A Component Architecture for System Extensibility

Antony Edwards

Undergraduate Thesis

School of Computer Science and Engineering
University of New South Wales

November 2000

Supervisor: A/Prof. Gernot Heiser

Abstract

Component-based programming has shown itself to be a natural way of constructing extensible software. Well-defined interfaces, encapsulation, late binding and polymorphism promote extensibility, yet despite this synergy, components have not been widely employed at the systems level. This is primarily due to the failure of existing component technologies to provide the protection and performance required of systems software. This thesis presents the design, implementation and performance of a component model for system extensions that allow users to to create and customise system services.

Effective access control is a crucial feature of any system. In an extensible system, however, where potentially any user can create and modify system services, access control is even more critical. Despite the increasing importance of access control due to extensibility and increased connectivity, the protection mechanisms provided by existing component systems remain primitive and ad hoc. This thesis presents the design, implementation and performance of a complete access control model for extensible systems.

Acknowledgements

As all theses, this work would not have been possible without the thousands of researchers who make their ideas, opinions, experience and results publicly available.

More directly, I would like to thank my supervisor Gernot Heiser, whose support has constantly been beyond the call of duty. I'd also like to thank the many other people who contributed their time to this thesis, in particular, Simon Winwood, Adam Wiggins, James Edwards, Jon Tidswell, David Clarke, Manuel Chakravarty and John Potter.

Finally, I'd like to thank my soul of inspiration, the eternally patient Lilika Limaye.

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Thesis Statement and Contributions . . . 3
1.3 Overview . . . 4

2 Background **5**

2.1 Mungi . . . 5
2.1.1 A Single Address Space . . . 5
2.1.2 Protection in a SASOS . . . 6
2.1.3 The Mungi Philosophy . . . 7
2.1.4 Mungi Overview . . . 8
2.1.5 Protection Domain Extension . . . 10
2.2 Component Software . . . 11
2.2.1 Motivation . . . 11
2.2.2 What Are Components? . . . 12
2.2.3 Component Concepts . . . 13
Instances . . . 13
Interfaces . . . 14
Encapsulation . . . 14
Subtyping . . . 14
Implementation Inheritance . . . 15
Polymorphism . . . 16
Events . . . 16
Late Binding . . . 16
2.2.4 Components $\neq$ Object-Oriented Programming . . . 17
2.2.5 Existing Component Architectures . . . 17
CORBA . . . 17
COM . . . 19
2.2.6 Relation of Existing Work to this Thesis . . . 20

3 Extensibility **21**

3.1 Overview . . . 21
3.2 Software Technology Requirements . . . 21
3.3 Security and Protection Requirements . . . 23
3.4 System Requirements . . . 23

3.5 Comparison of Extensible System 24

I A Component Model for System Extensibility 27

4 Component System Design 29
4.1 Overview 29
4.2 System Model 30
4.2.1 Conceptual Model 30
4.2.2 System Structure 32
Client 33
Component Implementation 33
Interface Objects 33
Component Stubs 34
Component Invoker and Component Adaptor 34
Component Locator 35
4.3 The Component Model 35
4.3.1 Interfaces 36
Interface Characteristics 36
Component Construction Model 38
Interface Objects 39
4.3.2 Encapsulation 39
4.3.3 Polymorphism 41
4.3.4 Code Reuse: From Inheritance to Composition 42
Inheritance 42
Implementation Inheritance vs Customisation 42
Implementation Inheritance 44
Composition and Aggregation 44
4.3.5 Customisation 46
Semantic Issues of Customisation 47
Interface Delegation 50
4.3.6 Events 51
4.3.7 Component Lifecycle 52
4.3.8 Dynamic Type Discovery 53
4.3.9 Exceptions 53
4.3.10 Static Interfaces 54
4.4 Interface Definition Language 55
4.5 Component Invoker and Component Adaptor 56
4.5.1 Type Declarations 56
4.5.2 Component Invoker Interface 57
ci_request 57
ci_reset 58
ci_get_opt and ci_set_opt 58
4.5.3 Component Adaptor Interface 59
4.6 Component Locator 60
4.6.1 Type Declarations 60

4.6.2 Component Locator Interface . . . 60
cl_get_entry_pt . . . 60
cl_set_entry_pt . . . 61
cl_get_component_class . . . 62
cl_reset . . . 62
cl_get_opt and cl_set_opt . . . 63

5 Component System Implementation 65
5.1 Overview . . . 65
5.2 System Model Implementation . . . 66
5.3 Component-Classes . . . 68
5.3.1 Component Structure . . . 68
5.3.2 Class Interface . . . 73
Constructor . . . 73
Destructor . . . 77
Create_Cicap . . . 77
5.3.3 Component Stub Code . . . 79
Removing Parameter Limitations . . . 80
Removing Return Value Limitations . . . 83
Procedural to Component-Oriented . . . 84
Access Control . . . 84
5.4 Interface Objects . . . 86
5.4.1 Interface Object Structure . . . 86
5.4.2 Interface Object Implementation . . . 87
Exceptions . . . 87
Creation Constructor . . . 89
CICAP Constructor . . . 89
Destructor . . . 92
Interface Method Stubs . . . 92
5.4.3 Client Example . . . 94
5.5 Component Model Implementation . . . 96
5.5.1 Interfaces . . . 96
5.5.2 Encapsulation . . . 96
5.5.3 Polymorphism . . . 97
5.5.4 Component Code Reuse . . . 98
Composition . . . 98
Aggregation . . . 98
5.5.5 Customisation . . . 102
Declaring a Delegatable Interface . . . 102
Registering the Overriding Instance . . . 103
Delegation Protocol . . . 105
5.5.6 Dynamic Type Discovery . . . 109
5.5.7 Persistence and Naming . . . 109
5.5.8 Static Interfaces . . . 109
Component Stubs . . . 110
Interface Objects . . . 111

5.5.9 Version Support . . . 111
5.6 Component Invoker and Component Adaptor . . . 113
5.6.1 ci_request . . . 114
5.6.2 ci_reset . . . 114
5.6.3 ci_get_opt and ci_set_opt . . . 114
5.7 Component Locator . . . 115
5.7.1 cl_get_entry_pt and cl_set_entry_pt . . . 115
5.7.2 cl_get_component_class . . . 115
5.7.3 cl_reset . . . 118
5.7.4 cl_get_opt and cl_set_opt . . . 118

6 Component System Performance 121
6.1 Overview . . . 121
6.2 Environment . . . 122
6.3 Mungi Micro-benchmarks . . . 122
6.4 MCS Micro-benchmarks . . . 123
6.4.1 Component Invoker and Component Locator . . . 123
6.4.2 Component Creation . . . 123
6.4.3 Method Invocation . . . 125
6.5 CORBA Comparison Project Micro-benchmarks . . . 127
6.5.1 Measurement Verification . . . 127
6.5.2 Dispatcher Performance . . . 128
6.5.3 Performance . . . 128
6.5.4 Multi-Threading . . . 130
6.5.5 Scalability . . . 131
6.6 Micro-benchmark Comparisons . . . 131
6.7 Macro-benchmarks . . . 133
6.7.1 OO1 . . . 133
6.7.2 An Extensible File System . . . 135
6.8 Conclusion . . . 137

II Access Control 139

7 Access Control in Component Systems 141
7.1 Overview . . . 141
7.2 Access Control Theory . . . 142
7.2.1 Glossary . . . 142
7.2.2 Access Control Matrix . . . 143
7.2.3 Role-based Access Control . . . 143
7.2.4 Domain and Type Enforcement . . . 144
7.2.5 Context Sensitivity . . . 145
7.2.6 Mandatory and Discretionary Access Control . . . 145
7.3 Access Control Mechanism . . . 147
7.3.1 Mandatory Access Control . . . 147
Entities, Operations and Validations . . . 147

Managing Domains and Types . . . 151
Discussion . . . 152
7.3.2 Discretionary Access Control . . . 153
7.3.3 Combined . . . 154
7.4 Examples . . . 154
7.4.1 Denning's Lattice . . . 154
Domains and Types . . . 155
Validate Access . . . 155
Legal Domains . . . 155
Legal Types . . . 156
7.4.2 Chinese Wall . . . 156
Domains and Types . . . 156
Validate Access . . . 157
Legal Domains . . . 157
Legal Types . . . 157
7.4.3 Containment . . . 158
Domains and Types . . . 158
Validate Access . . . 159
Legal Domains . . . 160
Legal Types . . . 160
Other Security Policies . . . 160
7.5 A Formal Model . . . 160
7.5.1 Terminology . . . 161
7.5.2 Access Modes . . . 162
7.5.3 Discretionary Access Control . . . 162
7.5.4 Mandatory Access Control . . . 164
7.5.5 Combined . . . 165

8 Access Control Implementation 167
8.1 Overview . . . 167
8.2 Policy Objects . . . 168
8.2.1 Policy Object Interface . . . 169
Legal Domains . . . 170
Legal Types . . . 170
Validate Access . . . 171
DomainStack . . . 171
8.2.2 Multiple Validations . . . 173
8.2.3 Registration . . . 174
8.2.4 Asynchronous Validation . . . 175
Issuing a Validation Request . . . 175
Handling Replies . . . 179
8.3 Kernel Mechanics . . . 181
8.3.1 Domain Assignment . . . 181
8.3.2 Type Assignment . . . 182
8.3.3 Object Accesses . . . 183
8.3.4 Invoke Operation . . . 185

8.4 Concluding Remarks . . . 188

9 Access Control Performance **189**
9.1 Overview . . . 189
9.2 Micro-benchmarks . . . 189
9.3 Macro-benchmarks . . . 191
9.4 Conclusion . . . 193

III Conclusion **195**

10 Conclusion **197**

IV Appendicies **201**

A Example Policy Objects **203**
A.1 Standard Definitions . . . 204
A.2 Denning's Lattice . . . 205
A.2.1 Header . . . 205
A.2.2 Implementation . . . 206
A.3 Chinese Wall . . . 207
A.3.1 Header . . . 207
A.3.2 Implementation . . . 208
A.4 Bell LaPadula . . . 209
A.4.1 Header . . . 209
A.4.2 Implementation . . . 210

Chapter 1

Introduction

> *The recent trend towards dynamically extensible systems holds the promise of more powerful and flexible systems.*
>
> **R. Grimm and B. Bershad, 1997** [GB97b]

1.1 Motivation

Extensibility is revolutionising system construction. Traditional monolithic operating systems have evolved in an ad hoc manner, making them large, complex, unreliable and slow. Extensible operating systems replace this chaos with a model for controlled evolution, resulting in smaller, faster, and more reliable systems.

Extensible operating systems allow units of code (extensions) to be dynamically added to the base system. These extensions can add new services to the system, and modify the behaviour of existing services. Extensibility provides three principle benefits.

- An extensible operating system kernel does not need to provide all the traditional system services, e.g. a file system, allowing it to be small, fast, robust and flexible.
- New system services and features can be easily added to a running system, allowing it to meet new requirements and take advantage of new methods and technologies. Traditionally, operating system developments have taken years to deploy in commercial systems; extensibility promises to reduce this latency dramatically.

- Application developers best know how to optimise and structure their applications. Extensible systems allow applications to customise system services to improve performance, correctness and simplicity [PB96]. For example, a real-time multimedia application could customise a general purpose network protocol, and a database server could customise the disk cache replacement algorithm.

Current work into extensible operating systems has focussed on two approaches [OSD94].

- Allowing user-developed modules to be dynamically added to the kernel. These modules export an interface that can be called by users. Such systems rely on 'safe' languages (e.g. Modula 3 [Nel91]), compile-time analysis and dynamic reference checks for safety. Prominent examples include SPIN [BSP⁺95] and VINO [SESS96].
- Providing a *trusted path* [DoD85, LSM⁺95] mechanism, such as a protected procedure call [DVH66] or IPC. Extensions execute as user tasks, using the standard system protection mechanisms for safety. Clients invoke extensions via the trusted path. For example, Amoeba [MT86] used a client/server extension model with an IPC based trusted path.

An extensibility mechanism requires flexibility, safety and performance. It is now widely accepted that flexibility and safety can be provided in user space, therefore, kernel-module based systems are motivated solely by performance [BSP⁺95]. Performance oriented research into μ-kernel construction, however, has resulted in trusted path mechanisms with overheads of 100-200 cycles [LES⁺97,GSB⁺99]. Given the disappointing performance of kernel-modules [BSP⁺95, SESS96], such systems are obsolete. Kernel-module based systems also require programmers to use a specific 'safe' language, and rely on a complex compiler for protection [Che94], resulting in a large *trusted computing base (TCB)*.

Trusted path based extensibility mechanisms are more suitable for the current generation of μ-kernel based operating systems. These systems provide flexibility, safety and performance at user level, however, as yet there has been limited investigation into the development of an *extension model* using these mechanisms. Such a model is essential for the interoperability of extensions. Without such interoperability, these systems simply translate the chaos of traditional systems to user level.

Component-based programming systems have shown themselves to be a natural way of constructing extensible software. Well-defined interfaces, encapsulation, polymorphism and late-binding, all encourage extensibility, yet despite this synergy, components have not been widely employed at the systems level. This is primarily due to the failure of existing component technologies to provide the protection and performance required of systems software. This thesis presents the design, implementation and performance of a component model for system extensions that allow users to to create, compose, customise, and extend system services.

Security protects a system from inappropriate use, with what constitutes 'inappropriate' being defined by a *security policy* [AGS83]. At the core of any security policy are restrictions on *who* can access *what* in *what way* [GB97b]. The specification and enforcement of such restrictions is called *access control*, and is essential for systems to be able to enforce effective security. Access control is becoming increasingly vital due to three key factors.

- Increased connectivity, driven by the popularity of the Internet, means that there is no such thing as a single user system.
- Increased data sharing, driven by the wide-spread integration of computer systems into core buisness *processes*, is creating a need for more sophisticated protection.
- Software is becoming more fine-grained, i.e. composed of a number of modules each performing a particular task. Modules come from a variety of sources and may not be entirely trusted, therefore, each should execute with least privilege.

Effective access control is therefore a crucial feature of any system. In an extensible system, however, where potentially any user can create and modify system services, access control is even more critical. Despite the increasing importance of access control due to extensibility and increased connectivity, the protection mechanisms provided by existing component systems remain primitive and ad hoc. This thesis presents the design, implementation and performance of a complete access control model for extensible systems.

1.2 Thesis Statement and Contributions

It is my thesis that a lightweight component model, implemented using a trusted path mechanism, can provide safe, effective, and efficient, system extensibility.

Addressing these claims, this thesis makes the following contributions.

- A component model for system extensions.
- An implementation of this component model on the Mungi OS, that takes full advantage of the abstractions provided, e.g. a single-address-space.
- An access control model capable of providing effective system, service, and user protection, in extensible environments.
- An implementation of the access control model on the Mungi OS, that allows policies to be specified by user-level objects.

1.3 Overview

Chapter 2 contains background material for this thesis. It describes the Mungi operating system and component-based programming. Background material relating to access control is presented as it becomes relevant in Chapter 7.

Chapter 3 presents the requirements of an extensibility model and an extensible operating system, providing a clear evaluation criteria for the model presented in Chapter 4.

Chapters 4, 5 and 6 respectively describe the design, implementation and performance of a component model for system extensibility.

Chapters 7, 8 and 9 respectively describe the design, implementation and performance of an access control model for extensible operating systems.

Finally, Chapter 10 contains some concluding remarks.

Chapter 2

Background

2.1 Mungi

Mungi is a research operating system being developed at the University of New South Wales by the Distributed Systems (DiSy) group. This section provides an introduction to the Mungi philosophy, abstractions and implementation. A more detailed description of Mungi can be found in [HEV+98], and the full Mungi paper trail can be accessed at:

`http://www.cse.unsw.edu.au/~disy`

A large proportion of the material presented in this section has been adapted from previously published Mungi descriptions [Voc98, Lau00, HEV+98, HLR98, HVGR99].

2.1.1 A Single Address Space

Mungi is a single-address-space operating system (SASOS), i.e. all processes on all computing nodes in the system execute within the same virtual address space. That address-space contains all data, transient as well as persistent. Within this *single-level store* data are identified through their 64-bit addresses.

A SASOS provides a number of advantages over the traditional per-process address-space model. These advantages were recognised and realised by systems such as IBM System/38 [HSH81] and Monads [RA85]. Most importantly [Fab74, Sol96, WMR+95]:

- Data sharing is made easier. The traditional, multiple address-space, model relies on the fact that one process cannot address data that is in another address-space, for protection. Sharing information between separate address-spaces requires the explicit intervention of the kernel and some other subsystem (usually the file system or IPC). In contrast, a single-address-space system gives all processes the ability to address all data in the system. This uniformity of naming, and lack of explicit kernel involvement in communications, makes a SASOS an ideal environment for sharing information between processes.

- Trivial process migration. In a *distributed* single-address-space system, process migration can be as simple as moving a thread control block from one node to another. As the process starts execution on the new node its working set is automatically faulted over to the new machine.

- No reliance on message passing. As the global address space is the natural conduit for the transfer of information, there is no need to introduce a separate IPC abstraction.

- Lower context switch overhead. As a result of the fact that all physical to virtual translations are the same for all processes, there is no need to flush caches and translation information when switching processes. A context switch in a single address-space only needs to change the active protection information.

Original single address-space systems used custom hardware to create a sufficiently large address space (32-bits is inadequate). Since then however, 'off the shelf' processors with a 64-bit address space have appeared, e.g. the DEC Alpha [Dig92] and MIPS R4000 [KH92], allowing single address-space systems to run on standard hardware. It is also interesting to note that the recently released IA-64 [IA-00], Intel's 64-bit processor architecture, contains *protection keys* to separate virtual address translation and memory protection. Such hardware is the foundation for an efficient SASOS implementation.

2.1.2 Protection in a SASOS

While a SASOS makes sharing data easy, this must not be at the expense of protection. Traditional operating system protection is based on isolating each process in a private address-space, and only allowing address-space boundaries to be crossed via the operating system. Access to all objects external to a process's address-space is therefore under the full control of the system. As there are no address-space boundaries in a SASOS, traditional protection methods cannot be employed.

As far as protection is concerned, the concept of an address-space is replaced by a *protection domain*, which is the set of objects a process is allowed to access. As in every virtual memory system, a process can only access areas of virtual memory for which a mapping to physical memory has been established, and every attempt to access unmapped memory will result in a page fault. When the system handles that fault, it can verify whether the process has permission to access the memory region, i.e. whether it is part of the process's protection domain. If it is not, the system will generate a *protection fault*. In essence, protection in a SASOS is provided by controlling what is *accessible*, rather than what is *addressable*.

Protection domains are defined as a set of *capabilities*, which confer to a holder the right to perform specific operations on a given object. Mungi capabilities are described below.

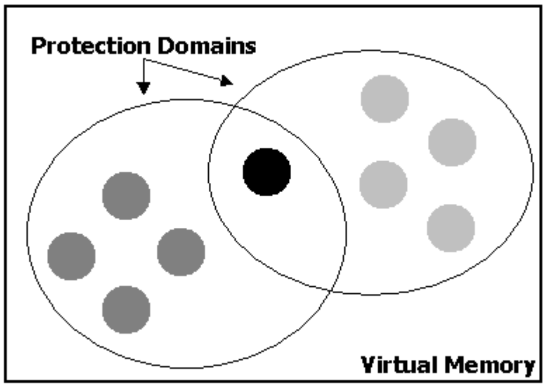


Figure 2.1: Mungi Protection Domains

2.1.3 The Mungi Philosophy

A number of goals that guided the design of Mungi, originally presented in [Voc98], are reproduced below:

Simplicity: Mungi should present a 'pure' single address-space. It is the premise that a huge, flat, address-space abstraction is sufficient to implement all the needed services, without introducing other name spaces. As a result, traditional kernel services, such as I/O and interprocess communication (IPC), are accomplished through the single, shared, address-space. The kernel, and all kernel data structures, are also part of this address space.

Usability: Mungi presents a simple, intuitive, interface to users, based on the single address-space paradigm.

Flexibility: Mungi should provide a simple set of basic abstractions, which can be used to build more complex abstractions. By keeping mechanism and policy separate, Mungi provides services that applications can use to implement policy.

Protection: Protection has to be simple, intuitive, robust and reliable. Consequently, protection must be designed into Mungi at the lowest level.

Performance: Performance is the overriding factor in almost any software. Simple and intuitive abstractions are irrelevant if their execution cost is prohibitive.

No reliance on custom hardware: Mungi has to run on standard 64-bit hardware. History has shown that OS projects involving new hardware have difficulty gaining acceptance, e.g. Intel 432 [Org93] and Monads [RA85]. Furthermore, 'off the shelf' hardware is less expensive and usually more reliable than custom hardware.

2.1.4 Mungi Overview

In addition to the single address-space discussed above, Mungi provides abstractions for *threads, objects, capabilities, exceptions* and *protection domains*. A *bank account* abstraction is also provided, which is used to impose limitations on resource usage.

Threads: are the execution abstraction in Mungi. A thread is a lightweight instruction stream, associated with exactly one, volatile, protection domain.

Mungi Objects and Capabilities. Virtual memory is allocated in contiguous, page-aligned segments called *objects*, which are also the unit of protection. A process is granted certain rights to all or none of an object.

Access is controlled via password capabilities [APW86]; when an object is created, the system returns a capability to the user which contains the object's base address and a password. Such a capability grants full (read, write, execute and destroy) rights to the object and is called an *owner capability*. A process holding an owner capability can register less powerful capabilities for an object.

Mungi capabilities can be freely stored or passed around without system intervention. They are protected from forgery by their password, which is registered in a global, distributed, data structure called the *object table* (OT). When validating a capability the system compares the capability's password with the list of valid passwords stored in the OT, and grants access if the requested operation is compatible with the access mode stored with the password in the OT. Validations in Mungi are cached for performance. The validation process is shown in Figure 2.2.

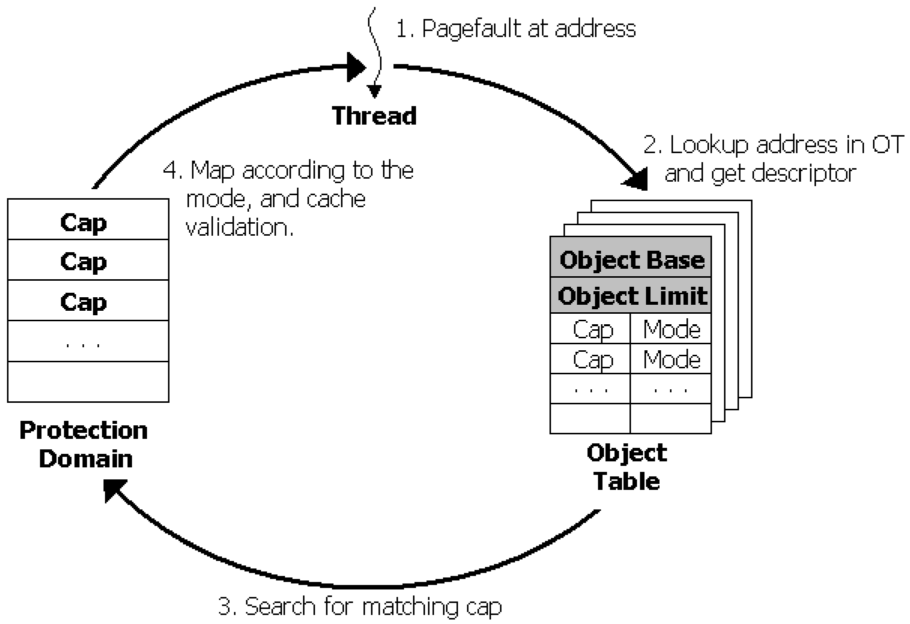


Figure 2.2: Mungi Access Validation

Exceptions: Exceptions are generated as a result of a program fault (e.g. division by zero or a protection fault). An exception handler can be registered to handle such an event. If an exception occurs for which a handler has been registered, that handler is called as an un-programmed function call of the thread which caused the exception. In particular, the exception handler executes within the faulting thread's protection domain, using the faulting thread's stack. If an exception occurs that is not associated with a particular thread, the system selects any thread within the faulting APD to handle the exception.

Protection Domains: define the objects that a thread may legally access. A protection domain is a set of pointers to *capability lists* (C-lists). Contrary to classical capability systems, these C-lists are not system objects, they are standard objects maintained by appropriate users. Multiple threads may execute within a single protection domain.

Bank Accounts: An object, once created, persists until explicitly destroyed, and may outlive its creator. To reduce a proliferation of garbage objects, a *kill list* is maintained for each task, containing all objects created by that task. When a task finishes, all the objects in the associated kill list are destroyed. An object may be removed from the kill list using an explicit system call.

While the kill list helps to reduce the amount of garbage objects in the system, this is not sufficient to prevent all secondary storage eventually filling up with unused objects. Automatic garbage collection does *not* provide a solution [HLR98]. As in a traditional file system, persistent objects are normally entered into a directory, which associates human-readable names with 64-bit object addresses. As long as the directory continues to contain a reference to an object, it cannot be automatically removed as garbage. The system must rely on users to manage their storage.

To assist users in managing their storage, a scheme derived from the *rent* model used in Monash University's Password Capability System [APW86] and the *bank accounts* used in Amoeba [MT86], is used. Whenever an object is created, a bank account must be supplied, and the bank account reference is recorded in the object's OT entry. A *rent collector* periodically charges the account for the disk storage used by the objects. A *paymaster* periodically deposits funds into each account. An empty or overdrawn account cannot be used to create new objects, forcing the user to clean up. An *account statement* is issued by the rent collector, to show users where their money goes.

Users may also create additional bank accounts. These accounts are funded by another bank account the user owns, rather than the paymaster. For example, a user may specify that a created bank account receives twenty percent of the funds deposited into their main account. By using different accounts for different groups of objects, users can exercise more precise control over their resource usage.

2.1.5 Protection Domain Extension

Extensibility is supported by a protected procedure call mechanism [DVH66] called *Protection Domain Extension* [VERH96]. PDX allows a thread's protection domain to be extended, in a controlled manner, for the duration of a procedure call. A Mungi object containing code can register a set of valid entry points into the object. Along with the entry points, a set of capabilities which will be added to the caller's domain for the duration of the call is specified. A thread holding a PDX capability to an entry point may invoke it via the system call:

```
int PdxCall( void *entry_pt, cap_t param, cap_t *ret, void *pd );
```

This invokes the method at `entry_pt` in a protection domain that is the union of the passed protection domain `pd`, and the C-list registered with the entry point. Figure 2.3 shows how the (shaded) active protection domain changes for the duration of a `PdxCall()`.

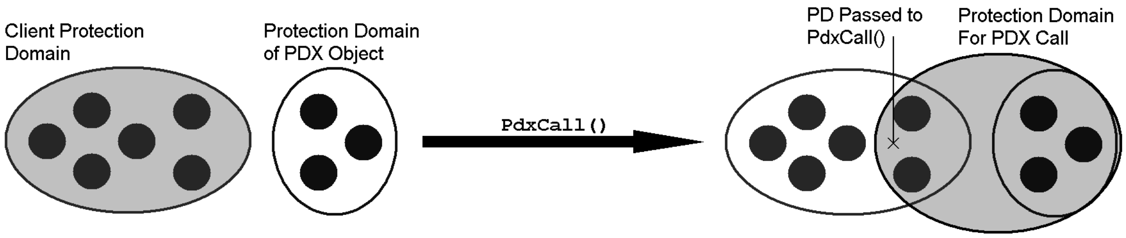


Figure 2.3: PDX Protection Domains

PDX domains are cached, by Mungi, so that a thread wishing to execute in a domain can do so quickly. Caching protection domains has similar performance benefits to those reported for LRPC [BALL89].

2.2 Component Software

> *Object-oriented computing has failed, but component software ... is succeeding. Here's why.*
>
> **Jon Udell, 1994** [Ude94]

2.2.1 Motivation

Components are a software engineering device motivated by the deficiencies of existing software construction techniques:

- Modern software is large, complex, and monolithic. It is excessively time-consuming and expensive to develop, maintain, and extend.
- Most software is bloated, containing every possible feature. Generally, people use less than 10% of an application's features [OHE96], so that the rest simply add complexity and overhead.

- Integration of applications is difficult. As new technologies are often required to communicate with existing software, this is a major concern of many organisations.
- Distributed software requires a unit of distribution.

These problems are not new. Increasing software size and complexity was identified over thirty years ago as the fundamental obstacle to quality software engineering, i.e. the 'software crisis'. An abstract solution to all these problems has also been known for thirty years - modularity. Modular software encourages reuse, customisation, scalability and maintainability. Modules are the software analogue of the integrated circuits (ICs) used in the digital electronics industry. Object-oriented programming claimed to provide these software ICs, however, it has failed to deliver. Software constructed using object technologies are still usually monolithic, and their parts are no more reusable than traditional procedural libraries. Object-orientation has failed to provide effective modules because:

- Object definitions are purely technical. Given the flexibility of software construction, a model for object use is required.
- Software modules are the units of abstraction, analysis, development, extension and maintenance. Objects are too fine-grained to assume these roles [Sut97].
- Object-oriented language designers lost focus. Rather than directly addressing the original issues, most efforts focussed on second level problems. In particular, a large number of projects focusing on inheritance semantics failed to relate their efforts back to basic goals such as module reuse and software extension.

Object-orientation was the first major attempt to provide effective software modules, and in *this* role it did not fail. It was unreasonable to expect that the first attempt would provide a complete solution. An indication of the success of object-orientation in this role is that many of its concepts have been incorporated into the next generation of software modules, i.e. components.

2.2.2 What Are Components?

So what are components? Learning from the mistakes of object-orientation, components are defined conceptually rather than technically. The focus is on the role of components within the software construction process.

- Jed Harris, ex-President of Component Integration (CI) Labs [OHE96].

  > *A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability.*

- A 1994 Meta Group white paper [Szy98].

  > *Software components are defined as prefabricated, pretested, self-contained, reusable software modules - bundles of data and procedures - that perform specific functions.*

- Clemens Szyperski in *Component Software: Beyond OO Programming* [Szy98].

  > *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and is subject to composition by third parties.*

Modularity is the fundamental concept in each definition. Components are software units that can be independently developed and then composed into larger services and applications. That component technologies have developed from this abstract definition *to* a technical definition, is a key feature of their success.

This thesis aims to develop a technical definition of a component that satisfies these abstract definitions. Chapter 4 presents this definition.

2.2.3 Component Concepts

This section briefly introduces the technical concepts commonly associated with software components. A detailed investigation of these concepts to produce a complete technical definition of a component is the subject of Chapter 4.

Instances

It is critical to distinguish between a *component-class* and a *component-instance*. A component-class is the *definition* of a component, which may include interface definitions, code, global data, resource capabilities, etc. A component-instance is an instantiation of the component-class, and is associated with some per-instance state data. The operations of a component-class use and modify this state data, which therefore determines the context in which operations execute. Operations are always invoked on component-instances.

This situation is analogous to object-oriented programming. A class provides the definition of the interface, an implementation of the operations and defines the per-instance data. Clients create instances of the class, i.e. *objects*, which contain a particular set of values for the per-instance data.

Interfaces

Interfaces are the glue that connect clients to component-instances. An interface is a description of a set of operations that a client may invoke on an instance of a particular component-class. Descriptions may include information such as the number and type of arguments, the return type, pre and post-conditions for state data, the maximum CPU time the operation may consume, or just about any other factor.

