed environment.

Threading is facilitated through the `process` class which components need to subclass. Regis processes are light-weight threads which exist within a single Unix process. Within each heavy-weight Unix process, henceforth referred to as a "Regis node", a non-preemptive scheduling policy is implemented. Thus the thread of execution is only switched when either a thread yields voluntarily or blocks waiting for communication. Distributed applications written using Regis, consist of many light-weight processes (components) contained in a number of Regis nodes running on different hosts.

Communication is provided through a set of template classes which facilitate the exchange of typed messages between components. The `port<T>` communication class defines a communication endpoint for receiving messages of type `T`. Upon arrival messages are queued on the port's input queue and delivered to the component for consumption when the `port<T>::in` method is invoked. The `portref<T>` class is the counterpart of `port<T>` and is used to send synchronous (`portref<T>::out`) or asynchronous (`portref<T>::send`) messages. A `portref` is a reference to a particular `port`. As part of the application elaboration portrefs on components are bound to relevant ports on other components thus establishing a communication link. The Regis communication system guarantees in-sequence delivery of messages.

6.2.2 Runtime hooks

As can be seen from figure 6.6 the reconfiguration service components - CM, RM and CDB each have corresponding hooks into the application which allow them to accomplish their respective tasks. Ideally the inclusion of these hooks into the system should be transparent to the application programmer. This is a highly desirable property as in this way the requirements which the dynamic reconfiguration service places on the runtime system can be discharged without having an impact on the application programmer.

Although the hooks required by the CM, RM and CDB need to exist for every

individual Regis component, the interactions they facilitate are not component specific. Thus in the prototype the desired transparency is achieved by providing alternative reconfiguration counterparts to the Regis `process`, `port<T>` and `portref<T>` classes which include the generic hook implementations. These reconfiguration classes are `rcprocess`, `rcport<T>` and `rcportref<T>` and can be used in exactly the same way as the original Regis classes by the application programmer.

The `rcprocess` class is depicted in figure 6.7. It inherits from the Regis `process` class and thus has all the functionality of its parent. In addition however it defines three communication ports `cmport`, `rmport` and `cdbport` which are used for interactions with the CM, RM and the CDB respectively. As part of the component creation process `rcprocess` sends a registration message to the CDB which signifies that it was created successfully and also registers the addresses if its `cm`, `rm` and `cdb` ports with the CDB for future interactions.

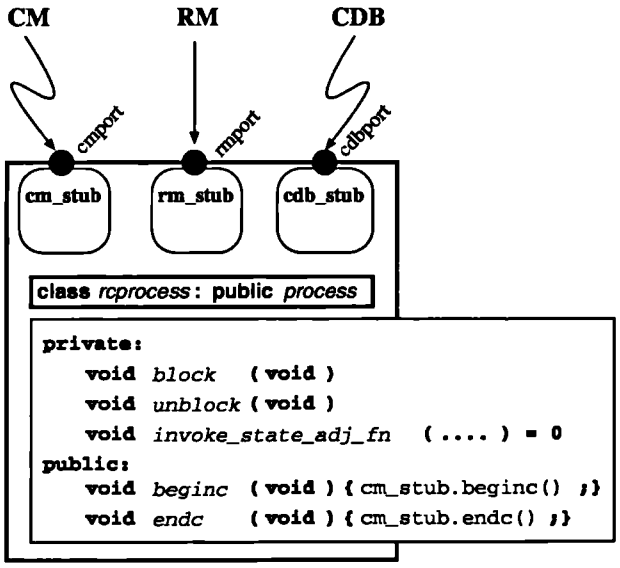


Figure 6.7: rcprocess class

Considering that the speed with which components process CM, RM and CDB messages govern the speed with which the application is reconfigured, it is desirable that these messages be processed quickly. To this end the `cm`, `rm` and `cdb` ports are instances of a specialised port class `iportbase<T>` which rather than queuing messages, processes them on arrival. The processing of these messages however is carried out by the `process_msg` of the corresponding stub components, `cm_stub`, `rm_stub` and `cdb_stub` which are additional members of the `rcprocess` class. The

operations of these stub components will be discussed in the next sections.

The `rcprocess` class also defines a number of additional methods. Notable among these are the `block`, `unblock`, `invoke_state_adj_fn`[1], `beginc` and `endc` methods.

The `block` and `unblock` methods are used by the `cm_stub` to stop and restart the component's thread of execution and the `invoke_state_adj_fn` method is used by the `rm_stub` to invoke user defined access-methods. These methods are `private` and thus not accessible to the application programmer whose component class is a subclass of `rcprocess`.

The `beginc` and `endc` methods however are defined as public methods in the `rcprocess` class and are therefore accessible to the application programmer. These methods are the means through which the component signals its state-transitions to the `cm_stub`. In fact as can be seen from figure 6.7 these methods simply invoke corresponding methods in the component's `cm_stub`. These methods will be discussed in more detail in section §6.2.2.2. But briefly, as discussed in chapter five §5.7.3, the application programmer encloses critical regions of the application code between `beginc` and `endc` method invocations which prevents reconfigurations from altering its communication partners while it is involved in a transaction. The case-study in chapter seven illustrates how these methods are used in practice.

6.2.2.1 CDB hooks - cdb_stub

The primary task of the `cdb_stub` is to process messages from the CDB. The arrival of messages on the `cdbport` (see figure 6.7) trigger the execution of the `cdb_stub::process_msg` method which does just that. There are three different types of requests messages which the CDB sends to components. These are `Delete`, `Bind` and `PortInfo`.

- **The `Delete` request -** is sent to the stub when the component has been targeted for removal.

  In this case a confirmation message is sent to the CDB before the component is terminated and its runtime representations discarded.

- **The `Bind` request -** is accompanied with a portref index and a port-reference value.

[1]The `invoke_state_adj_fn` method is in fact a pure virtual function and it is actually defined in the `rcprocessT` template class described later.

The CDB identifies ports and portrefs on a component's communication interface by unique integer indecies. Thus the portref index supplied as part of the bind request identifies a unique portref on the components communication interface. Once the selected portref is located and assigned the supplied value, the binding between the local portref and the remote port is established. A confirmation message is sent back to the CDB to signal the success or failure of the operation.

- **The `PortInfo` request -** is accompanied with a port index.

  In this case the stub locates the port, creates a port-reference object from it and sends the portref back to the CDB. A `null` portref signals the failure of the operation.

The `PortInfo` and `Bind` request are used in pairs by the CDB to realise bind operations. A `PortInfo` message is first sent to the cdb stub of the component which owns the port object. The returned portref is then sent as part of a `Bind` message to the `cdb_stub` of the component which owns the portref, thus establishing the binding.

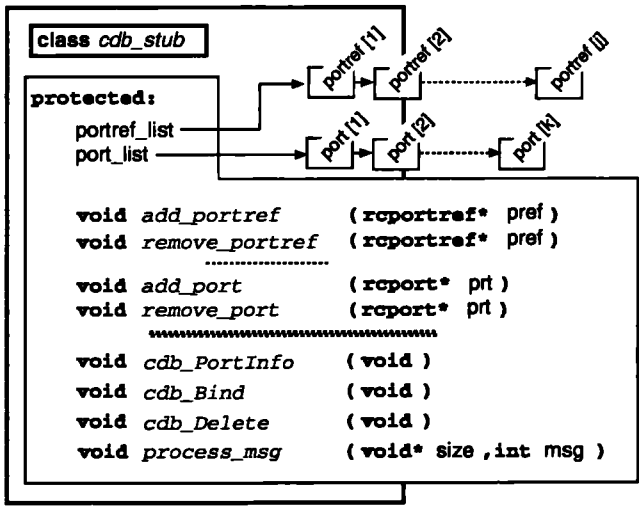


Figure 6.8: cdb_stub class

As can be seen from figure 6.8 the `cdb_stub` defines two list variables `portref_list` and `port_list` as well as "add" and "remove" methods for ports and portrefs. The `add_port` and `remove_port` methods operate on the `port_list` variable, adding to, or removing the supplied parameter from the list. The "add" and "remove" methods for portrefs perform an identical function on the `portref_list`.

These methods are invoked by the constructor and destructors of the `rcport<T>` and `rcportref<T>` classes mentioned in section §6.2.2. This is necessary so that the `cdb_stub` has access to the communication end-points of the component it is managing. The order in which ports and portrefs are added to their respective lists corresponds to the integer identifiers assigned to them in the CDB.

6.2.2.2 CM hooks - cm_stub

Together, the stub and the CM ensure that reconfigurations do not leave application components in mutually inconsistent states. As mentioned in chapter three the actual function of the `cm_stub` depends largely on the adopted mutual-consistency mechanism. In the prototype the CM and `cm_stub` implement the mutual-consistency mechanism proposed in chapter four.

In this scheme, the stub is purely reactive in behaviour and is activated only when certain events occur. The activation points, as described in section §5.7.2, are:

- **Transaction begin :** The point just before the component is about to become involved in a transaction.
- **Transaction end :** The point immediately after the termination of a transaction.
- **Message arrival :** When a message arrives at any of the component's communication ports.
- **Message receipt :** When the component accepts a message.

Table 6.1 gives the mapping between the triggering event and the invoked stub method. The `beginc` and `endc` methods are called when the application programmer invokes the `rcprocess` versions of these methods. It is through the side-effects of these two methods that the stub can determine whether the component is in an `Active` or `Idle` transaction state.

Transaction begin	**void beginc (void)**
Transaction end	**void endc (void)**
Message arrival	**void msg_arrive (proc_id& msg_id)**
Message receipt	**void msg_receipt (proc_id& msg_id)**

Table 6.1: Event activation methods

The msg_arrive and msg_receipt methods are invoked by the modified communication rcport<T> class. The msg_arrive method is triggered when a message is added to the input queue of an rcport<T> object and the msg_receipt method is called each time the rcport<T>::in method is called. The proc_id parameter supplied to these methods is the unique component identifier of the sender which is included in all message headers.

The actions performed by the stub at the activation points however, depend on its current management state. The stub begins its execution in the Normal management state but, depending on control messages from the CM or runtime events, it may change state to the Primed, Blocking or Blocked states. Since for each state the activation points result in a different stub behaviour, four different classes have been defined. Each of these classes encapsulates the stub's behaviour in one of the four management states. The stub contains instances of these classes in the form of the normal, primed, blocking and blocked hook variables.

As can be seen from figure 6.9 invocation of the stub activation methods simply map to identical calls on the object referenced by the curr_hook pointer member. The use of curr_hook introduces a level of indirection which allows the external interface of the stub to remain fixed while its behaviour changes. Such behavioural changes are brought about by the install_hooks method which changes the object referenced by curr_hook. Thus, depending on the management state of the stub, the curr_hook member is made to reference one of the hook variables normal, primed, blocking and blocked. The functionality of the hook classes follow the pseudo code given in section §5.7.2.

Other notable members of cm_stub include, node_state which records the transaction state of the components , PSet and BSet which correspond to the prime and block sets and the two list variables arivd_msgs and active_msgs which keep track of messages which have arrived and messages which have been consumed.

6.2.2.3 RM hooks

The rm_stub's only function is to facilitate the invocation of user defined state-access methods. The invocation control messages arriving at the rmport contain a string representation of the function name which is to be invoked, a remote reference for returning the results of the method invocation, the number of arguments taken by the method and the marshaled arguments in a packed buffer. Upon receiving such a control message the stub calls the owning process' invoke_state_adj_fn. The signature for this methods is given below.

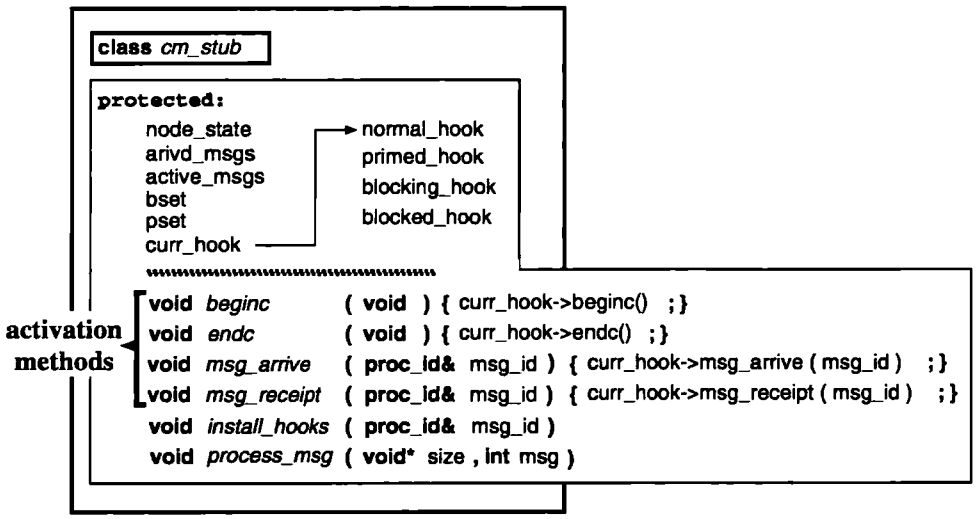


Figure 6.9: cm_stub class

invoke_state_adj_fn (**char** * fn_name, **int** num_args, unpacker & unpkr, packer & pkg)

The `unpacker` object `unpkr` contains the marshaled arguments and the `packer` object `pkg` is the packed buffer used to store the results of the invoked function. Packed buffers are used throughout the prototype wherever it has become necessary to transmit messages, the contents of which do not conform to a single data-type. The `packer` and `unpacker` classes enable controlled access to the contents of these buffers through the standard C++ input (<<) and output (>>) operators. Once the `invoke_state_adj_fn` returns, the `rm_stub` sends the `pkg` object, which now contains the results of the invocation, back to the RM.

State-access methods are defined by the application programmer as `public` member methods of their Regis components. These methods must conform to the following methods signature:

void *component*::*state_adj_fn* (**int** no_args, unpacker& u, packer& p)

It is the responsibility of the state-access method to extract the actual parameters of the function from the `unpacker` object `u` and then to put the results into the `packer` object `p`. Methods defined in this way are registered with the runtime system by invoking `add_st_adj_fn`. This method is a member of the `rcprocessT` template class which is the actual class user defined components inherit from (see figure 6.10). A template class was necessary as each user defined component class needed to have its own static list of state-access methods, rather than having a single

list for the whole application which could easily result in function name clashes. The `rcprocessT` class makes this possible and also provides the necessary functions to register, search for and invoke user defined member state-access methods.

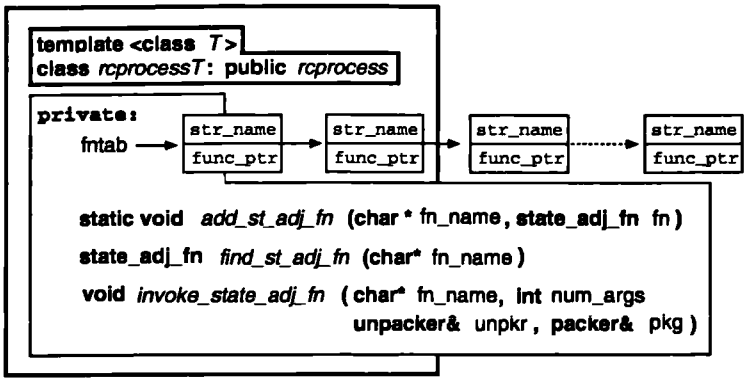


Figure 6.10: rcprocessT class

6.2.3 Regis and Java connectivity

Since the CM, CDB and RM were implemented in Java, it was necessary to facilitate communication between Java and Regis. To this end a runtime package mirroring that of Regis was written in Java such that messages could be exchanged easily between the two environments. This task was made complicated by the fact that Regis does not use an interface definition language and thus the format of the packets are architecture dependent.

The programmer interface in Java is very similar to that in Regis and communication is facilitated through Java variants of `port` and `portref` and `process` classes. A useful side-effect of this process was that Java components could now be included in the application configuration thus allowing Java and Regis components to interact and be bound together in a uniform manner.

6.2.4 RM

The tasks of the RM were to facilitate the expression of configuration and reconfiguration descriptions and to execute the reconfiguration scripts upon the arrival of triggering messages. In this section we will give an overview of how these tasks are supported by the prototype. The case-study described in chapter seven illustrates how these mechanisms are actually used.