An interface, and its semantic description, contain all the information a client requires to use a component-class. Therefore, a client is not concerned with how a component-class implements an interface, enabling easy composition.

Encapsulation

Encapsulation hides the internal operation of a module from clients. This involves making the state data inaccessible, *data encapsulation*, and forcing clients to operate on instances via the defined interface, *control encapsulation*. Encapsulation is required for the effective separation of specification and implementation.

Subtyping

A *type* is a set of values which an instance of that type may take on. For example, an instance of the `short` type in C may take on the values `{-32768...32767}`. A type T_1 is a *subtype* of type T_2, if T_1 is a superset of T_2. For example, a `long` in C may take on the value `{-2147483648...2147483647}`, therefore `long` is a subtype of `short`.

Crucially, a subtype is *substitutable* for its supertypes.

An object-oriented type, e.g. a class, defines a set of operation that may be performed on an instance, and the per-instance state data. A class C_1 is substitutable for C_2 if it exports all the operations, and contains all the public data members, of C_1.

Object-oriented languages (e.g. C++, Java, Oberon [Obe94], Smalltalk [GR89], Eiffel [Mey90]) almost always form subtypes using *interface inheritance*, i.e. a type T_1 declares itself as a subtype of an already existing type T_2. T_1 automatically 'inherits' all of T_2's operations and public data members. Although in most object-oriented languages the subtype also inherits the implementation of the operations, this is a separate issue, with a separate goal and is therefore discussed separately.

Implementation Inheritance

Implementation inheritance involves a type acquiring an implementation of certain operations it exports from another type. Implementation inheritance is best explained using an example, which is presented below in C++.

```
class base {
public:
    base();
    ~base();
    some_method();
};

class derived : public base {
public:
    derived();
    ~derived();
    some_other_method();
};
```

As C++ combines subtyping and implementation inheritance, both feature in this example. Class `derived` is specified as a subtype of `base`, so that it automatically exports the method `some_method()`. Implementation inheritance means that `derived` also inherits the implementation of `some_method()` provided by `base`. Therefore, if a client invokes `some_method()` on an instance of type `derived`, the code invoked is the implementation contained in `base`.

Implementation inheritance is the reuse model of almost all object-oriented languages.

Polymorphism

Polymorphism is the ability of a strongly typed client variable to refer to instances of different component-classes. Traditional languages handle variants, e.g. a union, by explicit case analysis. If a new variant is added, every case analysis must be upgraded, which is obviously not extensible or easily maintainable.

Polymorphism is closely related to the concept of substitutability described above. Therefore, object-oriented languages use subtyping to provide polymorphism.

Consider a user-interface toolkit where each element, e.g. a button, text-box, label or list, is an instance of a component-class corresponding to the element type. A *container* manages the elements of a form, including the position and size of each one. Without polymorphism, redrawing a form would require a different client variable type, and hence different code, to be used for positioning each element type. Furthermore, new element types could not be used without modifying the container code. Polymorphism therefore reduces the amount of code required, and encourages extensibility.

Events

A standard operation results in control being transferred from the client to the component implementation. Conversely, *events* are operations invoked by the component implementation, that transfer control *to* the client. Events are also known as *up-calls* and *callbacks*.

Events allow components to push information to the client, rather than requiring clients to continually poll for changes. This is especially useful for components that interact with asynchronous devices such as humans, printers and disks.

Late Binding

A polymorphic client variable may, by definition, refer to instances of different component-classes. *Late binding* allows the component-class referred to by such a variable to be determined at run-time. As components and clients are decoupled, the component implementation is not included in the standard linking process. Therefore, when the component-class is determined, it may need to be dynamically located, linked and loaded.

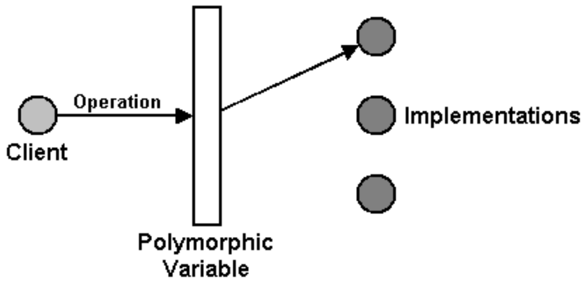


Figure 2.4: Late Binding

2.2.4 Components $\neq$ Object-Oriented Programming

Most of the component concepts introduced in the previous section are also present in object-oriented languages. For this reason, people often equate components with object-orientation. Despite their similarities however, they are different technologies, with different goals. Components are principally a conceptual model for software design, while object-orientation is a programming paradigm. Understanding this distinction is essential when evaluating a technical, rather than abstract, component model.

2.2.5 Existing Component Architectures

This section briefly introduces the dominant players in the component industry. Specific details of their standards are discussed in Chapter 4 as they become relevant.

CORBA

The *Common Object Request Broker Architecture (CORBA)* is a middleware standard developed by the Object Management Group (OMG), a consortium with over 700 member companies. The notable exception is Microsoft, which uses its own middleware architecture called the *Component Object Model (COM)*.

An *Object Request Broker (ORB)* is the heart of CORBA. It allows clients to send requests to, and receive responses from, objects. An ORB provides transparency to the client, who is unaware of where the object is located, its implementation language, its operating

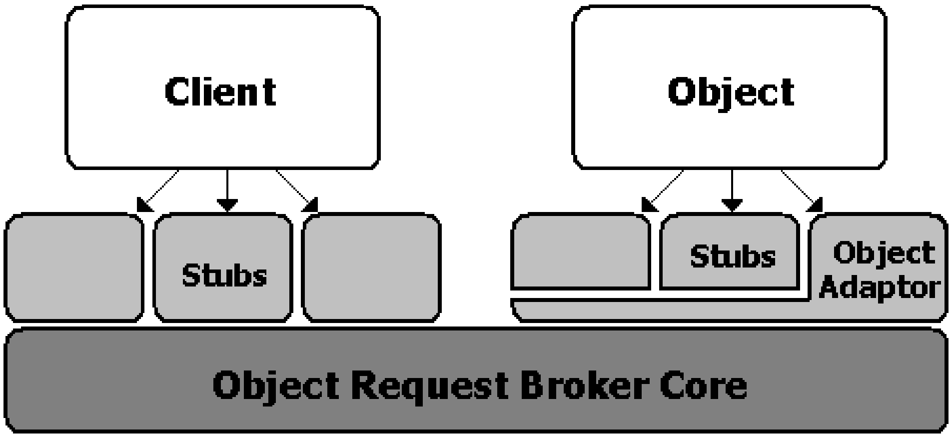


Figure 2.5: CORBA Architecture

system and all other environmental concerns. Objects export an interface defining the operations that clients may perform. CORBA interfaces are defined using a declarative language called the *Interface Definition Language (IDL)*. Clients can dynamically discover objects, and invoke methods on their interface.

Although other mechanisms exist, clients generally invoke operations using clients stubs. CORBA stubs are similar in concept to RPC stubs [BN84], in that they make a CORBA request appear as a standard language method invocation to the client program. Internally, stubs invoke the ORB interface to initiate a request and receive a reply. Object adaptors are responsible for activating object implementations, and providing object-side ORB independence.

CORBA was originally released in 1991 as version 1.1. Its goal was to provide a standardised client/server communications architecture, so that clients could easily use objects produced by different vendors. This remained the focus of the widely adopted CORBA 2.0 standard, released in July 1995. Since then, software construction models have progressed so that the strong client/server flavour of CORBA 2.0 no longer satisfies the definition of a software component. OMG have responded to this with the development of a new specification, *CORBA Components* [OMG99d], which creates a component environment above the existing CORBA architecture. This specification also includes a component model, based on Sun's *Enterprise Java Beans (EJB)* [Jav97], which was unfortunately missing from the original standards. *CORBA Components* has yet to be released as a standard.

COM

Microsoft are moving the entire Windows operating system into their component technology, called the *Component Object Model (COM)* [COM95]. COM is a very different standard to CORBA. In particular, it is a binary standard, that is (not surprisingly) unconcerned with issues of interoperability and platform independence.

COM has an architecture that is similar to, though somewhat simpler than, CORBA. This simplicity is due to COM only being required to support a single implementation in a known environment, while CORBA is must allow many implementations in diverse environments.

Clients interact with instances via interfaces defined in Microsoft IDL (MIDL). When an instance is created, the *Service Control Manager (SCM)* locates the implementation and activates it appropriately. Clients can then invoke methods on the object without the direct intervention of COM. Objects can either be in-process, out-of-process on the local machine, or out-of-process on a remote machine. If the object is in-process, then clients directly invokes the object's methods. Otherwise, the client invokes methods on an in-process *stub object* that performs a remote procedure call (RPC) with a corresponding stub in the object's process.

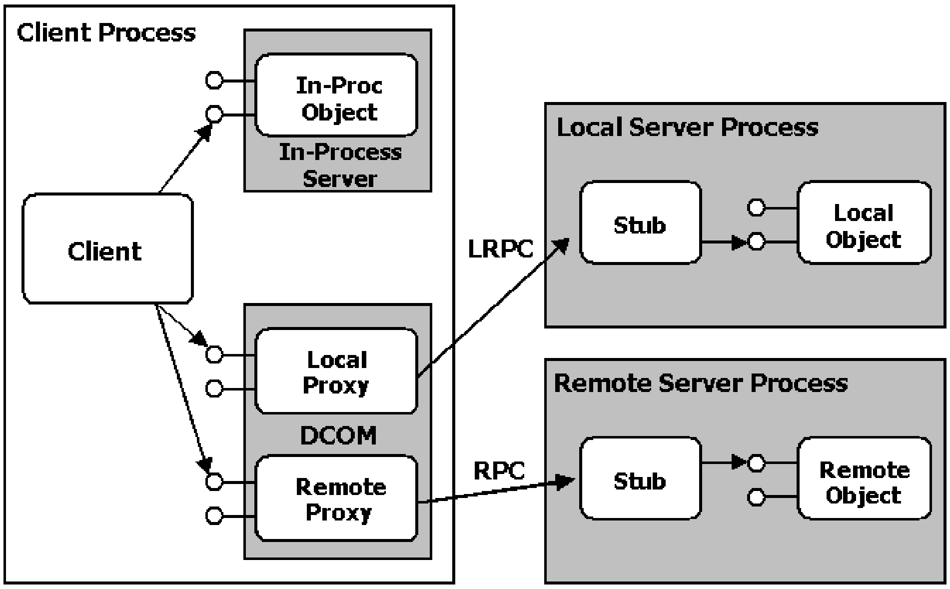


Figure 2.6: COM Communications

2.2.6 Relation of Existing Work to this Thesis

CORBA and COM define how components are located and how requests are communicated. They are designed to provide software interoperability. In contrast, this thesis is concerned with the component *model*, i.e. what components are, how they behave, and how they are used to construct software. A component model and a communication standard are complementary rather than competing, both are required for a complete component solution.

It is my intention that the component model described in this thesis will use the *CORBA Components* communications standard. Although this goal is not directly addressed due to time constraints, its future realisation is facilitated by the use of CORBA standards where appropriate. In particular, interfaces are described using a subset of OMG's IDL, and the OMG *C-Language Mapping Specification* [OMG99b] has been used. For clarity, these standards are described as they become relevant.

Chapter 3

Extensibility

Extensibility must be designed into a system.

Clemens Szyperski, 1998 [Szy98]

3.1 Overview

This chapter presents the requirements of an extensible operating system. The requirements are separated into three categories.

Software technology requirements specify the functionality that an extension model must provide. These requirements apply to both system and application extensions.

Security requirements distinguish application and system extensibility [Che94]. If extensions are to provide secure services, and become an integral part of the base system, the extensibility mechanism must provide adequate security and protection.

System requirements specify the features that an extensible *system* must provide.

3.2 Software Technology Requirements

Software technology requirements describe what extensions must be able to do.

- *Add services dynamically.* This is the most basic requirement of extensible software. Once an extension is activated, other entities in the system should immediately be able to invoke the extension code.

- *Add services polymorphically.* Polymorphism allows existing clients to use a new service without modifying their code. Without polymorphism, clients and services are tightly coupled, almost to the point of being fragments of the same application, e.g. an FTP server and an FTP client.

- *Be late-binding.* A client should be able to dynamically discover and use new services. In particular, clients should not have to re-link to use a new service. With full late-binding, the extension code to be invoked is determined dynamically using the service reference, linked and loaded.

- *Deploy updated versions.* An extension may require modifications to remove bugs, improve performance, improve flexibility, or take advantage of new technologies. Clients of the original extension should automatically migrate to the new extension.

- *Impose as few restrictions on programmers as possible.* Technologies that impose restrictions usually fail the psychological acceptance test. Restrictions, by definition, also reduce the flexibility of the mechanism, hence reducing its application domain.

- *Module extension.* Extensions are not stand-alone servers that can be transparently replaced. New extensions compose and customise existing extensions, building upon the supported functionality to provide new services. A new extension should be able to reuse and extend existing extensions [GB97a].

- *Perform fine-grained customisation.* Customisation allows the addition of domain specific knowledge to system services, improving performance, correctness and simplicity [GB97a]. The level of improvement depends on how specific the added knowledge is, which in turn depends on how fine-grained customisation is. For example, per-client customisation allows greater specialisation of a service than per-system customisation, resulting in greater benefits to applications.

- *Provide a semantic model, or standard, for extension construction.* Without a semantic model, clients must contain extension-specific knowledge, reducing the interoperability and reuse of extensions.
- *Provide acceptable performance.* No functionality can compensate for inadequate performance, especially at the systems level where performance is paramount.
- *Transparent interposition.* Extensions may be *interposed*, i.e. inserted between, two existing extensions, or an extension and a client.

3.3 Security and Protection Requirements

System extensions must provide additional features related to protection.

- *Execute in an amplified protection domain.* This allows extensions to provide secure access to privileged resources. For effective security, the amplified protection domain should follow the principle of least privilege.
- *Store privileged data.* Services must be able to store privileged data in persistent objects, i.e. extensions cannot be memoryless [Lam73].
- *Perform discretionary access control.* A client must only be able to invoke methods they have access to, on extensions they have access to.

3.4 System Requirements

An extensible operating system must provide the following features.

- *An extensibility mechanism* satisfying the above requirements.
- *Mechanisms allowing authorized extensions to perform privileged system operations.* If the system does not allow extensions to exert significant control, the power of extensions is greatly diminished.
- *Effective access control.* Extensible systems are inherently dynamic, and contain a (possibly) large number of fine-grained extensions. The system must be able to maintain effective access control in this environment.
- *A minimal trusted computing base (TCB).* Effective, reliable security requires a small, well structured, TCB [SS75].

3.5 Comparison of Extensible System

Table 3.1 compares a number of existing extensible systems in terms of the requirements presented above. The goal of this section is to identify the specific features that distinguish extensible operating systems from traditional operating systems. It does not aim to be a taxonomy of extensible systems, providing only a brief description of the systems examined. Performance results were obtained from a variety of published sources, using different hardware, and so should not be directly compared. Chapter 6 contains a comparison of existing systems.

- **Unix** A traditional, Unix style, operating system serves as a baseline for comparison, and illustrates what it is that extensible system do that traditional systems do not. Extensions in traditional systems are applications. An RPC is considered the equivalent of a method invocation; performance result taken from Chapter 6.
- **L**$_4$ [Lie96] is a second generation, IPC based, μ-kernel, implementing a custodial access control mechanism [Lie92]. L$_4$ is representative of a system providing a trusted path mechanism, but no extension model. Performance values are from [LES$^+$97], measured on a 100MHz MIPS R4600. A method invocation is equivalent to two IPCs plus one hundred cycles of stub code [HLP$^+$00].
- **SPIN** [BSP$^+$95] supports extensibility by allowing modules to be dynamically added to the kernel. Protection is based on the use of a type-safe language (Modula 3 [Nel91]), compile-time checks, and dynamic reference resolution. Performance values are from [BSP$^+$95], a cross-domain call on an Alpha 133MHz AXP 3000/400.
- **VINO** [SESS96] also allows user modules, called *grafts*, to be added to the kernel. Protection in VINO is based on software fault isolation [WLAG93], which involves compile-time checks of static memory references and inserting instructions before dynamically constructed references. A transaction system prevents resource hoarding. Performance values are from [SESS96]. As VINO overhead is produced by the inserted code, there is no standard overhead, and so a single value cannot be given.

	Unix	**L_4**	**SPIN**	**VINO**
Dynamic Services	$\surd$	$\surd$	$\surd$	$\surd$
Polymorphism	M	M	$\surd$	$\surd$
Late-binding	$\surd$	$\surd$	$\surd$	$\surd$
Version updates	M	M	$\surd$	$\surd$
Added Restrictions			Must use Modula-3	
Module Extension	$\surd$	$\surd$	$\surd$	$\surd$
Customisation	$\times$	M	$\surd$	$\surd$
Semantic Model	$\times$	M	$\surd$	$^1/_2$
Method Perf. (μs)	160	2.72	89	$103 \rightarrow 360$
Transparent Interposition	M	M	$\surd$	$\surd$
Least Privilege	$\times$	$\surd$	$\surd$	$\surd$
Store Priv. Data	$\surd$	$\surd$	$\surd$	UA
Add Acc. Control	M	$\surd$	$\times$	UA
Extension Control	$\times$	$\surd$	$\surd$	$\surd$
Fine Access Control	$\times$	$\surd$	$\surd$	UA
TCB (not incl. kernel)	‡	‡	*Compiler*	*Compiler*

M Though not naturally supported, this feature could be manually implemented.
UA This information is not available.
‡ Depending on policy, servers and chiefs may be added to the TCB.

Table 3.1: Comparison of Existing Extensibility Mechanisms

Part I

A Component Model for System Extensibility

Chapter 4

Component System Design

An object [component] is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

CORBA 2.0 Specification, 1999 [OMG99c]

4.1 Overview

This chapter presents the design of the Mungi Component System (MCS), a component-based programming system suitable for the development of system extensions.

A component-based programming system is not a single entity; it comprises a number of cooperating models and runtime elements. The purpose of this chapter is to describe in detail the models used, the roles of the runtime elements, and their client interfaces. The elements described in this chapter (in this order) are:

A System Model identifies the major elements of the system, the roles they play, and how they interact. It introduces the environment in which components and clients exist. Section 4.2 describes the system model.

A Component Model describes in detail what components are, how they behave, how they should be used, and how they interact with one another as well as clients. Section 4.3 describes the component model.

An Interface Definition Language (IDL) is a specification language used to unambiguously describe the external interface of components. MCS uses a subset of the

Object Management Group's (OMG) CORBA IDL. Details, references and examples are given in Section 4.4.

A Runtime is required to provide certain functionality. Runtime elements are identified in the system model (Section 4.2), and described in Sections 4.5 and 4.6. The MCS runtime is referred to throughout the chapter as MCS-RT.

Whilst reading this chapter, it is helpful to bear in mind the fundamental services provided by a component system:

- Component creation.
- Method invocation.

The system model and component model define the semantics of these two operations, i.e. *what* they do. MCS-RT is concerned with *how* they are done, and IDL specifies the syntax of the operations for each component-class.

4.2 System Model

A system model identifies the major elements of the system, the roles they play, and how they interact. It introduces the environment in which components and clients exist. This section describes the MCS system model.

4.2.1 Conceptual Model

Figures 4.1 and 4.2 illustrate the conceptual models for component creation and method invocation. Before describing these two services however, it is essential to reinforce the distinction between a *component-class* and a *component-instance*. An MCS component-class consists of the implementation code and the protection domain in which the implementation code executes. A component-instance contains a particular set of values for the state data, and is also a unit of access control (see Chapter 7). A component-class is analogous to a class in an object-oriented programming language, with a component-instance being equivalent to an object.

In Figure 4.1, a client requests that a new component-instance, of a certain component-class, be created. MCS locates the appropriate component implementation code, and

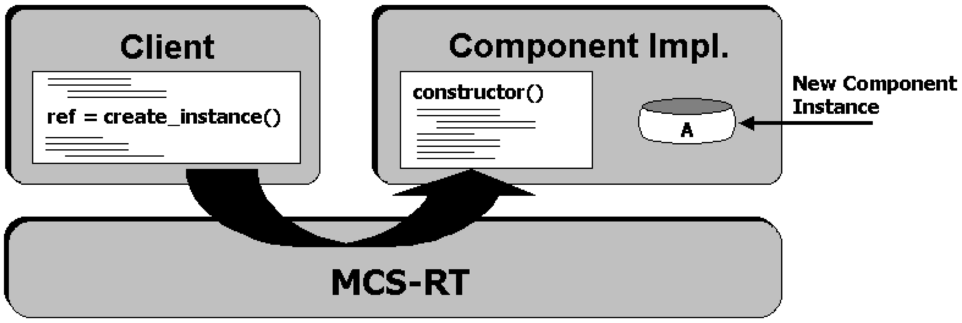


Figure 4.1: Component Creation

invokes a constructor within the component-class's protection domain. A new instance is then created *within the component-class's protection domain*, and a reference to this instance is returned to the client. The client uses this reference for method invocations, to specify the state data to be used. A client cannot directly access the component-instance as it is not within its protection domain.

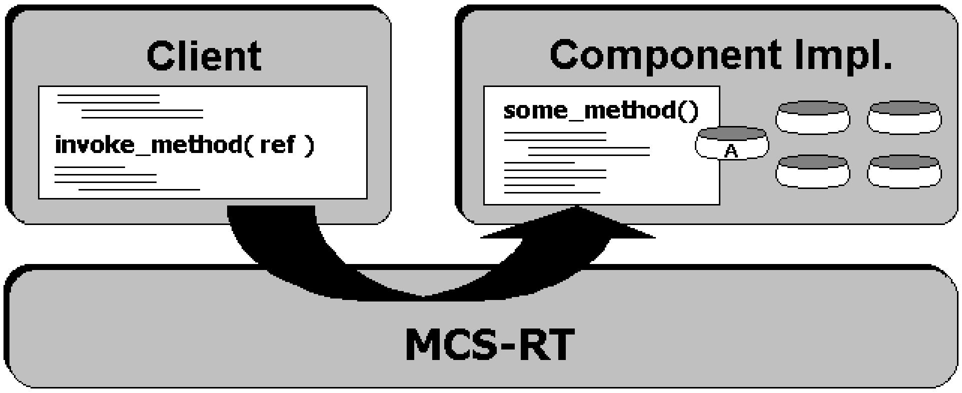


Figure 4.2: Method Invocation

In Figure 4.2, a client invokes a method using a reference to a valid component-instance. MCS determines the component-class of the reference, locates the component implementation, and invokes the requested method within the component-class's protection domain. The method executes using the state data stored in the specified component-instance. When the method completes, control is returned to the client.

Critical features of this model are:

- Client code and the component implementation execute in different, though not necessarily disjoint, protection domains. This allows components to provide secure access to privileged resources, by executing within an amplified protection domain. The component-class's protection domain contains the Mungi objects it needs to provide its service, the instances, and the client objects it is operating on. By providing the minimum required objects, this system model supports the principles of least privilege and mutual suspicion [CJ75].

- Component-instances do not exist in the client's protection domain. This means that clients cannot directly access instance state data, i.e. data is fully *encapsulated*. This encapsulation is ensured by the operating system rather than a compiler, making it suitable for storing privileged system data (e.g. a spooler file) [JLI98] .

- At the system interface level, component creation and method invocation are simply requests sent from one protection domain to another. Therefore, both these services can be implemented at user-level using a trusted path mechanism.

component implementations execute in a protection domain that is the union of a protection domain defined for the component-class, and a protection domain passed by the client. The portion of the protection domain defined for the component-class is referred to as the *protection context*, and is specified along with the component-class interfaces.

4.2.2 System Structure

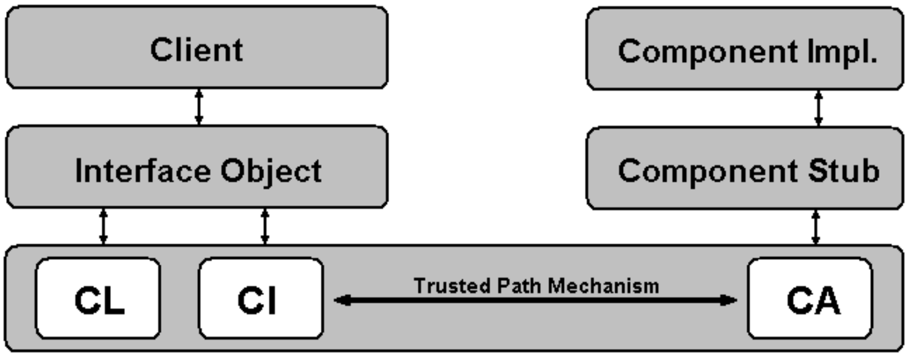


Figure 4.3: System Structure

Figure 4.3 decomposes the conceptual model presented in the previous section. MCS-RT has been split into three modules: the *Component Locator* (CL), the *Component Invoker* (CI) and the *Component Adaptor* (CA). In addition, both the client and the component implementation have been separated into two layers. The following sections describe each of these system elements.

Client

A client is any thread holding a reference to a component-instance, or requesting the creation of a new instance. Component implementations may themselves act as clients to other component-classes.

Component Implementation

A component implementation contains the code implementing a component's interfaces. Interfaces were introduced in Section 2.2.3, and are described for MCS in Section 4.3.1.

In the presence of events or upcalls (Section 2.2.3), the conceptual distinction between the component implementation and the client is obscured. The key distinguishing feature is that the component implementation has access to the state data of the instance being operated on, whilst the client does not.

Interface Objects

Clients do not directly call the MCS-RT interface. Instead, an abstraction layer is placed between the client and MCS-RT to provide:

- Transparency.
- Functionality.

On the client-side, this abstraction layer is called the *interface object*. An interface object is a representation of an interface of a component-*class*, using the client programming language's native object construct, e.g a C++/Java class, or a C abstract data type (ADT). Each interface object contains a reference for a component-*instance*. When a method is invoked on the interface object, it invokes the corresponding method on the component-instance. Components therefore appear to the client programmer as standard, native, objects. Interface objects are analogous to RPC client stubs [BN84], which make synchronous requests to remote servers appear to the client as normal procedure calls.

As MCS-RT only provides a standard interface to the underlying system's trusted path mechanism, interface objects also implement the semantics of the component model. A technical definition of the component model is the subject of Section 4.3.

Component Stubs

Component stubs serve the same purpose as interface objects, on the component-side. Stubs invoke the component implementation methods using native language conventions, so that the component can be implemented in a natural style. Stubs also implement the component model semantics.

Component Invoker and Component Adaptor

As its name suggests, the Component Invoker (CI) presents an interface that allows clients, or more likely interface objects, to invoke entry points within the component implementation. This service comprises two functions:

- Activating the component implementation.
- Transferring control, and limited request data, to the entry point.

Transferring the request to the component implementation is implemented using a, system provided, trusted path mechanism. The Component Adaptor (CA) is the destination of all requests. Together, the CI and the CA provide a standardised request mechanism, making component stubs and interface objects platform independent.

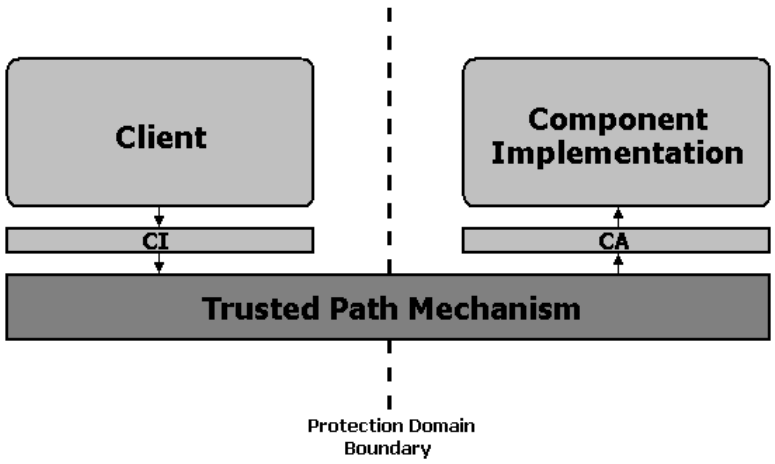


Figure 4.4: Component Invoker / Component Adaptor

Component Locator

Clients must provide the CI with the location of the component implementation, so that it can activate and invoke the appropriate code. Clients retrieve this information using the Component Locator (CL).

4.3 The Component Model

A component model provides the technical description of components, i.e. how they behave, how they are used, and how they interact with other elements. A component-based programming system's effectiveness is primarily determined by how well the technical definition correlates to the abstract definitions presented in Section 2.2. This section presents the MCS component model, which is based on four fundamental concepts:

Components = Interfaces + Encapsulation + Polymorphism + Composition

- **Interfaces** describe how clients interact with components. Although it may appear that interfaces are essentially a syntactic concern, they raise a number of critical issues related to the construction of components.
- **Encapsulation** enforces modularity.
- **Polymorphism** is an object-oriented software technology feature, that has been extremely effective in the construction of extensible software.
- **Composition** describes how components are used to construct software. Code reuse, an original motivation for components, is the primary concern of composition.

In addition to these, features such as events, exceptions and persistence, are also described. As stated in Section 2.2, many of the features of component-oriented software are also present in object-oriented programming languages. It is important that features of a component model are evaluated on whether they achieve the goals of component-oriented software presented in Section 2.2, and *not* on how 'object-oriented' they are.

4.3.1 Interfaces

Interface Characteristics

An *interface* is a description of a set of semantically related operations. MCS interface specifications contain the name, arguments and return type of each operation. A number of semantic modifiers may also be applied to constructs within a specification. These *modifiers* are described in the following chapter. Specifications are expressed in a subset of OMG IDL described in Section 4.4. An example of a simple interface specification is presented below, the OMG IDL syntax should be fairly intuitive.

```
interface IWeaponSystems {
    int AcquireTarget( target_desc_t *spec );
    target_desc_t FireAtTarget();
    void FirePattern( pattern_desc_t p );
    wstatus_t Status();
};
```

MCS interfaces are immutable. An interface provides a contract between clients and the component-class, specifying what functionality is available, and how it may be accessed. Allowing interfaces to change violates this contract, creating a number of problems, such as the syntactic fragile base class problem [IBM94]. Although solutions to *some* of these problems have been developed, e.g. by the SOM architecture [FCDR95], these are invariably complicated. Rather than allowing interfaces to change, MCS component-classes may *export* (see below) multiple interfaces. Interfaces may be added to a component-class, but never removed. This model provides all the flexibility of changeable interfaces, yet avoids all the problems. Note that not only may a component-class export multiple interfaces, a single interface may be exported by multiple component-classes.

MCS component-classes export one or more interfaces. Separating the methods of a component-class into semantic groups has a number of benefits in relation to polymorphism, code reuse and versioning. These benefits are described in their respective sections below. A component-class exports an interface by providing implementation code for *all* the operations of the interface, and *registering* their entry points with the Component Locator. An example component-class declaration is presented in Figure 4.5.

Every component-class also implicitly exports the *class interface.* This static interface (see Section 4.3.10) provides the methods required by the component model. Figure 4.6 contains the IDL specification of the class interface.

```
component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    provides IStellarCartography;
    provides IWeaponSystems;
};
```

Figure 4.5: Example Component Specification

```
exception MCSE_STE_INSFBUFFER { };

[static] interface IClassInterface {
    cicap_t constructor();
    void destructor( cicap_t ref );
    cicap_t create_cicap( cicap_t owner, short interfaces, char slot );
    void get_typeinfo( cap_t info, int size ) throws( MCSE_STE_INSFBUFFER );
};
```

Figure 4.6: Class Interface IDL

Interfaces cannot be subtyped (Section 2.2.3), i.e. an interface cannot be derived from an existing 'base' interface, as is allowed in most object-oriented programming languages, e.g. Java or C++. The exclusion of subtyping is discussed with polymorphism in Section 4.3.3.