6.2.4.1 (Re)configuration specifications

The expression of configuration and reconfiguration descriptions in Java are facilitated through a number of configuration classes. The classes visible to the reconfiguration programmer comprise `PNode`, `PPort`, `PPortref`, `CNode`, `CPort` and `CPortref`. Classes beginning with the letter "P" relate to primitive components and communication objects while those beginning with "C" relate to composite ones.

In order to specify the configuration representation of a primitive computational element, the application programmer defines a new Java class which inherits from `PNode`. The communication interface of the component is defined by declaring `PPort` and `PPortref` members to mirror those on the actual computational component. Any state-access methods supported by the computational elements are also defined in the body of the class as member functions. The configuration specification of the `Controller` component in chapter four (figure 4.15 page 104) gives a good indication of how these classes are used.

Primitive component constructors - The configuration classes have been written in such a way that their constructors have specific side-effects. In the case of `PNode` an update request is made to the CDB requesting the creation of an user-defined component record. Figure 6.11 depicts the constructor for the `Controller` primitive component.

```
public Controller (boolean frstPas, int loc, CNode pa, int reseedPrd )
  {
    super (pass, loc, pa, packParams(reseedPrd) ) ;

                    // create PPorts objects
    CUIn = new PPort ( this );          CLIn = new PPort ( this );
    CDispIn = new PPort ( this );       RNGIn = new PPort ( this );

                    // create PPortref objects
    CUOut = new PPortref ( this );      CLOut = new PPortref ( this );
    CDispOut = new PPortref ( this );   RNGOut = new PPortref ( this );
  }
```

Figure 6.11: Controller constructor

Notice that in addition to the instantiation parameters necessary for creating an instance of the `Controller` computational component (i.e `reseedPrd`), the constructor takes three other parameters. The `loc` and `pa` parameters are of relevance

to the current topic of discussion and identify the host at which the component should be created and the parent composite component of the primitive instance, respectively.[2] As can be seen, the first statement in the method is a call to the `PNode` constructor which passes all the relevant creation information to the CDB. The `PNode` constructor expects to receive the component specific creation parameters in an `Object`[3] array and this is the return value of the `packParams` method.

In the prototype, the component programmer must define this method manually. However, this method can be generated automatically by a simple preprocessor capable of parsing component constructor signatures. The `packParams` method for the `Controller` method is given below.

```
static Object [ ] packParams ( int prd )
  {
    Object p[ ] = new Object [1] ;
    p[0] = new Integer (prd) ;
    return p ;
  }
```

Figure 6.12: Controller.packParams method

The following statements in the constructor create `PPort` and `PPortref` objects to complete the components communication interface. The constructors for these objects take as their parameter a reference to the owning component. The side-effects from these statements result in update requests to the CDB, which augment the previously created component-record with the components communication interface. The CDB assigns the unique port and portref indecies referred to in section §6.2.2.1 based on the order these update requests arrive. Thus `CUIn` has index zero and `CLIn` is assigned index one and so on. The order that these objects are created must be the same as that of the `port` and `portref` declarations in the C++ `process` class.

Primitive state-access methods - The state-access method definitions in Java simply serve as a means of passing invocation parameters through to C++ component methods and retrieving the results in an accessible form for use by the change programmer. State-access method invocations are realised through a call to the

[2]The `pass` parameter is needed by the RM during the two pass execution cycle of reconfiguration scripts. Its use will be described in section 6.2.4.2).

[3]The `Object` class is the root of all Java classes

`PNode.invoke` method. The parameters to this method are the name of the method and an `Object` array containing all of the method parameters. The return type of the invoke method is an object of type `Unwrap` which is used to extract the results of the function call.

Figure 6.13 shows the `getReseedPrd` state-access method of the `Controller` component. The Java access-method has the same signatures as the C++ method defined in the corresponding Regis component. The `getReseedPrd` method does not take any parameters and hence a `null` value is supplied in place of the `Object` array. The function however does have an integer return value and the `unpack` method of the returned `Unwrap` object `res` is called to gain access to it. The mapping between C++ objects and Java objects is done by the `UnWrap` and `Packer` classes. These classes can be easily extended to map between complex objects and not just the default data-types.

```
public int getReseedPrd ( )
  {
    Unwrap res = invoke ( "getReseedPrd" , null /* num prms*/ ) ;
    return ((Integer) res.unpack ("Integer")).intValue () ;
  }
```

Figure 6.13: Controller.getReseedPrd state-access method

Figure 6.13 depicts the `setReseedPrd` method of the `Controller` component. In this instance there is no return value but the method parameters are inserted into an `Object` array and passed to the `invoke` method. As with the `packParams` method, the generation of state-access methods are good candidates for automated generation techniques.

```
public void setReseedPrd ( int Prd )
  {
    Object prms[ ] = new Object [1] ;
    prms[0] = new Integer (Prd) ;

    Unwrap res = invoke ( "setReseedPrd" , prms ) ;
  }
```

Figure 6.14: Controller.setReseedPrd state-access method

Composite components - Composite components are defined in the same way as primitive components with the distinction that the composite variants of the configuration classes are used. Composite component constructors however also need to establish the internal configuration of the component which is done by creating and binding child component together.

To do this however, the application designer needs access to the basic configuration actions. The basic actions necessary for the specification of initial and subsequent configurations of the system are provided as public member methods of the various configuration classes. Thus, composite and primitive portref classes provide `bind` and `unlink` methods while `Node` classes implement the `delete` method. The `new` operator is used to create instances of the user defined component classes.

Since composite component declaration can also define reconfiguration methods and composite state-access methods they also need access to the CDB query mechanism which enables them to gain handles to reconfiguration targets. These query methods are also implemented as member methods of the configuration classes. Thus the `ClassInstances` method used in chapter four is actually a member method of the `CNode` class. Other notable query methods include the `isBound` and `nodeAt` methods of the Portref classes. This topic will be explored in greater detail in section §6.2.6 which discusses the CDB implementation and the supported access methods.

Again the examples given in chapter four are indicative of how composite components are programmed (see figure 4.16 on page 105).

6.2.4.2 Two pass execution cycle

In the prototype, the two pass execution cycle of reconfiguration scripts relies on all reconfiguration methods to take an additional boolean parameter. This `frstPas` parameter indicates whether the script is being executed as part of the first or second phase. This parameter is also passed to all basic reconfiguration actions such as `bind`, `unlink`, `delete` and component constructors.

In the first pass the execution of the basic reconfiguration actions have two distinct side-effects. The first is to modify the CDB and the second is to create a flat-list of reconfiguration actions. This list is later used by the CM to calculate the BSet and PSet memberships. The database updates are tagged such that the CDB recognises them as temporary modifications and does not alter the actual application configuration.

To overcome the difficulty posed by state-access methods, a mechanism has been

put in place which enables the change programmer to indicate to the reconfiguration system which components will be affected by the planned change. Thus, when situations such as that in figure 6.4 arise, the change designer encloses the offending code segment in an `if` statement such that during the first phase the necessary components are added to the original block-set, while during the second phase the actual code segment is executed. Figure 6.15 depicts the code segment of figure 6.4 after the necessary modification have been made.

```
Node nodes [ ] = { a, b, c } ;
if ( new Invoker ( frstPas, nodes ).ok () )
  {
    if ( a.getStatus () )
      b.setStatus ( val1 ) ;
    else
      c.setStatus ( val2 ) ;
  }
```

Figure 6.15: State access methods

In the above example the `Invoker` object is used to signal to the reconfiguration runtime system that components `a`, `b` and `c` will be affected by the code-segment enclosed in the `if` statement. During the first pass the `Invoker.ok` method returns false and thus the state-access methods are not invoked. The `ok` method however returns true on the second pass and thus allows the previously skipped code-segment to be executed.

6.2.4.3 Triggering interface

The triggering of reconfiguration is done through the `reconfigPort` member which all composite components inherit from their CNode super class. Triggering messages are sent to this port by the event composition service. Triggering messages contain an integer "selector" field, which signifies which of the supported reconfiguration methods of the component is to be triggered, and optionally the scripts invocation parameters. Composite components are threaded objects which block waiting for triggering messages or direct method invocations via other reconfiguration scripts. Reconfiguration methods are declared using the `synchronized` keyword in Java which prevents multiple version of the same method from being invoked on the same object concurrently.

All composite components support the execution of evolutionary change scripts.

Setting the selector field to zero in a triggering message signifies the invocation of evolutionary change scripts. As illustrated in chapter four (see figure 4.26) evolutionary change scripts are wrapped in a class which implements the `EvolutionaryChange` interface. The name of this class is supplied as part of the triggering message to the composite component. The composite component loads this class' byte-code by using Java's dynamic loading facilities and then runs the evolutionary change-script by invoking the class' `execute` method.

The decoding of the triggering messages arriving on the `reconfigPort` and the subsequent invocation of the reconfiguration methods have not been automated in the prototype and need to be done manually by the change programmer. These however follow a set pattern and are good targets for automation techniques.

6.2.5 CM

The task of the consistency manager was to ensure that reconfigurations would leave application components in mutually consistent states. To ensure this the CM needed access to the flat list of planned reconfiguration actions and the current system configuration. In the prototype, as in the presented framework, the RM supplies the CM with the flat list of reconfiguration actions and the CDB provides it with up-to-date configuration information.

As stated in chapter three the system architects should not be restricted to a single mutual-consistency mechanism, but be allowed to choose the consistency scheme best suited to the target application domain. The prototype implementation of the CM facilitates this through the definition of a simple interface which classes embodying consistency algorithms must support. The interface requires that such classes provide implementations for the `waitSafe` and `signalEnd` methods. The `waitSafe` method is called by the CM prior to the actual execution of the reconfiguration, while the `signalEnd` method is invoked after the completion of the configuration changes. Thus, in the case of avoidance schemes the `waitSafe` method should only return after the reconfigurable-state has been brought about and the `signalEnd` method should reverse any actions taken by the `waitSafe` method, thereby returning the system to its normal mode of operation. System designers register their chosen mutual consistency object with the reconfiguration environment via the `CM.setCM` method.

Figure 6.16 depicts a typical interaction with the CM in the prototype. The code segments results in the safe execution of a reconfiguration script named `Grow`. The `Grow` reconfiguration script does not take any runtime parameters except for the

mandatory boolean "frstPas" parameter mentioned in section 6.2.4.2 and an integer transaction identifier, "tId", parameter. This parameter is not used by the CM but it is required by the CDB. Its significance will be explored later in section §6.2.6.3.

```
Integer tId = new Integer ( 0 ) ;
Grow ( true /* frstPas = */, tId ) ;
CM.setCM ( new CMan ( ) ) ;
CM.waitSafe ( ) ;
Grow ( false /* frstPas = */, tId ) ;
CM.signalEnd ( ) ;
```

Figure 6.16: CM interface

The first statement declares the `tId` variable and initialises it to zero. The next statement is an invocation of the `Grow` reconfiguration method with the `frstPas` parameter set to true. This accomplishes the first pass execution of the reconfiguration script, enabling the RM to generate the flat list of basic reconfiguration actions needed by the CM. As described in section §6.1.2, any configuration updates made during this phase are rolled back at the end of the first phase by the CDB and have no effect on the underlying application configuration.

The CM interactions begin with the third statement which registers an instance of the `CMan` class with the CM. `CMan` is the class which implements the consistency-mechanism of chapter five. Next, the CM's `waitSafe` method is invoked. This method calls the registered consistency class' `waitSafe` method (`CMan.waitSafe` in this case) which actually brings about the reconfigurable-state. Once the `CM.waitSafe` method has returned, it is safe to carry out the reconfiguration proper. This is done by invoking the `Grow` reconfiguration method once again, but this time with the `frstPas` parameter set to false. Configuration updates made during this second phase are permanent in effect and result in modifications to the underlying application configuration. The `CM.signalEnd` method invokes the `CMan.signalEnd` method which performs necessary cleanup operations and returns the system to its normal mode of operation.

The mutual-consistency mechanism adopted by the prototype is that presented in chapter five. The implementation of the `CMan` class therefore follows the pseudo code description given in section §5.7.1. The `CMan.waitSafe` method corresponds to the statements up to the end of the `while` loop in the consistency object (see figure 5.23 on page 149) and the `CMan.signalEnd` method corresponds to the statements following the execution of the reconfiguration script.

6.2.6 CDB

In the presented reconfiguration environment the CDB was required to maintain a consistent and up-to-date view of the application configuration, facilitate access to this information, and help parallelise script executions. The following sections describe how these issues are dealt with in the prototype implementation.

Although the CDB relies on well established database facilities such as transactions, rollbacks, concurrency control mechanisms and triggers, the CDB's interpretation of a number of these notions is sufficiently different from those in existing database systems that providing a new implementation seemed more feasible than making modification to an existing one. Furthermore, existing database systems are fairly heavyweight objects. While this is acceptable for systems which make constant use of the database, the imposed overhead on distributed application which accesses the CDB only occasionally could not be justified. Finally since the CDB is a repository for configuration information alone, a new implementation also afforded the possibility of organising the internal data-structures to best suit the specialised nature of the stored information and query and update requests.

6.2.6.1 Stored configuration information

The CDB contains six different record types, each corresponding to one of the following - composite components, primitive components, composite ports, composite portrefs, primitive ports and primitive portrefs.

A composite component record - Although composite components have no real representation in the actual application configuration, they are represented in the CDB data-structures. This is so that change designers can express reconfiguration scripts in terms of the design time composition hierarchy rather than the actual flattened system structure. The CDB maintains the following information for every composite component in the application configuration:

- a numeric identifiers signifying the host location where the component was created,
- a reference to the database record of the parent composite component,
- number of child components,
- a hash-table of all child-component database records,

- a list of references to database records for the components composite ports (`CPort`),
- a list of references to database records for the components composite portrefs (`CPortrefs`)

The hash-table containing child component-records, places components which have the same type in the same buckets. This allows queries which request reconfiguration targets based on their type (e.g. `ClassInstances`) to be serviced efficiently. Figure 6.17 depicts the component class hierarchy established when using the provided configuration classes for the lottery example.

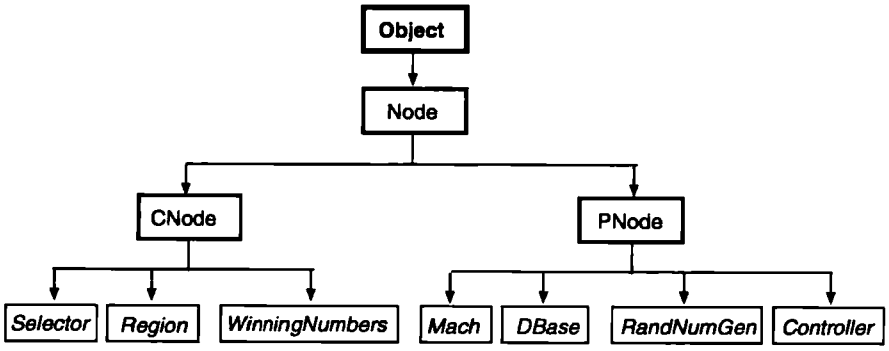


Figure 6.17: Component class hierarchy

The `Node` class is the root of all component classes, and `CNode` and `PNode` are the respective roots of composite and primitive components. Thus all of a composite's child components will appear in the hash-table's `Node` bucket and all composite and primitive child components will appear in the `CNode` and `PNode` buckets respectively. Additional buckets are created for user defined components types, thus all `RandNumGen` child components will also be placed in a separate bucket as will `Controller`'s, `Selector`'s, Region's etc. Aside from the component hierarchy depicted in figure 6.17 the application programmer may have set-up other useful type groupings or interface implementations (e.g. migratable). These will also be reflected in the hash-table. The necessary information regarding a components class hierarchy and the interfaces it implements are extracted through Java's runtime-type-information support.

A primitive component record - is similar to a composite component record except that fields related to child-components are absent and the lists of communica-

tion end-points maintain references to primitive port and portref records as opposed to composite ones. Primitive component records also maintain the addresses of the component's cm, cdb and rm ports. These port addresses are required by the reconfiguration runtime system.

Primitive and composite port records - retain a reference to their owning component and a list of portrefs which hold references to them. For example in figure 6.18 the port record of component *f* reflects that there are portrefs on *a*, *b* and *c* which hold references to it.