Interfaces are often referred to as contracts. OMG IDL interface specifications contain only the signatures of the operations. The notion of interfaces as contracts has been extended by a number of research projects [HHG90, Hol92] to include safety conditions (i.e. invariants), progress conditions [CM98], performance requirements and resource consumption limits. What should be included in a contract, how contracts are specified, and how contracts are enforced, is a complex area of research and outside the scope of this thesis.

An interface should contain a relatively small set of semantically related methods. In addition to this, methods should have no dependence on the implementation of methods of other interfaces. Such dependencies create numerous problems for component composition and code reuse. The importance of this is discussed in Section 4.3.4.

Component Construction Model

> *A refinement occurs once it is accepted that something does not have to be only what it appears to be.*
>
> **Clemens Szyperski [Szy98]**

Traditionally, an interface has *defined* the underlying module. In systems where modules may export multiple interfaces, and a single interface may be exported by multiple modules, this is no longer true. As a result, the relationship between interfaces and modules in such systems has become confused. This section presents a reference model for the construction of components that is assumed throughout this chapter.

Essentially, interfaces provide a *view* of a component-class. Each component-class implements some core functionality and data, that it allows access to via a *language-level* interface. Exported interfaces apply some additional processing to this core-interface, to present the functionality and data in a certain format. This model is best described by a diagram, as shown in Figure 4.7.

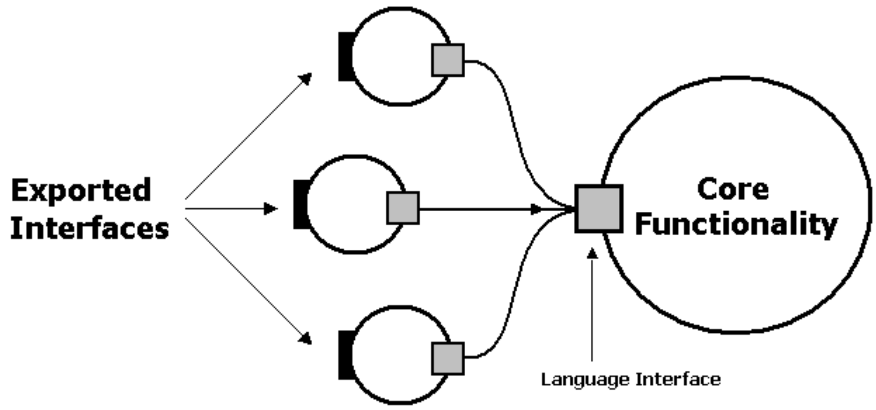


Figure 4.7: Component-Class Structure

Different views of a given component-class may exist for a variety of reasons. Views can provide functionality and data in different formats, e.g. file data as a raw byte stream or a sequence of strongly typed records, and they can provide different levels of access. Principally however, views present semantically disjoint sets of operations, providing a foundation for polymorphism (Section 4.3.3).

A crucial feature of this model is that it contains a concrete notion-of-self, i.e. there is a central module that defines the behaviour of the component-class. A clear notion-of-self is required if components are to be used as atomic units of composition. As interfaces are added horizontally, this model also allows interfaces to be added to a component-class without affecting existing interfaces, and hence existing clients.

Interface Objects

Clients always interact with a component-instance via one of the interfaces exported by its component-class. Interfaces are mapped into the client's programming language using the native 'object' construct. This abstraction, called an *interface object*, was introduced in the system model. As an example, the C++ interface object for the `IWeaponSystems` interface presented above would be:

```
class IWeaponSystems {
public:
    IWeaponSystems( cid_t cid );
    IWeaponSystems( cicap_t ref );
    int AcquireTarget( target_desc_t *spec );
    target_desc_t FireAtTarget();
    void FirePattern( pattern_desc_t p );
    wstatus_t Status();
};
```

Interface object constructors are described in detail in Chapter 5. Briefly, however, clients must either provide the interface object with an existing component-reference, or the identifier of the component-class of the new instance to be created by the interface object.

Each interface object contains a reference to a component-instance. When a method of the interface object is called, the reference's component-class is resolved, and the appropriate entry point in the component implementation is invoked. Dynamically resolving the implementation to invoke using the component-instance is called *late-binding*, and is an essential feature of component-oriented programming. Late-binding, shown in Figure 4.8, is the key to polymorphism and versioning, which are discussed later in this chapter.

4.3.2 Encapsulation

Encapsulation (introduced in Section 2.2) comes in two flavours, data and control. *Data encapsulation* ensures that only the component-class may access the context data of an

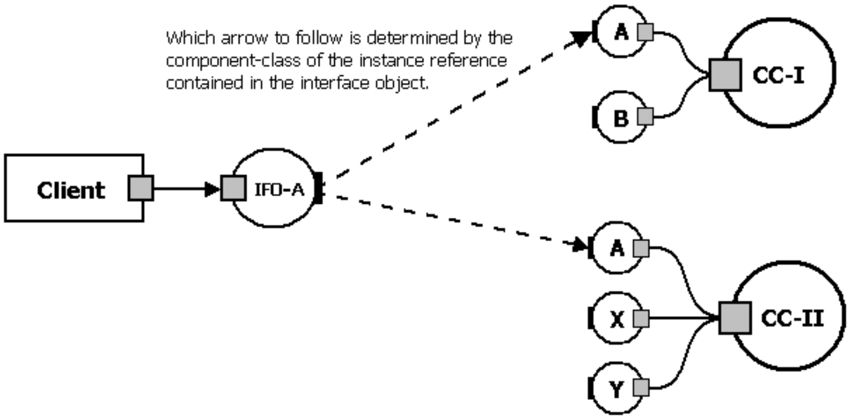


Figure 4.8: Interface Objects and Late Binding

instance. *Control encapsulation* ensures that the only way a client can operate on an instance is via the registered entry points. Encapsulation guarantees the separation of specification and implementation created by interfaces, providing three prime benefits:

- A component-class's implementation, and context data format, may be changed without (syntactically) affecting clients.
- Component-instances may contain privileged data. As the component implementation may execute within an amplified protection domain (Section 4.2) this is important. Without data encapsulation, the component implementation cannot trust the security or integrity of the instance data.
- Invariants can be guaranteed. This requires both data encapsulation, and control encapsulation, so that the control path cannot be interfered with.

As shown in the system model (Section 4.2), component-instances are accessible only from the component-class's protection domain, providing system enforced data encapsulation. Furthermore, the Component Adaptor will only invoke entry points that have been registered with the Component Locator, providing encapsulation of control. Consequently, MCS provides genuine encapsulation of both data and control. Encapsulation is required if components are to provide secure services.

Despite their claims, few existing component and object-oriented systems provide genuine encapsulation. For example, C++ encapsulation can be violated by any programmer using pointer arithmetic and Java encapsulation can be violated by hostile byte-code. COM does allow component-instances to be properly encapsulated using a separate address-space, however, this results in a significant performance penalty (see Chapter 6), so that most COM components are used in-process.

4.3.3 Polymorphism

Polymorphism is defined as the ability to assume multiple forms. In component software, this corresponds to the ability of a strongly typed client variable to refer to instances of different component-classes. Component-classes that may be referred to by the same client variable type are said to be *substitutable* for one another. A component model must define how substitutability is determined.

Object-oriented languages usually use subtypes, formed by interface inheritance (see Section 2.2.3), to determine substitutability. A subtype is substitutable for any of its supertypes. Despite its popularity, determining substitutability in this fashion is rejected. Subtyping imposes a hierarchical relationship between substitutable component-classes, which is too inflexible for component-oriented software. A component-class must be able to adopt unrelated views, and behave polymorphically with respect to these views.

MCS employs a polymorphic model that flows directly from the interface model described above. If two component-classes export the same interface, then a client can operate on instances of either using the same interface object. In the context of this interface, both component-classes appear the same to the client, i.e. interface objects are inherently polymorphic. There is no well-known term for this approach to polymorphism, which is referred to as *interface polymorphism* within this thesis. Polymorphism is closely related to late-linking, and the ability of a single interface object type to operate on different component-classes was alluded to in Figure 4.8.

Interface polymorphism provides the functionality traditionally implemented using *subtype polymorphism*. Other polymorphic models, e.g. *parametric polymorphism* [RS97], are often used in conjunction with subtype polymorphism to provide additional functionality. These models, however, have not shown themselves to be beneficial at the component level. A detailed explanation of many polymorphic models can be found in [AC96].

4.3.4 Code Reuse: From Inheritance to Composition

Code reuse is one of the prime motivations for component-oriented software. In fact, it was the observation that object-orientation had failed to deliver the promised levels of code reuse and shorter development times [Ude94], that led to the development of component-oriented software. Thus, how well a component model supports reuse is of utmost importance. This section begins with a discussion of inheritance (Section 2.2) as it is the reuse model of all current object-oriented programming languages.

Inheritance

Although object-oriented languages all support some form of inheritance, the exact model employed differs. These differences occur because languages use inheritance to fulfill a variety of goals. To reason clearly, these goals, and how inheritance achieves them, must be identified and separately addressed. Three 'cardinal facets' of inheritance are identified in [Szy98], a fourth has been added by the author:

- **Subclassing** is the inheritance of code, often called *implementation inheritance.*
- **Subtyping** was discussed in Section 4.3.3, and is known as *interface inheritance.*
- **Substitutability** was also discussed in Section 4.3.3. Most languages guarantee that subtypes are substitutable for supertypes, although Eiffel is an exception.
- **Customisation**. Inheritance allows the implementation of a method of a base-class to be replaced, e.g. virtual inheritance in C++. Such a mechanism is intended to allow a base-class to be to be customised or *specialised.*

Implementation Inheritance vs Customisation

Discussions of inheritance rarely distinguish between implementation inheritance and customisation. As these two facets serve entirely different purposes, this lack of distinction results in misleading observations and conclusions. This section clearly describes the semantics of each, and the purpose they serve.

Implementation inheritance allows a new component-class to reuse a method implementation from an existing component-class. Customisation allows a component-instance to replace the implementation of a method in another component-instance. The latter involves the modification, not reuse, of existing code. An example is the most effective way to clarify these ideas.

Figure 4.9 shows two component-classes. `base` provides two methods, `A` and `B`, while `derived` provides three methods, `A`, `B` and `C`. `derived` inherits its implementation of `A` from `base`, but implements `B` itself.

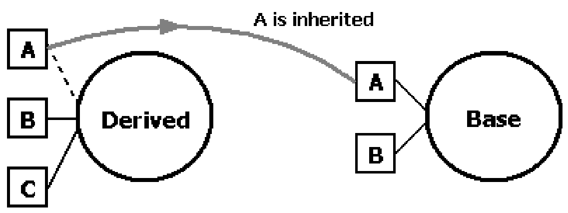


Figure 4.9: Implementation Inheritance

Figure 4.10(A) shows the flow of control when a client invokes `A` on an instance of `derived`. As the method implementation is inherited, control is passed to the component-class that actually implements the method, i.e. `base`. Obviously, this is code reuse; `derived` has reused `base`'s implementation of `A`. Now assume that `derived` has customised `base`'s implementation of `B`. Figure 4.10(B) shows the flow of control when a client invokes `B` on an instance of `base`. When shown this way, it becomes obvious that customisation is not code reuse. Implementation inheritance and customisation are, in fact, orthogonal. For example, a class may override a method of an existing class, with an implementation it inherited from another existing class.

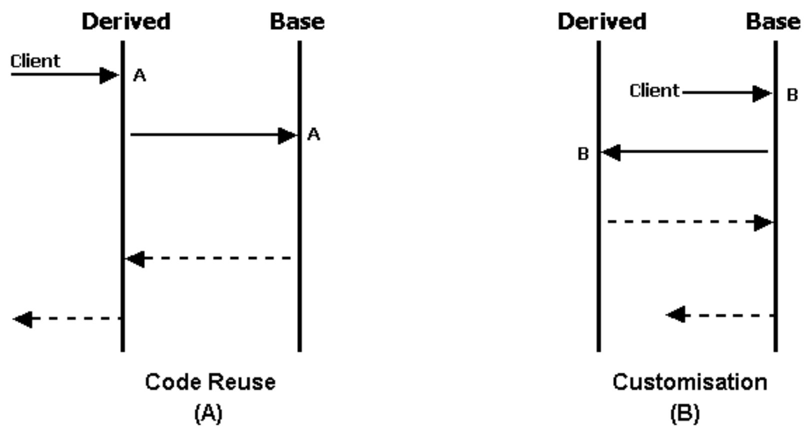


Figure 4.10: Control Flow of Reuse vs. Customisation

Implementation Inheritance

With the removal of customisation, implementation inheritance semantics becomes very simple, effectively equivalent to a cut-and-paste of code. Despite this simplicity, implementation inheritance still presents a number of problems:

- Inheritance is static. Component software is inherently dynamic, and instances of the same component-class are used in a variety of roles. This means that a component-class must be able to reuse different implementations depending on its context and state.
- Like interface inheritance, implementation inheritance imposes a hierarchical relationship. Such a relationship is inappropriate for component-based software, reducing its flexibility. A component-class should be able to reuse implementations from a variety of sources, specifying which implementations are provided by which sources. This style of reuse is known as *mixins* [BC90].
- Most importantly, it is unnecessary. Components are by definition reusable software modules. Introducing a new reuse model simply because a component-class is being used by another component-class, rather than a client, is nonsensical.

Related to the third point above, is the concept of a *specialisation interface* [KL92], i.e. the interface between a base-class and a sub-class. For example, in C++ the specialisation interface is the union of protected and public members, while the client interface is simply the public members. The existance of a specialisation interface that is different from the client interface implies that a sub-class is required to have greater knowledge of a base-class than a client. Code reuse is encouraged by minimizing the knowledge required for reuse, a specialisation interface clearly violates this rule.

Composition and Aggregation

Code reuse in MCS is based on *composition*, also referred to as *forwarding* and *containment*. Under composition, components reuse code exactly the same way clients do, by creating an instance and invoking methods on its interfaces. Composition relies on the natural reusability of the component model, eliminating the specialisation interface. As a component-class may use context and state data to decide which other component-class to call, composition is inherently dynamic. Furthermore, composition does not impose any structural constraints on the developer.

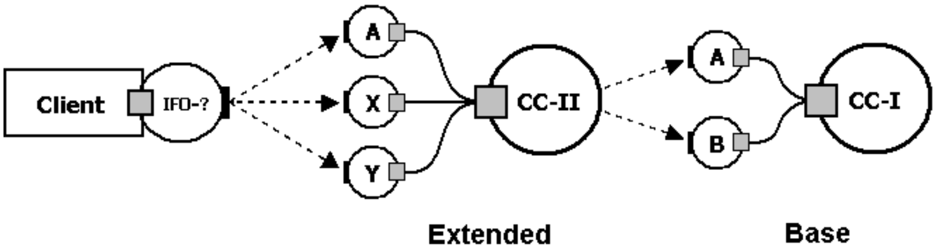


Figure 4.11: Reuse By Composition

Figure 4.11 shows a component-class, `Extended`, reusing another component-class `Base`. Clients invoke methods on instances of `Extended`, via the `A`, `X` and `Y` interfaces. The implementations of these method in the `Extended` component-class, invoke methods on the `A` and `B` interfaces of the `Base` component-class, hence reusing its code. Although this model is far simpler than implementation inheritance, i.e. no new concepts are introduced at all, it *is* code reuse, which was the original goal. Three distinct styles of code reuse can be identified in Figure 4.11.

- Traditional, library-style, reuse of the `B` interface. Any of the implementations in the `Extended` component-class can make use of the functionality already provided by the `B` interface. This is a natural form of reuse, employed by procedural languages.
- Extension of the `A` interface. `Extended` can *add* extra functionality to the implementation of `A` provided by `Base`. This is achieved by each method of the `A` interface in `Extended` calling the corresponding method in `base`, but performing additional processing before the call, after the call, or both.
- Directly exporting the functionality of the `Base` component-class. A common scenario is that the `Extended` component-class exports some functionality of the `Base` component-class without modification. This is a special case of the previous item, notable because it is common and can be optimised.

Aggregation is an optimisation of the case that a component-class C_{Ext} directly exports the functionality of another component-class C_{Base}. Applying composition directly, a method that is unchanged by C_{Ext} must still be relayed by C_{Ext} to C_{Base}. This incurs the overhead of an extra, and unnecessary, method invocation. Aggregation allows interfaces of a component-class to be directly exported by another component-class. When a client invokes a method of an aggregated interface of C_{Ext}, the request is sent directly to C_{Base} with an appropriate instance. Figure 4.12 illustrates the aggregation concept.

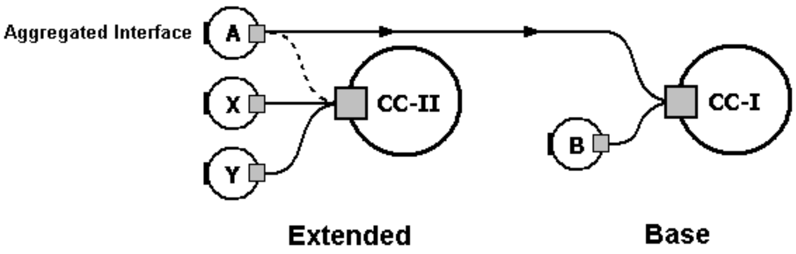


Figure 4.12: Component Aggregation

Aggregation must be transparent to both clients and components. The most important implication of this is that aggregation must be transitive, i.e. a component-class must be able to aggregate an interface from a component-class which itself aggregates the interface. Furthermore, this should not result in any additional overhead. Transitive aggregation is shown in Figure 4.13.

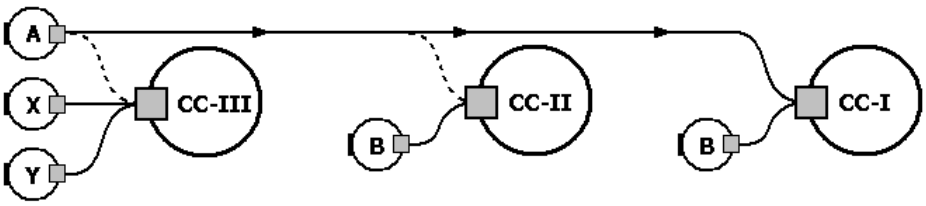


Figure 4.13: Transitive Aggregation

Composition and aggregation both operate at interface granularity. Section 4.3.1 stated that 'An interface should contain a relatively small number of semantically related operations'. Component-classes should therefore be able to independently reuse any interface of another component-class, without creating semantic problems. A more precise discussion of the semantic issues related to interfaces is contained in Section 4.3.5 below.

4.3.5 Customisation

Customisation allows a component-instance to replace the implementation of a method in another component-instance. It is the component analogue of virtual inheritance. Figure 4.10 illustrated the semantics of customisation.

Object-oriented programming languages primarily use customisation for polymorphism.

Languages typically implement polymorphism using subtypes formed by interface inheritance. A given object of type T_A may be used (substituted) whenever the expected type T_E is a supertype of T_A. When it is substituted, the object is cast to the expected type (T_E). This means that when a method is called, it is the implementation defined by T_E and not T_A that is invoked. Customisation overcomes this problem by allowing T_A to replace the implementation of the method in T_E, when T_E is being used as a base-class for T_A. Other instances of T_E retain the original implementation. Therefore, when an object of type T_A is cast to type T_E, which is whenever it is being used polymorphically, the correct implementation is still invoked.

Customisation is therefore used in object-oriented languages to work-around the deficiencies of their polymorphism model, in particular, the lack of separation between type and implementation inheritance. As MCS clearly separates polymorphism and code reuse, this problem does not occur. A component-class can export any interface, and provide a custom implementation of its methods. When a method is invoked on an interface object, the component-class of the contained reference is determined and the appropriate implementation invoked.

Although the traditional motivation for customisation is already satisfied by the polymorphism model, component-oriented software creates new motivations for customisation. Most of these motivations come from the desire to be extensible. For example, in a component-based file system, a user may wish to extend the directory component-instances they own to implement a customised file cache. Also, a system administrator may extend a printer component-instance to handle an error in the driver or implement fair scheduling. Allowing such modifications is a core feature of extensible systems that can improve the performance, correctness and simplicity of applications [GB97a]. Customisation is concerned with individual *instances* not component-classes.

Semantic Issues of Customisation

Customisation creates a number of semantic difficulties that must be understood and addressed by the component model. This section describes the nature of these difficulties. As an introduction, consider the C++ code fragment in Figure 4.14.

In this example, a class `derived` customises the method `some_method()` from class `base`. If a client invokes `some_other_method()` on an instance of class `derived`, the implementation in `base` will be invoked. This is implementation inheritance. When `some_other_method()` calls `some_method()` on itself, the implementation in `derived`

```
class base {
private:
    int flag;
public:
    void some_method()
    {
        // some code
        flag = value;
        // some more code
    }

    void some_other_method()
    {
        // some code
        some_method();
        // some more code
    }
};

class derived {
public:
    void some_method( int param )
    {
        // code
    }
};
```

Figure 4.14: C++ Customisation Example

will be invoked. This is customisation. Problems occur if `some_other_method()` depends on `some_method()` setting the `flag` state variable, which the overriding method does not (and in this example cannot) do.

Customisation causes problems when methods of a class invoke other methods on themselves, e.g. as `some_other_method()` does. When an object calls a method on itself, it is called *self-recursion*. Except in trivial cases, self-recursion almost always creates semantic dependencies between the *implementations* of methods within a class. Problems could be avoided by including a complete semantic description of each method in the interface specification, so that an overriding implementation would know exactly what it was required to do. Unfortunately, due to the infinite number of ways semantic dependencies can be introduced, the creation of such descriptions is, at present, impractical. Currently, a developer wishing to override a method in a base-class must manually inspect the source of the the existing implementation and the implementation of any other methods in the base-class that call the method. This is a tedious, time-consuming, and error-prone task at best. When the base-class is embedded in a class hierarchy in which customisation is being heavily used, finding all dependencies becomes impossible.

As subtle, undefined, semantic dependencies are an intrinsic feature of programming, simple customisation appears unachievable. Encapsulation has traditionally been used to overcome such problems, and it is here the problem lies. Customisation violates encapsulation. Figure 4.14 clearly shows encapsulation of control being violated, i.e. class `derived` modifies the control flow of an existing method. Others have made the observation that *"inheritance breaks encapsulation"* [Sny86], however, it is customisation and not inheritance per se, that violates encapsulation.

Customisation $\propto^{-1}$ Encapsulation

A clarifying example of this statement is provided by the *semantic fragile base class problem* (semantic FBC), which is concerned with how a base-class implementation can be updated without breaking sub-classes. Issues associated with re-compilation and entry-point location are part of the *syntactic FBC*. When a base-class implementation is modified, semantic relations between its methods are created and destroyed. To avoid problems, overriding methods in sub-classes must also be updated to include the new semantics. Therefore, modifying the *implementation* of a base-class results in all classes down the hierarchy having to be modified. Clearly, encapsulation is not respected.

Interface Delegation

Encapsulation is violated because customisation is performed on syntactic, rather than semantic, entities. An approach to customisation, based on this observation, was proposed by Lamping [Lam93]. In this scheme, each method in an interface specification must declare what other methods it has a dependency on, i.e. other methods within the class it calls. Using these annotations, a dependency graph is constructed at compile time, and methods are collected into *groups*. Customisation is performed on groups rather than individual methods, hence avoiding any semantic dependencies. A similar approach is taken by MCS, however, interfaces are used as the semantic entities, rather than groups.

Customisation in MCS is based on *interface delegation*, shown in Figure 4.15. A component-class declares certain interfaces as *delegatable*, which can then be overridden at runtime by another component-instance (C_D). Responsibility for handling methods invocations is therefore *delegated* to C_D. If a client invokes a method on a delegated interface, the call is redirected to the delegating component-instance C_D.

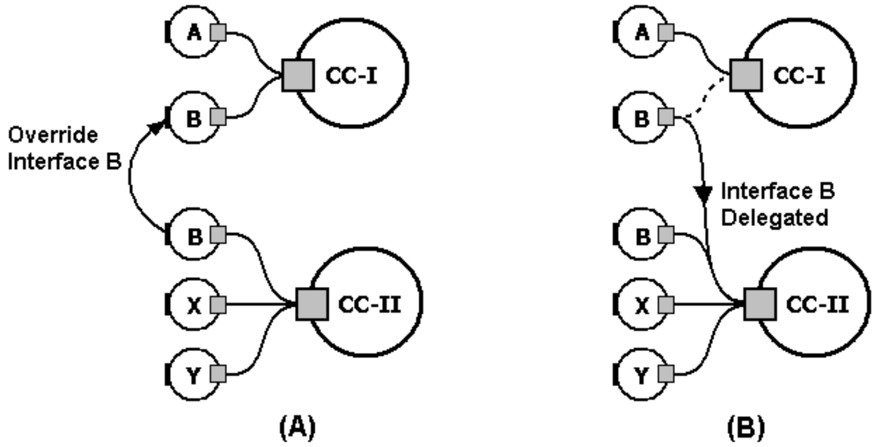


Figure 4.15: Interface Delegation

Section 4.3.1 described interfaces as ‘a set of semantically related operations’. As a component-instance must override an entire interface, encapsulation is maintained, avoiding the semantic problems described above.

4.3.6 Events

Component interfaces have so far been called by clients, transferring control to the component implementation. Such interfaces are *incoming interfaces*. Conversely, the methods of *outgoing interfaces* are called by the component implementation, and transfer control to the client. The methods of outgoing interfaces are called *events*, and are implemented by an *event sink*. Figure 4.16 illustrates the concept. A component exports both an incoming `I` and an outgoing `E` interface. Arrows show the direction of control flow, originating from the invoker and pointing towards the implementer.

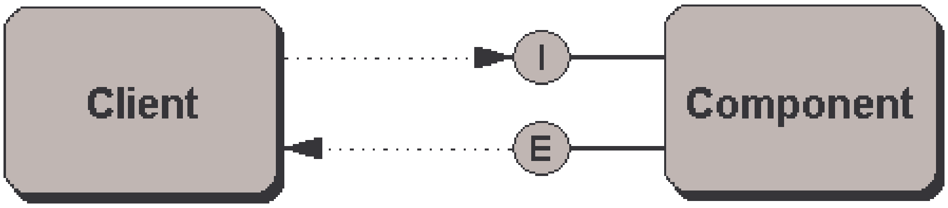


Figure 4.16: Incoming vs Outgoing Interfaces

Events are typically used to send status information to a client, avoiding the need for clients to poll components. This is especially useful when components are being used to interact with asynchronous devices such as humans, printers and disks.

MCS employs a *publish and subscribe* event model. Component-classes *publish* outgoing interfaces in their IDL specification, as shown in the following example:

```
component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    provides IStellarCartography;
    provides IWeaponSystems;

    publishes IHail;
};
```

Clients *subscribe* for events by calling an event registration method on the component-instance of interest. A reference to an event sink must be passed to the registration method. An event sink can be an instance of any component-class that implements as an incoming interface, the outgoing interface of interest. Figure 4.17 clarifies the model.

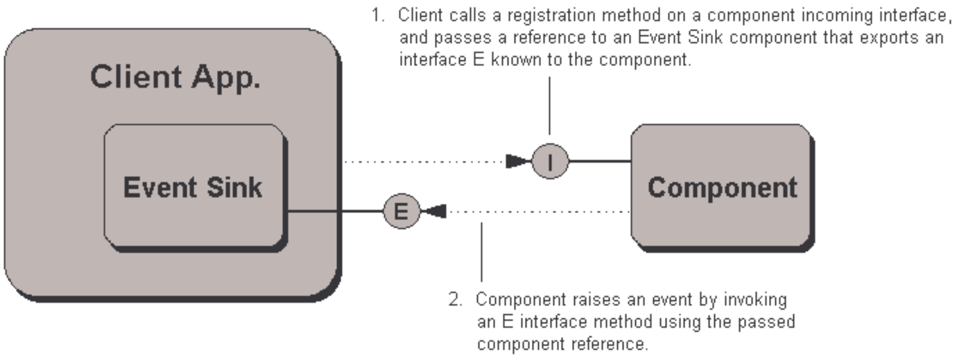


Figure 4.17: Events Using Sink Components

When a component-instance raises an event, it invokes the appropriate method on the incoming interface of every registered event sink. Each registration method has a corresponding unregister method that allows clients to remove an event sink.

4.3.7 Component Lifecycle

Component-instances are explicitly created, and may be explicitly destroyed, by clients. As a single component-instance may be used by a number of non-cooperating users, however, it is not always obvious when an instance should be destroyed, and who should destroy it. Object-oriented languages solve this problem by automatically destroying objects when there are no clients holding a reference to the object. This is not suitable for MCS. Component-instances do not exist in the client programmer's model, only interface objects do, and so instances never go out of scope. Furthermore, references to component-instances may be freely distributed without system or library intervention, therefore automatic reference counts cannot be maintained.

As MCS components are persistent, garbage collection is performed in a fashion similar to that of files in traditional operating systems, i.e. explicit reference counting. Clients must explicitly register, and revoke, their interest in a component-instance. When all clients have revoked their interest, the instance is destroyed.

Methods for incrementing and decrementing an instance's reference count are not defined by the MCS component model. This service can be built on top of the underlying component model, as is done in CORBA using the *Life Cycle Service* [OMG98].

4.3.8 Dynamic Type Discovery

Clients generally know statically which interface they should use to communicate with a passed component-reference. For example, a client will always use the (e.g.) `IByteStream` interface when it expects the passed reference to be a generic data file. If the component-class does not export the interface, then an exception will be thrown (Section 4.3.9).

There are situations, however, in which a client uses different interfaces depending on the component-class. For example, if an existing interface (I_O) becomes inadequate, a new version of the interface (I_N) with additional methods may be developed. Clients may prefer to use I_N to take advantage of the new functionality, however, to be backwardly compatibly they should also accept I_O. When a client is passed a reference, it must know which interface object to use. It could first try I_N, and then I_O if an exception is thrown, however, a cleaner solution is to provide clients with access to a component's type information. Publishing type information at runtime is called *reification*.

Every MCS component provides type information via an entry point of the class interface (Section 4.3.1). This type information includes the component-class identifier and the identifier of every interface exported. A set of flags convey additional information about each exported interfaces, such as whether the interface is aggregated, delegatable, outgoing, and if it is actually delegated for this instance. The syntax of this entry point, and the format of the data returned, is described in the following chapter.

Dynamic type discovery also allows a client to quickly determine whether a component-class exports a specific set of interfaces (known as a *category*). Both CORBA and COM publish type information at runtime, CORBA uses the interface repository, while COM uses the `IProvideClassInfo2` interface which each component must implement.

4.3.9 Exceptions

Runtime errors can be raised by MCS-RT, component stubs, and interface objects. For example, MCS-RT will return an error if the client does not hold a PDX capability to the invoked entry point, component stubs will return an error if the passed reference is invalid, and an interface object will return an error if it does not contain a component-reference. A mechanism is required to report these errors, known as *exceptions*, so that they can be distinguished from successful return values. In MCS, the CI (Section 4.5) returns an indication of whether an exception has occurred. Interface objects check this indication, and convert exceptions into the native language format. As MCS uses OMG IDL, component implementations are also allowed to explicitly raise exceptions. These exceptions must be declared in a component's IDL specification. MCS exceptions can therefore be divided into three categories:

System exceptions are thrown by the MCS-RT. For example, protection faults, or any unhandled operating system exception encountered while executing the method.