Recall that composite ports have two sets of bindings. One set of bindings established the bindings of external components to the composite and the other established the actual binding between the virtual composite port and the actual concrete port provision on one of the composite's child components. Thus in addition to the information maintained by primitive port-records, composite port entries also record the internal port to which they are bound. In the figure, the composite port-record of component *K* records the binding between itself and the port on *K*'s child component *f*.

Primitive and composite portref records - retain a reference to their owning component and record the port to which they are bound[4]. In the figure, the portref record of component *a* would reflect its binding to the port on component *f*.

As with composite ports, composite portrefs also possess two sets of bindings. Thus composite portrefs maintain an additional list of internal bindings which record the list of internal portrefs which are bound to the composite. In the figure, the composite portref record of *K* retains a list which contains the portrefs on *f*, *e* and *d*.

6.2.6.2 Query and update interface

The CDB's query and update interface is exposed to the change programmer through methods of the reconfiguration classes. These methods pass additional parameters to the CDB and help preserve the design-time view of the application configuration exported by the CDB. Note that the implemented query methods are by no means exhaustive, and cover a minimal set of query operations. At implementation time it was unclear what other query operations would prove useful. If necessary, it is

[4]In Regis a portref can only be bound to a single port.

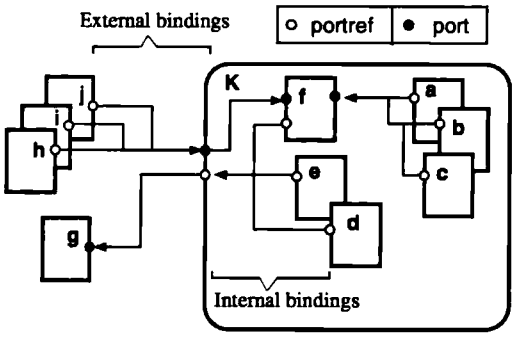


Figure 6.18: Internal and external bindings

possible for application programmers to combine the results of a selection of these basic query operations to derive more specialised configuration information.

Tables 6.2 through to 6.7 chart the different methods of the reconfiguration classes and give a brief outline of their function.

Primitive components methods - (PNode table 6.2) The `loc` and `paw` methods are database read operations served locally by the CDB. The `delete` method is an update operation which removes the component's record from the CDB and sends a delete message to the component's `cdb_stub` which removes its actual counterpart from the application configuration. The `delete` method takes a boolean `frstPas` parameter which is used during the two phase execution of reconfiguration scripts.

PNode	
int *loc* ( )	returns the host location where the component was created
CNode *paw* ( )	returns a handle to the parent composite component
void *delete* (boolean frstPas)	removes the component from the application configuration

Table 6.2: PNode methods

Composite components methods - (CNode table 6.3) All of `CNode`'s methods, with the exception of `delete`, are database read operations which are served locally by the CDB. The `loc` and `paw` methods are as described for `PNode`. The

`ClassInstances` methods query the child components of the composite based on their type. The type of child-components can be specified either as a string or by a Class variable. The `childTypes` method returns a list of component types which exist within the composite component.

The `delete` method, unless overridden by an user-defined `delete` method, unlinks all of the composite's internal components and invokes their respective `delete` methods. As with all database update methods, `delete` takes the mandatory boolean `frstPas` parameter. The role of the `TBegin` and `TEnd` functions will be explained in section §6.2.6.3.

CNode
int *loc* ( ) returns the host location where the component was created
CNode *paw* ( ) returns a handle to the parent composite component
int *numChildren* ( ) returns the number of child-components contained by the composite.
Enumeration *ClassInstances* (String) returns the list of components contained by the composite which have the type specified in the String parameter.
Enumeration *ClassInstances* (Class) returns the list of components contained by the composite which have the type specified in the Class parameter.
Enumeration *childTypes* ( ) returns the list of component types contained by the composite
void *delete* (boolean frstPas) removes the composite component from the application configuration.
void *TBegin* (boolean frstPas, Integer tId) void *TEnd* (boolean frstPas, Integer tId) CDB reconfiguration transaction begin and end wrapper functions

Table 6.3: CNode Methods

Primitive port methods - (PPort table 6.4) Each primitive port belongs to the communication interface of a single primitive component. The `owner` method of a `PPort` object returns the component which owns the port. The `portrefsAt` method returns the list of all portrefs which reference this port and the `nodesAt` method returns a list of all components whose communication interface contains a reference to this port. In the prototype, the `nodesAt` method is used by the CM when calculating the Block-set membership (e.g. all components which have

connection arcs to a component targeted for deletion must be in BSet)

PPort	
PNode *owner* ( )	returns the primitive component which owns this port.
Enumeration *portrefsAt* ( )	returns a list of all portrefs which references this PPort.
Enumeration *nodesAt* ( )	returns a list of all components which have references to this PPort.

Table 6.4: PPort methods

Composite port methods - (CPort table 6.5) Composite port methods are similar to those of primitive ports. In particular the `owner`, `portrefsAt` and `nodesAt` methods behave exactly as in `PPort`. Composite ports however need to provide additional methods for manipulating their internal interface bindings (see figure 6.18).

CPort	
CNode *owner* ( )	returns the composite component which owns this port.
void *bind* (boolean frstPas, PPort) void *bind* (boolean frstPas, CPort)	binds the CPort's internal interface to that of a child component's primitive or composite portref.
void *unlink* (boolean frstPas, CPort) void *unlink* (boolean frstPas, PPort)	unlinks the composite-port from its internal primitive or composite port
Provision *internalPortAt* ( )	returns the internal primitive or composite port bound to the CPort.
Node *internalNodeAt* ( )	returns the internal component bound to the CPort.
Node *portrefsAt* ( )	returns a list of external portrefs bound to the CPort.
Enumeration *NodesAt* ( )	returns a list of all external components which have reference to this CPort.

Table 6.5: CPort methods

For this purpose two `bind` methods have been provided which enable the change designer to link the `CPort`'s internal interface to a `PPort` or a `CPort` belonging to a child component. The `unlink` method performs the reverse of a `bind` operation and breaks a previously established link. The `internalPortAt` and `internalNodeAt`

methods are used to query a CPort's internal binding. Thus, internalPortAt returns the port to which the CPort's internal interface is bound and internalNodeAt returns the child component which owns that port. All bind and unlink CDB update methods take the mandatory frstPas parameter.

Primitive portref methods - (PPortref table 6.6) Like ports, each portref belongs to a single primitive component and it is a handle to this component which is returned by the owner method. Communication paths are established by linking PPortrefs to primitive or composite ports. This operation is facilitated through the two bind methods and the unlinking of such bindings is catered for by the unlink methods. Additionally it is possible to check whether a portref is bound or not through the isBound method. If the PPortref is bound, then the portAt method returns the referenced port and the nodeAt method returns the component which owns that port. All bind and unlink CDB update methods take the mandatory frstPas parameter.

PPortref	
PNode *owner* ( )	returns the primitive component which owns this portref.
void *bind* (boolean frstPas, PPort) void *bind* (boolean frstPas, CPort)	binds the primitive-port to primitive or composite port
void *unlink* (boolean frstPas)	unlinks the portref from the composite or primitive port it is currently referencing.
boolean *isBound* ( )	returns true if the portref is bound, false otherwise.
Provision *portAt* ( )	returns a reference to the port object referenced by the portrefence.
Node *nodeAt* ( )	returns the component which owns the port referenced by this portref.

Table 6.6: PPortref methods

Composite portref methods - (CPortref table 6.7) The CPortref methods owner, bind, unlink, isBound and nodeAt behave exactly as those described for PPortref.

However, as with composite ports, composite portrefs also need to provide additional methods to establish their internal interface bindings. To this end two

additional `bind` methods have been defined. These methods take portrefs as their parameters and establish the binding between the `CPortref`'s internal interface and any `PPortrefs` or `CPortrefs` on child components. Appropriate `unlink` methods have also been provided which break the link between the `CPortref`'s internal interface and the specified composite or primitive portref.

CPortref
CNode *owner* ( ) returns the composite component which owns this portref.
void *bind* (boolean frstPas, PPort) void *bind* (boolean frstPas, CPort) binds the composite-port to an external primitive or composite port
void *unlink* (boolean frstPas) unlinks the portref from the composite or primitive port it is currently referencing.
void *bind* (boolean frstPas, CPortref) void *bind* (boolean frstPas, PPortref) binds the CPortref's internal interface to that of a child component's composite or primitive portref.
void *unlink* (boolean frstPas, PPortref) void *unlink* (boolean frstPas, CPortref) unlinks the portref from the internal composite or primitive ports which reference it.
boolean *isBound* ( ) returns true if the portref is bound, false otherwise.
Enumeration *internalPortrefsAt* ( ) returns a list of internal components which have portrefs bound to the composite portref.
Enumeration *internalNodesAt* ( ) returns a list of internal components which have portrefs bound to the composite portref.
Provision *portAt* ( ) returns a reference to the port object referenced by the portrefence.
Node *nodeAt* ( ) returns a reference to the component owning the externally referenced port object.

Table 6.7: CPortref methods

The `internalPortrefsAt` and `internalNodesAt` methods are used to query the internal bindings of a `CPortref`. Thus, the `internalPortrefsAt` method returns all portrefs bound to the `CPortref`'s internal interface and `internalNodesAt` returns the child components which own those portrefs. All `bind` and `unlink` CDB update methods take the mandatory `frstPas` parameter.

In addition to the presented methods a number of convenience functions have also been defined. Variants of the `unlink` method allow all bindings of a communication end-point to be broken and `rebind` methods combine the `unlink` and `bind` operations.

6.2.6.3 Transactions

As mentioned before, the CDB treats the execution of each reconfiguration script as a transaction. To facilitate this, reconfiguration scripts must take a mandatory integer parameter `tId` and enclose their method bodies within the `CNode` methods `TBegin` and `TEnd`. The `TBegin` and `TEnd` methods call appropriate methods of the CDB and signal the beginning and end of reconfiguration transactions, respectively.

Figure 6.19 show the skeleton for reconfiguration scripts in this scheme. Figure 6.20 depicts the `newSelector` reconfiguration method of chapter four (figure 4.21) written as it would appear in the prototype.

```
public void reconf_method (
            boolean frstPas,
            Integer tId,
            p1, p2, ..., pn)
{
  TBegin ( frstPas, tId ) ;
    ...
    ...
    ...
    ...
  TEnd ( frstPas, tId ) ;
}
```

Figure 6.19: Change script skeleton

```
public void newSelector (
            boolean frstPas,
            Integer tId,
            int loc )
{
  TBegin ( frstPas, tId ) ;
    Enumeration slctrs = ...;
    Selector sel, tmp ;
    ...
    CDispIn.bind ( frstPas, ... ) ;
  TEnd ( frstPas, tId ) ;
}
```

Figure 6.20: NewSelector script

The scheme works as follows. Top-level reconfiguration scripts are always called with the `tId` parameter set to zero (see figure 6.16). When the `TBegin` method is invoked with this parameter the CDB recognises it as a new transaction and updates the `tId` variable with a unique transaction identifier. All subsequent reconfiguration actions and queries carry this identifier, thereby allowing the CDB to distinguish between query and update requests originating from different transactions. Nested reconfiguration method-calls are passed the `tId` of their parent transaction by the caller and are thus recognised as nested transaction by the CDB. The CDB keeps track of the number of `TBegin` and `TEnd` method invocations for each transaction

identifier and interprets the final TEnd call as a commit request.

Updates made by transactions which invoke the TBegin method with the frstPas parameter set to true are treated as temporary updates by the CDB. The results of these updates are rolled back when the final TEnd method is called. This is in step with the two phase execution policy applied by the RM. By the same token update requests made during the second execution pass (i.e those made by transactions which have invoked TBegin with the frstPas parameter set to false) are reflected onto the application configuration immediately.

The prototype implementation of the CDB does not deal with the execution of multiple concurrent transactions and thus transactions are always allowed to commit. However in circumstances where a reconfiguration transaction cannot be included in a serialisable schedule by the adopted concurrency control mechanism, the transaction's TEnd commit request would be rejected. The CDB would then roll back the transaction and schedule it for later execution. The only requirement on this process is that reconfiguration transactions which are allowed to commit during their first pass executions must also be allowed to complete in their second phase.

6.2.6.4 Concurrency

The CDB facilitates concurrency within single reconfiguration scripts by parallelising the execution of the basic change actions. Under this scheme the RM continues to execute reconfiguration scripts sequentially, however the update requests to the database succeed immediately by modifying the database records and spawning threaded objects which carry out the actual modifications to the application configuration asynchronously. Bind and unlink operations are carried out by binder threads, state-access methods by invoker threads and create requests by threaded God objects.

God objects forward create requests to the create-proxy residing on the appropriate Regis node and await the create confirmation message from the new component. This message contains the port addresses for the component's cdb and rm ports which binder and invoker threads operating on the component need before they can proceed. Such binder and invoker threads suspend their execution and wait until they are notified by the God thread that the component has been created successfully. Access to a component's records (e.g its ports and portreferences) is facilitated through synchronized Java methods which implement semaphores and thus sequentialise concurrent access requests. God object wake suspended threads in the same order as they are added to the waiting queue and thus the sequential

behaviour of reconfiguration script execution is maintained for change actions which operate on the same targets.

The only time the execution of the reconfiguration script by the RM needs to synchronise with that of the slave threads in the CDB is during state-query methods whose return values determine the flow of control in the change script.

The prototype CDB implementation also included a simple graphical interface which allowed the database records to be viewed and modified interactively.

6.3 Chapter summary

This chapter addressed some outstanding implementation issues pertaining to the proposed reconfiguration environment and went on to describe the details of a prototype dynamic reconfiguration service. The prototype design followed the logical partitioning set out in the framework of chapter three and provided implementations for the runtime reconfiguration hooks, CM, RM and the CDB. The prototype implementation incorporated the consistency preserving mechanism of chapter five and the proposed state-access interface required for re-establishing invalidated invariants in the CM and RM respectively. Existing object oriented features of Java were coupled with a set of configuration class libraries to meet the requirements of the reconfiguration programming paradigm. The configuration query and update interface exported by the CDB was added to this augmented language environment, thereby enabling the expression of hierarchic and reusable change scripts. Additionally, the CDB implementation facilitated concurrency within individual change scripts by parallelising the execution of the basic reconfiguration actions.

Chapter 7

Case study - The Evolving Philosophers

This chapter describes an implementation of the "Evolving Philosophers" problem over the prototype reconfiguration service. The case study serves to demonstrate how the various elements of the reconfiguration environment fit together and gives an accurate insight into the process of writing reconfigurable applications in the proposed framework. Screen grabs of the case study implementation are given in Appendix A.

7.1 The evolving philosophers

The evolving philosophers problem [KM90] is a modification of the "Diners", first described in [Dij72]. In the static diners problem, a fixed number of philosophers are arranged in a ring with neighbouring philosopher sharing exactly one fork. Philosophers are either hungry, thinking or eating. However, before a philosopher can eat he must first acquire his left-hand and right-hand forks. Solutions to this problem need to stipulate fork allocation and sharing strategies which prevent philosophers from starving. For example if every philosophers has one fork but refuses to share it until after he has eaten, the philosophers will all starve. The evolving philosophers introduces dynamism into the problem by allowing philosophers to join and leave the ring. The consistency requirements however still apply and so neighbouring philosophers must share exactly one fork and share it in such a way that they do not starve.

The solution adopted by this implementation follows that of [KM90] which is an extension of Chandy and Misra's solution presented in [CM84]. In their "hygienic"

solution forks are categorised as either being clean or dirty. Forks which have been used for eating are dirty and since philosophers are hygienic they clean forks before sending them to their neighbours. The fork sharing strategy requires philosophers in possession of dirty forks to service requests from their neighbours, while deferring requests for forks which are clean, until after they have eaten. In effect this solution implements a precedence graph such that an edge directed from a philosopher u to v indicates that u has precedence over v (see figures 7.1 and 7.2).