Stub exceptions are returned by the component stub. Stub exceptions are often handled by the interface object, e.g. when the called interface is aggregated, however, other stub exceptions, e.g. invalid reference, are returned to the client. Stub exceptions and system exceptions do not have to be declared in the component's IDL.

User exceptions are thrown explicitly by the component implementation code. How exceptions are passed from the component implementation to the component stub, depends on the language mapping. Each method lists all exceptions it may throw in its IDL specification. If an undeclared exception is thrown, the component stub converts it into an 'unknown user exception' stub exception.

Stub and user exceptions may return exception data to the client. Section 4.5 describes how exceptions are raised and passed to an interface object, while Section 5.4.2 describes how interface objects pass exceptions to clients.

4.3.10 Static Interfaces

Method invocations execute within a (data) context defined by the component-instance on which the method is invoked. Contexts are completely disjoint. These are the standard object-oriented method semantics. Occasionally, however, it is convenient for certain methods to always execute in a single, global, context. These methods are called *static methods*. In MCS, static methods are grouped into *static interfaces*.

As they only ever execute in a single context, static methods do not require a reference for method invocation. Therefore, static methods are procedural rather than object-oriented.

Static interfaces are motivated by the desire to used component-based programming features, e.g. polymorphism and late-binding, in a procedural model. They are useful in an object-oriented model for allowing configuration and management of the component-class as a whole. For example, constructors are effectively static methods.

4.4 Interface Definition Language

An *interface definition language (IDL)* is used to specify interfaces and component-classes. This specification serves two purposes:

- It provides an unambiguous, implementation language independent, specification of a component-class, allowing clients and components to be developed in isolation.
- As the specification is unambiguous, it can be used to automatically generate the code for interface objects and component stubs.

MCS uses a subset of the OMG CORBA 2.3.1 IDL [OMG99c] and the extensions specified in the March 1999 CORBA Components submission [OMG99d]. As the CORBA Components submission is still volatile, it is possible that examples in this thesis may become invalid.

Elements of the CORBA 2.3.1 IDL *not* supported are:

- Valuetypes.
- Subtyping of interfaces.

Elements of the CORBA Components IDL which *are* supported:

- Grammar rules 2 $\rightarrow$ 24 (incl.) from pages 5-35 and 5-36 of [OMG99d]. These rules pertain to the declaration of component-classes, and the interfaces they export.

4.5 Component Invoker and Component Adaptor

Section 4.2.2 introduced the Component Invoker (CI) and the Component Adaptor (CA) as a standardised request mechanism interface, comprising two functions:

- Activating the component implementation.
- Transferring control, and limited request data, to the entry point.

Figure 4.4, reproduced below as Figure 4.18, shows the position of the CI and CA within the system model. This section describes the client interface to the CI, and the interface between the component stubs and the CA, using C syntax.

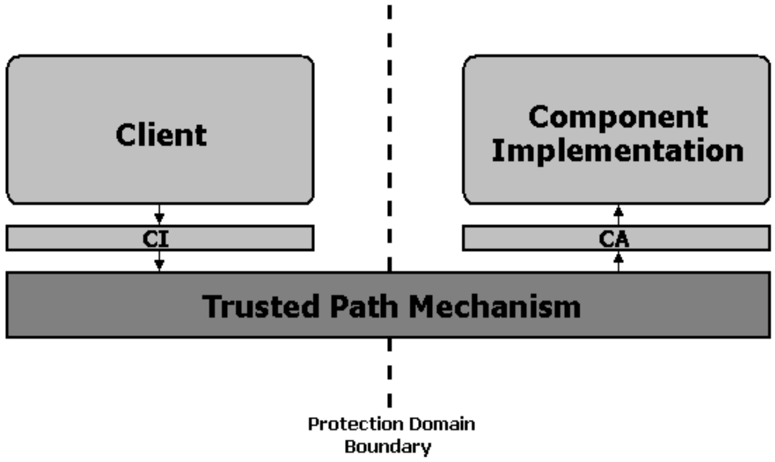


Figure 4.18: Component Invoker / Component Adaptor

4.5.1 Type Declarations

This section describes the additional types introduced by the CI interface.

```
typedef struct {
    unsigned long data1;
    unsigned data0 : 62;
    unsigned type  : 2;
} res_desc_t;
```

A `res_desc_t` structure is returned by each method invocation request to the CI. The `data1` and `data0` fields either contain the method's return value, or exception data, depending on the value of `type`. The `type` field value has the following meanings:

0 Successful method invocation; `data1` and `data0` contain the method's return value.

1 User exception thrown; the upper 16-bits of `data1` contains an exception identifier (EID), with the remaining bits, and `data0`, containing the exception structure defined in the IDL. This imposes a 110-bit limit on the size of exception structures.

2 Stub exception thrown; `data1` and `data0` are as for user exceptions.

3 System exception thrown; `data1` and `data0` are as for user exceptions.

4.5.2 Component Invoker Interface

ci_request

```
int ci_request( entry_pt_t ept, cap_t param, res_desc_t *ret, void *pd );
```

Parameters:

`ept`	[in]	A reference for the entry point to invoke, normally retrieved using the Component Locator.
`param`	[in]	A parameter passed by-value to the component stub.
`ret`	[out]	A pointer to a buffer that contains the return value of the component stub on return. The format of this descriptor was described in Section 4.5.1 above.
`pd`	[in]	A protection domain (PD) object that is merged with the protection context of the component-class for the duration of the method. Two special values, described below, may be passed instead of an actual PD object.

These values, defined in `mcs_ci.h`, may be passed instead of an actual PD object:

PD_EMPTY	None of the client's protection domain is added to the protection context.
PD_MERGE	All of the client's protection domain is added to the protection context.

Description:
This function invokes an entry point, `ept`, in the component implementation with the parameter `param`. The entry point executes in a protection domain that is the union of the protection context and `pd`. On return, `ret` contains a valid result descriptor.

A client, or more likely an interface object, uses this function to invoke entry points in the component implementation. The entry point is passed

Return Value:
On success the function returns zero. On failure one of the following error values (defined in `mcs_ci.h`) is returned:

MCSE_EXCEPTION	An exception has been thrown, details in `ret`.
MCSE_FAIL	Unspecified error.

ci_reset

```
int ci_reset();
```

Description:
A client uses this function to re-initialise the Component Invoker. What re-initialisation involves depends on the implementation.

Return Value:
On success the function returns zero, otherwise it returns `MCSE_FAIL`.

ci_get_opt and ci_set_opt

```
int ci_get_opt( int option_id, void *optval, int *optlen );
int ci_set_opt( int option_id, void *optval, int optlen );
```

Parameters:

`option_id`	[in]	Identifies the option being manipulated.
`optval`	[in]/[out]	Points to a buffer for the option value. For `set`, this is an [in] parameter containing the new option value. For `get`, it is an [out] parameter containing the existing option value when the function returns successfully.
`optlen`	[in]/[inout]	The size, in bytes, of the value. For `set`, this is an [in] parameter containing the size of the passed `optval` buffer. For `get`, this is an [inout] parameter, initially containing the size (in bytes) of the passed `optval` buffer, and containing the size (in bytes) of the value stored in `optval` when the function returns successfully.

Description:
`get_ci_opt` and `set_ci_opt` manipulate the options associated with the Component Invoker. Options may effect the behaviour of all instances of the CI running in the system, or only of the instance called.

Currently no CI options defined.

Return Value:
On success the function returns zero and the result parameters are set appropriately. On failure one of the following error values is returned:

MCSE_SMALL_BUFFER	The supplied buffer is too small for the value (`ci_get_opt` only).
MCSE_FAIL	Unspecified error.
MCSE_UNKNOWN_OPT	The CI does not recognise the `option_id`.

4.5.3 Component Adaptor Interface

When a method is invoked using `ci_request`, the appropriate component implementation is loaded and a request sent to the CA via the trusted path. An instance of the CA executes in the protection domain of every active component implementation. It is the responsibility of the CA to invoke the entry point identified with the passed parameter.

When the entry point returns, the CA sends the return value back to the CI via the trusted path. Therefore, the only interface syntax that needs to be defined is that of the invoked entry points. Each MCS entry point must have the type:

```
typedef res_desc_t (ca_stub_t*)( capt_t param );
```

4.6 Component Locator

An interface object must pass an entry point reference to `ci_request`. This information is retrieved using the Component Locator (CL).

4.6.1 Type Declarations

This section describes the additional types introduced by the CL interface.

```
typedef unsigned long  cid_t;
typedef unsigned long  iid_t;
typedef unsigned short mid_t;
```

4.6.2 Component Locator Interface

cl_get_entry_pt

```
int cl_get_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t *epr );
```

Parameters:

`cid`	[in]	Component-class identifier.
`iid`	[in]	Interface identifier.
`mid`	[in]	Method identifier.
`epr`	[out]	A pointer to a buffer in which the entry point reference will be returned on success.

Description:
`cl_get_entry_pt()` returns the entry point reference for a method. Methods are uniquely identified by their `(cid,iid,mid)` tuple. An interface object statically knows the `IID` and `MID` of each of its methods, and given a component-reference it can obtain `cid` using the `cl_get_component_class()` function described below. The definition of `entry_pt_t` depends on the MCS-RT implementation, and is located in `mcs_cl.h`.

Return Value:
On success the return value is zero and the structure pointed to by `epr` contains the method's entry point reference. On failure the return value is non-zero and contains one of the following error codes:

MCSE_ARG_INVALID	`epr` is NULL.
MCSE_FAIL	Unspecified error.
MCSE_IID_NOT_EXPT	Component-class `CID` does not export interface `IID`.
MCSE_MID_NOT_EXPT	Interface `IID` does not contain the method `MID`.
MCSE_PROT_FAIL	The client does not have permission to access the location information for this component-class (if it exists).

cl_set_entry_pt

```
int cl_set_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t epr );
```

Parameters:

`cid`	[in]	Component-class identifier.
`iid`	[in]	Interface identifier.
`mid`	[in]	Method identifier.
`epr`	[in]	New entry point reference.

Description:
Sets the registered entry point for a method. Future calls to `cl_get_entry_pt()` using

the same `(cid,iid,mid)` tuple will return `epr`.

Return Value:
On success the return value is zero. On failure the return value is non-zero and contains one of the following error codes:

MCSE_ARG_INVALID	`epr` is invalid.
MCSE_FAIL	Unspecified error.
MCSE_PROT_FAIL	The client does not have permission to set the location information for this component-class.

cl_get_component_class

```
cid_t cl_get_component_class( cicap_t ref );
```

Parameters:

`ref`	[in]	A reference for a component-instance.

Description:
`cl_get_component_class()` returns the identifier of the reference's component-class. The `cicap_t` type is described in the following chapter.

Return Value:
If the return value is non-zero, the reference is for a valid component-instance and the return value is the identifier of the instance's component-class. If the return value is zero then the reference is not to a valid component-instance.

cl_reset

```
int cl\_reset();
```

Description:
A client uses this function to re-initialise the Component Locator. What re-initialisation involves depends on the implementation.

Return Value:
On success the function returns zero, otherwise it returns `MCSE_FAIL`.

cl_get_opt and cl_set_opt

```
int cl_get_opt( int option_id, void *optval, int *optlen );
int cl_set_opt( int option_id, void *optval, int optlen );
```

Parameters:

`option_id`	[in]	Identifies the option being manipulated.
`optval`	[in]/[out]	Points to a buffer for the option value. For `set`, this is an [in] parameter containing the new option value. For `get`, it is an [out] parameter containing the existing option value when the function returns successfully.
`optlen`	[in]/[inout]	The size, in bytes, of the value. For `set`, this is an [in] parameter containing the size of the passed `optval` buffer. For `get`, this is an [inout] parameter, initially containing the size (in bytes) of the passed `optval` buffer, and containing the size (in bytes) of the value stored in `optval` when the function returns successfully.

Description:
`cl_get_opt` and `cl_set_opt` manipulate the options associated with the Component Locator. Options may specify the behaviour of all instances of the CL running in the system, or only of the instance called.

Currently no CL options defined.

Return Value:
On success the function returns zero and the result parameters are set appropriately. On failure one of the following error values is returned:

MCSE_SMALL_BUFFER	The supplied buffer is too small for the value (`cl_get_opt` only).
MCSE_FAIL	Unspecified error.
MCSE_UNKNOWN_OPT	The CL does not recognise the `option_id`.

Chapter 5

Component System Implementation

Amdahl's Law: Make the common fast.

5.1 Overview

Chapter 4 presented a conceptual design of MCS and justified the choices made. This chapter describes an efficient implementation of the component architecture on Mungi.

System Model Implementation describes how the component environment is implemented using the Mungi abstractions presented in Section 2.1.

Component-Classes describes a skeleton component structure, an implementation of the class interface, and standard component stubs.

Interface Objects describes the implementation of C-language interface objects. Although the syntax is different for each client language, the semantics are the same.

Component Model Implementation describes how the features of the component model presented in Section 4.3 are implemented.

Component Invoker, Component Adaptor and Component Locator describe an implementation of the MCS-RT interfaces.

5.2 System Model Implementation

A system model identifies the major elements of the system, the roles they play, and how they interact. Section 4.2 introduced the MCS system model, and identified its critical features. Figure 5.1 reproduces the basic system architecture.

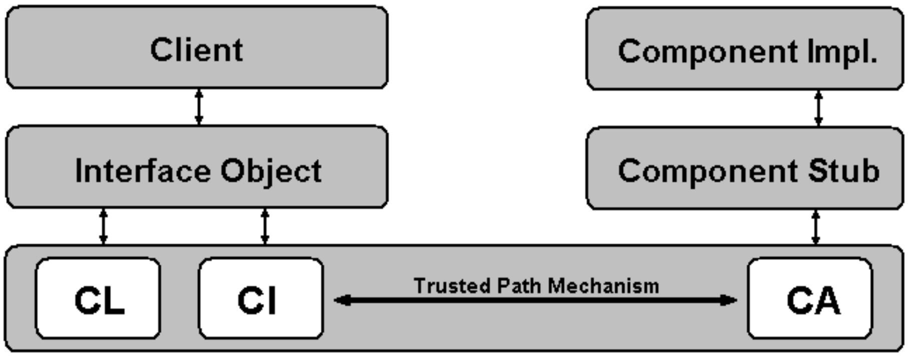


Figure 5.1: System Structure

A component implementation, and its corresponding component stubs, are placed in a Mungi PDX object. As described in Section 2.1, clients can only invoke the code in a PDX object using the `PdxCall()` system call.

```
int PdxCall( void *entry_pt, cap_t param, cap_t *ret, void *pd );
```

Legal entry points into a PDX object are registered by the object's owner using the `ObjCrePdx()` system call. Mungi stores these entry points in the Object Table (OT), and generates an associated *PDX capability.* A client may only invoke an entry point using `PdxCall()` if the entry point is present in the OT, and they possess the corresponding PDX capability. When an owner of a PDX object registers entry points using `ObjCrePdx()`, it also specifies a protection domain called the *protection context.* A `PdxCall()` executes in a protection domain that is the union of the protection context, and a protection domain specified by the client in the `pd` parameter.

Component stubs are registered as the entry points into a component's PDX object, with a protection context containing the component's resources. Therefore, the trusted path mechanism provided by `PdxCall()` forces clients to use a component's defined interface, and allows its methods to execute in an amplified protection domain.

Component-instances require a location to store their data. As *all* data in Mungi is stored in Mungi objects, when a new instance is constructed, it is placed in a Mungi object *within the protection context*. A constructor then returns the virtual address of the instance data to the client, which is a unique identifier in a SASOS. As component-implementations never distribute capabilities to their instance data, instances are data encapsulated.

So far, to invoke a method, a client requires a PDX capability to the method's component stub, a reference to the instance, but no *capability* to the instance. Therefore, there is no protection against a client with access to a component stub, invoking operations on arbitrary references in the hope of revealing some private data, or performing a privileged action. The problem is that the system capabilities are providing protection at component-class granularity, whereas protection is required at instance granularity.

As Mungi is a capability system, the natural solution is to create a capability that confers upon the holder the right to invoke methods on a particular instance. Unfortunately, this solution is inadequate. Mungi capabilities derive their protection from the memory management unit (MMU), which deals with memory at page granularity, and therefore, provides protection at page granularity. To protect instances with Mungi capabilities, each component-instance would have to be placed on its own page. As the majority of components are quite small, this leads to extremely inefficient use of memory and dismal translation lookaside buffer (TLB) performance.

A reference to the component-instance is explicitly presented to a component stub on *every* method invocation, i.e. component stubs have *complete mediation* [SS75]. Therefore, this implementation uses component stubs to perform discretionary access control. Constructors create a random 64-bit password that is returned to the client along with the reference, as well as being stored in the component-instance itself. This `(ref, password)` tuple is called a *component-instance capability (CICAP)*, and must be presented by the client on each method invocation. A passed CICAP's password is validated by a simple integer comparison with the password stored in the component-instance. As CICAPs provide protection at instance granularity, component-instances for different clients may be safely stored in the same Mungi object. Figure 5.2 shows the structure of a CICAP.

CICAPs allow discretionary (see Section 7.2.6) access control at component-instance granularity. They do not address the wider problem of access control in extensible systems, which is the focus of Chapters 7, 8 and 9.

```
typedef struct {
    unsigned long password;
    void *ref;
} cicap_t;
```

$\textbf{Password}_{(64)}$	$\textbf{Instance Address}_{(64)}$

Figure 5.2: CICAP Structure

5.3 Component-Classes

This section describes a standard component-class implementation, including the structure, class interface, and component stubs. Although the details depend on the language and the component-class, the concepts remain the same.

5.3.1 Component Structure

This section presents a skeleton component implementation that provides a foundation for the rest of this chapter. As all interfaces have similar structure and component stubs, only code for the `IWeaponSystems` interface is presented. Figure 5.3 contains an IDL specification for the `CStarfleetGalaxyClass` component-class, and Figures 5.4, 5.5, 5.6 and 5.7 present header and implementation files corresponding to this specification.

- `CStarfleetGalaxyClass.h`, shown in Figure 5.4, contains type definitions and constants from specification. This file is used by both component implementations and interface objects related to the specification.

- `CStarfleetGalaxyClass_cis.h`, shown in Figure 5.5, is the header file for the component implementation. It contains a location for the developer to define the component-core module (`csgc_t` and functions), as well as prototypes for the interface functions and their associated component stubs.

- `CStarfleetGalaxyClass_cs.c`, shown in Figure 5.6, implements the class interface and component stubs for the `CStarfleetGalaxyClass` component-class. As the class interface and component stubs are described in the following sections, their implementations have been left deliberately blank.

- `CStarfleetGalaxyClass_cis.c`, shown in Figure 5.7, contains a skeleton for the implementation of the component-core and interface modules.

```
/****************************************************************************
 * CStarfleetGalaxyClass.idl
 *
 * Specification for the CStarfleetGalaxyClass component.
 ****************************************************************************/
#include "other_interfaces.idl"

typedef enum{ WS_OFFLINE, WS_STANDBY, WS_READY } wstatus_t;

interface IWeaponSystems {
        int AcquireTarget( target_desc_t *spec );
        target_desc_t FireAtTarget();
        void FirePattern( long power, char class, int n_rounds, pattern_t p );
        wstatus_t Status();
        void InitialiseWeapon();
        void RetireWeapon();
};

component CStarfleetGalaxyClass {
        provides IEngineeringConsole;
        provides IEngines;
        provides IHolodeck;
        provides INavigation;
        provides ISensors;
        provides IStellarCartography;
        provides IWeaponSystems;
};
```

Figure 5.3: CStarfleetGalaxyClass.idl

```
/******************************************************************************
 * CStarfleetGalaxyClass.h
 *
 * Common client and component implementation definitions from CStarfleetGalaxyClass.idl.
 ******************************************************************************/
#ifndef __CSTARFLEETGALAXYCLASS_H__
#define __CSTARFLEETGALAXYCLASS_H__

/* component-class identifier */
#define CID_CSTARFLEETGALAXYCLASS 264

/* type definitions */
typedef enum{ WS_OFFLINE, WS_STANDBY, WS_READY } wstatus_t;

#endif /* __CSTARFLEETGALAXYCLASS_H__ */
```

Figure 5.4: CStarfleetGalaxyClass.h

```
/******************************************************************************
 * CStarfleetGalaxyClass_cis.h
 *
 * Component implementation side interface and component definitions
 * from CStarfleetGalaxyClass.idl.
 ******************************************************************************/
#ifndef __CSTARFLEETGALAXYCLASS_CIS_H__
#define __CSTARFLEETGALAXYCLASS_CIS_H__

#include "other_interfaces_cis.h"
#include "CStarfleetGalaxyClass.h"

/* component data */
typedef struct {
    /* FILL-IN: Component developer defines the data members */
} csgc_t;

/* component methods */
    /* FILL-IN: Component developer defines the component-core operations */

/* IWeaponSystems interface methods */
int csgc_iws_AcquireTarget( csgc_t *cthis, target_desc_t *spec );
target_desc_t csgc_iws_FireAtTarget( csgc_t *cthis );
void csgc_iws_FirePattern( csgc_t *cthis, long power, char class, int n_rounds, pattern_t p );
wstatus_t csgc_iws_Status( csgc_t *cthis );
void InitialiseWeapon( csgc_t *cthis );
void RetireWeapon( csgc_t *cthis );

/* Interface methods for all other exported interfaces (not incl. class-interface) would follow */

/* IWeaponSystems component stubs */
res_desc_t csgc_iws_stub_AcquireTarget( cap_t param );
res_desc_t csgc_iws_stub_FireAtTarget( cap_t param );
res_desc_t csgc_iws_stub_FirePattern( cap_t param );
res_desc_t csgc_iws_stub_Status( cap_t param );
res_desc_t csgc_iws_stub_InitialiseWeapon( cap_t param );
res_desc_t csgc_iws_stub_RetireWeapon( cap_t param );

/* Component stubs for other interfaces (incl. the class-interface) would follow */

#endif /* __CSTARFLEETGALAXYCLASS_CIS_H__ */
```

Figure 5.5: CStarfleetGalaxyClass_cis.h

```
/***************************************************************************
 * CStarfleetGalaxyClass_cs.c
 *
 * Component stubs for CStarfleetGalaxyClass
 ***************************************************************************/
#include "CStarfleetGalaxyClass_cis.h"

/* IWeaponSystems component stubs */
res_desc_t csgc_iws_stub_AcquireTarget( cap_t param )
{
}

res_desc_t csgc_iws_stub_FireAtTarget( cap_t param )
{
}

res_desc_t csgc_iws_stub_FirePattern( cap_t param )
{
}

res_desc_t csgc_iws_stub_Status( cap_t param )
{
}

res_desc_t csgc_iws_stub_InitialiseWeapon( cap_t param )
{
}

res_desc_t csgc_iws_stub_RetireWeapon( cap_t param )
{
}

/* Component stubs for other interfaces would follow */

/* Implementation of the class-interface would follow */
```

Figure 5.6: CStarfleetGalaxyClass_cs.c

```
/*************************************************************************
 * CStarfleetGalaxyClass_ci.c
 *
 * Component Implementation for CStarfleetGalaxyClass
 *************************************************************************/
#include "CStarfleetGalaxyClass_cis.h"

/*************************************************************************
 *
 * Component Implementation
 *
 *************************************************************************/
/* FILL-IN: Component developer defines and implements component operations */

/*************************************************************************
 *
 * Interface Implementations
 *
 *************************************************************************/

/* IWeaponSystems Implementation */
/* FILL-IN: Component developer must implement interface methods */
int csgc_iws_AcquireTarget( csgc_t *cthis, target_desc_t *spec )
{
        int retval;
        return retval;
}

target_desc_t csgc_iws_FireAtTarget( csgc_t *cthis )
{
        target_desc_t retval;
        return retval;
}

void csgc_iws_FirePattern( csgc_t *cthis, long power, char class, int n_rounds, pattern_t p )
{
}

wstatus_t csgc_iws_Status( csgc_t *cthis )
{
        wstatus_t retval;
        return retval;
}

void csgc_iws_InitialiseWeapon( csgc_t *cthis )
{
}

void csgc_iws_RetireWeapon( csgc_t *cthis )
{
}
/* Implementations of other interfaces (not incl. class interface) would follow */
```

Figure 5.7: CStarfleetGalaxyClass_ci.c

```
[static] interface IClassInterface {
    cicap_t constructor();
    void destructor( cicap_t ref );
    cicap_t create_cicap( cicap_t owner, short interfaces, char slot );
    void get_typeinfo( cap_t info, int size ) throws( MCSE_STE_INSFBUFFER );
};
```

Figure 5.8: Class Interface IDL

This implementation structure is designed to reflect the component-class construction model illustrated in Figure 4.7. A 'component-core' module is defined by the developer. This module presents a language interface that is called by the interface modules, that are invoked by the component stubs. Interface methods are passed a reference to a data structure (`csgc_t`), defined by the component developer, that contains the instance data.

5.3.2 Class Interface

Every component-class must export and implement the class interface, which contains methods required for the correct operation of the component model. Figure 5.8 shows class interface's IDL specification. This section describes a standard implementation of the first three methods, with `get_typeinfo()` described in Section 5.5.6. The class interface is assigned interface identifier (IID) zero, and its methods assigned method identifiers (MIDs) in top-down order from zero, e.g. `destructor()` is identified by `(CID,0,1)`.

Constructor

Clients invoke `constructor()` to create a new instance of a component-class. Essentially, this involves creating a new language object to represent the instance, and returning a reference to this object to the client. Two additional operations are also performed:

- Every component-instance is associated with some metadata, that is stored in a header above the language object, and initialised by the constructor. The header format, and a description of its fields, its presented in Figure 5.9.
- Component implementations may wish to be notified when a new instance is created. A component developer can specify a method to be invoked upon construction, by prefixing the **CONS** modifier to its IDL specification, as shown in Figure 5.10. **CONS** methods, which must have no parameters, are called by the constructor.

<table>
<tr><td>HV1$_3$</td><td colspan="2">Domain Label$_{61}$</td><td>HV2$_3$</td><td colspan="4">Type Label$_{61}$</td></tr>
<tr><td colspan="3">Owner CICAP$_{64}$</td><td colspan="5">E1-CICAP$_{64}$</td></tr>
<tr><td colspan="3">E2-CICAP$_{64}$</td><td colspan="5">E3-CICAP$_{64}$</td></tr>
<tr><td colspan="2">VO$_{32}$</td><td>VL$_{32}$</td><td colspan="2">EF1$_{16}$</td><td>EF2$_{16}$</td><td>EF3$_{16}$</td><td>Flags$_{16}$</td></tr>
<tr><td colspan="8">Di-CICAP$_{128}$ . . .</td></tr>
</table>

- **HV1 & HV2** concatenated, form the Header Format Version (6 bits) which identifies the layout of the following header. The above layout if format version one.
- **Domain Label** is used for mandatory access control, described in Chapter 8.
- **Type Label** is also used for mandatory access control.
- **Owner CICAP**, generated by the constructor, is used to validate passed CICAPs. An owner CICAP allows access to all interfaces of the instance.
- **E*i*-CICAP** are the extended CICAPs, created using the `create_cicap()` method.
- **VO** is the minor version number of the component-class that created the instance.
- **VL** identifies the format of the instance data. Minor versions may change the data format of a component-class, therefore, this field is required to allow an implementation to identify, and handle, instances created by a different minor versions.
- **EF*i*** is a flag field associated with Ei-CICAP, described later in this section.
- **Flags** provides status information about the instance. Currently only the LSB is defined, indicating that an instance is valid when set.
- **D*i*-CICAP** are the delegation CICAPs, described in Section 5.5.5. A Di-CICAP field exists for each interface marked as delegatable in the IDL specification.

Figure 5.9: MCS Component-Instance Header Format

```
interface IWeaponSystems {
    int AcquireTarget( target_desc_t *spec );
    target_desc_t FireAtTarget();
    void FirePattern( pattern_desc_t p );
    wstatus_t Status();
    [CONS] void InitialiseWeapon();
    void RetireWeapon();
};
```

Figure 5.10: CONS Modifier Example

```
typedef struct {
    unsigned hv1    :  3;
    unsigned domain : 61;
    unsigned hv2    :  3;
    unsigned type   : 61;
    unsigned long ocicap, ecicap[3];
    unsigned int vo, vl;
    unsigned short ef[3], flags;
    unsigned long dcicap[1];
} inst_hdr_t;
```

Figure 5.11: Instance Header Structure

As described in Section 5.2, component-instances are stored in Mungi objects. Constructors, therefore, require a method of locating a Mungi object, and an address within the object, where the instance can be stored. Locating the Mungi object involves a protocol between the constructor, and the creator of the PDX object. Every protection context (Section 5.2) must provide a capability to a Mungi object in which component-instances can be stored as its first element. A protection context is implemented as a capability-list (c-list) in the active protection domain (APD) of the component-class. As there is no guarantee regarding the order of c-lists within the APD, protection contexts must have the `format` field in their c-list header set to `CL_PROTCTX` by the creator. Therefore, using `ApdGet()`, and inspecting the header of each c-list in their APD, a constructor can locate the Mungi object in which component-instances can be stored. Constructors store the address of this object in a static variable to amortize the lookup cost.

As component-instances are of fixed size, locating an address within the object at which a new instance can be stored is simple. Starting at the second word, the object is divided into blocks capable of holding a header and an instance. Free blocks are chained, with the first word of a free block containing the address of the first word of the next free block. A pointer to the first free block is contained in the first word of the object.

Figure 5.12 contains the implementation of `constructor()` for the example component-class (`CStarfleetGalaxyClass`), with `InitialiseWeapon()` marked **CONS**, and one delegatable interface (which one is irrelevant to the constructor). Constructors for different component-classes differ in the size of the instance data, the number of Di-CICAP slots initialised, the **CONS** methods called, and the version number stored in the header. Constructors are invoked directly by `ci_request()`, and so have the MCS required signature. Although CICAPs are defined as 128 bits, instances are word-aligned making the lower two bits of the address redundant. This means that CICAPs only carry 126 bits of information, and so do not require a parameter buffer.

```
static void *instobj;

res_desc_t csgc_ci_stub_constructor( cap_t param )
{
	apddesc_t apd;
	clist_t *clist;
	inst_hdr_t *hdr;
	int i;
	res_desc_t res;
	csgc_t *ref;

	/* Find an address to store the new instance */
	if( instobj == 0 ) {
		/* find the instance object */
		ApdGet( &apd );
		for( i = 0; i < apd.n_apd; i++ ) {
			clist = (clist_t*)apd.clist[i].address;
			if( clist != 0 && clist->format == CL_PROTCTX ) {
				instobj = clist->caps[0].address;
				break;
			}
		}
		/* check that the object has been found */
		if( instobj == 0 ) {
			res.type = MCS_STUB_EXCEPTION;
			res.data1 = MCS_STE_NO_INSTOBJ;
			return res;
		}
	}
	hdr = *((inst_hdr_t**)instobj);
	*((inst_hdr_t**)instobj) = *((inst_hdr_t**)hdr);

	/* Initialise the static header and the d1-cicap */
	memset( hdr, 0, sizeof(inst_hdr_t) );
	hdr->opasswd = generate_password();
	hdr->hv2 = 1;
	hdr->vo = hdr->vl = CSGC_MINOR_VERSION;
	hdr->flags = MCS_IF_VALID;

	/* Create a new language instance */
	ref = (csgc_t*)&(hdr->di[1]);
	memset( ref, 0, sizeof(csgc_t) );

	/* Invoke the CONS methods */
	csgc_iws_InitialiseWeapon( ref );

	/* Return */
	res.type = 0;
	res.data1 = hdr->ocicap;
	res.data0 = ((unsigned long)ref)>>2;
	return res;
}
```

Figure 5.12: Constructor

```
interface IWeaponSystems {
    int AcquireTarget( target_desc_t *spec );
    target_desc_t FireAtTarget();
    void FirePattern( pattern_desc_t p );
    wstatus_t Status();
    [CONS] void InitialiseWeapon();
    [DSTR] void RetireWeapon();
};
```

Figure 5.13: DSTR Modifier Example

Destructor

Clients invoke `destructor()` to destroy an instance, which frees the memory associated with the instance and invalidates further accesses. Like the constructor, component implementations may wish to be notified when an instance is about to be destroyed, so that resources can be released. A component developer can specify a method to be invoked upon destruction, by prefixing **DSTR** to its IDL specification, as shown in Figure 5.13. Figure 5.14 contains the destructor for the `CStarfleetGalaxyClass` component-class.

Create_Cicap

A constructor returns an *owner CICAP*, which allows access to *all* interfaces of the component-instance. Often, however, clients need to provide other clients with access to only a subset of the interfaces of an instance. For example, in a component-based file system, users should be able to access the `IReadDirectory` interface of the `/home` directory, but not its `IModifyDirectory` interface. Partial access is also essential for secure aggregation and delegation, as will be discussed in Sections 5.5.4 and 5.5.5 respectively.

`create_cicap()` allows a client holding the owner CICAP to create extra CICAPs (E-CICAPs) with access to a subset of the exported interfaces. A 16-bit bit-field passed to the method specifies the interfaces the new CICAP will have access to. Each bit represents an interface 'provided' by the component-class (the class interface is *not* represented). The least-significant-bit (LSB) represents the first 'provided' interface, the second-LSB represents the second interface, etc. For example, referring to Figure 5.3, a bit-field corresponding to the integer value `0x05` would create a new CICAP with access to the `IEngineeringConsole` and `IHolodeck` interfaces. As there is a limit of three E-CICAPs for each component-instance, and no `delete_cicap()` method, clients must also specify which `ecicap` field should be used to store the E-CICAP.

```
static void *instobj;

res_desc_t csgc_ci_stub_destructor( cap_t param )
{
        apddesc_t apd;
        cicap_t _cicap;
        inst_hdr_t *hdr;
        res_desc_t res;

        /* Validate the passed CICAP */
        _cicap = *((cicap_t*)&param);
        hdr = (inst_hdr_t*)_cicap.ref;
        if( ((hdr->flags & 0x1) == 0) || hdr->ocicap != _cicap.password ) {
                res.type = MCS_STUB_EXCEPTION;
                res.data1 = MCS_STE_PROT;
                return res;
        }

        /* Call the DSTR methods */
        csgc_iws_InitialiseWeapon( (csgc_t*)&hdr->dcicap[1] );

        /* Get the address of the instance object */
        if( instobj == 0 ) {
                ApdGet( &apd );
                for( i = 0; i < apd.n_apd; i++ ) {
                        clist = (clist_t*)apd.clist[i].address;
                        if( clist != 0 && clist->format == CL_PROTCTX ) {
                                instobj = clist->caps[0].address;
                                break;
                        }
                }
                if( instobj == 0 ) {
                        res.type = MCS_STUB_EXCEPTION;
                        res.data1 = MCS_STE_NO_INSTOBJ;
                        return res;
                }
        }

        /* Free the block */
        hdr->flags = 0;
        *((inst_hdr_t**)hdr) = *((inst_hdr_t**)instobj);
        *((inst_hdr_t**)instobj) = hdr;

        /* Return */
        res.type = 0;
        return res;
}
```

Figure 5.14: Destructor

```
res_desc_t csgc_ci_create_cicap( cap_t param )
{
    cicap_t cicap;
    char slot;
    unsigned short ef;
    inst_hdr_t *hdr;
    res_desc_t res;
    void **pb;

    /* Unpack parameters */
    pb = (void**)param.address;
    cicap = *((cicap_t*)pb);
    pb += 2;
    slot = *((char*)pb);
    pb += 1;
    ef = *((unsigned short*)pb);

    /* Validate the passed CICAP */
    hdr = (inst_hdr_t*)cicap.ref;
    if( ((hdr->flags & 0x1) == 0) || hdr->ocicap != cicap.password ) {
        res.type = MCS_STUB_EXCEPTION;
        res.data1 = MCS_STE_PROT;
        return res;
    }

    /* Create new E-CICAP */
    hdr->ecicap[slot%3] = generate_password();
    hdr->ef[slot%3] = ef;

    /* Return */
    res.type = 0;
    res.data1 = hdr->ecicap[slot%3];
    return res;
}
```

Figure 5.15: Create_Cicap

When a component stub is invoked, it checks to see whether the passed CICAP matches the owner CICAP, or any of the E-CICAPs with access to the interface. Figure 5.15 contains the `create_cicap()` implementation for any component-class. As the arguments of this method are more than 128-bits, a *parameter buffer* must be used. Argument, and return value, passing protocols are described in Section 5.3.3, and can be ignored here.

5.3.3 Component Stub Code

Every interface method exported by a component-class registers an entry point with the Component Locator. Interface objects invoke methods by retrieving this entry point, and passing it to the Component Invoker, which performs a `PdxCall()`. Rather than being the address of the actual interface method, the registered entry point is the address of a short *component stub* procedure, that calls the interface method, as well as performing some additional operations. There are five reasons why a stub procedure is used.

- As described in Section 4.5.3, MCS entry points must have the signature:

  `res_desc_t stub_name( cap_t param );`

  For transparency, component stubs remove this restriction.
- MCS-RT provides a procedural request mechanism, whereas the desired semantics are to invoke a method on a component. Component stubs retrieve the instance from the parameter list and invoke the corresponding method on the instance.
- As described in Section 5.2, component stubs perform instance access control.
- Some component model features, such as delegation, require additional code. Component stubs provide a convenient location for this code, clearly separating the component model code from the component implementation code. This separation makes the component mechanics as transparent as possible, and is essential if the component model code is to be generated by an IDL compiler.
- In a heterogenous system component stubs may need to convert the binary format of data into a common 'on-the-wire' data representation. Examples include the *eXternal Data Representation (XDR)* used by Sun RPC [Sun85], and the *Common Data Representation (CDR)* used by CORBA [OMG99c].

This section describes component stubs for basic methods, and therefore only addresses the first three points. Component model features are described in Section 5.5, while heterogeneity issues, of both language and architecture, are beyond the scope of this thesis.

Removing Parameter Limitations

MCS defines the parameter of an entry point as two words (128 bits). As component methods may have an arbitrary number of parameters, a protocol is required between component stubs and interface objects to remove this limitation.

If a non-static method has no arguments the component stub parameter contains the CICAP. Otherwise, the parameter is a capability for a *parameter buffer (PB)* containing

```
void FirePattern( long power, char class, int n_rounds, pattern_t pattern );
```

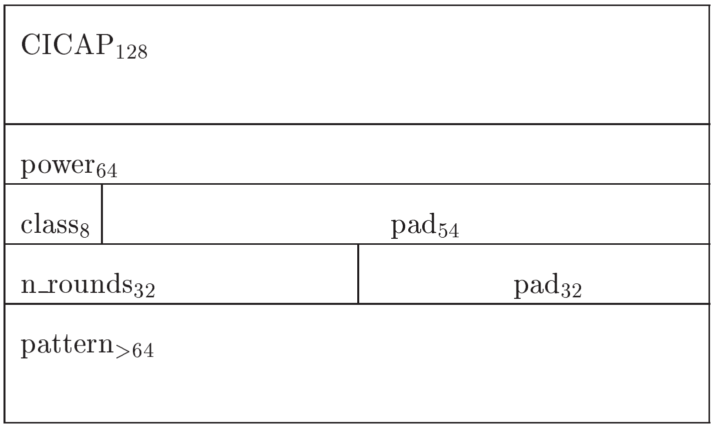


Figure 5.16: Parameter Buffer Format

```
res_desc_t csgc_iws_stub_Status( cap_t param )
{
        cicap_t _cicap;

        /* unpack parameters */
        _cicap = *((cicap_t*)&param);

        /* other stub code ... */
}
```

Figure 5.17: Direct Parameter Transfer

the method's arguments. As component stubs are individually developed, or generated, from an IDL specification, they statically know the method's signature, and therefore whether to expect a parameter buffer. If a parameter buffer is used, the first two words contain the CICAP, followed by the arguments in left-to-right order. Arguments are all stored at word aligned addresses. Figure 5.16 shows the layout of a parameter buffer passed to the `FirePattern()` method of the `IWeaponSystems` (Figure 5.3). Figures 5.17 and 5.18 demonstrate direct and buffered parameter transfer respectively.

Figure 5.18 shows that to retrieve parameters from a PB, the component stub simply obtains the address of the buffer and starts unpacking. If the buffer is not in the component-class's protection domain, a protection fault will occur. Component stubs could avoid this protection fault by performing an `ApdLookup()` system call [HVGR99], which searches the current protection domain for a given capability, on every method invocation, and adding the passed capability to the protection domain if it is not found. A protection fault will still occur if the passed capability is invalid, however, this results in a system

```
res_desc_t csgc_iws_stub_FirePattern( cap_t param );
{
        cicap_t _cicap;
        void **_pb;
        long power;
        char class;
        int n_rounds;
        pattern_t pattern;

        /* unpack parameters */
        _pb = (void**)param.address;
        _cicap = *((cicap_t*)_pb);
        _pb += 2;
        power = *((long*)_pb);
        _pb += 1;
        class = *((char*)_pb);
        _pb += 1;
        n_rounds = *((int*)_pb);
        _pb += 1;
        pattern = *((pattern_t*)_pb);

        /* other stub code ... */
}
```

Figure 5.18: Buffered Parameter Transfer

exception being returned by `ci_request()` to the interface object, whose responsibility it is to pass a valid capability. Obviously, performing a system call that scans (possibly) the entire protection domain on every method invocation creates significant overhead. Instead, when a PDX protection domain is initialised, it registers an exception handler for protection faults (`ST_PROT` exceptions) using Mungi's `ExcptReg()` system call. As described in Section 2.1, handlers execute in the faulting thread's protection domain, and with the faulting thread's stack. The `ST_PROT` exception handler locates the capability on its stack, and searches the protection domain using `ApdLookup()`. If the capability *is not* found, then it is added to the protection domain, and the component stub is restarted. If the capability *is* found, then the interface object has passed an invalid capability, and the method invocation is aborted, resulting in a system exception being returned by `ci_request()`.

Clients reuse parameter buffers across calls, amortizing the cost of handling the exception.

```
res_desc_t csgc_iws_stub_Status( cap_t param )
{
        cicap_t _cicap;
        wstatus_t _result;
        res_desc_t _retval;

        /* unpack parameters */
        _cicap = *((cicap_t*)&param);

        /* other stub code ... */

        /* set result descriptor */
        _retval.type = 0;
        _retval.data1 = _result;
        return _retval;
}
```

Figure 5.19: Direct Return Transfer

Removing Return Value Limitations

A similar situation exists for the return value. MCS entry points must have a return value type of `res_desc_t`, shown below. This is a 128 bit structure, with two bits used to indicate the exception status of the method invocation, leaving 126 bits for data.

```
typedef struct {
    unsigned long data1;
    unsigned data0 : 62;
    unsigned type  : 2;
} res_desc_t;
```

If the return value is $\leq$ 64 bits, then it is placed in `data1`. If the return value is greater than 64 bits, but less than 126 bits, its upper 64 bits are placed in `data1`, and its lower bits are placed in the *most significant bits* of `data0`. Otherwise, the interface object must use a parameter buffer when calling the component stub, and the return value is copied into the start of this buffer. Therefore, parameter buffers are used if *either* the arguments are greater than 128 bits, or the return value is greater than 126 bits. Figures 5.19 and 5.20 demonstrate direct and buffered return transfer respectively.

```
res_desc_t csgc_iws_stub_FireAtTarget( cap_t param )
{
        cicap_t _cicap;
        void **_pb;
        target_desc_t _result;
        res_desc_t _retval;

        /* unpack parameters */
        _pb = (void**)param.address;
        cicap = *((cicap_t*)_pb);

        /* other stub code ... */

        /* set result descriptor */
        *((target_desc_t*)_pb) = _result;
        _retval.type = 0;
        return _retval;
}
```

Figure 5.20: Buffered Return Transfer

Procedural to Component-Oriented

MCS-RT provides a procedural request mechanism, whereas the desired semantics are to invoke a method on a component. All (non-static) methods pass a CICAP as the first parameter, which contains a pointer to the location of a native language 'object' on which the method should be invoked. A component stub simply casts the reference and invokes the method. For example, in C++ [ES90], the `FireAtTarget()` method is invoked by:

```
_res = _ref->FireAtTarget();
```

As C does not have an object construct, 'invoking a method' means passing the pointer to the interface function, e.g.:

```
_res = csgc_iws_FireAtTarget( _ref );
```

Access Control

Section 5.2 introduced the concept of a CICAP, and Section 5.3.2 showed how additional CICAPs can be created by the owner. A client may only invoke a method on an interface if it possesses the owner CICAP, or an E-CICAP with access to the interface. This validation is performed by the complete `Status()` component stub shown in Figure 5.21.

```
extern __inline__ validate_cicap( inst_hdr_t *hdr, cicap_t ucicap )
{
    return ((hdr->flag & MCS_IF_VALID) != 0)
             && (hdr->ocicap != _cicap.password
                || (hdr->ecicap[0] != _cicap.password && (hdr->ef[0] & CSGC_EF_IWS))
                || (hdr->ecicap[1] != _cicap.password && (hdr->ef[1] & CSGC_EF_IWS))
                || (hdr->ecicap[2] != _cicap.password && (hdr->ef[2] & CSGC_EF_IWS)));
}

res_desc_t csgc_iws_stub_Status( cap_t param )
{
    cicap_t _cicap;
    res_desc_t _retval;
    inst_hdr_t *hdr;

    /* unpack parameters */
    _cicap = *((cicap_t*)&param);

    /* validate access */
    hdr = (inst_hdr_t*)_cicap.ref;
    if( validate_cicap(hdr, _cicap) == 0 ) {
        _retval.type = MCS_STUB_EXCEPTION;
        _retval.data1 = MCS_STE_PROT;
        return _retval;
    }

    /* invoke method */
    _retval.data1 = (unsigned)csgc_iws_Status( (csgc_t*)&(hdr->dcicap[1]) );

    /* set result descriptor */
    _retval.type = 0;
    return _retval;
}
```

Figure 5.21: CICAP Access Control

5.4 Interface Objects

Interface objects were introduced in Sections 4.2.2 and 4.3.1 as an abstraction mapping components into the client's programming language, using the native 'object' construct. Clients always interact with a component-instance via an interface object. As their name suggests, an interface object represents an interface rather than a particular component-class. This is consistent with construction model presented in Figure 4.7, and essential for the implementation of features such as polymorphism. This section describes the implementation of an interface object for the `IWeaponSystems` interface presented in Figure 5.3. Although details depend on the language, the concepts remain the same.

5.4.1 Interface Object Structure

An interface object is an object in the traditional sense, i.e. a per-instance data block with an associated set of operations. All interface objects contain the same data members.

- An *actual CICAP*, assigned when the interface object is constructed, which does not change for the life of the interface object.
- An *effective CICAP*, which determines the component-class to be invoked, and is the CICAP that is passed on a method invocation. Why an effective CICAP is required, rather than just the actual CICAP, is related to the aggregation and delegation protocols described in Sections 5.5.4 and 5.5.5.
- A capability for a parameter buffer. As described above, this buffer is used for passing arguments and return values too large for the standard MCS signature.
- A cached component-class identifier (CID) for the effective CICAP, passed to the Component Locator (CL) when retrieving entry points. This is initially obtained from either the client or the `cl_get_component_class()` method.
- A list of entry points. Entry points are resolved lazily using the CL, and placed into this list. Obviously, the list must be flushed when the effective CICAP changes.

Figure 5.22 presents the C structure for the data of an interface object. An interface object has a method corresponding to each method of the interface's IDL specification. In addition to these, every interface object also provides two constructors, a component destructor, and a method to retrieve the actual CICAP. A standard implementation of these methods is presented in Section 5.4.2.

```
typedef struct {
    cicap_t acicap;
    cicap_t ecicap;
    cid_t cid;
    cap_t pb;
    entry_pt_t epts[MAX_EPTS];
} interface_object_t;
```

Figure 5.22: Interface Object Structure

Figure 5.23 presents a header file for the `IWeaponSystems` interface. Translating IDL defined methods into a client implementation language, is done according to a language mapping specification. MCS follows existing CORBA standards where possible, therefore, this example uses OMG's *C-Language Mapping Specification* [OMG99b], with two deviations. Firstly, the 'CORBA_' namespace has been omitted, e.g.:

`CORBA_long` → `long`

Secondly, namespace prefixes are initialised, and made lower case, e.g.:

`IWeaponSystems_Status` → `iws_Status`

5.4.2 Interface Object Implementation

This section describes an implementation of the `IWeaponSystems` interface object. Constructors and destructors of different interface objects differ only in the constants passed to the Component Locator, and the type of the pointer returned. Methods corresponding to those defined in IDL obviously differ according to their arguments and return value, however, as for component stubs, the process is identical.

Exceptions

Exceptions signal that an unexpected error has occurred during a method invocation, without interfering with the return value of a successful invocation. As described in Section 4.5.1, they are passed from the component stub to the interface object in the `res_desc_t` structure. Interface objects must either handle the exception, or pass it to the client using the standard language mechanism. Since the C language does not provide

```
/*************************************************************************************
 * IWeaponSystems_io.h
 *
 * Interface object definition for the IWeaponSystems interface.
 *************************************************************************************/
#ifndef __IWEAPONSYSTEMS_H__
#define __IWEAPONSYSTEMS_H__

#include "mcs_std_io.h"
#include "other_types_cis.h"

/* Interface constants */
#define IID_IWEAPONSYSTEMS      2475
#define MID_START               101
#define MID_ACQUIRETARGET       101
#define MID_FIREATTARGET        102
#define MID_FIREPATTERN         103
#define MID_STATUS              104
#define MID_INITIALISEWEAPON    105
#define MID_RETIREWEAPON        106

/* Alias the interface object */
#define IWeaponSystems_t interface_object_t

/* Standard methods */
IWeaponSystems_t *iws_create_constructor( cid_t cid, environment_t *ev );
IWeaponSystems_t *iws_cicap_constructor( cicap_t cicap, environment_t *ev );
IWeaponSystems_t *iws_clone_constructor( IWeaponSystems_t *io, environment_t *ev );
void iws_destructor( IWeaponSystems_t *cthis, environment_t *ev );
#define iws_get_cicap(cthis) (cthis->acicap)

/* IDL defined methods */
int iws_AcquireTarget( IWeaponSystems_t *cthis, target_desc_t *spec, environment_t *ev );
target_desc_t iws_FireAtTarget( IWeaponSystems_t *cthis, environment_t *ev );
void iws_FirePattern( IWeaponSystems_t *cthis, long power, char class, int n_rounds,
                      pattern_t p, environment_t *ev );
wstatus_t iws_Status( IWeaponSystems_t *cthis, environment_t *ev );
void iws_InitialiseWeapon( IWeaponSystems_t *cthis, environment_t *ev );
void iws_RetireWeapon( IWeaponSystems_t *cthis, environment_t *ev );

#endif /* __IWEAPONSYSTEMS_H__ */
```

Figure 5.23: IWeaponSystems_io.h

native exception handling support, clients receive exceptions via a parameter passed to each interface object method. This parameter has type `environment_t`.

```
typedef struct {
    char _major;
    unsigned short _minor;
    char data[MAX_DATA];
} environment_t;
```

When an interface object method returns, `_major` indicates whether an exception has occurred, containing one of the values MCS_NO_EXCEPTION, MCS_USER_EXCEPTION, MCS_STUB_EXCEPTION or MCS_SYSTEM_EXCEPTION. `_minor` contains the specific exception identifier, and `data` contains any exception data.

Creation Constructor

```
IWeaponSystems_t *iws_create_constructor( cid_t cid, environment_t *ev );
```

Creation constructors create a new component-instance. As there is a many-to-many relationship between interfaces and component-classes, clients must specify the component-class of the instance they wish to create. As shown in Figure 5.4, the component-class identifier is located in the 'common' header file. Constructors are reasonably straightforward, and therefore, are explained by way of an example presented in Figure 5.24.

CICAP Constructor

```
IWeaponSystems_t *iws_cicap_constructor( cicap_t cicap, environment_t *ev );
```

CICAP constructors are used to create new interface objects that are connected to an existing component-instance. Clients use this constructor when they wish to invoke methods on multiple interfaces of an instance, or when a CICAP is received from an external source, such as another client or a persistent store. As a CICAP is supplied by the client, the constructor does not need to invoke the component-class constructor, otherwise it performs the same operations as the creation constructor, as shown in Figure 5.25.

CICAP constructors do not check that the passed cicap is valid, or that the component-class of the CICAP exports this interface. Such errors will result in a stub exception being returned to the client on the first method invocation.

```
#include "IWeaponSystems_io.h"
#include "mcs_cl.h"
#include "mcs_ci.h"

IWeaponSystems_t *IWS_create_constructor( cid_t cid, environment_t *ev )
{
        IWeaponSystems_t *new_io;
        entry_pt_t ept;
        int r;
        res_desc_t res;
        cap_t param = {0,0};

        /* clear the exception parameter */
        ev->_major = MCS_NO_EXCEPTION;

        /* allocate memory for the new interface object - legal error so no exception */
        new_io = malloc( sizeof(IWeaponSystems_t) );
        if( new_io == NULL )
                return NULL;

        /* get the entry point for the constructor */
        r = cl_get_entry_pt( cid, IID_CLASS_INTERFACE, MID_CONSTRUCTOR, &ept );
        if( r != 0 ) {
                ev->_major = MCS_STUB_EXCEPTION;
                ev->_minor = r;
                goto error;
        }

        /* invoke the constructor */
        r = ci_request( ept, param, &res, PD_EMPTY );
        if( r != 0 ) {
                ev->_major = res.type;
                ev->_minor = res.data1 >> 48;
                goto error;
        }

        /* fill in the interface object data */
        new_io->acicap.ref = (res.data0<<2);
        new_io->acicap.password = res.data1;
        new_io->ecicap = new_io->acicap;
        new_io->cid = cid;
        memset( new_io->epts, 0, sizeof(entry_pt_t)*MAX_EPTS );
        new_io->pb = create_parameter_buffer();

        /* done */
        return new_io;

error:
        free( new_io );
        return NULL;
}
```

Figure 5.24: Creation Constructor

```
#include "IWeaponSystems_io.h"
#include "mcs_cl.h"
#include "mcs_ci.h"

IWeaponSystems_t *IWS_cicap_constructor( cicap_t cicap, environment_t *ev )
{
        IWeaponSystems_t *new_io;
        int r;

        /* clear the exception parameter */
        ev->_major = MCS_NO_EXCEPTION;

        /* allocate memory for the new interface object */
        new_io = malloc( sizeof(IWeaponSystems_t) );
        if( new_io == NULL )
                return NULL;

        /* fill in interface object */
        new_io->acicap = new_io->ecicap = cicap;
        new_io->cid = cl_get_component_class( cicap );
        memset( new_io->epts, 0, sizeof(entry_pt_t)*MAX_EPTS );
        new_io->pb = create_parameter_buffer();

        /* done */
        return new_io;
}
```

Figure 5.25: CICAP Constructor

Destructor

```
void iws_destructor( IWeaponSystems_t *cthis, environment_t *ev );
```

Destructors destroy both the component-instance and the interface object. Destructors are always invoked on the component-class of the *actual* CICAP, rather than the effective CICAP, as this is the component-class that constructed the instance. A destructor is essentially just a method invocation to the component-class destructor. As method invocation has not yet been described an example destructor is not presented.

Interface Method Stubs

An interface method stub is generated for each method defined in the interface's IDL specification, providing clients with a native language interface. These stubs are the client-side equivalent of component stubs, and serve a largely similar purpose.

- Implement the parameter passing protocol described in Section 5.3.3, and include the CICAP as the first parameter to any non-static method invocation.
- Locate the component stub entry point of the appropriate method.
- Implement component model features, such as delegation.
- Convert data into a standard binary format in heterogenous systems.

Section 5.5 describes the implementation of the component model, and heterogeneity is beyond the scope of this thesis. Therefore, only the first two features are addressed here.

Interface methods stubs locate the appropriate entry point using the Component Locator. Each entry point is uniquely identified by its `(CID,IID,MID)` tuple, which is easily obtained by the stubs. The `CID` is stored in the interface object by the constructor, and the last two elements are statically known. If the component-class does not export the interface, the Component Locator will return an error, and the interface object will return this error to the client as a stub exception. Parameter passing was described in detail in Section 5.3.3, and so will not be repeated here. Figure 5.26 shows the complete interface method stub for the `FirePattern()` method of the `IWeaponSystems` interface.

```
#include "IWeaponSystems_io.h"
#include "mcs_cl.h"
#include "mcs_ci.h"

void iws_FirePattern( IWeaponSystems_t *cthis, long power, char class, int n_rounds,
                                        pattern_t p, environment_t *ev )
{
        void **_pb;
        entry_pt_t _ept;
        int _r;
        res_desc_t _res;

        /* pack parameters */
        _pb = (void**)cthis->pb.address;
        *((cicap_t*)_pb) = cthis->ecicap;
        _pb += 2;
        *((long*)_pb) = strength;
        _pb += 1;
        *((char*)_pb) = class;
        _pb += 1;
        *((int*)_pb) = n_rounds;
        _pb += 1;
        *((pattern_t*)_pb) = p;

        /* get the entry point */
        _ept = cthis->epts[MID_FIREPATTERN-MID_START];
        if( _ept == NULL ) {
                _r = cl_get_entry_pt( cthis->cid, IID_IWEAPONSYSTEMS, MID_FIREPATTERN, &_ept );
                if( _r != 0 ) {
                        ev->_major = MCS_STUB_EXCEPTION;
                        ev->_minor = _r;
                        return;
                }
                cthis->epts[MID_FIREPATTERN-MID_START] = _ept;
        }

        /* invoke the entry point */
        _r = ci_request( _ept, cthis->pb, &_res, PD_EMPTY );
        ev->_major = res.type;

        /* check exceptions */
        if( _r != 0 ) {
                ev->_minor = (res.data1 >> 48);
                memcpy( ev->_data, &(((char*)&(res.data1))[2]), 13 );
                return;
        }

        /* done */
        return;
}
```

Figure 5.26: Interface Method Stub

5.4.3 Client Example

Interface objects are an abstraction designed to allow clients to program naturally with components. Figure 5.27 presents an example of a client creating a new component-instance, invoking some methods on this instance, creating a new interface object for the same component-instance, and invoking some methods on the new interface.

```
#include <stdio.h>
#include <assert.h>
#include "INavigation_io.h"
#include "IWeaponSystems_io.h"
#include "CStarfleetGalaxyClass.h"

int main( void )
{
	IWeaponSystems_t *iws;
	INavigation_t *in;
	target_desc_t *target, result;
	nav_desc_t nav;
	environment_t ev;
	wstatus_t status = WS_STANDBY;
	int r = -1;

	/* create a new component, and return the weapon systems interface */
	iws = iws_create_constructor( CID_CSTARFLEETGALAXYCLASS, &ev );
	assert( iws != NULL && ev->_major == MCS_NO_EXCEPTION );

	/* wait for weapons to come online */
	while( status == WS_STANDBY ) {
		status = iws_Status( iws, &ev );
		assert( ev->_major == MCS_NO_EXCEPTION );
	}
	assert( status != WS_OFFLINE );

	/* acquire target and fire */
	target = get_target_spec();
	while( r != 0 )
		r = iws_AcquireTarget( target, &ev );
	result = iws_FireAtTarget();

	/* create a new interface object, connected to the same instance */
	in = in_cicap_constructor( iws_get_cicap(iws), &ev );
	assert( in != NULL && ev->_major == MCS_NO_EXCEPTION );

	/* invoke methods on the new interface */
	nav = get_navigation_course();
	in_engage_engines( in, nav, &ev );

	/* done */
	return 0;
}
```

Figure 5.27: Client Example

5.5 Component Model Implementation

This section describes the implementation of the component model presented in Section 4.3. Many features of this component model are a natural result of the interface model implemented in the previous sections, and therefore, only require a description of how component-classes and interface objects should be used. Other features, such as delegation, require code to be added to the component stubs and interface methods stubs.

5.5.1 Interfaces

Sections 5.3 and 5.4 described in detail the implementation of IDL interfaces, the class interface, the component construction model, component stubs and interface objects.

5.5.2 Encapsulation

Constructors create a new instance of the component inside a Mungi object that exists only within the component's protection domain. It then passes the *address* of the instance, which uniquely identifies an object in a SASOS, to the client, but does not return a (Mungi) capability. The component implementation is therefore the only domain holding a capability to the new instance, and as such is the only domain that can directly access the instance data. Instances are therefore data encapsulated.

Encapsulation of control ensures that clients can only operate on an instance via the defined entry points, and that they cannot modify the control flow of these entry points, e.g. they cannot force a method to abort prematurely. Combined, data and control encapsulation allow state invariants to be guaranteed. Component implementations are contained in Mungi PDX objects, with `PdxCall()` providing the trusted path mechanism. `PdxCall()` will only invoke operations that are registered in the Mungi Object Table (OT), and so by only registering the component stubs, control is encapsulated.

Delegation allows an interface of a component-instance (C_1) to be overridden by another component-instance (C_2) at runtime. Methods invoked on this interface of C_1 will be redirected to C_2. C_2, however, does not have access to the instance data stored in C_1, therefore, delegation allows clients to change which data operation is invoked, rather than allowing clients to modify the control flow of a particular data operation.

5.5.3 Polymorphism

Polymorphism relates to the ability of a strongly typed client variable to refer to instances of different component-classes, and therefore, have different implementations for different instances. Component-classes that can be referenced by the same client type are *substitutable* for one another. As the same client code can be used to manage instances of different component-classes, polymorphism can reduce the size of applications. Also, a new component-class that is substitutable for an existing component-class, can be added without modifying the client code, therefore, polymorphism encourages extensibility.

Interfaces objects are inherently polymorphic. A client can interact with two component-classes exporting the same interface using the same interface object type. When a method of an interface object is called, the interface method stub uses the component class identifier (CID) of the contained CICAP to determine the component implementation to invoke. If the component-class of the CICAPs contained in two interface objects of the same type are different, then a different `(CID,IID,MID)` tuple will be passed to the Component Locator. In general, this means that a different entry point reference will be returned, and so a different component implementation will be invoked. Therefore, two component-classes may appear (syntactically) as the same to a client, i.e. interface objects are polymorphic. Figure 5.28 illustrates how interface objects provide polymorphism.