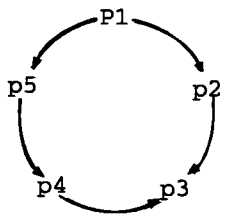


Figure 7.1: p1 hungry

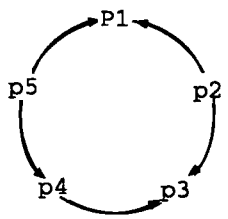


Figure 7.2: p1 eating

A philosopher u has precedence over philosopher v if i) u holds a clean fork, or ii) v holds a dirty fork or iii) the fork is in transit from v to u. Chandy and Misra showed that as long as the precedence graph was acyclic, it would remain so and starvation would not occur. By the same token all that reconfigurations need to do is ensure that the addition and removal of philosophers do not introduce cycles in the precedence graph.

7.2 Implementation

In this implementation of the evolving diners a top-level composite `Ring` component encloses the ring of dining `Philosophers` and in addition to the constructor method for establishing the initial ring, provides the reconfiguration methods for adding and removing `Philosopher` components dynamically.

The `Philosopher` composite component encapsulates the behaviour of a philosopher and is itself made up of the `Phil`, `Fork_server` and `Display` primitive components. The `Phil` and `Fork_server` components are Regis primitive components implemented in C++[Str93] and provide the computational behaviour of a philosopher. The `Display` component is a Java primitive component which manages the visual representation of the philosopher.

The following sections give detailed descriptions of the computational and con-

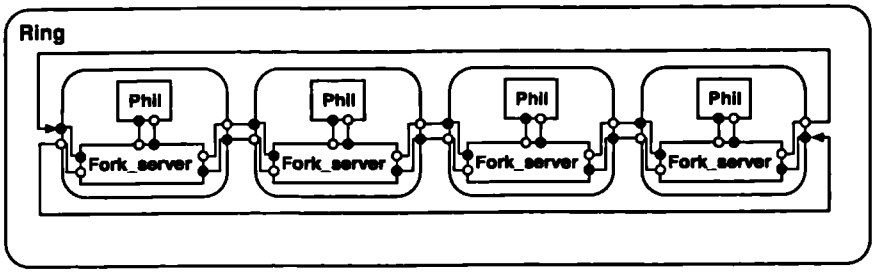


Figure 7.3: Ring of Dining Philosophers

figurational aspects of these components.

7.2.1 Phil primitive component

Figure 7.4 depicts the C++ header file for the `Phil` Regis component. It declares the communication interface of the component which consists of the port `rep` and port-references `req` and `disp_pref`. The `rep` and `req` communication end-points are used in communications with the `Fork_server` component. The `disp_pref` port-reference is used to inform the `Display` component of state changes (e.g hungry to eating, eating to thinking etc.).

```
class Phil : public rcprocessT<Phil>
{
public:        // communication interface
  rcport<int> rep ;
  rcportref<int> req ;
  rcportref <int> disp_pref ;
    // constructor and create function
  Phil ( portrefbase pr, int nid, int pid )
  static void create_fn ( portrefbase pr, int nid, int pid,
                int num_prms, unpacker& up)
private:       // auxiliary functions
  void eat ( void )
  void think ( void )
  void hungry ( void )
  int d_time ( void )
};
```

Figure 7.4: Phil C++ header file

Notice that `Phil` inherits from the template class `rcprocessT` and uses the reconfiguration alternatives to Regis communication classes, `rcport` and `rcportref`

to declare its communication end-points `rep`, `req` and `disp_pref`.

The header file also declares the components constructor and a create function `create_fn` which is used by the reconfiguration runtime system to create components of type `Phil`. Both methods take three default parameters `pr`, `nid` and `pid`, needed by the reconfiguration runtime system.

Additionally a number of auxiliary functions are declared. The `eat`, `think` and `hungry` methods are used to keep the visual representation of the philosopher in tune with its actual state. These methods change the state of the `Phil` component and send appropriate messages to the `Display` component via `disp_pref`. The `d_time` method returns a random number between the two class constants `min_t` and `max_t`[1] and is used to determine the period spent eating and thinking by the philosopher.

7.2.1.1 Phil reconfiguration description

Figure 7.5 depicts the corresponding Java `Phil` class which encapsulates the component's reconfiguration description. Since `Phil` does not define any state-access methods, its reconfiguration description simply captures its communication interface. The `Phil` constructor has been included for completeness. As described in the previous chapter, it performs necessary housekeeping operations.

```
public class Phil extends PNode
{
  PPort rep ;
  PPortref req ;
  PPortref disp_pref;

  public Phil (boolean pass, int loc, CNode pa )
    { // no user defined params hence null
      super (pass, loc, pa, null ) ;
        // instantiate PPort and PPortref objects
      rep = new PPort ( this ) ;
      req = new PPortref ( this ) ;
      disp_pref = new PPortref ( this ) ;
    }
}
```

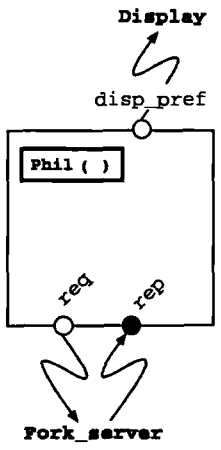


Figure 7.5: Phil configuration description

[1] not shown here.

7.2.1.2 Phil computational behaviour

The `Phil` component implements the philosophers basic behaviour by cycling through the hungry, eating and thinking states. As mentioned before a philosopher cannot eat until it has possession of both its left-hand and right-hand forks. When hungry a `Phil` component waits until it receives a transaction initiation message from its `Fork_server` component. The transaction initiation message is sent to `Phil`'s `rep` port and signals that the philosopher is in possession of both its forks and can begin to eat. Once the component has finished eating the forks are returned to the `Fork_server` via a message sent on the `req` port-reference and the transaction is terminated.

This behaviour is captured in the C++ definition of the `Phil` component constructor, depicted in figure 7.6. Notice the use of the `beginc` and `endc` methods to mark critical region of the component's code. Having categorised the interactions with the fork-server as a transaction the component programmer can be sure that the `Phil` component will not be reconfigured while it is in possession of the forks. The particular `beginc` construct used is one which combines the receipt of a message and the beginning of the critical region as a single operation.

```
Phil::Phil ( portrefbase pr, int nid, int pid )
            : rcprocessT<Phil> ( pr, nid, pid )
{
  int forks ;
  while ( true )
    {
    hungry () ;
    rep.beginc ( forks ) ;        // begin interaction with Fork Server
      {                           // once fork arrives
      eat () ;
      delay ( d_time () ) ;
      req.send ( forks ) ;        // return forks
      think () ;
      }
    endc () ;                     // end interaction with Fork Server
    delay ( d_time () ) ;         // delay for think time.
    }
}
```

Figure 7.6: Phil constructor

Also notice that the component constructor passes the previously mentioned default parameters `pr`, `nid` and `pid` to its super-class' constructor. The `nid` para-

meter is the unique identifier of the component and `pid` is the unique identifier of its parent composite component. The super-class constructor sends a creation confirmation message via the `pr` port-reference. Messages sent on this port-reference arrive at the CDB *God* thread which requested the creation of the component. As described in chapter six, this message contains port-references to the component's cdb, rm and cm portrefs and is used by the CDB to update the component's record.

Recall that components were created by a proxy on each host through the invocation of pre-registered create functions. These functions received the components runtime parameters in a packed buffer and after unpacking the parameters created an instance of the object. Since the `Phil` component does not take any user defined parameters its create function simply creates a new Phil component by passing the default parameters to the constructor. The definition of the `Phil::create_fn` is give in figure 7.7.

```
void Phil::create_fn ( portrefbase pr, int nid, int pid, int num_prms,
                       unpacker& up )
{
    // no user defined params to unpack
  new Phil (pr, nid, pid);
}
```

Figure 7.7: Phil create function

7.2.2 Fork_server primitive component

Figure 7.8 depicts the C++ header file for the `Fork_server` Regis component. Like the `Phil` component it inherits from `rcprocessT` and declares its communication interface using the `rcport` and `rcportref` classes. The `preq` and `prep` port and port-reference pair are used for communication with the `Phil` component whose forks the `Fork_server` manages. The `rreq`, `rrep` and `lreq`, `lrep` pairs are used for communicating with the philosopher's right and left neighbours respectively.

The `Fork_server`'s constructor takes two parameters in addition to the three mandatory arguments. The `l` and `r` runtime parameters are used to assign left and right forks to the `Fork_server` respectively. Assigned forks are always clean to begin with. A zero integer value is interpreted as no-fork.

A close inspection of the class definition reveals that in addition to the obligatory `create_fn` the `Fork_server` component defines two state-access methods for setting and getting the fork values. A number of auxiliary members and methods have also been defined. The `lf` and `rf` members determine the state of the component's forks (e.g. Clean, Not-Available, Dirty). The send-fork methods send forks

```
class Fork_server : public rcprocessT<Fork_server>
{
public:          // communication interface
  rcport<int> preq, lreq, rreq ;
  rcportref<int> prep, lrep, rrep ;
  rcportref<int> disp_pref ;
    // constructor & create function
  Fork_server (portrefbase pr, int nid, int pid, int l, int r )
  static void create_fn ( portrefbase pr, int nid, int pid,
                          int num_prms, unpacker& up)
    // state-access methods
  void set_fork ( int no_args, unpacker& u, packer& p )
  void get_fork ( int no_args, unpacker& u, packer& p )
private:         // auxiliary members & functions
  forkstates lf, rf ;
  void send_left_fork ( void )
  void send_right_fork ( void )
  void update_display ( void )
}
```

Figure 7.8: Fork_server C++ header file

to the appropriate neighbour and update the values of the `lf` and `rf` members. The `update_display` method is used to inform the `Display` component of relevant changes.

7.2.2.1 Fork_server reconfiguration description

```
public class Fork_server extends PNode
{
  PPort preq, lreq, rreq ;
  PPortref prep, lrep, rrep ;

  Fork_server (bool pass, int loc, CNode pa,
               int l, int r )
  void set_fork ( int lr, int f )
  boolean get_fork ( int lr )
  static Object [ ] packParams ( int l, int r )
}
```

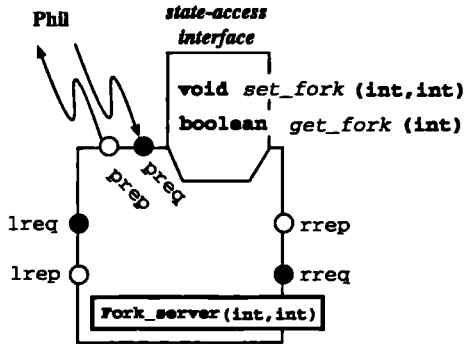


Figure 7.9: Fork_server configuration specification

The corresponding Java class definition which encapsulates the `Fork_server`'s reconfiguration description is depicted in figure 7.9. It declares the primitive port

and portref objects which define the component's communication interface and also exposes the `set_fork` and `get_fork` state access-methods.

Since the component constructor takes additional runtime arguments, (`l` and `r`), it is necessary to provide a `packParams` method to insert the runtime parameters into an object array. This array can then be used to create the packed buffer transmitted as part of create requests. Figure 7.10 depicts the Java constructor for the `Fork_server` component. The `packParams` method is used as part of the call to the super-class constructor which actually uses the returned object array to create the packed buffer. The constructor also populates the component's communication interface by initialising the appropriate `PPort` and `PPortref` objects.

```
public Fork_server (boolean pass, int loc, CNode pa, int l, int r )
  {
    // use packParams to pack object array of user defined
    // runtime params
  super (pass, loc, pa, packParams (l, r)) ;
    // instantiate PPort and PPortref objects
  preq = new PPort ( this ) ;          lreq = new PPort ( this ) ;
  rreq = new PPort ( this ) ;          prep = new PPortref ( this ) ;
  lrep = new PPortref ( this ) ;       rrep = new PPortref ( this ) ;
}
```

Figure 7.10: Fork_server constructor

The Java `packParams` method and the corresponding C++ `create_fn` are given in figures 7.11 and 7.12 respectively.

```
static Object [ ] packParams ( int l ,
                               int r )
{
  Object p [ ] = new Object [2] ;
  p [0] = new Integer (l) ;
  p [1] = new Integer (r) ;
  return p ;
}
```

Figure 7.11: Java `packParams`

```
void create_fn ( portrefbase pr, int nid,
                 int pid, int num_prms,
                 unpacker& up )
{
  int lf, rf ;
  up >> lf ;
  up >> rf ;
  new Fork_server ( pr, nid, pid, lf, rf ) ;
}
```

Figure 7.12: C++ `create_fn`

The parameters placed in the `Object` array by the Java `packParams` method are transformed and placed in a packed buffer by the *God* thread which is in charge of the components creation. All that the component's `create_fn` does is to unpack the user defined parameters and invoke the component constructor with them. Notice

that the packing process preserves the order in which parameters are placed in the Object array and thus the first object in the buffer is the left-fork value.

7.2.2.2 Fork_server get/set_fork State access-methods

Figures 7.13 and 7.14 depict the Java and C++ `set_fork` state-access method pairs, respectively. The Java code inserts the arguments to the method into an `Object` array and uses the `invoke` method to call the actual C++ state-access function. The C++ function unpacks the packed parameters and sets the designated fork to the provided value.

```
public void set_fork ( int lr, int f )
{
    // create array of size two elements
  Object prms[ ] = new Object [2] ;

    // insert Object variants of params
    // into the array
  prms [0] = new Integer (lr) ;
  prms [1] = new Integer (f) ;

    // provide invoke method with the
    // name of the method and its params
  Unwrap res = invoke ("set_fork", prms );
}
```

Figure 7.13: Java `set_fork`

```
void set_fork ( int no_args, unpacker& u,
                packer& p)
{
    // unpack params
  int which_fork, fval ;
  u >> which_fork ;
  u >> fval ;
    // set new fork value
  if ( which_fork == LEFT_FORK )
    lf = fval ;
  else
    rf = fval ;
    // inform display of change
  update_display ( ) ;
}
```

Figure 7.14: C++ `set_fork`

Similarly, figures 7.15 and 7.16 depict the Java and C++ `get_fork` state-access method pairs, respectively. As before the Java code inserts the arguments to the method into an `Object` array and uses the `invoke` method to call the actual C++ state-access function. In this case the parameter specifies which of the forks, left (0) or right (1), the method queries. The boolean return value when true signifies that the queried fork is in the possession of the `Fork_server` and when false it indicates the absence of the fork. Note that in this instance information is passed both from the Java method to the C++ function and vice-versa. The return value of the `invoke` method is an `UnWrap` object which contains the results of the C++ function invocation. The Java method calls the `unpack` method of this object to extract typed objects from the buffer. In this instance the return value is of type `Integer`. The boolean conversion considers any number greater than zero as true.

```
public boolean get_fork ( int lr )

{
  Object prms [ ] = new Object [1] ;
  prms [0] = new Integer (lr) ;
  Unwrap res = invoke ( "get_fork",
                        prms );
  // unpack & return result
  return ((Integer) res.unpack
  ("Integer")).intValue () > 0 ;
}
```

Figure 7.15: Java `get_fork`

```
void get_fork ( int num_args, unpacker& u,
                packer& pkr)
{
  int which_fork ;
  u >> which_fork ;
    // Nope has val 0 ( false ) ,
  if ( which_fork == Right_Fork )
    pkr << rf ;
  else // Left_Fork
    pkr << lf ;
}
```

Figure 7.16: C++ `get_fork`

7.2.2.3 Fork_server computational behaviour

As mentioned before `Fork_server` components coordinate the fork-sharing strategy adopted by the philosophers. In this implementation `Fork_server` components participate in three different transactions. The first type is initiated by the `Fork_server` when it is in possession of two clean forks. Figure 7.17 depicts the code segment which embodies this interaction.

```
if ( lf == Clean && rf == Clean )
  {
  beginc () ;
    {
      // send forks to Phil
    prep.send ( forks ) ;
      // wait to get them back
    preq.in ( forks ) ;
      // send forks to neighbours
    send_left_fork () ;
    send_right_fork () ;
    update_display () ;
    }
  endc () ;
  }
```

Figure 7.17: `Phil` transaction

In it a message is sent to the `Phil` component informing it that it can begin eating. The `Fork_server` then waits for a reply message from the `Phil` component signalling that the philosopher has finished eating with the forks. At this point the forks are sent to the neighbouring philosophers and the transaction is terminated.