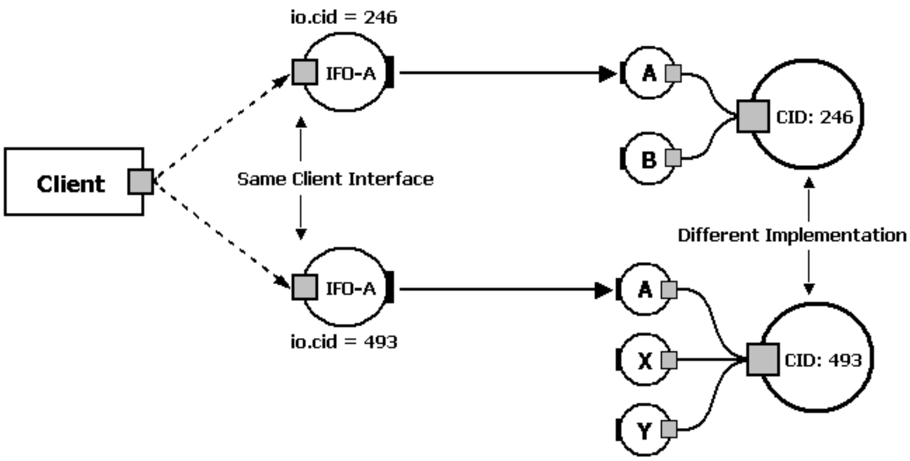


Figure 5.28: Polymorphism Using Interface Objects

As interface method stubs determine the `CID`, and the corresponding entry point, dynamically using the Component Locator, and `PdxCall()` dynamically loads and links the component implementation, interface objects are late binding. Late binding and polymorphism allow applications to use new component-classes without recompiling.

5.5.4 Component Code Reuse

Code reuse in MCS is based on *composition* and *aggregation*.

Composition

Under composition, a component-class (C_{outer}) reuses the implementation of interfaces in another component-class (C_{inner}) by creating an instance of C_{inner} and invoking methods on its interfaces. Composition, therefore, relies upon the natural reusability of components. Usually, each instance of C_{outer} uses a different instance of C_{inner}, created by its constructor, or more precisely, by a **CONS** method. Instances of C_{outer} never distribute the CICAP to their instance of C_{inner}, so that the inner-component is effectively *contained* within the outer-component. Composition does not require any implementation.

Aggregation

Aggregation is used when a component-class (C_{outer}) wishes to export an interface implementation provided by another component-class (C_{inner}) unchanged. Using composition, a method that is unchanged by C_{outer} must still be relayed by C_{outer} to C_{inner}, resulting in additional overhead. Aggregation allows clients to, transparently, invoke methods directly on C_{inner}. Figure 4.12 illustrates the aggregation concept.

As a given interface may be aggregated by some component-classes, but not by others, a protocol is required that allows an interface object to identify aggregated interfaces and obtain a CICAP for the appropriate instance of C_{inner}. This section describes this protocol and its implementation. Four requirements are identified for the protocol.

- C_{inner} must not be required to take part in the protocol, as this would violate encapsulation. A component-class's behaviour should not depend on the caller.
- Different instances of C_{outer} must be able to aggregate a given interface from different component-classes, i.e. one instance of C_{outer} may aggregate an interface from C_{in1}, while another instance of C_{outer} may aggregate the same interface from C_{in2}.
- Aggregation must be transitive, i.e. C_{outer} should be able to aggregate an interface from C_{trans}, which itself aggregates the interface from C_{inner}. A limitation should not be placed on the length of this aggregation chain.

- Aggregation must not violate containment. Therefore, clients should only be able to invoke methods on the interfaces of the inner instance that have been aggregated.

Figure 5.29 summarises the aggregation protocol, described in detail below.

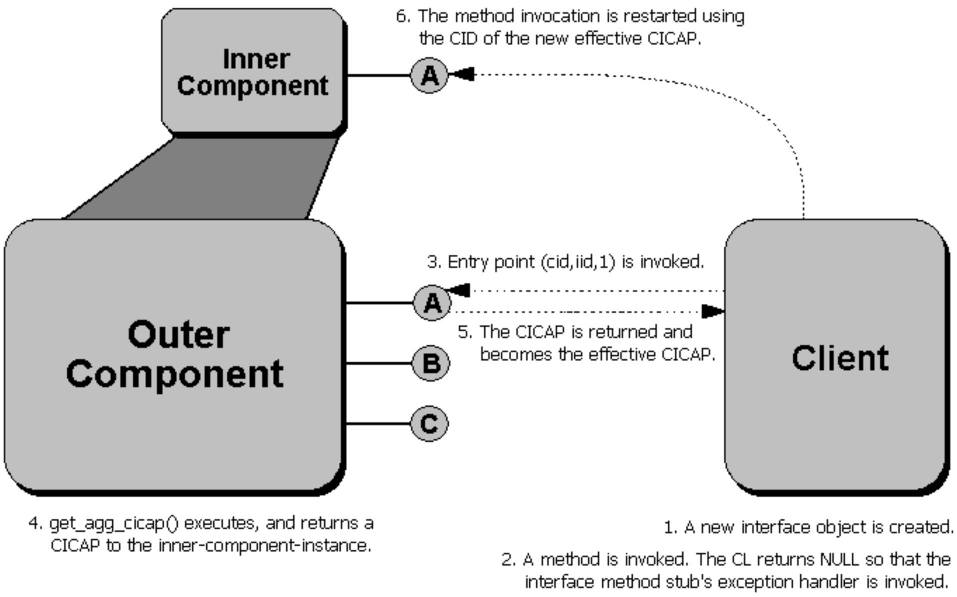


Figure 5.29: Aggregation Protocol

1. A client creates an interface object using any constructor. Components cannot aggregate the class interface or any interface containing a **CONS** method, therefore, construction is not affected by aggregation.

2. A method is invoked on the interface object. The initial parts of the interface method stub are the same as presented in Section 5.4.2. It packs the parameters, retrieves the entry point from the Component Locator and calls `ci_request()`. Methods of aggregated interfaces, however, are not registered for the outer-component-class, and so the returned entry point will always be `NULL`.

3. As there is no Mungi object at `NULL`, `ci_request()` will return a protection fault as a system exception. This invokes the interface method stub's exceptions handler (shown in Figure 5.31), which checks whether the entry point was `NULL`. If so, it retrieves the entry point for `(cid,iid,1)`, which must be registered by all aggregated interfaces for the *outer*-component-class. This entry point, whose signature

```
component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    provides IStellarCartography;
    [AGG] provides IWeaponSystems;
};
```

Figure 5.30: AGG Modifier Example

is shown below, calls a developer implemented method that returns a CICAP for this interface's inner-component-instance.

```
cicap_t get_agg_cicap()
```

4. `get_agg_cicap()` has a standard component stub that invokes a method implemented by the component developer. This method has no arguments, and returns the CICAP for the inner-component-instance to the interface object.
5. This CICAP becomes the effective reference of the interface object, and is stored in its `ecicap` data member. `cl_get_component_class()` is then used to determine the CICAP's component-class identifier, which is stored in the `cid` member.
6. The interface object restarts the original method invocation using the new CID and CICAP returned by the outer-component. If the inner-component also aggregates this interface, then the protocol returns to Step 2.

A component-class specifies that an interface is aggregated by prefixing the **AGG** modifier to its 'provides' declaration, as shown in Figure 5.30. Aggregated interfaces create only one method, `get_agg_cicap()`, in the outer-component-class. An interface method stub implementing the aggregation protocol is shown in Figure 5.31. As the code for packing the arguments into a parameter buffer and retrieving the entry point is unchanged from Figure 5.26, it has been omitted due to space constraints. Some important features of the protocol are discussed below.

- No code is added to the critical path in the interface method stub. This is why the check for a `NULL` entry point is delayed until the exception handler.
- `acicap` in the interface object always contains the original CICAP. This is essential as the client requires the original CICAP to access other interfaces.

```
#include "IWeaponSystems_io.h"
#include "mcs_cl.h"
#include "mcs_ci.h"

/* interface method stub for the get_agg_cicap() method */
cicap_t iws_get_agg_cicap( IWeaponSystems_t *cthis, environment_t *ev );

void iws_FirePattern( IWeaponSystems_t *cthis, long power, char class, int n_rounds,
                                              pattern_t p, environment_t *ev )
{
        void **_pb;
        entry_pt_t _ept;
        int _r;
        res_desc_t _res;
        cicap_t _adcicap;

AD_RESTART:
        /* pack parameters ... */

        /* get the entry point ... */

        /* invoke the entry point */
        _r = ci_request( _ept, cthis->pb, &_res, PD_EMPTY );
        ev->_major = res.type;

        /* check exceptions */
        if( _r != 0 ) {

                /* check if the interface is aggregated */
                if( _ept == NULL ) {
                        _adcicap = iws_get_agg_cicap( cthis, ev );
                        if( ev->_major != MCS_NO_EXCEPTION )
                                return;
                        cthis->ecicap = _adcicap;
                        cthis->cid = cl_get_component_class( _adcicap );
                        goto AD_RESTART;
                }

                /* other exceptions are just returned to the client */
                ev->_minor = (res.data1 >> 48);
                memcpy( ev->_data, &(((char*)&(res.data1))[2]), 13 );
                return;
        }

        /* done */
        return;
}
```

Figure 5.31: Interface Method Stub With Aggregation

- As the protocol updates the effective reference, it is only executed on the first call to an aggregated interface, amortizing its cost.
- Using `create_cicap()`, an outer-component can create a CICAP for the inner-component, with access to only the aggregated interfaces. If only this CICAP is given to clients, via the `get_agg_cicap()` method, then containment is maintained.

5.5.5 Customisation

Customisation in MCS is based on *interface delegation*. Delegation allows a component-*instance* (CI_E) to override an interface of another component-instance (CI_B) at runtime. Client invocations to this interface of CI_B are redirected (or delegated) to CI_E. Delegation allows new components to be transparently added to the system, and also allows behaviour to be modified at instance, rather than class, granularity. Without delegation, all clients must explicitly migrate to the new component-instance.

Delegation and aggregation are similar mechanisms. Both allow an interface of a component-class to become an alias for the corresponding interface of another component-class. There are, however, two crucial differences between delegation and aggregation, described below. Figure 5.32 illustrates the similarities and differences between delegation and aggregation.

- Delegation occurs at instance, rather than class, granularity, i.e. an interface may be delegated by one instance of a component-class, but not by another. As delegation is used to implement customisation, this is essential. Conversely, a given interface is either aggregated for all instances of the component-class, or for none, although instances may aggregate from different component-classes. This means that an interface cannot statically know if it is delegated.
- Delegation is requested by the inner component, whereas aggregation is requested by the outer component. As method requests are initially sent to the outer component, it must be aware that delegation has occurred. Therefore, a mechanism is required to inform the outer component that an interface has been delegated, and to whom.

Declaring a Delegatable Interface

Interfaces are declared as delegatable by prefixing the **DLG** modifier to its 'provides' declaration in the component-class specification, as shown below.

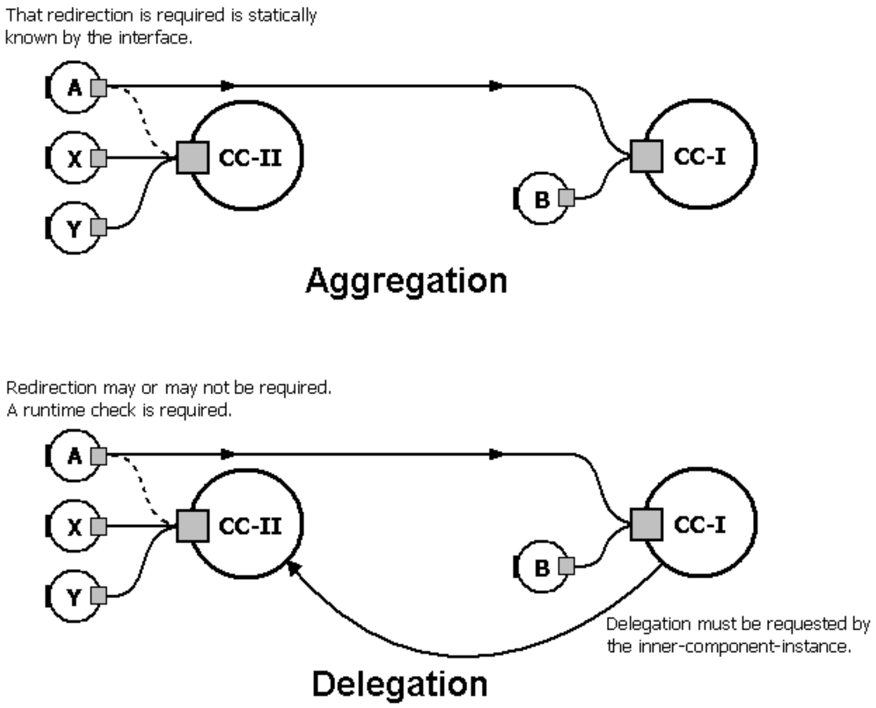


Figure 5.32: Aggregation vs Delegation

```
component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    provides IStellarCartography;
    [DLG] provides IWeaponSystems;
};
```

Registering the Overriding Instance

For every delegatable interface exported by a component-class, a corresponding *registration interface* is exported, and a Di-CICAP field created in the instance header (Section 5.9). Registration interfaces have the (implicit) IDL specification shown below.

```
interface IDelegationRegistration {
    void register_inner_component( cicap_t cicap );
};
```

```
res_desc_t csgc_idr_iws_stub_reg_inner_component( cap_t param )
{
        cicap_t _cicap, _r_cicap;
        void **_pb;
        res_desc_t _res;
        inst_hdr_t *hdr;

        /* unpack parameters */
        _pb = (void**)param.address;
        _cicap = *((cicap_t*)_pb);
        _pb += 2;
        _r_cicap = *((cicap_t*)_pb);

        /* validate access */
        hdr = (inst_hdr_t*)_cicap.ref;
        if( validate_cicap(hdr,_cicap) == 0 ) {
                _res.type = MCS_STUB_EXCEPTION;
                _res.data1 = MCS_STE_PROT;
                return _res;
        }

        /* store the passed cicap in the instance header */
        hdr->dcicap[IWS_DCICAP_FIELD] = _r_cicap;

        /* done */
        _res.type = 0;
        return _res;
}
```

Figure 5.33: register_inner_component

A component-instance overrides a delegatable interface by passing its CICAP to the `register_inner_component()` method of the corresponding registration interface. This method, whose implementation is shown in Figure 5.33, stores the passed CICAP in the D*i*-CICAP field for this interface. Delegation can be reset by passing a CICAP with a `NULL` reference. `register_inner_component()` is placed in a separate interface, as interfaces are the unit of access control, and the capability to override an interface is obviously a more privileged operation than invoking methods of the interface. In general, the creator of a component-instance should not distribute CICAPs with access to registration interfaces.

Delegation Protocol

Delegation is implemented using a protocol similar to the aggregation protocol. Due to the differences identified in Figure 5.32, however, two modifications are required.

- As an interface may not be delegated, the outer-component-class must provide an implementation of the interface. Therefore, unlike aggregation, these methods will have valid entry points registered with the Component Locator, and hence the initial method invocation will succeed. Component stubs of delegatable interfaces must raise an `MCS_STE_DELEGATED` exception if the interface is actually delegated.
- As CICAPs are too large to fit in the exception data area, interface objects must still invoke an entry point to obtain the CICAP for the inner-component-instance. This entry point is registered with the key `(cid,iid,2)`. As the CICAP to be retrieved is stored in a known field within the instance header, this component stub doesn't need to call the component implementation like `agg_get_cicap()`.

Figure 5.34 summarises the delegation protocol, with a detailed description of the steps given below. Figure 5.35 presents an example component stub for a delegatable interface, and Figure 5.36 presents an interface method stub implementing the delegation protocol.

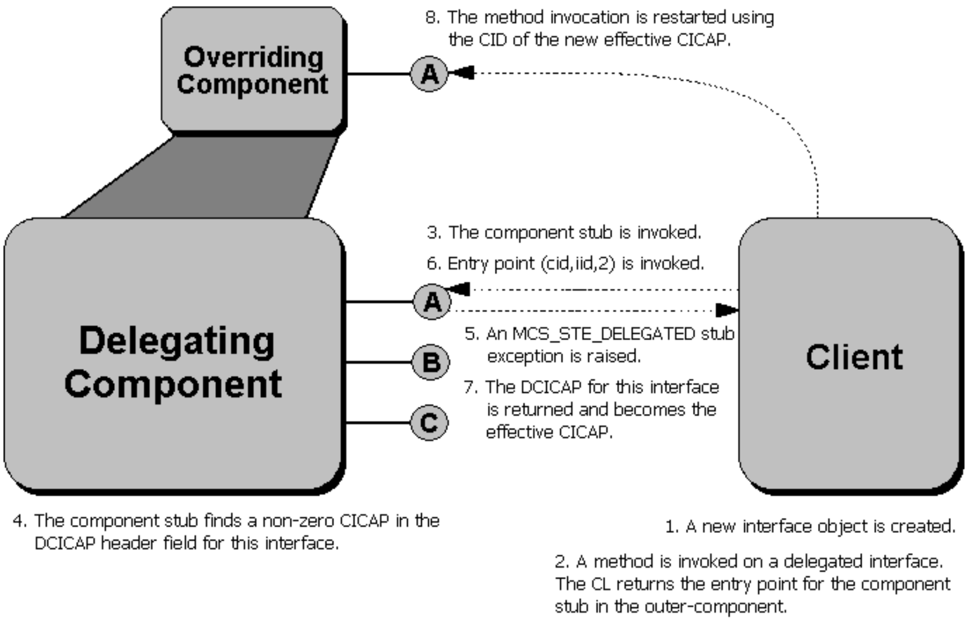


Figure 5.34: Delegation Protocol

1. A client creates an interface object using any constructor. Components cannot delegate the class interface or any interface containing a **CONS** method, therefore, construction is not affected by delegation.

2. A method is invoked on the interface object by a client.

3. As delegated interfaces are implemented by the outer-component, a valid entry point is retrieved from the CL, and the corresponding component stub invoked.

4. Component stubs for delegatable interfaces check the corresponding D-CICAP field in the instance header. If this field contains a CICAP with a non-zero `ref`, an `MCS_STE_DELEGATED` stub exception is raised.

5. This exception is returned to the interface object, invoking its exception handler. This checks whether the exception was `MCS_STE_DELEGATED`, and if so, retrieves the entry point for `(cid,iid,2)`, whose signature is given below.

   ```
   cicap_t get_del_cicap()
   ```

6. `get_del_cicap()` is invoked.

7. `get_del_cicap()` returns the CICAP in the corresponding D-CICAP field. This CICAP becomes the effective reference of the interface object, and is stored in its `ecicap` data member. `cl_get_component_class()` is then used to determine the CICAP's component-class identifier, which is stored in the `cid` member.

8. The interface object restarts the original method invocation using the new CID and CICAP returned by the outer-component.

Combined, aggregation and delegation create the possibility of cycles within the interface implementation chain. For example, in Figure 5.37, a component-instance CI_2 aggregates an interface from CI_1, which in-turn has delegated the interface to CI_1. In this situation, the interface object will loop forever. Currently, it is the developer's responsibility to ensure that such cycles do not form. Interface objects can avoid endless looping by placing a limit on the maximum number of iterations of the protocol.

Although this section has highlighted the similarities of delegation and aggregation, it is important to remember that they serve very different purposes. Aggregation allows a component-class to reuse an implementation of an interface provided by another component-class. Delegation allows a component-*instance* to be customised at runtime.

```
res_desc_t csgc_iws_stub_Status( cap_t param )
{
	cicap_t _cicap;
	res_desc_t _retval;
	inst_hdr_t *hdr;

	/* unpack parameters */
	_cicap = *((cicap_t*)&param);

	/* validate access */
	hdr = (inst_hdr_t*)_cicap.ref;
	if( validate_cicap(hdr,_cicap) == 0 ) {
		_retval.type = MCS_STUB_EXCEPTION;
		_retval.data1 = MCS_STE_PROT;
		return _retval;
	}

	/* check if this interface is delegated */
	if( hdr->dcicap[CSGC_IWS_DCICAP_FIELD].ref != NULL ) {
		_retval.type = MCS_STUB_EXCEPTION;
		_retval.data1 = MCS_STE_DELEGATION << 48;
		return _retval;
	}

	/* invoke method */
	_retval.data1 = (unsigned)csgc_iws_Status( (csgc_t*)&(hdr->dcicap[1]) );

	/* set result descriptor */
	_retval.type = 0;
	return _retval;
}
```

Figure 5.35: Delegatable Component Stub

```
/* interface method stub for the get_agg_cicap() and get_del_cicap() methods */
cicap_t iws_get_agg_cicap( IWeaponSystems_t *cthis, environment_t *ev );
cicap_t iws_get_del_cicap( IWeaponSystems_t *cthis, environment_t *ev );

#define IS_DELEGATED(x) (_res.type == MCS_STUB_EXCEPTION \
                        && ((_res.data1 >> 48) == MCS_STE_DELEGATION))

wstatus_t iws_Status( IWeaponSystems_t *cthis, environment_t *ev )
{
        void **_pb;
        entry_pt_t _ept;
        int _r;
        res_desc_t _res;
        cicap_t _adcicap;

AD_RESTART:
        /* pack parameters ... */

        /* get the entry point ... */

        /* invoke the entry point */
        _r = ci_request( _ept, cthis->pb, &_res, PD_EMPTY );
        ev->_major = res.type;

        /* check exceptions */
        if( _r != 0 ) {

                /* check if the interface is aggregated or delegated */
                if( _ept == NULL )
                        _adcicap = iws_get_agg_cicap( cthis, ev );
                else if( IS_DELEGATED(_res) )
                        _adcicap = iws_get_del_cicap( cthis, ev );

                /* if the interface is aggregated or delegated, reissue the call */
                if( ev->_major == MCS_NO_EXCEPTION ) {
                        cthis->ecicap = _adcicap;
                        cthis->cid = cl_get_component_class( _adcicap );
                        goto AD_RESTART;
                }

                /* other exceptions are just returned to the client */
                ev->_minor = (res.data1 >> 48);
                memcpy( ev->_data, &(((char*)&(res.data1))[2]), 13 );
                return;
        }

        /* done */
        return (wstatus_t)res.data1;
}
```

Figure 5.36: Interface Method Stub With Delegation

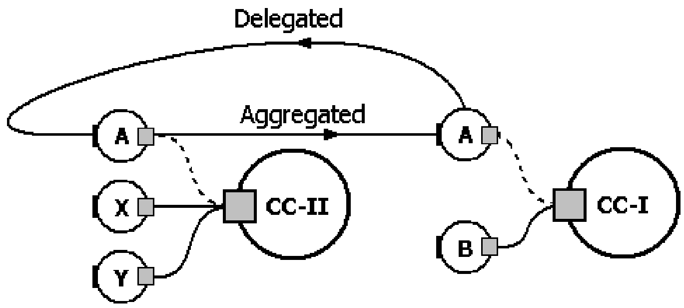


Figure 5.37: Aggregation / Delegation Cycle

5.5.6 Dynamic Type Discovery

Every MCS component-class provides type information via the `get_typeinfo()` method of the class interface. This method copies the component-class's type information into a client supplied buffer according to the format shown in Figure 5.38. If the buffer passed is too small to hold all the type information, as much information as possible is copied into the buffer and an `MCSE_STE_INSFBUFFER` exception is returned. All type information is statically known to the component-class.

5.5.7 Persistence and Naming

Mungi objects are persistent, and remain at a fixed address within the single-address-space. As component-instances are contained within Mungi objects, they are automatically persistent, and their virtual addresses provide globally unique names. By providing these services, Mungi demonstrates its suitability as a component platform.

5.5.8 Static Interfaces

A component-class can either provide an interface as static or non-static. Interfaces are provided as non-static unless prefixed with the **STATIC** modifier as shown in Figure 5.40. Methods of static interfaces always execute in a single, global, context. As they are not associated with an instance, static methods require component stubs and interface objects to be modified. These modifications are listed in the following sections.

CID_{64}					
$Version_{32}$			Interface $Count_{32}$		
IID-1_{64}					
. . .					
IID-i_{64}					
. . .					
IID-n_{64}					
$F1_8$	...	Fi_8	...	Fn_8	

CID: the component-class identifier.

Version: the minor version number.

Interface Count: the number of incoming and outgoing interfaces.

IID-i: the interface identifier of the ith interface.

Fi: a set of flags describing the ith interface, shown in Figure 5.39 (MSB at left).

Figure 5.38: Type Information Format

Component Stubs

- No CICAP is passed to a static method. This means that static methods may have up to two words of arguments before a parameter buffer needs to be used, and the first argument is stored at the start of the buffer.

- As there is no CICAP, component stubs perform no access validation. Static methods are sufficiently protected by Mungi PDX capabilities.

- Component implementation signatures do not expect a reference to be passed.

Non-Static	`wstatus_t csgc_iws_Status( csgc_t *cthis );`
Static	`wstatus_t csgc_iws_Status( );`

AGG	DLG	OUT	REG	Undefined

AGG: Aggregated.

DLG: Delegatable.

OUT: Outgoing, i.e. this is an event interface.

REG: A registration interface for a delegatable interface. This interface always immediately follows the corresponding delegatable interface.

Figure 5.39: Flag Field Format

```
component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    [STATIC] provides IStellarCartography;
    provides IWeaponSystems;
};
```

Figure 5.40: STATIC Modifier Example

Interface Objects

- Interface objects for static interfaces do not contain the CICAP data members.
- As interface objects for static interfaces have no CICAP, they also have no CICAP constructor. Clone constructors are the same for static interfaces.
- As no instance needs to be created, creation constructors do not invoke the component implementation. The creation constructor for a static interface is the non-static creation constructor, Figure 5.24, with lines 18-32 (inclusive) removed.

5.5.9 Version Support

Component-classes are modified for a variety of reasons in a variety of ways. For example, bugs fixes, improving performance, improving structure, adding interfaces to provide extended functionality, and modifying an interface to change the functionality of a component-class. Version support in MCS has two goals.

- If a new version of a component-class is backwardly compatible, then existing clients should automatically use the new implementation and protection context. The change-over must take place without requiring all clients to be halted. Backwardly compatible changes to a component-class are called *minor version* changes.
- If a new version of a component-class is *not* backwardly compatible, the new version should be able to coexist with existing versions. Clients should transparently use the appropriate component-class. Modifications to a component-class that are not backwardly compatible are called *major version* changes.

Support for multiple versions of a component-class is primarily a concern of the Component Locator. An interface object must be able to determine the appropriate entry points for a given component-class identifier. When a component implementation registers its entry points with the Component Locator, it specifies a component-class identifier (CID). In effect, the component implementation is registering itself as the handler for component-instances with this CID.

A new minor version of a component-class retains the same CID as the previous version, and registers itself as the handler for instances of this CID. When an interface object next requests an entry point for this CID, it will receive the new entry point, and so automatically use the new version. Provided that the previous minor version remains on the system for a short period after the new version has been installed, so any in-progress methods can complete, clients do not have to be halted. The **VO** field in the instance header allows a minor version to distinguish between instances created by different versions. The **VL** field allows a minor version to indicate that it has already seen this component-instance, and performed any updates required, e.g. to its data format.

Major version changes are considered to be different component-classes by MCS, therefore, they are registered using a new CID. Major versions are related only by their human-readable name, e.g. `CStarfleetGalaxyClass`. As the methods of a new major version are registered with a new CID, existing component-instances continue to use the same component implementation, i.e. major versions of the same component-class can coexist.

Version numbers are included in a component-class's IDL specification, as shown in Figure 5.41. Major version numbers are separated into two parts, a 16 bit *vendor ID* and a 16 bit *local major version*. This allows disjoint development of a component-class, without conflicts arising if multiple versions appear on the same system.

When a new version of a component-class is developed, the developer must determine whether this is a major or minor update. Table 5.1 lists some common ways a component-

```
[VER(VID_DISY,2,4)] component CStarfleetGalaxyClass {
    provides IEngineeringConsole;
    provides IEngines;
    provides IHolodeck;
    provides INavigation;
    provides ISensors;
    provides IStellarCartography;
    provides IWeaponSystems;
};
```

Figure 5.41: Version IDL Example

class is modified, and whether they are major or minor updates.

Component Modification	Major / Minor
Implementation bug fix.	Minor
Implementation enhancement.	Minor
Implementation change effecting method semantics.	Major
Addition or Removal of a constant, type or exception definition from the component-class specification (this cannot effect an interface).	Minor
Addition of an interface to the component-class.	Minor
Removal of an interface from the component-class.	Major

Table 5.1: Major vs Minor Version Updates

Interfaces can be added to a component-class because, as described with the component construction model in Section 4.3.1, interfaces do not define a component, they only provide a *view* of the core component functionality. As long as the semantics of the core remain the same, interfaces are independent, therefore, new interfaces can be added.

5.6 Component Invoker and Component Adaptor

Together, the Component Invoker (CI) and the Component Adaptor (CA) provide a standard request mechanism interface, comprising two services:

- Activating the component implementation.
- Transferring control, and limited request data, to the entry point.

```
extern __inline__ int ci_request( entry_pt_t ept, cap_t param, res_desc_t *ret, void *pd )
{
        int r;
        r = PdxCall( ept, param, (cap_t*)ret, pd );
        if( r != 0 ) {
                ret.type = MCS_SYSTEM_EXCEPTION;
                ret.data1 = r << 48;
        }
        return r;
}
```

Figure 5.42: ci_request

The CI is implemented as a library.

5.6.1 ci_request

`int ci_request( entry_pt_t ept, cap_t param, res_desc_t *ret, void *pd );`

An MCS request has almost identical semantics to a Mungi `PdxCall()`, making the implementation of `ci_request()` trivial, and avoiding the need for a Component Adaptor entirely. Figure 5.42 presents the complete implementation of `ci_request()`, which consists of a `PdxCall()` followed by a check to see if a system exception should be raised.

5.6.2 ci_reset

`int ci_reset();`

As the Component Invoker has no state, this function just returns zero.

5.6.3 ci_get_opt and ci_set_opt

```
int ci_get_opt( int option_id, void *optval, int *optlen );
int ci_set_opt( int option_id, void *optval, int optlen );
```

As there are currently no options defined, both functions return `MCSE_UNKNOWN_OPT`.

5.7 Component Locator

Interface objects use the Component Locator to retrieve the entry point for a method. The Component Locator is implemented as library.

5.7.1 cl_get_entry_pt and cl_set_entry_pt

```
int cl_get_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t *epr );
int cl_set_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t epr );
```

These functions provide a lookup service that maps a key `(cid,iid,mid)` to a 64 bit value, i.e. the entry point address. Rather than directly providing this service itself, the Component Locator uses a nameserver component to actually store the entries. This component must statically export the `ICLNameServer` interface.

```
interface ICLNameServer {
    entry_pt_t getname( cid_t cid, iid_t iid, mid_t mid );
    int setname( cid_t cid, iid_t iid, mid_t mid, entry_pt_t data );
};
```

As the nameserver component cannot be used to lookup its own entry points, these values are stored in the environment variables `MCS_NS_EPT_GETNAME` and `MCS_NS_EPT_SETNAME`. `ci_reset()`, which creates the nameserver interface object used by the CL, enters these entry points into the `epts` member of the interface object. Figures 5.43 and 5.44 present implementations of `cl_get_entry_pt()` and `cl_set_entry_pt()` respectively.

5.7.2 cl_get_component_class

```
cid_t cl_get_component_class( cicap_t ref );
```

`cl_get_component_class()` provides a mapping from a CICAP to its component-class identifier (CID), which must be passed to `cl_get_entry_pt()`. This function is only required if the interface object is created using the CICAP constructor, as otherwise the CID is provided by the client. Figure 5.25 shows the CICAP constructor.

```c
#include <stdlib.h>
#include "mcs_cl.h"
#include "mcs_ci.h"
#include "ins_io.h"

/* interface object for the nameserver component - created by cl_reset() */
INameServer *ins;

int cl_get_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t *epr )
{
	environment_t ev;

	/* lookup the entry point */
	*epr = ins_getname( cid, iid, mid, &ev );
	if( ev.type != MCS_NO_EXCEPTION )
		return MCSE_FAIL;

	/* if the call succeeded, by the entry wasn't found, determine whether the class,
	 * interface or method doesn't exist */
	if( *epr == NULL ) {

		/* lookup the constructor, which must exist for every legal class */
		*epr = ins_getname( cid, IID_CLASS_INTERFACE, MID_CONSTRUCTOR, &ept );
		if( ev.type != MCS_NO_EXCEPTION )
			return MCSE_FAIL;
		if( *epr == NULL )
			return MCSE_CID_NOT_EXIST;

		/* lookup MID_START, which corresponds to some method for every non-empty interface */
		*epr = ins_getname( cid, iid, MID_START, &ept );
		if( ev.type != MCS_NO_EXCEPTION )
			return MCSE_FAIL;
		if( *epr == NULL )
			return MCSE_IID_NOT_EXPT;

		/* otherwise it must be the method */
		return MCSE_MID_NOT_EXPT;
	}

	/* done */
	return 0;
}
```

Figure 5.43: cl_get_entry_pt

```
int cl_set_entry_pt( cid_t cid, iid_t iid, mid_t mid, entry_pt_t epr )
{
        environment_t ev;

        /* set the entry point */
        *epr = ins_setname( cid, iid, mid, epr, &ev );
        if( ev.type != MCS_NO_EXCEPTION )
                return MCSE_FAIL;
        return 0;
}
```

Figure 5.44: cl_set_entry_pt

```
cid_t cl_get_component_class( cicap_t cicap )
{
        objinfo_t obj;
        int r;

        /* get the object descriptor */
        r = ObjInfo( cicap.address, 0, &obj );
        if( r != 0 )
                return 0;
        return obj.userinfo;
}
```

Figure 5.45: cl_get_component_class

This, CICAP $\rightarrow$ CID, mapping is implemented using the Mungi Object Table (OT), in cooperation with the component-class. When a Mungi object is created, a corresponding object descriptor is added to the OT. This descriptor contains object metadata, including a `userinfo` field that is set by the owner, and can be accessed using the `ObjInfo()` system call. As described in Section 5.3.2, the Mungi object in which instances of a component-class are to be stored is the first capability in the protection context. Therefore, this object is created by the client registering the component-class. When creating this object, the owner must set the `userinfo` field in its descriptor to the CID. As a CICAP contains an address within this object, the Component Locator can retrieve the appropriate object descriptor using `ObjInfo()`, and access its public information, including the CID. This imposes the restriction that all instances stored within a single Mungi object must be instances of the same component-class, however, in practice this is not a problem.

Figure 5.45 presents the implementation of `cl_get_component_class()`.

5.7.3 cl_reset

```
int cl_reset();
```

`cl_reset()` must create the interface object for the nameserver component, and initialise its entry points. The component identifier for the nameserver is stored in an environment variable `MCS_NAMESERVER_CID`. Figure 5.46 presents the implementation of `cl_reset()`.

5.7.4 cl_get_opt and cl_set_opt

```
int cl_get_opt( int option_id, void *optval, int *optlen );
int cl_set_opt( int option_id, void *optval, int optlen );
```

As there are currently no options defined, both functions return `MCSE_UNKNOWN_OPT`.

```
#include <stdlib.h>
#include "mcs_cl.h"
#include "mcs_ci.h"
#include "ins_io.h"

/* interface object for the nameserver component - created by cl_reset() */
INameServer *ins;

int cl_reset()
{
        environment_t ev;
        char *strenv;
        cid_t cid;
        entry_pt_t getname, setname;

        /* retrieve the CID for the nameserver */
        strenv = getenv( MCS_NAMESERVER_CID );
        if( strenv == NULL )
                return MCSE_FAIL;
        atob( (unsigned long*)&cid, strenv, 10 );

        /* lookup the entry points */
        strenv = getenv( MCS_EPT_GETNAME );
        if( strenv == NULL )
                return MCSE_FAIL;
        atob( (unsigned long*)&getname, strenv, 16 );
        strenv = getenv( MCS_EPT_GETNAME );
        if( strenv == NULL )
                return MCSE_FAIL;
        atob( (unsigned long*)&setname, strenv, 16 );

        /* create an interface object, and manually set the entry points. This is a static
         * interface, therefore, the constructor will not try and invoke the component */
        if( ins == NULL ) {
                ins = create_constructor( MCS_NAMESERVER_CID, &ev );
                if( ins == NULL )
                        return MCSE_FAIL;
        }
        ins->epts[0] = getname;
        ins->epts[1] = setname;

        /* done */
        return 0;
}
```

Figure 5.46: cl_reset

Chapter 6

Component System Performance

Performance is the master.

Jochen Liedtke [Lie93]

6.1 Overview

MCS provides a component model that can be used to create, compose, customise, and extend system services. This chapter demonstrates that such flexibility need not sacrifice performance, by presenting micro and macro-benchmark results. A brief outline of this chapter is given below.

- **Mungi Micro-benchmarks** presents the cost of Mungi system calls used by MCS.
- **MCS Micro-benchmarks** presents the performance of MCS-RT, component-instance creation and method invocation. Costs associated with component stubs, interface objects, aggregation, and delegation are isolated.
- **'CORBA Comparison Project' Micro-benchmarks** evaluates MCS using a published suite of micro-benchmarks developed for CORBA implementations.
- **Micro-benchmark Comparison** uses the results from the previous section to compare MCS to some existing component technologies.
- **Macro-benchmarks** demonstrate that micro-benchmark performance improvements result in end-to-end performance benefits.

6.2 Environment

All Mungi/MCS results were obtained on a hardware platform developed at UNSW, featuring a 100MHz MIPS R4600 processor and 64MB of RAM on a 50MHz memory bus, which is rated at approximately 1.9 SPECint-95[1].

6.3 Mungi Micro-benchmarks

This section presents the cost of Mungi system calls, allowing a precise analysis of the results presented in the following sections. Results are the average of a large number of iterations, and therefore, represent ideal cache conditions. As there is a significant difference between the performance of the first `PdxCall()` to a protection domain, and subsequent calls, results are shown for both cases. An analysis of the performance of Mungi system calls is presented in [Voc98].

Operation	Time (μs)
Null system call	6
ObjCreate()	30
ObjDelete()	44
ObjInfo()	14
ApdGet()	13
PdxCall() - first call	887
PdxCall() - repeated call	29

Table 6.1: Mungi System Call Costs (μs)

[1] Jeroen Vochteloo [Voc98] reports that this platform exhibits performance that is roughly equivalent to a 100MHz MIPS R4600-based SGI Indy workstation rated at 1.9 SPECint-95.

6.4 MCS Micro-benchmarks

As described in Section 4.1, all component architectures provide two fundamental services: component creations and method invocation. This section presents the performance of these two services, as well as that of MCS-RT.

6.4.1 Component Invoker and Component Locator

Table 6.2 presents the performance of the Component Invoker and the Component Locator. Again, all results are the average of a large number of iterations.

Referring back to the implementations presented in Sections 5.6 and 5.7, these results are all as expected. As `ci_request()` and `cl_get_entry_point()` are respectively on the critical paths of method invocation and component creation, it is essential that they perform well. Comparing their costs to that of a `PdxCall()`, i.e. the dominant operation of both functions, it can be seen that MCS imposes little additional overhead.

Procedure	Time (μs)
ci_request()	29
ci_reset()	0
ci_get_opt()	0
ci_set_opt()	0
cl_get_entry_point()	34
cl_set_entry_point()	34
cl_get_component_class()	15
cl_reset()	40
cl_get_opt()	0
cl_set_opt()	0

Table 6.2: MCS-RT Performance (μs)

6.4.2 Component Creation

A new component-instance is created using the creation constructor (Figure 5.24) of an interface object. Table 6.3 presents the performance of a creation constructor, and identifies its major costs.

Operation	Time (μs)
Retrieve the constructor entry point (cid,0,0)	34 / 2
Invoke the constructor	32
Create a parameter buffer	30 / 0
Other	4
Total	**110 / 38**

Table 6.3: Profile of the Creation Constructor (μs)

An initial result of 100μs represented unacceptable overhead. Therefore, two optimisations were conceived and implemented.

- A per-process entry point cache was implemented in `cl_get_entry_pt()`. Repeated constructions, therefore, do not invoke the name-service component, avoiding the overhead of an additional `ci_request()`. Cache entries must be timed-out, so that a process does not have to restart to use a new minor version. As shown in Table 6.3, this optimisation reduces the cost of a (repeated) component creation by 32μs.

- Originally, each interface object created its own Mungi object to use as a parameter buffer, to protect itself from other (buggy) code. Unfortunately, as this protection incurs a 30μs penalty, it must be reconsidered. Therefore, the creation constructor is modified to accept a parameter buffer from the client.

  ```
  IO_t *io_create_constructor( cid_t cid, cap_t *pb, int size, env_t *ev );
  ```

  Clients may pass a parameter buffer to the interface object via the `pb` argument. The constructor validates that the `size` of this buffer is sufficient, and if so, uses it as its parameter buffer. If `size` is zero, then the constructor creates a new Mungi object to use as a parameter buffer. Essentially, responsibility for protecting parameter buffers is passed to the client, reducing the cost of creation by 30μs.

These two optimisation reduce the fast-case component construction cost to 38μs, which represents an acceptable overhead. This cost is dominated by the `ci_request()` to the component-class constructor. The remaining time is consumed initialising the instance header, creating an owner CICAP, and initialising the interface object.

Interface objects provide an additional constructor, the CICAP constructor, which creates a new interface object that is connected to an existing component-instance. As the CICAP constructor also created a new Mungi object to use as a parameter buffer, it is modified in a similar way to the creation constructor.

```
IO_t *io_cicap_constructor( cicap_t cicap, cap_t *pb, int size, env_t *ev );
```

A CICAP constructor incurs a cost of 3μs, representing the time taken to create and initialising a new interface object.

6.4.3 Method Invocation

This section evaluates the performance of method invocation using four methods. These four methods, shown in interface `IBenchMethodInvocation` below, represent the four permutations of direct vs. buffered parameter transfer, and direct vs. buffered return value transfer. Section 6.5 provides a more rigorous investigation of the effects of different method signatures.

```
typedef struct {
    long a, b, c, d;
} big_return_t;

interface IBenchMethodInvocation {
    void dd();
    big_return_t dl();
    void ld( long a, long b, long c, long d );
    big_return_t ll( long a, long b, long c, long d );
};
```

That `dl()` exhibits the same performance as `dd()`, indicates that there is no inherent penalty associated with using a parameter buffer. `ld()` and `ll()` incur an additional 1μs penalty since they must explicilty copy the arguments into the parameter buffer. The different performance of `dl()` and `ld()` indicate that it is the extra instructions, rather than the memory operations, that create the overhead. This suggests that if a large number of parameters are to be passed, a structure should be used. Table 6.4 also confirms that there is no overhead associated with repeated calls to aggregated and delegated interfaces, and that the extra code required in the component stubs of delegatable interfaces does not create a significant overhead.

Method	Time (μs)
dd()	32
dl()	32
ld()	33
ll()	33
Aggregated dd() (repeated call)	32
Delegated dd() (repeated call)	32
Delegatable dd() (repeated call)	32

Table 6.4: Method Invocation Performance (Repeated Calls) (μs)

Table 6.5 presents the performance of the *first* method call on a new interface object.

Constructor	Method Called	Time (μs)
Creation constructor	dd()	32
Creation constructor	dl()	32
Creation constructor	Aggregated dd()	62
Creation constructor	Delegated dd()	92
CICAP constructor	dd()	32
CICAP constructor	dl()	33
CICAP constructor	Aggregated dd()	62
CICAP constructor	Delegated dd()	92

Table 6.5: Method Invocation Performance (First Call) (μs)

Interface objects for aggregated interfaces must call the outer-component-instance to retrieve a CICAP for the inner-component-instance, therefore, two `ci_request()` operations are required, essentially doubling their cost. As described in Section 5.5.5, delegated interfaces require three `ci_request()` operations for the first call. Table 6.5 also shows that performance is independent of the constructor used to create the interface object.

6.5 CORBA Comparison Project Micro-benchmarks

The *CORBA Comparison Project* [Bub99] is an attempt to provide a suite of simple micro-benchmarks to evaluate the performance of a CORBA implementation. Although MCS is not an ORB, it does provide the same basic functionality, which can be evaluated using these benchmarks. There are three motivations for applying these benchmarks.

- Standard, published, benchmarks provide more credible results.
- Standard benchmarks provide results that can be compared to other systems.
- These benchmarks evaluate features not evaluated above, e.g. scalability.

Each benchmark is described as it is performed below. These descriptions have been adapted from [Bub99].

6.5.1 Measurement Verification

A benchmark, *CALL*, is used to verify that there is no significant relation between the method invocation pattern and the method invocation performance. Such a relation would invalidate the results of other benchmarks in the suite which have oversimplified invocation patterns. CALL employs five invocation strategies.

- *As Thru, Rev Thru* - a single component-instance is called 10 times, repeated for 1000 instances in their creation order and reverse creation order.
- *As Proxy, Rev Proxy* - 1000 component-instances are called once in their creation order and reverse creation order. This is repeated 10 times.
- *Random* - 10000 calls are made to 1000, randomly selected, component-instances.

Results from the CALL benchmark, shown in Figure 6.1, indicate that there is no significant relation between method invocation pattern and invocation times. Therefore, the results of the benchmarks in this suite are valid for different invocation patterns.

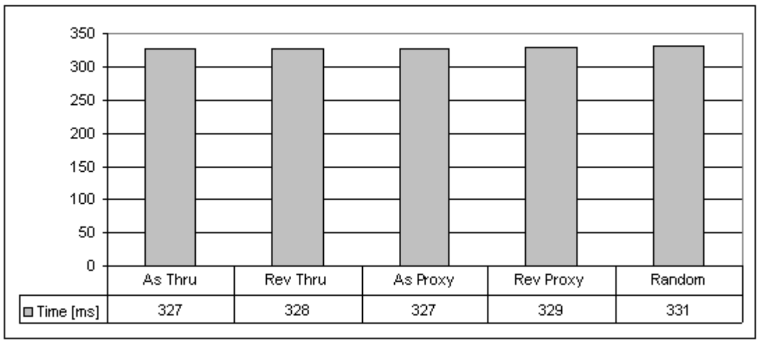


Figure 6.1: CALL Results - 10000 calls

6.5.2 Dispatcher Performance

CORBA implementations involve a *dispatcher*, which is used to locate the appropriate object implementation and entry point for a method invocation. Three benchmarks, *IDL.NAMES, IDL.LONG, and IDL.DEEP*, are used to discover relationships between the structural complexity of an interface and invocation performance. As MCS directly invokes specific component stubs, rather than requiring a message loop, and caches entry points in interface objects, these benchmarks do not apply.

6.5.3 Performance

IDL.ENCAP compares the times needed to pass data individually, in an array, and in a structure. A single component-instance is created, and each of the methods of the `IEncap` interface invoked 1000 times. Figure 6.2 shows the average time per invocation.

```
typedef long arr_t [100];
typedef struct {
    long x0, x1, ..., x99;
} str_t;

interface IEncap {
    void no_encap( long x0, long x1, ..., long x99 );
    void arr_encap( arr_t a );
    void str_encap( str_t a );
};
```

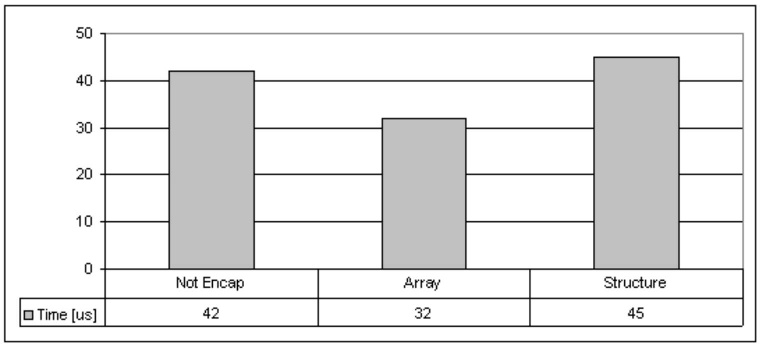


Figure 6.2: IDL.ENCAP Results - 1 call

IDL.ENCAP shows that MCS passes arrays much faster than structures or individual parameters. This is because MCS makes use of the single-address-space, and passes arrays as pointers. Therefore, only one word needs to be passed in the parameter buffer.

THROUGH.IN determines how fast an ORB transfers data. Each basic data type is sent individually, and in a 1kB array. For example, for the `short` basic type, the following three methods are called.

```
typedef short short_sarr_t [1024/sizeof(short)];
typedef short short_larr_t [4096/sizeof(short)];

interface IThrough {
    long in_short( short a );
    long ar_short( short_sarr_y a );
    long ar_short( short_larr_y a );
    . . .
};
```

Each method is invoked 1000 times on 5 objects. The average performance of each operation is shown in Figures 6.3 and 6.4.

THROUGH.IN shows that there is no relation between the size of the array being passed and the method invocation time. This is because, as stated above, arrays in MCS are passed as pointers. THROUGH.IN also shows that there is no relation between data type and invocation time; the maximum deviation is 3%, most likely due to cache effects.

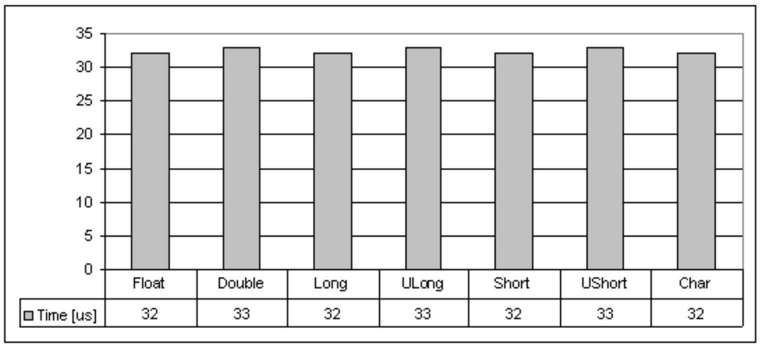


Figure 6.3: THROUGH.IN Basic Data Types - 1 call

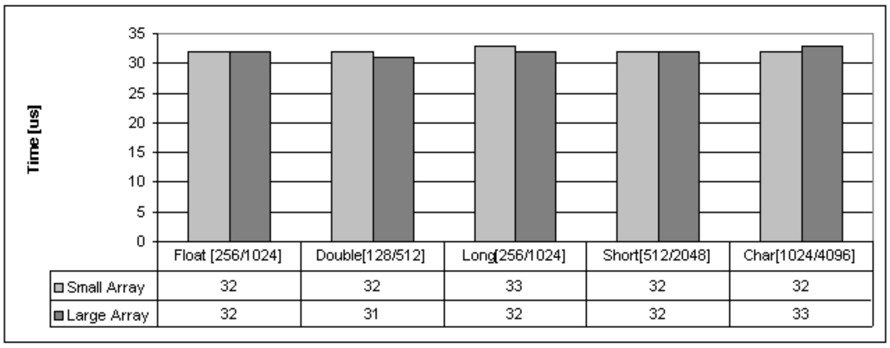


Figure 6.4: THROUGH.IN Arrays - 1 call

6.5.4 Multi-Threading

MT.CLIENT tests the multi-threading capabilities of clients. A client creates five threads, each of which creates an instance of a *different component-class*, and invokes a method that takes 10 seconds to complete. A multithreaded client architecture will take 50 seconds to complete the benchmark, while a serialised client architecture will require 250 seconds.

MCS completes in 50 seconds.

MT.CONN tests the multi-threading capabilities of clients when accessing the same component-class. This benchmark is similar to MT.CLIENT, however, all client threads create an instance of the same component-class and invokes the method five times. An architecture that allows the requests to be handled in parallel will take 50 seconds to complete, a serialised architecture will take 250 seconds.

Again, MCS completes in 50 seconds.

6.5.5 Scalability

Scalability is measured using the *PROXY* benchmark. PROXY measures the time taken to create a new component-instance and invoke two methods on it. This is measured for 1000, 2000 and 3000 instances already existing in the component-class. Figure 6.5 shows the results for the PROXY benchmark. These results show that there is no relation between performance and the number of existing instances. They also confirm the result from Section 6.4.3, that there is no difference between the time taken by the first and subsequent invocations for standard methods.

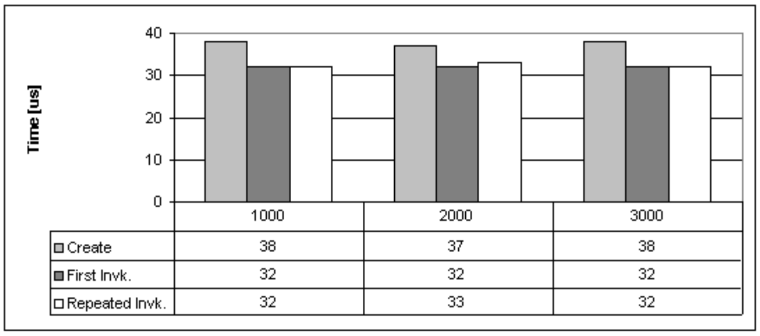


Figure 6.5: PROXY Results

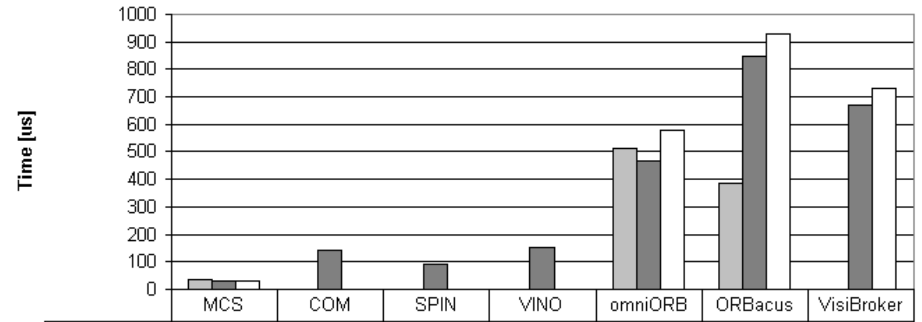
6.6 Micro-benchmark Comparisons

This section uses the results from the previous section to compare MCS to some existing component technologies. Where available, three results have been presented for each system: the time to create a new instance/object, to invoke a method with no arguments, and to invoke a method passing an array of 256 longs. Unfortunately, due to time and availability constraints, the results presented are from a variety of platforms. Therefore, Figure 6.6 presents the raw result, and Figure 6.7 normalises the results according to the platform's SPECint-95 rating. Whilst this is by no means a precision result, the performance differences involved are far greater than the error. SPEC-95 ratings were obtained from `http://www.specbench.org/osg/cpu95/results/cpu95.html`.

- Mungi's platform was described in Section 6.2, its SPEC-95 rating is 1.9.
- COM [COM95] results were obtained on a PentiumII 400MHz PC with 128MB of RAM, running Windows NT Workstation 4.0 SP3. Its SPEC-95 rating is 8.56. COM

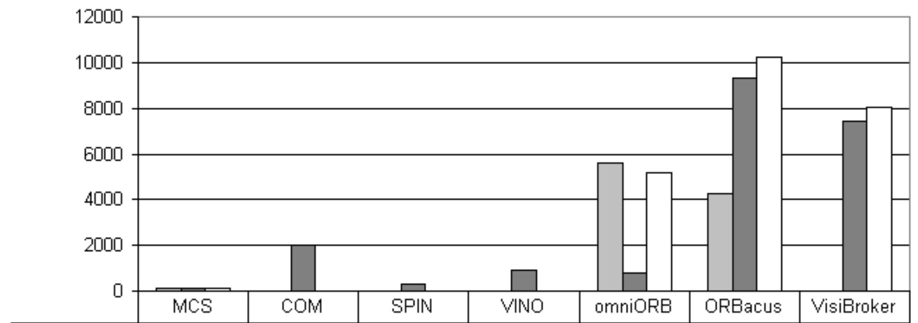
results are for out-of-process COM classes, i.e. equivalent to an MCS component.

- SPIN results are from [BSP+95], measured on an Alpha 133MHz AXP 3000/400. its SPEC-95 rating is 2.1.
- VINO results are from [SESS96], measured on a 120MHz PentiumI with 32MB of 60ns EDO DRAM. Its SPEC-95 rating is 3.6.
- All ORBs were evaluated by [OMG99a] on a 200MHz PentiumI with 64MB of RAM, running Windows NT Workstation 4.0. Its SPEC-95 rating is 6.7.



	MCS	COM	SPIN	VINO	omniORB	ORBacus	VisiBroker
Create	38				512	385	
Invoke	32	142	89	150	469	846	672
Invoke Array	32				580	931	732

Figure 6.6: Micro-benchmark Comparison



	MCS	COM	SPIN	VINO	omniORB	ORBacus	VisiBroker
Create	118				5622	4240	
Invoke	100	1993	307	885	768	9319	7403
Invoke Array	100				5167	10255	8063

Figure 6.7: Normalised Micro-benchmark Comparison

Figure 6.7 shows that MCS clearly outperforms the other component architectures, especially in bulk data sharing. The micro-benchmarks presented indicate three reasons for these performance advantages.

- Mungi provides superior performance to the other platforms used, especially in the critical area of cross-domain call overhead.
- MCS does not impose a significant overhead.
- A SASOS allows data to be passed by reference, avoiding marshalling costs.

6.7 Macro-benchmarks

Macro-benchmarks demonstrate that micro-benchmark performance improvements result in end-to-end performance benefits. This section presents two macro-benchmarks, the OO1 benchmark [CS92] that simulates a 'typical' operations on an object-oriented database, and the Andrew Benchmark [HKM$^+$88] that emulates a software development workload on a file system.

6.7.1 OO1

The OO1 benchmark [CS92] is designed to simulate typical operations on an object-oriented database. The data stored in the database should be accessible to the user only via the defined interface functions, and so is a natural situation for components. A database component is constructed with appropriate access methods, and an instance is created to hold the database information. The database used consisted of 20,000 parts. Four operations are performed, lookup, forward traversal, reverse traversal and insert.

Table 6.6 shows the results from the OO1 benchmark running on Irix, Linux and Mungi. Irix and Linux use an RPC trusted path, with a client/server software construction model. Irix and Linux results were obtained by [Voc98] on a 100MHz R4600-based SGI Indy workstation with 64MB of RAM. All times are in milliseconds. *L.* is lookup, *F.T.* is forward traversal, *R.T.* is reverse traversal and *I.* is insert.

Mungi components outperform Linux by a factor of eleven and Irix by a factor of twenty-one. Total execution time for OO1 can be separated in three categories.

- **Executing application code**, which includes the client logic and the database operations. As application code is the same for all three systems, and contains no

System	L.	F.T.	R.T.	I.	Time (ms)
Irix	949	1,409	1,411	203	3,972
Linux	344	467	461	842	2,114
Mungi	88.8	27.1	33.0	38.9	187.8

Table 6.6: OO1 benchmark results (in ms)

system calls, this cost should be constant across systems. By placing the database in the same protection domain as the client, and re-executing the benchmark, it was confirmed that the cost was constant, at 28μs. As this is over three orders of magnitude less than the total execution time, application code overhead is irrelevant.

- **Cross-domain call overhead**. Each operation results in (at least one) cross-domain call. Table 6.7 compares the cost of a cross-domain call on each system.

Mechanism	Time (μs)
Mungi PdxCall	30
Linux RPC	160
Irix RPC	450

Table 6.7: Cross-domain call performance

As 5869 operations are performed by the benchmark, the total cross-domain call overhead (X-Dom) can be calculated. These values are presented in Table 6.8.

System	X-Dom. (ms)	Other (ms)	Total (ms)
Irix	2641	1331	3972
Linux	939	1175	2114
Mungi	176	11.8	187.8

Table 6.8: Division of overhead

- **Model overhead** is the remaining difference between the three systems. Irix and Linux both use a client/server model in a multiple-address-space environment, while Mungi uses a component model in a single-address-space environment. Model overhead is primarily parameter marshalling and message dispatch, e.g. a message loop.

As Irix and Linux use the same software construction model, which does not involve system intervention, it is expected that both should incur a similar model overhead. Section 6.4 shows that, for Mungi, the model overhead is 6.7% ($\frac{2}{30}$) of the cross-domain cost. For the 176ms cross-domain cost reported in Table 6.8, this corresponds to 11.8ms, which is exactly the value in the **Other** column. This confirms that cross-domain call latency and model overhead are indeed the differentiating factors for the OO1 benchmark. Therefore, the **Other** column of Table 6.8 can justifiably be used as reporting the model overhead. Irix and Linux have a similar model overhead, though the 11.7% ($\frac{1331-1175}{1331}$) difference is greater than expected.

OO1 demonstrates that the excellent micro-benchmark performance of MCS results in concrete, end-to-end, performance gains. It also confirms the conclusion of the previous section, that these performance gains are due to superior system performance (i.e. `PdxCall()`) combined with a lightweight component model. Figure 6.8 shows that *both* provide significant performance benefits.

6.7.2 An Extensible File System

Although MCS clearly performs well compared to existing component, and RPC-based, architectures, system services are predominantly implemented in a monolithic fashion. Therefore, to show that MCS can be employed at the systems level, an MCS based system service must demonstrate performance comparable to an equivalent monolithic service. Given comparable performance, MCS is clearly a preferable software technology.

This section compares the performance of a component-based file system, in which files and directories are implemented as MCS components, and the Linux `ext2` file system. The systems are compared using the Andrew Benchmark [HKM+88], which emulate a 'typical' software development workload, by performing operations in five phases.

- Creates 20 subdirectories recursively.
- Copies a source tree containing 70 files into the created tree.
- Examines the status of every file in the tree.
- Examines every byte of data in all files.
- Compiles and links the program.

As `make` has not yet been ported to Mungi, the fifth phase has been omitted. Furthermore, this section is interested in comparing the software overhead, rather than the I/O performance, of the two systems. Therefore, the benchmark is performed entirely in memory. On Mungi, this is achieved by executing the benchmark in a non-persistent region of the single-address-space, while on Linux it is achieved by executing the benchmark on a RAM disk, created using the commands shown in Figure 6.8.

```
[ant@parrot /] mke2fs -v -m 0 /dev/ram0 4096
mke2fs 1.10, 24-Apr-97 for EXT2 FS 0.5b, 95/08/09
Linux ext2 filesystem format
Filesystem label=
1024 inodes, 4096 blocks
0 blocks (0.00%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
1024 inodes per group

Writing inode tables: done
Writing superblocks and fullsystem accounting information: done
[ant@parrot /] mount -t ext2 /dev/ram0 /mnt/ram
```

Figure 6.8: Creating a RAM Disk on Linux

Table 6.9 shows the results from ten executions of the benchmark. Linux results were obtained on a Pentium 200MHz PC with 32MB of RAM, running Linux kernel 2.0.34. Rather than merely showing comparable results, MCS performs a factor of six faster than Linux. This performance is due to two key factors.