The `send_left_fork` and `send_left_fork` methods accomplish the sending of the forks and also reset the fork variables `lf` and `rf`. The `update_display` method updates the `Display` components with the new state information.

```
beginc () ;
  {
  int f ;
  lreq.in ( f ) ;
  lrep.send ( 1 ) ; // ack
  lf = Clean ;
  update_display () ;
  }
endc() ;
```

Figure 7.18: L'neighbour trans'n

```
beginc () ;
  {
  int f ;
  rreq.in ( f ) ;
  rrep.send ( 1 ) ; // ack
  rf = Clean ;
  update_display () ;
  }
endc() ;
```

Figure 7.19: L'neighbour trans'n

The other two transactions are started by the philosopher's neighbours when they send clean forks to the `Fork_server`. Figures 7.18 and 7.19 depict the code segments for transactions started by the left and right neighbours respectively. The Fork_server engages in these transactions when messages arrive at the `lrep` and `rrep` ports. The actions performed in both cases consist of receiving the fork token, sending an acknowledgement message to the initiator of the transaction and updating the appropriate fork variable. The final call to the `update_display` method ensures that the `Display` component is apprised of the state changes.

Note that because of the way transactions have been set-up, the safe-state comes about when the `Phil` component is not in possession of the forks. Also any forks in possession of the `Fork_server` will be in the clean state. Thus, a true boolean value returned by the `Fork_server::get_fork` method signifies the existence of a *clean* fork.

7.2.3 Function registrations

As mentioned is chapter six, create functions and state-access methods need to be registered in the relevant list objects. This is done by defining a method of the system proxy object which is left intentionally undefined in the library. The `proxy::register_classes` method defined in figure 7.20 adds the `Phil::create_fn` and `Fork_server::create_fn` create functions to the proxy's list of create functions and also adds the `set_fork` and `get_fork` state-access methods for the `Fork_server` component to its type-specific state-access method-list.

```
void proxy::register_classes (void)
{
  add_cfn ( "phils.Phil" , Phil::create_fn ) ;
  add_cfn ( "phils.Fork_server" , Fork_server::create_fn ) ;

  Fork_server::add_st_adj_fn ( "set_fork", Fork_server::set_fork ) ;
  Fork_server::add_st_adj_fn ( "get_fork", Fork_server::get_fork ) ;
}
```

Figure 7.20: function registration

7.2.4 Philosopher composite component

The `Philosopher` composite component combines the primitive thinking, hungry, eating behaviour of a `Phil` component with the fork-sharing capabilities of the `Fork_server` component to create a "civilised" `Philosopher`. Figure 7.21 depicts the reconfiguration specification of a `Philosopher`. It defines the composite communication objects used by the philosopher to communicate with its neighbouring philosophers and also exposes the `set_fork` and `get_fork` state-access methods of the `Fork_server` component it encloses.

```
public class Philosopher extends CNode
{
    // communication interface
  CPort l_fil_in, r_fil_in ;
  CPortref l_fil_out, r_fil_out ;
    // reconfiguration interface
  public Philosopher ( boolean pass, int loc,
                       CNode pa,int lf, int rf )
  public void set_fork ( int lr , int f )
  public boolean get_fork ( int lr )
}
```

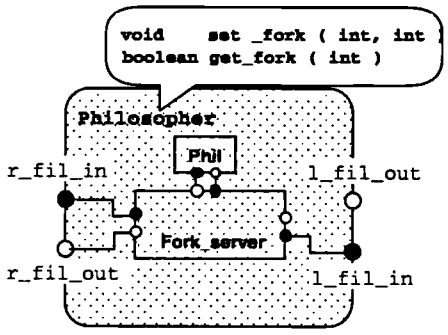


Figure 7.21: Philosopher composite reconfiguration description

7.2.4.1 Philosopher constructor

The Philosopher constructor depicted in figure 7.22 takes two user defined parameters `lf` and `rf` which determine the initial fork-allocation of the philosopher. The constructor body instantiates the composite port objects which constitute the composite's exported communication interface and instantiates and binds the subordinate `Display`, `Phil` and `Fork_server` primitive components.

```
public Philosopher ( boolean pass, int loc, CNode pa, int lf, int rf )
{
  super (pass, loc, pa, null) ;
        // instantiate CPort and CPortref objects
  l_fil_in = new CPort ( this ) ;          r_fil_in = new CPort ( this ) ;
  l_fil_out = new CPortref ( this ) ;      r_fil_out = new CPortref ( this ) ;
        // instantiate components
  Display d = new Display ( pass, loc (), this, lf, rf ) ;
  Phil p = new Phil ( pass, loc (), this ) ;
  Fork_server fs = new Fork_server ( pass, loc (), this, lf, rf );
        // establish Display bindings
  p.disp_pref.bind ( pass, d.in ) ;        fs.disp_pref.bind ( pass, d.in ) ;
        // establish Phil & Fork_server bindings
  p.req.bind ( pass, fs.preq ) ;           fs.prep.bind ( pass, p.rep ) ;
        // establish composite interface bindings
  fs.lrep.bind ( pass, l_fil_out ) ;       fs.rrep.bind ( pass, r_fil_out ) ;
  l_fil_in.bind ( pass, fs.lreq ) ;        r_fil_in.bind ( pass, fs.rreq ) ;
        // begin composite thread
  T.start ();
}
```

Figure 7.22: Philosopher constructor

7.2.4.2 Run method : reconfiguration message handler

The final statement in the constructor code-segment of figure 7.22, `T.start()`, starts the thread associated with the composite component which deals with reconfiguration method-invocation messages arriving at the component's `reconfigPort` (see section §6.2.4.3). The actions of this thread are determined by the body of the `run` method which composite components need to override. However since the `Philosopher` composite component does not define any reconfiguration methods it simply inherits the default run method provided by the `CNode` class. This method is depicted in figure 7.23.

The method body receives triggering messages arriving on the `reconfigPort` and interprets them based on the `methodId` field. A zero value in this field signifies an evolutionary change. Under such circumstances the name of the class which encapsulates the evolutionary change script is stored in the `evolClassName` field of the triggering message. This name is passed to the `Class.forName` method so that the code for the evolutionary change can be loaded dynamically. Once this has happened, an instance of the specific `EvolutionaryChange` class is created and its `execute` method, which encapsulates the change script, is invoked. In addition to the standard `frstPass` and `tId` reconfiguration method parameters the `execute` method

```
public void run ()
{
while (true)
  {
  TrigMessage tm = (TrigMessage) reconfigPort.in ("TrigMessage") ;
  switch ( tm.methodId )
    {
    case EVOL : // EVOLutionary change = 0
      {
        // load class using name stored in evolClassName field
      Class evolClass = Class.forName ( tm.evolClassName ) ;
      EvolutionaryChange ec = (EvolutionaryChange) evolClass.newInstance () ;
        //invoke reconfiguration method
      Integer tId = new Integer ( 0 ) ;
      ec.execute ( /*frstPass =*/true, tId, this ) ;
      CM.setCM ( new CMan () ) ;
      CM.waitSafe () ;
      ec.execute ( /*frstPass =*/false, tId, this ) ;
      CM.signalEnd () ;
      break ;
      }
    }
  }
```

Figure 7.23: run method

is also passed a reference to the composite component. This is so that the change script can gain handles to its reconfiguration targets such as the `WinningNumbers` component in the chapter four lottery example (see figure 4.26 on page 115).

7.2.4.3 Philosopher get/set_fork composite state access-methods

The `set_fork` and `get_fork` composite state-access methods are depicted in figures 7.24 and 7.25 respectively.

```
public void set_fork ( int lr , int f )
{
  Enumeration fSrvrs =
    ClassInstances ("phils.Fork_server") ;
  Fork_server fs ;
  fs = (Fork_server) fSrvrs.nextElement();
  fs.set_fork ( lr, f ) ;
}
```

Figure 7.24: set_fork access-method

```
public boolean get_fork ( int lr )
{
  Enumeration fSrvrs =
    ClassInstances ("phils.Fork_server") ;
  Fork_server fs ;
  fs = (Fork_server) fSrvrs.nextElement();
  return fs.get_fork ( lr ) ;
}
```

Figure 7.25: get_fork access-method

These simply invoke identically named methods on the subordinate `Fork_server` component and handle the passing of parameters and the returning of results. A handle to the enclosed `Fork_server` instance is gained through a parameterised call to the `ClassInstances` method. The parameter to the `ClassInstances` method is a string representation of the component type `Fork_server` defined in the `phils` package - hence `phils.Fork_server`.

7.2.5 Ring composite component

The `Ring` composite component is the top-level component in this application and is responsible for creating the ring of dining philosophers. This is done by its constructor. Additionally, it defines the `Grow`, `Shrink` and `GrowBy` reconfiguration methods. Figure 7.26 depicts the `Ring` reconfiguration description.

```
public class Ring extends CNode
{
  static final int ringMin = 2 ;
  public Ring ( boolean frstPass, int loc,
                CNode pa, int size )
    // reconfiguration interface
  public void Grow (bool frstPass,int tId)
  public void Shrink (bool frstPass,int tId)
  public void GrowBy ( bool frstPass,int tId,
                       int n )
    // overridden TrigMessage handler
  public void run ()
}
```

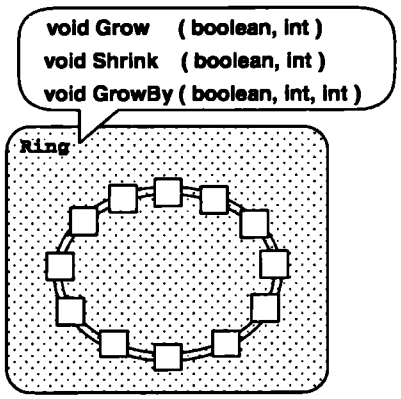


Figure 7.26: Ring composite component reconfiguration description

7.2.5.1 Ring constructor

The constructor method depicted in figure 7.27 takes the size of ring as a parameter and establishes the composite's internal configuration as depicted earlier in figure 7.3. The adopted fork allocation strategy assigns two clean forks to every other philosopher. Clean forks are designated by an integral value of one. The size of the ring is constrained to be at least `ringMin` elements where `ringMin` is a class constant. As with all composite components, the last statement in the constructor starts the thread which deals with triggering messages.

```
public Ring ( boolean frstPass, int loc, CNode pa, int size )
{
  super ( frstPass, loc, pa, null ) ;
  // there is a minimum ring size
  if ( size < ringMin )
    size = ringMin ;
  Philosopher fils[ ] = new Philosopher[size] ;
  for (int i = 0; i < size; i++)
    {
    fils[i] = new Philosopher
      ( frstPass, i, this, (i+1)%2, ((size-1)==i && i != 0)?0:(i+1)%2) ;
    if ( i > 0 )
      {
      fils[i].l_fil_out.bind (frstPass, fils[i-1].r_fil_in ) ;
      fils[i-1].r_fil_out.bind (frstPass, fils[i].l_fil_in ) ;
      }
    }
  fils[size-1].r_fil_out.bind ( frstPass, fils[0].l_fil_in ) ;
  fils[0].l_fil_out.bind ( frstPass, fils[size-1].r_fil_in ) ;
  T.start () ;
}
```

Figure 7.27: Ring constructor

7.2.5.2 Shrink reconfiguration method

The `Shrink` reconfiguration method which reduces the size of the ring of dining philosophers by removing an arbitrary philosopher is depicted in figure 7.28. The script begins by gaining handles to the philosopher targeted for removal, its left and right neighbours and additionally the left neighbour of its left neighbour - `target`, `lNeighbour`, `rNeighbour` and `llNeighbour` respectively in the script.

The script as written can only shrink rings that are at least three elements in size. This is checked for in the script and a `ReconfigurationException` thrown if the application configuration does not satisfy the scripts structural assumptions. If the script is allowed to proceed, the target philosopher is eliminated from the ring by binding its left and right neighbours to each other. Next the philosopher is unlinked and deleted from the application configuration.

Finally the necessary state-adjustments are carried out such that application invariants are preserved. This is done by assigning the shared fork between `lNeighbour` and `rNeighbour` to `lNeighbour`, thus ensuring that the number of philosophers and forks remain the same. The acyclicity of the precedence graph is guaranteed by giving `lNeighbour` the fork it shares with its left neighbour `llNeighbour` as well

```
public void Shrink (boolean frstPass, Integer tId) throws ReconfigurationException
{
  TBegin ( frstPass, tId ) ;
    {
        // gain handles to reconfiguration targets
    Enumeration fils = ClassInstances ( "phils.Philosopher" ) ;
    Philosopher lNeighbour = (Philosopher) fils.nextElement () ;
    Philosopher target = (Philosopher ) lNeighbour.r_fil_out.nodeAt () ;
    Philosopher rNeighbour = ( Philosopher ) target.r_fil_out.nodeAt () ;
    Philosopher llNeighbour = ( Philosopher ) lNeighbour.l_fil_out.nodeAt () ;
        // make sure that there are at least three fils
    if ( lNeighbour == rNeighbour )
      throw new ReconfigurationException ("Can't shrink Rings smaller than 3") ;
        // eliminate target from ring by binding its neighbours to each other
    lNeighbour.r_fil_out.rebind ( frstPass, rNeighbour.l_fil_in ) ;
    rNeighbour.l_fil_out.rebind ( frstPass, lNeighbour.r_fil_in) ;
        // unlink and delete target
    target.r_fil_out.unlink ( frstPass ) ;
    target.l_fil_out.unlink ( frstPass ) ;
    target.delete ( frstPass ) ;
        // adjust state to preserve application invariants
    Node nodes [ ] = { llNeighbour, lNeighbour, rNeighbour } ;
    if ( new Invoker ( frstPass, nodes ).ok () )
      {
        // preserve n forks n philosophers
      rNeighbour.set_fork ( left, noFork ) ;
      lNeighbour.set_fork ( right, cleanFork ) ;
        // preserve acyclicity of graph, liveness
      llNeighbour.set_fork ( right, noFork ) ;
      lNeighbour.set_fork ( left, cleanFork ) ;
      }
    }
  TEnd ( frstPass, tId ) ;
}
```

Figure 7.28: Shrink reconfiguration method

(i.e `lNeighbour` will have two clean forks).

7.2.5.3 Grow reconfiguration method

Figure 7.29 depicts the `Grow` method provided by the `Ring` component. It adds a new `Philosopher` component to the `Ring`. This is done by selecting an arbitrary `Philosopher` component from the ring and designating it as the new philosopher's left neighbour. By invoking the `nodeAt` method on the left-neighbour's `r_fil_out` port a handle is gained to what will be the new philosophers right-hand neighbour. Next a new Philosopher component, `newP`, is created with two clean forks and inserted in between the `lNeighbour` and `rNeighbour` components. The `rebind` method is used to accomplish this task. As described in chapter six, the `rebind` method unlinks any previous bindings of the communication end-point it is invoked upon before establishing new ones.

```
public void Grow (boolean frstPass, Integer tId )
{
  TBegin ( frstPass, tId ) ;
    {
        // gain handles to left & right neighbours of the new philosopher
    Enumeration fils = ClassInstances ( "phils.Philosopher" );
    Philosopher lNeighbour = (Philosopher) fils.nextElement ();
    Philosopher rNeighbour = (Philosopher) lNeighbour.r_fil_out.nodeAt () ;
        // create the new philosopher with two clean forks
    Philosopher newP = new Philosopher ( frstPass, lNeighbour.loc(), this,
                                         cleanFork, cleanFork ) ;
        // insert the new philosopher between its neighbours
    newP.l_fil_out.rebind ( frstPass, lNeighbour.r_fil_in ) ;
    newP.r_fil_out.rebind ( frstPass, rNeighbour.l_fil_in ) ;
    lNeighbour.r_fil_out.rebind ( frstPass, newP.l_fil_in) ;
    rNeighbour.l_fil_out.rebind ( frstPass, newP.r_fil_in) ;
        // remove forks as new philosopher has both
    Node nodes [ ] = { lNeighbour, rNeighbour } ;
    if ( new Invoker ( frstPass, nodes ).ok () )
      {
      lNeighbour.set_fork ( right, noFork) ;
      rNeighbour.set_fork ( left, noFork) ;
      }
    }
  TEnd ( frstPass, tId ) ;
}
```

Figure 7.29: Grow reconfiguration method

The application invariant is preserved through the invocation of `set_fork` methods on left and right neighbours which remove their right and left forks respectively. This actions ensures that the number of forks and philosophers remains the same. Also by assigning two clean forks to the new philosopher the precedence graph remains acyclic and thus starvation is prevented.

Notice that the reconfiguration method is enclosed within `TBegin` and `TEnd` methods and that an `Invoker` object is used to manually add elements affected by state-access methods to the block-set.

7.2.5.4 GrowBy reconfiguration method

The `GrowBy` method allows the ring of dining philosophers to grow by n where n is a runtime parameter of the reconfiguration script. The `GrowBy` method works by invoking the Grow reconfiguration method n times. The code for this function is depicted in Figure 7.30. A similar `ShrinkBy` method can also be provided which simply invokes the `Shrink` reconfiguration method n times.

```
public void GrowBy ( boolean frstPass, Integer tId, int n )
{
  TBegin ( frstPass, tId ) ;
    {
    for ( int i = 0 ; i < n ; i++ )
      Grow ( frstPass ) ;
    }
  TEnd ( frstPass, tId ) ;
}
```

Figure 7.30: GrowBy reconfiguration method

7.2.5.5 Run method : reconfiguration message handler

The `run` method, figure 7.31, decodes triggering messages and calls the appropriate reconfiguration method. Since the `Ring` component has defined new reconfiguration methods it overrides the `run` method defined in `CNode` which only handles evolutionary change requests. The code for running the `Grow` reconfiguration method has been given. The code follows the pattern described in chapter six section §6.2.5.

For simplicity we have not shown error checking code and catch clauses which deal with exceptions such as bind or reconfiguration exceptions thrown by change scripts.

```
public void run ()
{
RingLeader rl = new RingLeader ( this ) ;
while (true)
  {
  TrigMessage tm = (TrigMessage) reconfigPort.in ("TrigMessage") ;
  switch ( tm.methodId )
    {
    case EVOL :              // EVOLutionary change = 0
      ... as Philosopher
    case GROW :              // GROW = 1
      {
      Integer tId = new Integer ( 0 ) ;
      Grow ( /*frstPass =*/ true, tId ) ;
      CM.setCM ( new CMan () ) ;
      CM.waitSafe () ;
      Grow ( /*frstPass =*/ false, tId ) ;
      CM.signalEnd () ;
      break ;
      }
    case SHRINK :            // SHRINK = 2
    case GROWBY :            // GROW = 3
    case default :           // throw exception. No such script
    }
    ...
```

Figure 7.31: Ring run method

The depicted `run` method also creates a `RingLeader` object. In the implementation this object simulates the event composition service by allowing the user to generate triggering messages through a simple graphical interface. Figure 7.32 depicts the RingLeader object.

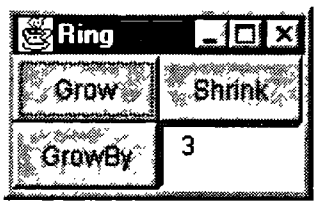


Figure 7.32: Generating triggering messages

7.3 Chapter Summary

This chapter presented the evolving philosophers case-study as implemented over the prototype reconfiguration service. The case-study demonstrated how the facilities of the proposed reconfiguration environment, encapsulated by the CM, CDB and RM runtime objects, could be united under a single language and used successfully to write reconfigurable applications.

The reconfiguration environment facilitated the development of consistency preserving reconfigurable distributed applications. This was accomplished through the use of an expressive object oriented language (Java) with augmented support for configuration queries, runtime state-access and mutually-consistent-states schemes.

Chapter 8

Conclusion

Increased reliance on long-running computer systems means that it is highly undesirable to interrupt the often indispensable services provided by distributed applications, in order to make alterations to them. Dynamic reconfiguration is essential for such systems as it allows changes to be performed on a fraction of a running distributed application while unaffected sections continue to provide a partial service.

For dynamic reconfiguration to be feasible, reconfiguration environments must provide the facilities for the specification and management of dynamic change. Furthermore dynamic reconfiguration must be constrained such that it does not leave the target application in an inconsistent state or results in large disruptions to the system. Unfortunately existing reconfiguration languages lack the constructs necessary for the specification of complex structured and reusable reconfigurations. Moreover the issues of consistency and minimum disruption, which are deciding factors in the viability of dynamic reconfiguration, are ignored by many of the reconfiguration environments in existence today.

This dissertation presented an environment for the specification and execution of consistency preserving dynamic reconfiguration with a special view to minimising system disruption. This chapter summarises the main contributions of the work and suggests possible directions for future research.

8.1 Summary and comparison

Chapter one identified the main responsibilities of the dynamic reconfiguration service to be the provision of support for the specification and management of dynamic reconfiguration and the preservation of consistency. The two criteria identified by the thesis for the preservation of consistency were that reconfiguration should i) re-

establish application invariants invalidated as a result of configuration changes, and ii) ensure that reconfigurations leave application components in mutually-consistent states. An Additional requirement placed on the reconfiguration environment was that it should minimise the disruption to the remainder of the system during reconfiguration.

Specification of change As mentioned before existing reconfiguration languages fail to provide the constructs necessary for the specification of complex structured and reusable reconfiguration scripts. The thesis addressed this problem by proposing the *reconfiguration programming paradigm.* The reconfiguration programming paradigm is novel in the way it encapsulates both the static (*configuration*) and dynamic (*reconfiguration*) behaviour of distributed applications in a unified model.

In keeping with this goal the specification and evaluation of triggering conditions, which were previously the domain of reconfiguration languages, are relegated to an external event composition service. Embedding the triggering conditions into the reconfiguration language, as is done in a number of contemporary change specification languages, is disadvantageous as it binds together the triggering condition and the reconfiguration script in such a way that even a minor modification to the condition will require alterations to the change script.

Additionally, the task of re-establishing invalidated application invariants which in existing environments was discharged by adding extra code to the computational elements of the application, is elevated to the reconfiguration language level. Embedding the interactions for reinstating particular application invariants in the code of the application is inflexible, and increases the functional complexity of the application. Moreover this approach makes little or no provisions for evolutionary reconfigurations, which become necessary after the application code has been compiled and is executing.

In order to facilitate the re-establishment of invalidated application invariants the paradigm requires that computational components implement a set of general state-access methods which allow an external agent to query and alter a selected subset of their internal state at runtime. The configuration descriptions of these components are extended such that in addition to the component communication interface and instantiation parameters, they now also include the component's state-access interface. These extended reconfiguration descriptions allow the change designer to identify and re-establish invalidated application invariants by invoking component state-access methods from within reconfiguration scripts. This is an important

property as it offers a practical alternative to embedding extra code into computation components and provides both programmed and evolutionary changes with a general mechanism for re-establishing invalidated application invariants.

The paradigm incorporates both inheritance and composition as part of its design methodology and in so doing facilitates the specification of hierarchic and generic reconfiguration specifications. Previously composition was used simply as a convenient means of specifying the initial configuration of a number of related components. In the reconfiguration programming paradigm however a composite component not only specifies the initial configuration of its subordinate components but also encapsulate their state and subsequent reconfiguration behaviour. This is accomplished by extending the configuration descriptions of composite component such that they include the definition of composite state-access and reconfiguration methods.

Composite state access-methods combine the invocation of a number of child state-access methods with some computation over the returned values in order to bring about a complex state adjustment or to return a compound value. Reconfiguration methods defined for composite component types capture internal reconfigurations which may become useful over the life-time of the component. Such configuration descriptions promote encapsulation and data-hiding and permit the definition of hierarchic reconfiguration scripts. Change designers will no longer need to delve into the internal structure of a composite component and can think in terms of composite components with well defined state and reconfiguration behaviour. In addition to the composition hierarchy, inheritance allows change designers to establish an inheritance hierarchy which facilitates the definition of generic configuration and reconfiguration descriptions (see section §4.6 on page 116).

Reconfiguration specifications written in this paradigm are reusable by virtue of the fact that they gain handles to their reconfiguration targets by querying the system configuration. These queries select components which match the configuration properties exhibited by the reconfiguration targets. Reconfiguration descriptions written in this way are independent of specific application configurations and can be used throughout the application's life-cycle. This is in contrast to scripts written in many existing reconfiguration languages that use access schemes which lead to disposable change specifications.

Additionally in the reconfiguration programming paradigm the application's composition hierarchy is maintained and the change programmer is allowed to express changes by applying reconfiguration actions to composite components. This enables the reconfiguration programmer to think and express changes in terms of the

same abstractions used originally during system design. Such a facility is absent in existing specification environments which require that change designers think about and express reconfigurations in terms of the flat view of interconnected application components.

The utility of this approach was illustrated through the use of an extended example, "The UK National Lottery".

Management of change To deal with the management of the change process the thesis proposed a framework which realises the reconfiguration service as a collection of interacting runtime entities. The runtime entities comprise the reconfiguration manager (RM), configuration database (CDB), consistency manager (CM) and a number of application hooks. In the framework the RM undertakes the task of executing reconfiguration scripts and coordinates the execution of the reconfiguration scripts with the CM and CDB such that reconfigurations do not interfere with each other and leave components in mutually consistent states. The CM encapsulates the safety mechanism used to ensure that components are left in mutually consistent states after reconfiguration. Changes to the configuration of the application are brought about through the CDB which exports an interface that can be used to query and modify the configuration of the underlying application at runtime.

By specifying the interactions between these and existing system services, the thesis arrived at a clear definition of their individual responsibilities and consequently that of the reconfiguration service as a whole. This logical division of responsibilities has the benefit that the framework may be used as a design template for the implementation of flexible reconfiguration services whose constituent components can be easily modified, replaced or reused. In the proposed scheme, the reconfiguration service is itself created as part of the application configuration and can thus be customised to suit the specific needs of the application it is helping to manage (e.g. the default consistency manager component (CM) can be eliminated from the reconfiguration service or replaced by a specialised variant depending on the consistency requirements of the target application).

Such a framework was noticeably missing from existing research efforts. Consequently, in the absence of a clear division of responsibilities, many existing reconfiguration environments either do not address important issues such as consistency and minimum disruption or invest elements of the environment with responsibilities which logically belong elsewhere. Another drawback of such systems is that the reconfiguration environments are inflexible and cannot be adapted easily to the

requirements of the target application.

Access to consistent and hierarchic configuration information Access to consistent and up-to-date configuration information is a prerequisite of the change specification process. However, gaining access to consistent and up-to-date configuration information about a distributed application which may be undergoing dynamic change poses serious practical problems. The thesis overcomes this problems by proposing that the configuration of the system be maintained and modified through a configuration database (CDB in the framework).

The CDB not only maintains the configuration of the system but also facilitates alterations to it by accepting reconfiguration requests. Since *all* modifications to the system configuration are carried out through the CDB, in the absence of failures, the information held in the CDB will always be up-to-date. Furthermore the CDB is in a position to exploit opportunities for leveraging concurrency within the execution of individual and across that of multiple reconfigurations.

Additionally, the CDB maintains the mapping between the design time hierarchic view of the application configuration and the actual flat configuration realised subsequent to system elaboration. This feature is a requirement of the reconfiguration programming paradigm and permits reconfigurations to be expressed in terms of change actions applied to composite components.

Mutually consistent states property One of the consistency preserving criteria was that reconfigurations should leave components in mutually consistent states. Performing configuration changes on a running system is an intrusive process which may interfere with ongoing interaction between components. In the framework the task of ensuring that this did not happen was assigned to the CM which could choose to implement any suitable mutual-consistency scheme.

Existing mutual-consistency schemes however, were found to be unsatisfactory as they either made restrictive assumptions about the services provided by the underlying distributed runtime system or imposed an excessive burden on the component programmer. Moreover, these schemes did not attempt to minimise the disruption to the system.

The thesis addressed these issues by proposing a novel mutual-consistency scheme, suited to a particular class of distributed applications which automatically imposes the safe state necessary for ensuring mutual-consistency in the presence of dynamic reconfiguration. The safe-state imposed is such that it caters for the requirements

of the invariant preservation mechanism proposed in the reconfiguration programming paradigm and can be used in conjunction with composite components. It was shown that Labeled transition systems in conjunction with automated analysis tools can be used to verify that a target system adheres to the requirements of the proposed consistency scheme. The approach takes advantage of the runtime behaviour of the application to minimise the disruption to the remainder of the system and compares favourably with existing schemes in this regard. It is largely transparent to the application level programmer and whilst guaranteeing the reachability of the safe state makes no assumptions about the ability of components to recover from interruptions caused by reconfigurations. Additionally, the scheme can be used to cater for multiple concurrent reconfigurations.

Minimisation of system disruption - The thesis addressed the issue of minimising system disruption through the CDB's support for concurrency and the proposed mutually-consistent states scheme.

Prototype and case-study As proof of concept the thesis presented a prototype implementation of the proposed reconfiguration environment and illustrated its utility through the use of a case-study. The implemented reconfiguration environment incorporated the proposed mutual-consistency mechanism and realised the reconfiguration programming paradigm by augmenting Java and using it as the reconfiguration specification language. The execution of evolutionary changes is also supported through an interface for loading reconfiguration descriptions dynamically.

8.2 Further Work

This section describes aspects of the presented work which would benefit from further work and suggests directions for future research.

Dealing with Failure The reconfiguration environment as presented does not deal with failures. In case of failure, the result of reconfiguration scripts which did not complete will have to be rolled back or rolled forward. In its current form the CDB can act as a stable image of the system configuration and can be used to reinstate the application's configuration prior to failure. However, before this can occur the CDB and the other components in the reconfiguration service must first become resilient to failures themselves.

Additionally the CDB will need to be modified such that it can deal with forced updates which host failures will impose upon it. The CDB can translate host failures to more fine grain composite component failures which can in turn trigger default scripts which will perform consistency checks on the CDB and for example remove links to non-existent components. Components and links removed as a result of failures can then be reinstated. Of course the CDB can only restore the configuration of the application. The computational state of the application must also be recovered from stable storage.

CDB query interface As it stands, the CDB configuration query interface is fairly primitive. As a result, non-basic query operations require that the change programmer write an imperative program which combines the basic queries with some computation in order to glean the desired information. A more sophisticated query interface would benefit the change designer by simplifying the task of locating reconfiguration targets. The potential for executing concurrent reconfigurations would also be increased as the CDB would be in a better position to optimise the processing of the queries and thus ensure that fewer records are read per query operation.

Application invariants One of the shortcomings of the reconfiguration programming paradigm is that it does not encapsulate application invariants. Strictly speaking the application's invariants should be expressed as part of its configuration specification. Given this description an automated inference engine could then generate the imperative state-adjustments necessary for re-establishing invalidated invariants. Currently this task is performed by the change designer.

A major obstacle in reaching this ideal however, is the absence of formal specification methods that can be used to model the behaviour of moderately sized applications without the specification process taking a disproportionate amount of time or becoming excessively complicated.

Mutual-consistency schemes The proposed mutual-consistency scheme imposes a number of behavioural restrictions on its target systems and thereby limits the range of application domains to which it can be applied. However it is these same behavioral restrictions which give rise to the beneficial aspects of the mechanism. Given that no one scheme is going to be suitable for all applications, arriving at the "correct" set of behavioural restrictions for the target application is the determining factor in the success of this and future consistency schemes. It would therefore be

of great benefit if future research would permit distributed applications to be classified, even broadly, according to the behavioural restriction to which they adhere. In the absence of such information, it is difficult to judge the scope of any consistency mechanism. Given these categories however, it would be possible to devise consistency schemes which exploit the specific characteristics of each application domain and are therefore ideally suited to them.

Appendix A

Screen Grabs: Philosophers

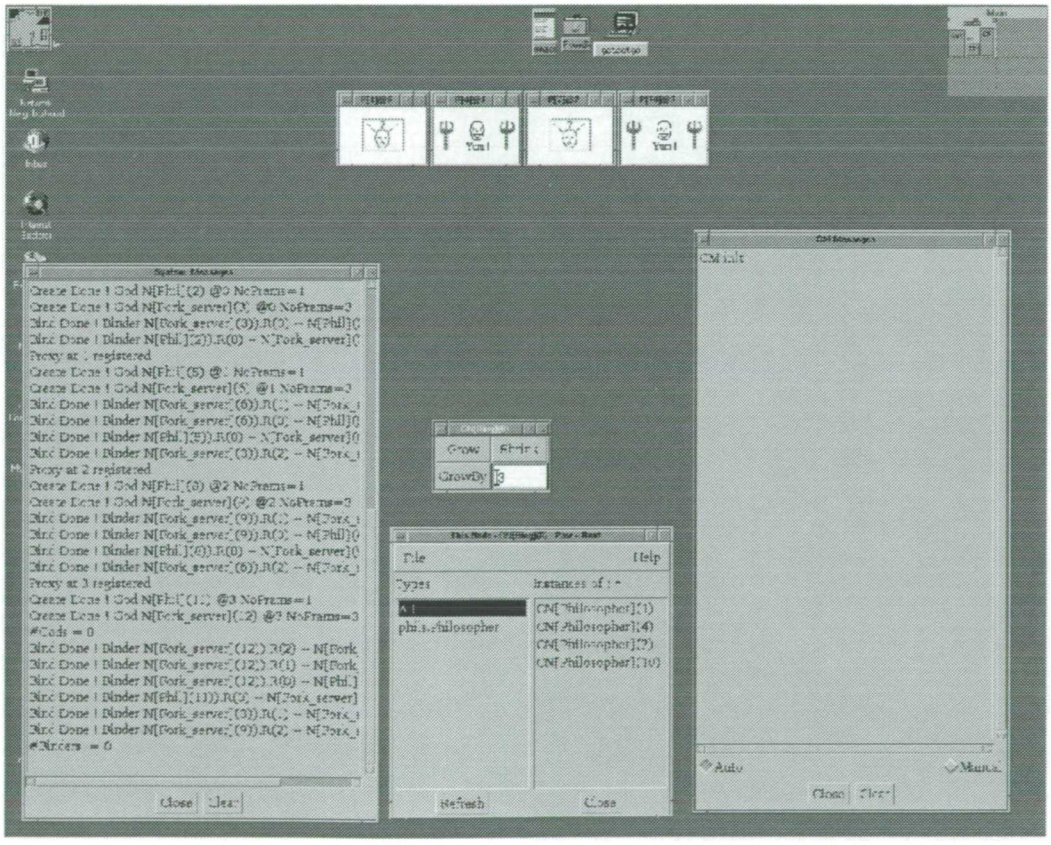


Figure A.1: Screen shot of four dining Philosophers

Figure A.1 shows the evolving philosophers example with four dining philosophers, from left to right P[1]@0, P[4]@1, P[7]@2 and P[10]@3. These names have been constructed by the `Display` component by appending the unique ID of the Philosopher's composite component with the integer identifier of the host upon which it

is executing. Thus the composite philosopher P[4]@1 has the unique identifier 4 and was created on host number 2. The mapping between the host ID's and actual host names can be controlled by the user via a ".hosts" file which contains integer ID and host name pairs.

The screen shot shows that philosophers P[4]@1 and P[10]@3 are eating and are in possession of both forks. Philosophers P[1]@0 and P[7]@2 are thinking. Although not represented graphically philosophers p[1]@0 and P[10]@3 are neighbours and complete the ring of dining philosophers. The "RingLeader" window is also visible in the centre and as described earlier in section §7.32, is used to trigger the programmed reconfigurations of the dining philosophers system. Three other system windows are also visible these, going from left to right are the "Systems Messages" window, the "CDB Viewer" and the "CM Messages" window.

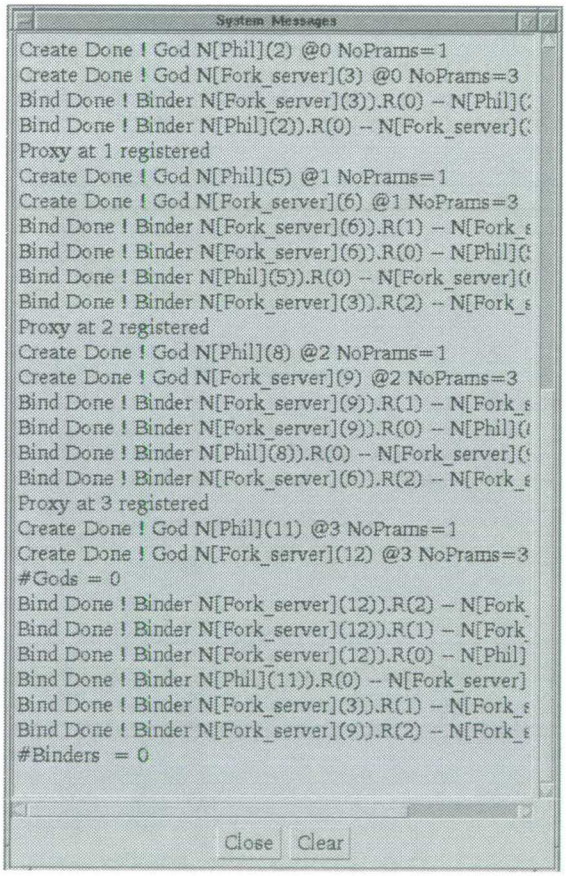


Figure A.2: System messages window

The systems messages window, figure A.2, displays the operations carried out by the runtime system as part of establishing the initial and subsequent configurations of the system. Creating the initial dining philosophers configuration involves the

creation of the `Phil` and `Fork_Server` subcomponents of each of the four composite `Philosophers` and the establishment of the various bindings between them. The progress of these operations can be observed in the systems messages window. The runtime system keeps track of the *God* and *Binder* threads spawned during this process. The application configuration will have been realised once all of these threads have terminated successfully.

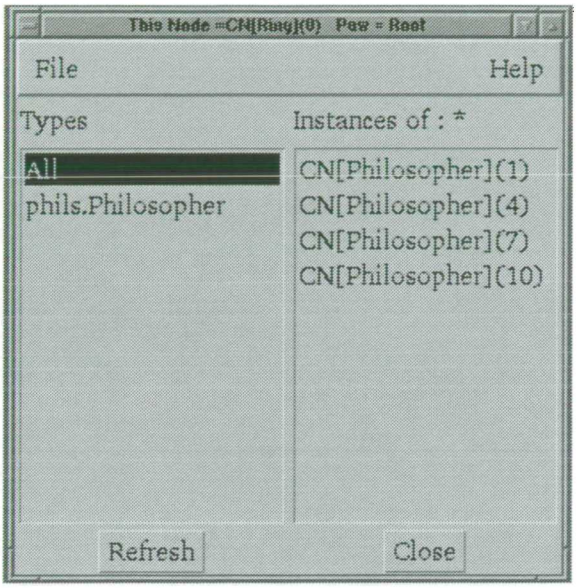


Figure A.3: CDB Viewer showing top-level Ring component

The CDB viewer, allows the change designer to view the configuration database at runtime. The default view shows the internal configuration of the top-level composite component, in this case the `Ring` component. The title of the CDB viewer window displays the type, ID and parent component of the displayed record. In this case the viewer is displaying a composite node of type `Ring` with ID 0, hence CN[Ring](0). The parent of this component is "root", indicating that it is the top-level component. For composite component the CDB viewer allows the user to view the sub-components contained within the composite grouped according to their type. In figure A.3 the category "All" has been selected which displays instances of all subcomponent types. The `Ring` component contains only instances of the `Philosopher` composite component type. These instances, which correspond to the four dining philosophers, are displayed.

The CM messages window captures the interactions of the consistency manager and the components in the system during reconfigurations. Since the system has

yet to be reconfigured this window is empty in figure A.1.

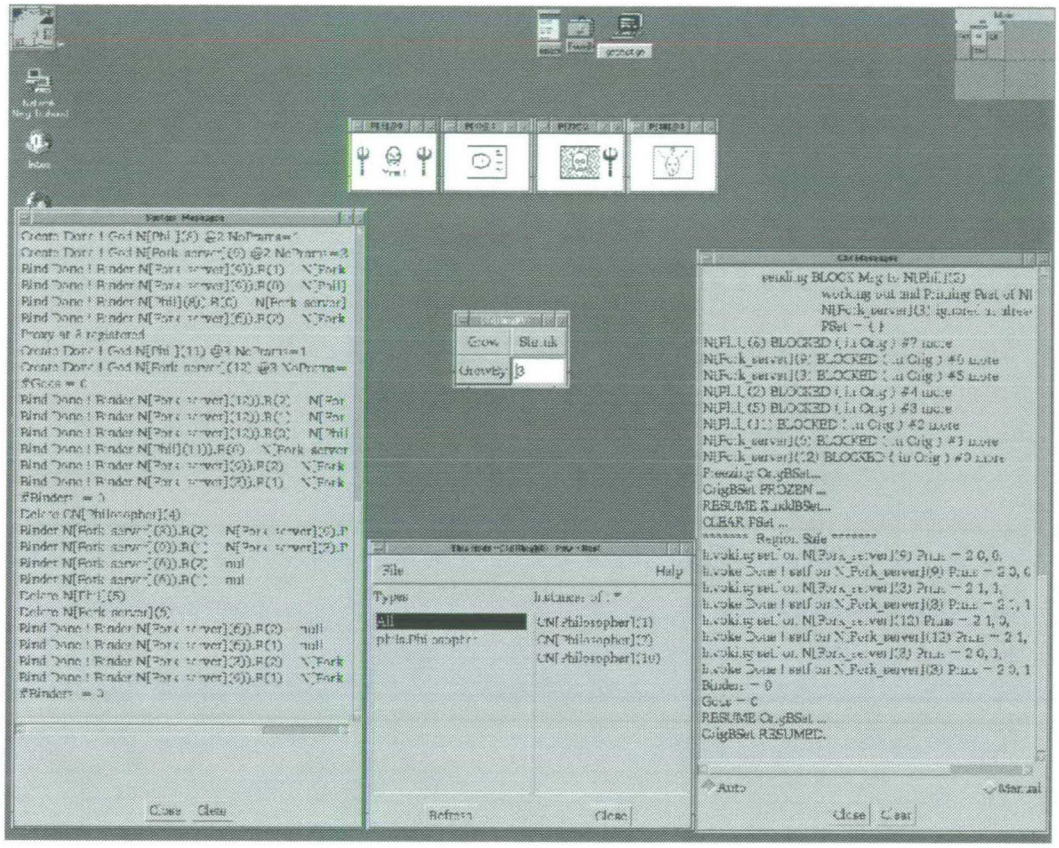


Figure A.4: The dining Philosophers subsequent to a shrink reconfiguration

Figure A.4 depicts the dining philosophers system of figure A.1 subsequent to the execution of a "shrink" reconfiguration. This reconfiguration was triggered by the system designer who generated the triggering event through the "RingLeader". As a result the philosopher P[4]@1 has been removed from the ring along with its shared fork. The ring has been reestablished by binding P[4]@1's neighbours, P[1]@0 and P[7]@2 together. In the captured image P[4]@1 is shown to have died, P[1]@0 is eating, P[7]@2 is hungry but cannot eat as it only has one clean fork, and P[10]@3 is thinking. The systems messages window, CDB viewer and CM messages window also reflect that the configuration of the system has been altered.

The CM messages window, figure A.5, displays the progress of the consistency algorithm. The window shows how the safe-state for reconfiguration is first imposed on the system, and also how the system is restored to its normal mode of operation after the reconfiguration has been performed. As mentioned before the RM coordinates with the CM before executing reconfigurations. In the window this interaction

is shown through the highlighted regions. The first highlighted line is where the CM signals the RM to proceed with the reconfiguration as the safe-state has been imposed. The next two highlighted lines are where the RM has signalled the end of the reconfiguration to the CM and is requesting that the CM return the system to its normal mode of operation.

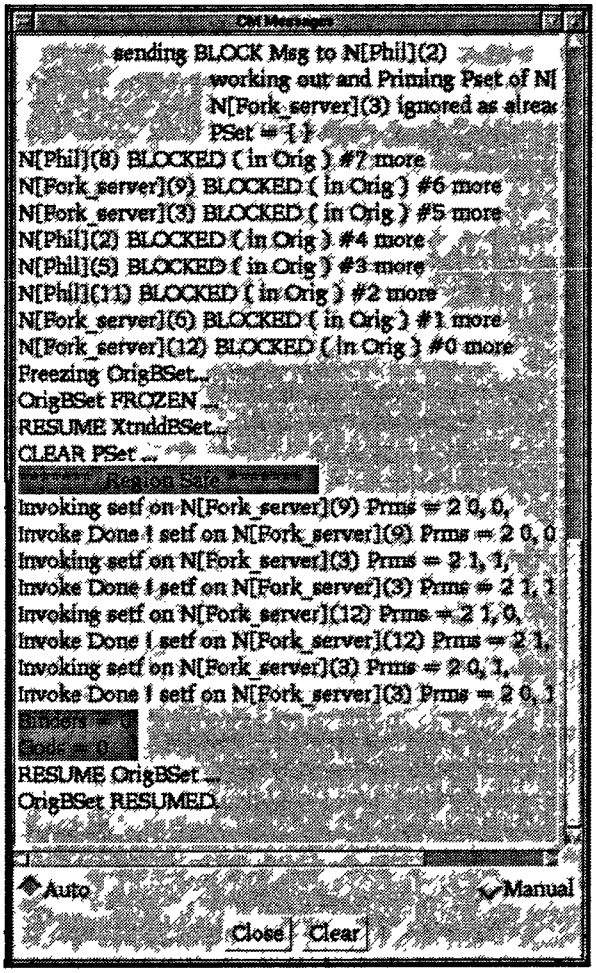


Figure A.5: CM messages subsequent to a shrink reconfiguration

The lines in between the highlighted regions show the state-access invocations performed by the RM as part of executing the shrink reconfiguration script. The window shows the multiple invocations of the `setf` method which correspond to the `set_fork` invocations in the shrink reconfiguration script depicted in figure 7.28. It is these invocations that preserve the invariants of the dining philosophers application.

The systems messages window displays the delete, unlink and bind actions which the runtime system carried out in order to affect the shrink reconfiguration. These consist of the removal of the P[4]@1 composite `Philosopher` component and thereby

that of its subcomponents N[`Fork_server`](6) and N[`Phil`](5), and also the bind operations necessary to link P[1]@0 and P[7]@2 together. Note that unlink operations are carried out by performing a bind operation to a `null` reference.

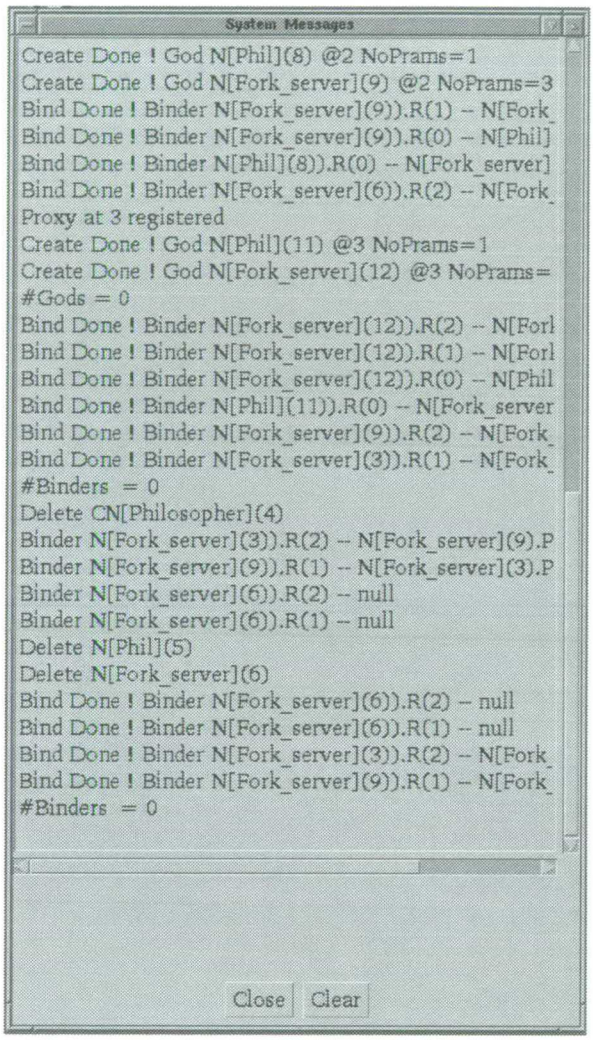


Figure A.6: System messages subsequent to a shrink reconfiguration

The CDB viewer can be used to browse the modified system configuration and if required verify that the reconfiguration has gone ahead correctly. The sequence of figures A.7 through to A.9 show the CDB viewer being used to examine the new system configuration.

The default top-level view of the CDB viewer, figure A.7, shows the absence of P[4]@1 from the list of subcomponents contained by the Ring composite component. This is consistent with the execution of the shrink reconfiguration which has resulted in the deletion of P[4]@1. Selecting CN[Philosopher](7) display the CDB record for

the **Philosopher** P[7]@2.

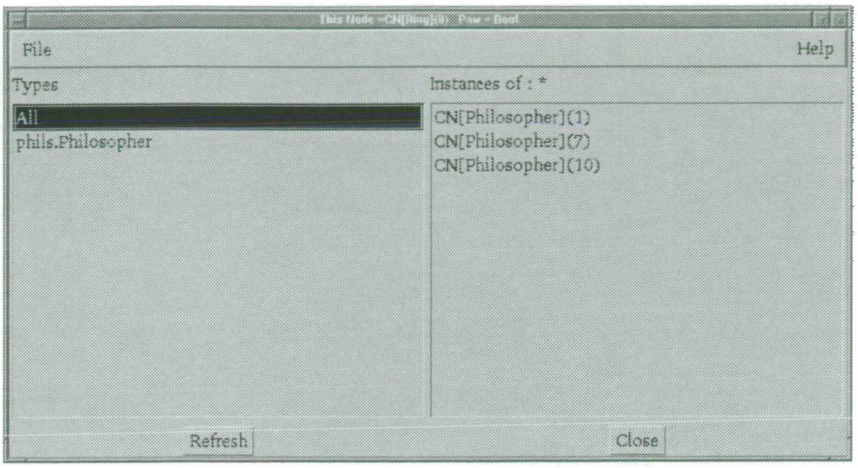


Figure A.7: CDB view - Ring all child types

P[7]@2 is a composite component and hence when selected the CDB viewer lists the subcomponent types in the left frame, and instances of "All" types in the left. As is evident from figure A.8, the P[7]@2 contains a **Phil** and a **Fork_server** component. These two subcomponent are primitive components. Selecting the primitive component N[Fork_server](9) displays its CDB record in the viewer.

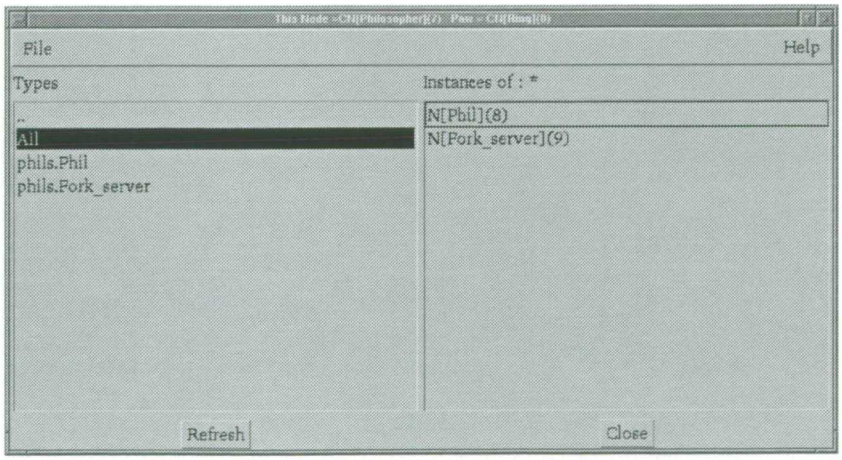


Figure A.8: CDB view - Philosopher 7 all child types

Figure A.9 displays the CDB record for the N[Fork_server](9) primitive component. Selecting the "neighbours" entry on the left frame allows us to examine the concrete bindings of the component's port-references. These are shown in the

left frame and show that the component is bound to a `Phil` and two `Fork_server` components through its R(0), R(1) and R(2) port-reference.

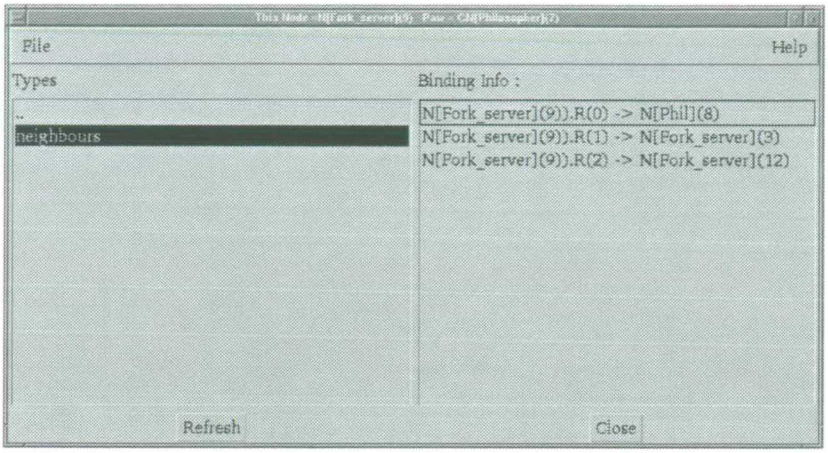


Figure A.9: CDB view - ForkServer 8 neighbours

The implemented CDB viewer was used only as a debugging facility and does not demonstrate the full capabilities of the CDB.

Bibliography

[AHP94] B. Agnew, C. Hofmeister, and J. M. Purtilo. Planning for change: A reconfiguration language for distributed systems. In *2nd International Workshop on Configurable Distributed Systems*, pages 15–22, Pittsburg, March 1994. IEEE.

[AHU87] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structure and Algorithms.* Addison-Wesley publishing Company, 1987.

[BBD+91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. Ledot, M. Meysembourg, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[BCM90] K. Birman, R. Cooper, and K. Marzullo. Isis and meta projects: Progress report. Technical Report TR 90-1103, Cornel University, March 1990.

[BD92] T. Bloom and M. Day. Reconfiguration in argus. In *Internation Workshop on Configurable Distributed Systems*, pages 176–187, London, March 1992. Imperial College of Science Technology and Medicine, IEE.

[BD93] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 2(8):102–108, March 1993.

[BDLM96] J. Berghoff, O. Drobnik, A. Lingnau, and C. Monch. Agent-based configuration management of distributed applications. In *3rd International Conference on Configurable Distributed Systems*, pages 52–59. IEEE, Maryland, May 1996.

[BDW+92] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, M. J. Gardner, and R. W. lichota. Building fault tolerant distributed applications with durra. In *1st International Workshop on Configurable Distributed Systems*, pages 140–151, London, March 1992. IEE.

[Blo83] T. Bloom. *Dynamic Module Replacement in a Distributed System.* PhD thesis, MIT Laboratory for Computer Science, March 1983.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 2nd edition edition, February 1994.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight casual and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), 1991.

[Buz94] L. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* PhD thesis, University of Newcastle Upon Tyne, October 1994.

[CK95] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In G. E. Kaiser, editor, *Proc. of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20 of *Software Engineering Notes*, pages 140–150, Washington D.C., October 1995. ACM.

[CL83] R. P. Cook and I. Lee. Dymos: A dynamic modification system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, pages 201–202, August 1983.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CN92] T. Coatta and G. Neufeld. Configuration management via constraint programming. In *1st International Workshop on Configurable Distributed Systems*, pages 90–101, London, March 1992. IEE.

[CN94] T. Coatta and G. Neufeld. Distributed configuration management using composite objects and constraints. In *2nd International Workshop on Configurable Distributed Systems*, pages 112–122, Pittsburg, March 1994. IEEE.

[Dat95] C. J. Date. *An introduction to database systems, Sixth Edition.* Addison-Wesley, 1995.

[Day87] M. S. Day. Replication and reconfiguration in a distributed mail repository. Technical Report MIT/LCS/TR-376, MIT Laboratory for Computer Science, April 1987.

[Dij72] E. W. Dijkstra. Hierarchical ordering of sequential processes. In C. A. R. Hoar and R. H. Perrot, editors, *Operating Systems Techniques*, New York, 1972.

[End92] M. Endler. *A Language for High-Level Programming of Dynamic Reconfiguration.* PhD thesis, Karlsruhe, Germany, August 1992.

[Etz92] G. Etzkorn. Change programming in distributed systems. In *1st International Workshop on Configurable Distributed Systems*, pages 140–151. IEE, London, March 1992.

[EW92] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *1st International Workshop on Configurable Distributed Systems*, pages 68–79, London, March 1992. IEE.

[FS91] O. Frieder and M. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems and Software*, 14(2), February 1991.

[FS96] H. Fossa and M. Sloman. Implementing interactive configuration management of distributed systems. In *3rd International Conference on Configurable Distributed Systems*, pages 44–51. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[GKC98] D. Giannakopoulou, J. Kramer, and S. C. Cheung. Analysing the behaviour of distributed systems using tracta. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1), 1998.

[GM95] J. Gosling and H. McGilton. Java white papers. Technical report, Sun Microsystems, 1995.

[Hil98] M. A. Hiltunen. Configuration management of highly-customizable services. In *4th International Conference on Configurable Distributed Systems*, pages 197–205, Annapolis, May 1998. IEEE.

[HL82] M. Herlihy and B. H. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[HP93] C. Hofmeister and J. M. Purtilo. Dynamic reconfiguration in distributed systems : Adapting software modules for replacement. In *13th International Conference on Distributed Computing Systems*, pages 101–110. IEEE, May 1993.

[HS94] P. T. Homer and R. D. Schlichting. Configuring scientific applications in a heterogeneous distributed system. In *2nd International Workshop on Configurable Distributed Systems*, pages 159–168, Pittsburgh, March 1994. Carnegie Mellon University, IEEE.

[HW96] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *3rd International Conference on Configurable Distributed Systems*, pages 70–80. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[HWP93] C. Hofmeister, E. While, and J. M. Purtilo. Surgeon: A packager for dynamically reconfigurable distributed applications. *Software Engineering Journal*, 8(2):95–101, March 1993.

[JC92] G. R. R. Justo and P. R. F. Cunha. Programming distributed systems with configuration languages. In *1st International Workshop on Configurable Distributed Systems*, pages 118–127, London, March 1992. IEE.

[Kin92] T. Kindberg. Equus: an environment for reconfigurable distributed computations. Technical Report 591, Queen Mary & Westfield College, Department of Computer Science, Mile End Road, London E1 4NS, 1992.

[Kin93] T. Kindberg. Reconfiguring client-server systems. Technical Report QMW-DCS-1993-630, Queen Mary and Westfield College, Department of Computer Science, Mile End Road, London E1 4NS, March 1993.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions On Software Engineering*, 15(11):1293–1306, November 1990.

[KM98] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *4th International Conference on Configurable Distributed Systems*, pages 99–100, Annapolis, May 1998.

[KMNS93] J. Kramer, J. Magee, K. Ng, and M. Sloman. The system architect's assistant for design and construction of distributed systems. In *4th IEEE Workshop on Future Trends of Distributed Computing*, pages 284–290, 1993.

[KMSD92] J. Kramer, J. Magee, M. Sloman, and N. Dulay. Configuring object-based distributed programs in rex. *IEE Software Engineering Journal*, 7(2):139–149, 1992.

[Kra90] J. Kramer. Configuration programming - a framework for the development of distributed systems. In *COMPEURO '90*, International Conference on Computer Systems and Software Engineering, Israel, May 1990. Imperial College of Science Technology and Medicine London UK, IEEE.

[Leh82] M. M. Lehman. Program evolution. In *Symposium on Empirical Foundations of Information and Software Science*, Atlanta, November 1982.

[Lim93] A. S. Lim. *A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems.* PhD thesis, University of Wisconsin-Madison, 1993.

[Lim96] A. S. Lim. Abstraction and composition techniques for reconfigurable of large-scale complex applications. In *3rd International Conference on Configurable Distributed Systems*, pages 186–193. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[Lis88] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lit96] S. M. Wheater M. C. Little. The design and implementation of a framework for configurable software. In *3rd International Conference on Con-*

figurable Distributed Systems, pages 136–143. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LW98] M. C. Little and S. M. Wheater. Building configurable applications in java. In *4th International Conference on Configurable Distributed Systems*, pages 172–179, Annapolis, May 1998.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA*, volume 22, pages 147–155. ACM, 1987.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying software architectures. In *5th European Software Engineering Conference*, pages 137–153, Sitges, September 1995. Spring-Verlag.

[MDK94a] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed systems. In *2nd International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburg, March 1994. IEEE.

[MDK94b] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(5):304–312, September 1994.

[MGK96] K. Moazami-Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *3rd International Conference on Configurable Distributed Systems*, pages 62–69. IEEE, Maryland, May 1996.

[MJ94] S. Menon and R. J. LeBlanc Jr. Object replacement using dynamic proxy updates. In *2nd International Workshop on Configurable Distributed Systems*, pages 82–91, Pittsburg, March 1994. IEEE.

[MKS89] J. Magee, J. Kramer, and M. S. Sloman. Constructing distributed systems in conic. *Transactions on Software Engineering*, 6(15):663–675, June 1989.

[MS95] M. Mansouri-Samani. Gem - a generalised event monitoring language for distributed systems. Technical Report DOC 95/8, Imperial College of Science, Technology and Medicine, July 1995.

[MU97] N. H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In *Second International Conference on Coordination Models and Languages*, Berlin, September 1997.

[MUWZ96] N. H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *3rd International Conference on Configurable Distributed Systems*, pages 89–97. IEEE, Maryland, May 1996.

[MWCB90] K. Marzullo, M. Wood, R. Cooper, and K. Birman. Tools for distributed application management. Technical Report TR 90-1136, Cornel University, July 1990.

[Obj95] The Object Management Group, OMG Headquarters, 492 Old Connecticut Path,Framington, MA 01701, USA. *The Common Object Request Broker: Architecture and Specification*, July 1995. Version 2.0.

[OdP96] I. Oueicheck and X. Rousset de Pina. Dynamic configuration management in the guide object-oriented distributed system. In *3rd International Conference on Configurable Distributed Systems*, pages 28–36. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[PH91] J. M. Purtilo and C. Hofmeister. Dynamic reconfiguration of distributed programs. In *11th International Conference on Distributed Computing Systems*, pages 560–571. IEEE, May 1991.

[PSWL95] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, 1995.

[Pur94] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser.

Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle, April 1992. Usenix.

[SD96] I. Sommerville and G. Dean. Pcl: A configuration language for modelling evolving system architectures. *Software Engineering Journal*, 11(2), 1996.

[SF88] M. E. Segal and O. Frieder. Dynamic program updating in a distributed computer system. In *Proceedings of the Conference on Software Maintenance*, pages 198–203, October 1988.

[Str93] B. Stroustrup. *The C++ Programming Language - Second Edition.* Addison Wesley, 1993.

[SW90] S.K. Shrivastava and S.M. Wheater. Implementing fault-tolerant distributed applications using objects and Multi-Coloured actions. In *Tenth International Conference on Distributed Computing Systems*, pages 203–210, Paris, May 1990.

[Ves94] S. Vestal. Mode changes in a real-time architecture description language. In *2nd International Workshop on Configurable Distributed Systems*, pages 136–146, Pittsburg, March 1994. IEEE.

[Wer98] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. In *4th International Conference on Configurable Distributed Systems*, pages 111–118. IEEE, Maryland, May 1998.

[WS96] I. Warren and I. Sommerville. A model for dynamic configuration which preserves application integrity. In *3rd International Conference on Configurable Distributed Systems*, pages 81–88. University of Maryland Institute for Advanced Computer Studies, IEEE, May 1996.

[You91] A. Young. *A Structural Approach To Fault and Change management in Distributed Systems.* PhD thesis, Imperial College London, May 1991.

[ZD94] M. Zimmermann and O. Drobnik. Specification and implementation of reconfigurable distributed applications. In *2nd International Workshop on Configurable Distributed Systems*, pages 23–34, Pittsburg, March 1994. IEEE.

BIBL LONDON UNIV.