- As the component-based file system is implemented as a stand-alone service, rather than in a large monolithic operating system, its design and implementation is much simpler.

- MCS provides component-based simplicity without imposing a significant overhead.

- A Mungi file system can use the single-address-space to reduce the number of cross-domain calls required.

Phase	Mungi (ms)	Linux (ms)
Create	97	260
Copy	223	730
Status	45	1470
Read	307	1510
Total	672	3970

Table 6.9: Andrew Benchmark, Mungi vs Linux (ms)

6.8 Conclusion

The Andrew benchmark demonstrates that MCS can provide performance comparable to that of existing, monolithic, system services. Micro-benchmarks showed that, for repeated calls, delegation does not impose any additional overhead. Combined, these results indicate the validity of the claim that *'MCS provides a component model that can be used to create, compose, customise, and extend system services.'*

Micro-benchmark results from Sections 6.3, 6.4 and 6.5, indicate that MCS performance is derived from three key factors: the superior performance exhibited by Mungi, a lightweight component-model, and use of the single-address-space to allow bulk data sharing. OO1 confirmed the importance of *both* the excellent cross-domain performance of Mungi, and the lightweight component model provided by MCS for end-to-end performance gains.

Part II

Access Control

Chapter 7

Access Control in Component Systems

Current security efforts suffer from the flawed assumption that adequate security can be provided in applications with the existing security mechanisms of mainstream operating systems. In reality, the need for secure operating systems is growing due to substantial increases in connectivity and data sharing.

P. Loscocco et. al., 1995 [LSM+95]

7.1 Overview

Security protects a system from inappropriate use, with what constitutes 'inappropriate' being defined by a *security policy* [AGS83]. At the core of any security policy are restrictions on *who* can access *what* in *what way* [GB97b]. The specification and enforcement of such restrictions is called *access control* and is the subject of this chapter.

Access control is required for systems to be able to enforce effective security policies; it is becoming increasingly vital due to three key factors.

- Increased connectivity, driven by the popularity of the Internet, means that there is no such thing as a single user system.
- Increased data sharing, driven by the wide-spread integration of computer systems into core buisness *processes*, is creating a need for more sophisticated protection.

- Software is becoming more fine-grained, i.e. composed of a number of modules each performing a particular task. Modules come from a variety of sources (this is especially true of extensible software) and may not be entirely trusted.

Despite the increasing importance of access control, the mechanisms of existing component systems remain primitive as Java continues to demonstrate [Lad97, MF97].

This chapter introduces a new access control mechanism based on a modified version of *Domain and Type Enforcement*(DTE) [BK85], combined with capability based protection. This mechanism can be used to implement a wide variety of security policies at various administrative levels. Section 7.2 presents the relevant theory, Section 7.3 describes the mechanism, Section 7.4 contains examples of implementing security policies, and Section 7.5 presents a formal access control model.

7.2 Access Control Theory

This section presents the theory required for an appreciation of the access control mechanism presented, as well as a feel for the current state of play.

7.2.1 Glossary

While the gratuitous use of jargon serves only to obfuscate, clarity requires the use of well defined terms. The following terms are used throughout this chapter and the next.

- **Object** A passive entity that is operated upon, i.e. it is the target of an operation. O denotes the set of all objects in a system.
- **Operation** An action performed on a passive entity by an active entity. M denotes the set of all operations in a system.
- **Subject** An active entity, e.g. a thread, which performs operations on objects. S denotes the set of all subjects in a system.
- **User** Any user who interacts directly with a computer system and thus has threads of control executing some code on their behalf. U denotes the set of all users.

7.2.2 Access Control Matrix

Access control matricies were introduced in Lampson's seminal paper *Protection* [Lam74]. An access control matrix lists all the subjects in a system along the vertical axis, all the objects in the system along the horizontal axis, and its cells contain the access rights of subjects to objects. Denoting the access control matrix A, A_{ij} lists the operations that S_i may perform on O_j. As most systems allow threads to perform operations on other threads, e.g. signals, subjects often appear along both axis. Figure 7.1 shows an example access control matrix for a Unix style environment.

	O_1	O_2	S_1	S_2
S_1	read		terminate	wait, signal, send
S_2		read, write, execute	wait, signal, terminate	
S_3				wait, signal, receive
S_4	execute	write	control	

Figure 7.1: Portion of an Access Control Matrix

Access control matricies express access control as a function: $S \times O \rightarrow P(M)$. All existing access control policies can be expressed using this function, and so almost all commercial operating systems provide a matrix-based protection mechanism, usually access control lists (ACLs). For practical reasons, ACLs are normally implemented using two functions: $S \rightarrow U$, and $U \times O \rightarrow P(M)$ [TOP99].

Due to the immense number of subjects and objects that exist in most systems, the complete matrix is too large to store explicitly. Fortunately, it is very sparse and can therefore be substantially compressed by removing empty entries. Compression down columns produces *access control lists* (ACLs), which define the subjects that may access a given object. Every object is associated with exactly one ACL. Compression across rows produces *capability lists*, which define the objects that a subject may access. Every subject is associated with exactly one capability list, often called a *protection domain.*

7.2.3 Role-based Access Control

Role-based access control (RBAC) [San95] is an extension of standard matrix-based protection. RBAC is based on the observation that security policies are usually not concerned with users per se, but rather their role in an organisation. For example, in a hospital, the security policy permits nurses in the cardiology ward access to certain files; it is not concerned with the identity of the nurse. RBAC introduces a new set, $ROLES$, and is based on three functions. A boolean set is also defined.

$ROLES$; the set of all roles in the system.
$B = \{\ \textsc{True}, \textsc{False}\}$

$\textsc{User}$: $S \rightarrow U$
$\textsc{LegalRoles}$: $U \rightarrow P(ROLES)$
$\textsc{ValidateAccess}$: $ROLES \times O \rightarrow P(M)$

These three mappings are maintained by a security administrator. When subject s' performs operation m' on object o', the system validates the access as follows:

$$\begin{array}{|l}
\textsc{ValidateAccess} : S \times M \times O \rightarrow B \\
\hline
u' = \textsc{User}(s') \\
rs' = \textsc{LegalRoles}(u') \\
Allowed = \exists r \in rs' \ \bullet \ m' \in \textsc{ValidateAccess}(r', o')
\end{array}$$

Note that the '•' symbol reads as 'such that'.

7.2.4 Domain and Type Enforcement

Domain and type enforcement (DTE) [BSS+95a, BSS+95b, BK85], was introduced as a more flexible alternative to military style, lattice-based [Den76], protection, for supporting system wide security policies. DTE associates each subject with a label called a *domain*, and each object with a label called a *type*. Domains and types are abstract labels that do not necessarily relate to system constructs such as protection domains or object classes. DTE introduces two new sets, D and T, and is based on three functions:

D; the set of all domains.
T; the set of all types.

$\textsc{Domain}$: $S \rightarrow D$
$\textsc{Type}$: $O \rightarrow T$
$\textsc{ValidateAccess}$: $D \times T \rightarrow P(M)$

These three mappings are maintained by a security administrator, i.e. system users cannot alter the types of the objects they own. When subject s' performs operation m' on object o', the system validates the access as follows:

$$
\begin{array}{|l}
\textsc{SysValidateAccess} : S \times M \times O \to B \\
\hline
d' = \textsc{Domain}(s') \\
t' = \textsc{Type}(o') \\
\mathit{Allowed} = m' \in \textsc{ValidateAccess}(d', t')
\end{array}
$$

7.2.5 Context Sensitivity

Context-sensitive mechanisms [Jae98] allow permissions to be specified relative to a context, i.e. the environment in which a module is executing. As modules are used in a variety of contexts, least privilege requires that permissions are context sensitive.

Access control mechanisms flexible enough to support context sensitive permission assignment are not used in practice. Existing approaches assign permissions based solely on the module identity and its author. Mechanisms for enforcement of limited rights for modules has been investigated in language-specific systems [GMPS97, OLW98], however, the effectiveness of these mechanisms is limited because they depend on a large trusted computing base (TCB), are language-specific, depend on unproven language separation features and do not control resource utilisation [Jae98].

Context-sensitive access control has received an increasing amount of attention recently, especially in role-based access control (RBAC) models [GI97, LS97]. These models parameterise modules, so their permissions can be derived from the runtime context.

7.2.6 Mandatory and Discretionary Access Control

Apart from on a small number of research systems, most notably SPIN [GB97a], that provide mechanisms for access control at various administrative levels, security policies are enforced using *one* of two techniques.

- *Mandatory access control* (MAC) mechanisms are those where the assignment of security attributes is controlled by a central authority, e.g. a security administrator. The “Orange Book” [DoD85] provides the original definition of MAC, which is tightly bound to the U.S. Department of Defense's multi-level security policy. The definition used here is a generalisation that has become popular in the access control literature [SCC97].

- *Discretionary access control* (DAC), on the other hand, allows users to define security attributes. All mainstream operating systems, e.g. Unix [LMKQ89] and Windows NT [Cus93], use discretionary access control. In these systems, the owner of a resource is responsible for the access control list (ACL) of that resource.

How a security policy should be enforced depends on who defines the policy. System-wide policies, such as Denning's Lattice [Den76] and Chinese Wall [San93], are defined by the organisation, and only the security administrators are trusted to maintain it. Therefore, system policies require mandatory access control. User policies, such as who can access the files in your home directory, are defined independently by each user, who should not trust others to maintain their policy. Therefore, user policies require discretionary access control. Essentially, for effective security, only those who define a policy should be responsible for maintaining it, i.e. apply the principle of least privilege.

It was previously stated that all existing access control policies can be expressed using the access matrix function: $S \times O \rightarrow P(M)$. This has led many operating system developers to claim that their system provides support for arbitrary security policies, because they implement a matrix-based protection, i.e. full ACLs or capabilities. Invariably however, such systems only provide discretionary access control mechanisms. This means they cannot enforce system-wide security policies, because such mechanisms cannot defend against careless or malicious users. These systems can achieve any legal state of a given policy, however they cannot prevent the system from entering an illegal state.

Although the above discussion focussed on system and user policies, in general, security policies can exist at any level. For a component environment, three essential levels are identified, *system* policies, *software* policies and *user* policies. Software policies refer to the context-sensitive policies applied to fine-grained modules, which are especially important in extensible systems. A general-purpose operating system must at least provide support for policies at these levels, and allow such policies to coexist. Therefore, they require mechanisms for enforcing *both* mandatory and discretionary access control.

7.3 Access Control Mechanism

This section describes the access control model. The mechanisms for mandatory and discretionary access control are discussed separately, and combined in Section 7.3.3.

7.3.1 Mandatory Access Control

This section presents a new mandatory access control model based on domain and type enforcement (DTE). It focuses on two new features, introduced below, providing support for extensible environments. These represent the principle contribution of this chapter.

- Adding a 'component' entity, and its associated operations, to the protection model.
- Allowing subjects to assign security attributes, i.e. domains and types, in a system controlled manner. This allows the specification of context sensitive software policies, which is essential for maintaining least privilege in extensible software.

Entities, Operations and Validations

In the original DTE model, which forms a subset of the model described here, subjects perform operations on objects. Subjects are threads of control, and are associated with a domain (hereafter referred to as an *m-domain* for clarity). Objects are memory segments, and are associated with a type (*m-type*). Four operations, read, write, execute, and destroy, are defined. No 'create' operation is defined, as non-existent objects do not have security attributes. An access in DTE is validated by obtaining the m-domain of the subject, the m-type of the object, and verifying that the operation is allowed. For any given m-domain and m-type, the operations allowed are given by the function:

$$\textsc{ValidateAccess}\colon D \times T \rightarrow P(M)$$

It is helpful to visualise this function in matrix form, with the m-domains listed along the vertical axis, the m-types listed long the horizontal axis, and the cells containing the operations allowed. Figure 7.2 presents an example DTE access matrix.

	T_1	T_2	T_3	T_4
D_1	read,write, execute		read, execute	read
D_2		read,write, execute		read, write
D_3			read,write, execute	
D_4	read, destroy			read,write, execute

Figure 7.2: DTE Access Matrix

Components introduce a new operation, called *invoke*. From an access control perspective, this operation takes the form:

$\textsc{InvokeMethod}(instance, method)$

An invoke operation, if validated successfully, allows the m-domain associated with a subject to be changed for the duration of a method call. Invoke can therefore be viewed as the mandatory analogue of a protected procedure call [DVH66]. The m-domain in which the method is executed is defined by the component-instance used. Therefore, a component-instance is associated with both an m-domain and an m-type. The m-domain specifies the permissions with which its methods execute, and the m-type specifies which m-domains may invoke methods on the instance. An example situation is presented in Figure 7.3. Here, subjects in m-domain `X` may invoke methods on the component-instance, as its m-type is available to the m-domain. When a method is invoked, the calling subject is transported to m-domain `Y` for the duration of the method call.

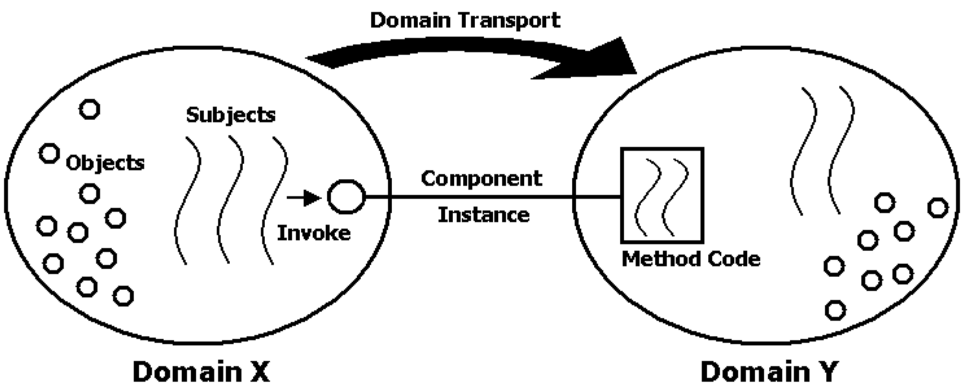


Figure 7.3: Invoke Operation Domain Switch

Component-instances need to be characterised to determine how they should be modeled in the system. Tables 7.1 and 7.2 respectively show the access control properties associated with component-instances, and how they participate in access operations.

Entity	**Domain**	**Type**
Subject	$\surd$	$\times$
Object	$\times$	$\surd$
Component-Instance	$\surd$	$\surd$

Table 7.1: Entity Properties

Table 7.1 shows that component-instances cannot be classified as either subjects or objects. This is not too surprising, as a feature of component, and object-oriented, programming is the vanishing distinction between code and data. Rather than modify one of the existing constructs, a new entity class is created:

I; the set of all component-instances in a system.

From a mandatory access control perspective, a component-instance is simply a unique identifier, associated with an m-domain and an m-type, that allows the m-domain of a subject to be temporarily changed. In particular, mandatory access control does not contain knowledge of, or influence, the component model.

Table 7.2 shows that component-instances always behave as passive entities, i.e. they are always the target of an operation. Furthermore, it also shows that the operation can be used to determine whether the passive entity is an object or a component-instance.

Again, it is helpful to visualise the situation in matrix form. As instances are passive entities, they are listed on the horizontal axis with objects. The cells of instance columns contain two items of information, the operations allowed (invoke or nothing) and the m-domain in which the instance's methods execute. Figure 7.4 contains an example matrix.

The access control model now contains two passive entity types, objects and component-instances. To assist the expression of the model formally, a new set representing all passive

Active Entity	**Operation**	**Passive Entity**
Subjects	read, write, execute, destroy	Objects
Subjects	invoke	Component-Instances

Table 7.2: Access Operations

	T_1	T_2	T_3	T_4	I_1	I_2	I_3
D_1	r w x -	- - - -	r - x -	- - - -	- - - i, D_3	- - - -	- - - -
D_2	- - - -	r w x -	- - - -	r w - -	- - - i, D_3	- - - -	- - - -
D_3	- - - -	- - - -	r w x -	- - - -	- - - i, D_3	- - - -	- - - -
D_4	r - - -	- - - -	- - - -	r w x -	- - - i, D_3	- - - i, D_1	- - - i, D_3

Figure 7.4: DTE Access Matrix

entities in the system is created.

$\Phi = O \cup I$; the set of all passive entities in the system.

An access now consists of a subject s', an operation m' and a passive entity ϕ'. To obtain the type of the passive entity, the system has to be able to determine whether it is an object or a component-instance. Table 7.2 showed that this could be determined from the operation. Therefore, the standard DTE functions must be modified to use the new set Φ and to make the operation available to the Type function.

Function	**Standard DTE**	**Component DTE**
Domain	$S \to D$	$S \to D$
Type	$O \to T$	$\Phi \times M \to T$
ValidateAccess	$D \times T \to P(M)$	$D \times T \to P(M)$

When subject s' performs operation m' on passive entity ϕ', it is validated by the process:

$\textsc{DteValidate} : (S, M, \Phi) \to B$

$d = \textsc{Domain}(s')$

$t = \textsc{Type}(\phi', m')$

$Return = m' \in \textsc{ValidateAccess}(d, t)$

Managing Domains and Types

This section describes how subjects and component-instances are associated with m-domains, and how objects and component-instances are associated with m-types. Subjects are created in two ways, by a thread create system call, and by a method invocation on a component-instance. These two cases are discussed separately below.

- Non-component subjects have their m-domain set by their parent as an argument to the thread create system call. A system function:

  $\textsc{LegalDomains} : D \rightarrow P(D)$

  defines the m-domains a subject in a given m-domain can legally assign.

  When a subject, s_p, attempts to create another subject, s_c, in m-domain d_c, the system validates the argument as follows:

  $d_p = \textsc{Domain}(s_p)$
  $\mathit{Allowed} = d_c \in \textsc{LegalDomains}(d_p)$

  The first subject of a user session cannot obtain its m-domain this way. It is assumed that the m-domain of this subject will be obtained from a user profile following a successful authentication, and set by a login manager.

- Objects have their m-type set by their creator as an argument to the object create system call. A system function:

  $\textsc{LegalTypes} : D \rightarrow P(T)$

  defines the m-types a subject in a given m-domain can legally assign.

  When a subject, s', attempts to create an object, o', with type t', the system validates the argument as follows:

  $d' = \textsc{Domain}(s')$
  $\mathit{Allowed} = t' \in \textsc{LegalTypes}(d')$

- Component-instances are not created by a system call, they are created by a constructor which returns a unique name for the instance to the client. As this occurs without system intervention, the m-domain and m-type of the instance cannot be validated on creation, and hence must be validated on-demand. To perform this lazy validation, the kernel must implement two functions.

  $\textsc{InstanceCreator} : I \rightarrow D_c;$

supplies the m-domain of the subject who created an instance, and:

$$\text{InstanceDomain} : I \rightarrow D_i;$$

supplies the m-domain associated with an instance. This is the m-domain in which methods invoked on the component-instance execute, and is defined by the creator of the component-instance. When a subject, s', invokes a method on an instance, ϕ', the system validates the access as follows:

$$\begin{aligned}
d_c &= \text{InstanceCreator}(\phi') \\
t_i &= \text{Type}(\phi', \texttt{invoke}) \\
&Allowed_1 = t_i \in \text{LegalTypes}(d_c) \\
&Allowed_2 = \text{DteValidate}(s', \texttt{invoke},\ \phi') \\
&d_i = \text{InstanceDomain}(\phi') \\
&Allowed_3 = d_i \in \text{LegalDomains}(d_c) \\
&Allowed = Allowed_1 \wedge Allowed_2 \wedge Allowed_3
\end{aligned}$$

The above validation comprises three sub-validations, all of which must succeed. The first validation, $Allowed_1$, ensures that the instance's m-type is legal, i.e. the creator of the instance has permission to specify the given m-type. The second validation, $Allowed_2$, is the standard DTE validation from the previous section. This validates that the subject may invoke methods on the component-instance. The third validation, $Allowed_3$, ensures that the instance's m-domain is legal, i.e. the creator of the instance has permission to specify the given m-domain.

If a method invocation is validated successfully, the method executes in the m-domain d_i, i.e. the m-domain of the component-instance.

Discussion

The access control model described is DTE with two extensions, a 'component' entity and an associated `invoke` operation, and the ability for users to assign m-domains and m-types in a system controlled way.

The `invoke` operation allows the m-domain associated with a subject to be changed for the duration of a method call, and the component-instance entity is the means of controlling this change. These features allow extensibility to be expressed in the system-wide policy, and are thus essential for *system* extensibility.

Component-instances provide the context data for the component implementation, i.e. they represent the context the module is being used in. By using component-instances as units of access control, this model is inherently context sensitive. Allowing the creator to specify the m-domain and m-type of the instance, within constraints, adds further context sensitivity since the creator can use state information when determining these values.

User specification of m-domains and m-types allows services to impose additional restrictions, and thereby implement their policies. Although users set security attributes, the values are controlled by a system-wide mapping, and so the model is still a form of mandatory access control. Section 7.4 demonstrates how some traditional, 'pure', mandatory policies can be implemented using the access control model described in this section.

Static methods were neglected in the above discussion. As static methods have no instance, they cannot provide an m-type for validation, or an m-domain for the method to execute in. Therefore, static methods are not considered `invoke` operations, and execute in the same m-domain as the client. Although they execute in the same m-domain, static methods may, and do in MCS, execute in a different discretionary protection domain.

Enforcing service-level policies using a system defined mapping, i.e. ValidateAccess, that is (potentially) different on every system, creates the problem that a service does not know how to express its abstract policy in terms of this system defined function. For example, a service cannot statically know what m-domain label to assign to a component-instance is wishes to confine. A simple solution to this problem is for the system to provide a dynamic library which supplies sufficient information for a service to determine appropriate labels.

7.3.2 Discretionary Access Control

On a general-purpose, multi-user, operating system, user policies are equally as important as system and service policies. Whilst the system-wide policy may allow users to share the files they create with all other users, in general, users will want to restrict this access. If users cannot enforce such policies, then the system becomes far less general-purpose. As stated above, user policies are ad hoc (e.g. try to specify a policy for access to the files in your home directory) and should be under the control of the user themselves, i.e. they should be enforced by a discretionary access control mechanism.

Discretionary access control is provided by password capabilities [APW86]. Mungi allows the discretionary protection domain of a subject to be extended for the duration of a procedure call, using a mechanism called *protection domain extension (PDX)* [VERH96]. A description of PDX was presented in Chapter 2.1, and will not be repeated here.

Password capabilities, unlike segregated capabilities [Tan92], can be freely distributed without system intervention. A result of this is that once a user passes out a capability to another user, they cannot control further distribution of the capability. Mandatory access control mechanism described above, however, permits the creator of an object to specify its m-type and hence restrict its accessibility. This allows the owner of an object to retain some control over access to an object after the capability has been shared.

7.3.3 Combined

Together, the mandatory and discretionary mechanisms described above support the specification and enforcement of system, software and user security policies. Importantly, it allows these policies, particularly system and software, to coexist. An access is allowed iff it is allowed by all three policies.

7.4 Examples

This section illustrates how some well-known security policies can be realised using the model described above. There are four steps to implementing a security policy using the above access control mechanism:

1. Define the structure of the m-domain and m-type labels.
2. Define the ValidateAccess function.
3. Define the LegalDomains function.
4. Define the LegalTypes function.

7.4.1 Denning's Lattice

Denning's Lattice [Den76] models military classification systems. It is a purely mandatory access control scheme, and does not involve discretionary security. This section assumes that all subjects possess full access capabilities to all objects in the system. Different access modes are also not considered.

Every object, o, has a *classification*, $C(o)$, denoting a security level, e.g. unclassified, confidential, secret or top secret. Every subject, s, has *clearance*, $C(s)$, to access certain security levels. Every subject and object is also affiliated with a *compartment*, K, denoting the role of that entity, e.g. engineering, personnel, medical or command.

An access is validated iff:

$C(s) \geq C(o)$ *sufficient clearance*, **and**
$K(s) \subseteq K(o)$ *need to know*

Domains and Types

COMPARTMENT	CLEARANCE

Figure 7.5: Domain and Type Labels

Compartment is a bit field, with each bit representing a particular compartment. If the bit is one, then the m-domain or m-type belongs to that compartment. Clearance is an integer, with a higher value indicating a more privileged clearance.

Validate Access

$\textsc{ValidateAccess} : D \times T \rightarrow B$

An access is validated by extracting the compartment and clearance values from the labels and comparing them. Iff the clearance value of the m-domain is greater than, or equal to, the clearance value of the m-type, **and** the m-domain belongs to at least every group the m-type belongs to, the validation is successful.

Legal Domains

$\textsc{LegalDomains} : D \rightarrow P(D)$

A subject may only create other subjects in an equal or less privileged m-domain, i.e. with an equal or lower clearance level, and in a subset of the compartments it belongs to. Since m-types and m-domains have the same structure, this function is exactly the same as the ValidateAccess function.

Legal Types

$\text{LegalTypes} : D \rightarrow P(T)$

A subject may only create an object with an m-type that it may access. Therefore, this function is also exactly the same as the ValidateAccess function.

7.4.2 Chinese Wall

Preventing conflicts of interest is important in a corporate environment, and is the motivation behind the *Chinese Wall* policy [San93], based on the concept of *conflict classes*. Two objects are placed in the same conflict class if it is deemed that knowledge of both would create a conflict of interest. A subject may initially access any object, however, once an object is accessed, all other objects in this conflict class become unavailable.

Consider the object partitioning shown in Figure 7.6. If a subject accesses object A.2, it may no longer access objects A.1 or A.3, though all objects in conflict classes B and C are still available. If this subject then accesses B.1, object B.2 will also become unavailable.

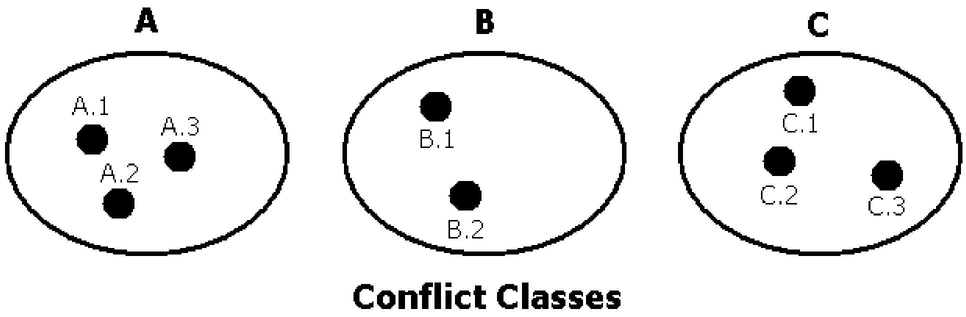


Figure 7.6: Chinese Wall Policy

Chinese Wall differs from other security policies, in that the validity of an access if based on the access history of the subject, i.e. it is a state-based policy.

Domains and Types

M-domain and m-type labels are simply unique identifiers with no particular structure. M-domains identify the user on whose behalf the subject is executing, and m-types identify the conflict class of the object.

Validate Access

$\textsc{ValidateAccess} : D \times T \rightarrow B$

As stated above, Chinese Wall is a stated-based policy. Therefore, the $\textsc{ValidateAccess}$ function *must* maintain state. Here, the state information is obtained using the function:

$$\textsc{AccessedTypes} : D \rightarrow P(T)$$

The returned set contains all types which have been recently accessed. Using this function, the Chinese Wall validation can be implemented as:

$$\begin{array}{|l}
\textsc{ValidateAccess} : D \times T \rightarrow B \\
\hline
\mathit{Allowed} = t \notin \textsc{AccessedTypes}(d) \\
\textsc{AccessedTypes}'(d) = \textsc{AccessedTypes}(d) \cup t
\end{array}$$

The second line in the implementation specifies that following validation, the m-type is added to the accessed m-types list for the m-domain.

Legal Domains

$\textsc{LegalDomains} : D \rightarrow P(D)$

M-Domains indicate the user on whose behalf the subject is executing. Therefore, users may only create subjects with the same m-domain.

Legal Types

$\textsc{LegalTypes} : D \rightarrow P(T)$

Objects in a system using the Chinese Wall policy are created and classified by specially privileged users, who may assign any m-type. Normal users may not create new objects.

7.4.3 Containment

Denning's Lattice and the Chinese Wall policy are information flow models, i.e. they restrict the flow of information out of the system. *Containment* is a relationship that enforces layering of components. It can be used to ensure encapsulation, and hence integrity, of data structures spanning several objects and component-instances.

Consider the situation shown in Figure 7.7. In this example, an instance of the `binary_tree` component-class constructs a binary tree using instances of the `bt_node` component-class. Clients perform operations on the tree by calling methods of the `binary_tree` component. If a client is allowed to directly access one of the node components, data can be modified in an uncontrolled manner, and so data integrity is lost. Containment prevents clients from accessing contained component-instances.

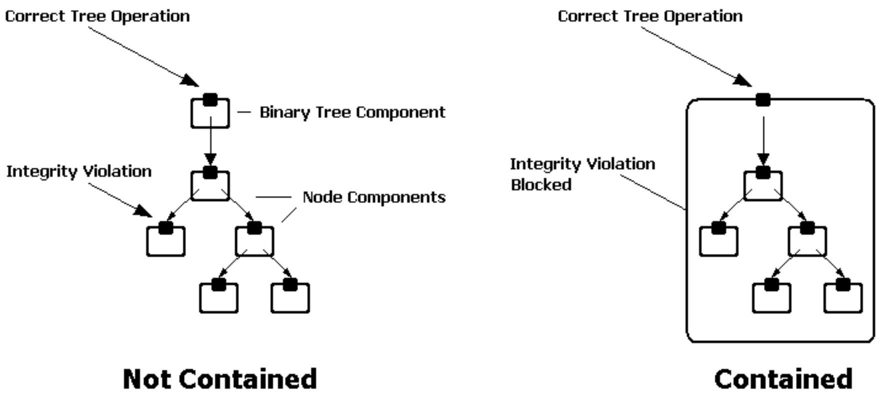


Figure 7.7: Containment

Domains and Types

A set of objects and components that must be kept consistent form a group known as a *constrained data item (CDI)* [CW87]. The upper bits of m-domain and m-type labels identify the CDI, while the lower bits indicate a *containment level* within the CDI. An object can only be accessed by a subject in the same CDI with an equal or higher containment level. An object with an m-type label equal to zero can be accessed by all subjects.

Figure 7.9 shows the m-domain, m-type, relation. Objects are shown as circles, subjects

CDI Identifier	C. Level

Figure 7.8: Domain and Type Labels

are shown as squares and component-instances are shown as a connected circle (indicating its m-type) and square (indicating its m-domain). CDIs are shown as shaded regions. Subjects may only access objects, or invoke methods on instances, whose m-type is in an ancestor node of the subject. Component-instances with their m-type in the root m-domain, and their m-domain in a CDI, are the top level (client) interfaces to the CDIs. A subject can only access data in a CDI via one of these client interfaces. As component-instances only allow clients to execute specific methods, data integrity can be assured. For example, in Figure 7.9, standard clients can only access the data in CDI-A via the component-instance marked `A.1`. This protection is guaranteed even if a client colludes with a trusted agent to obtain a capability to the data.

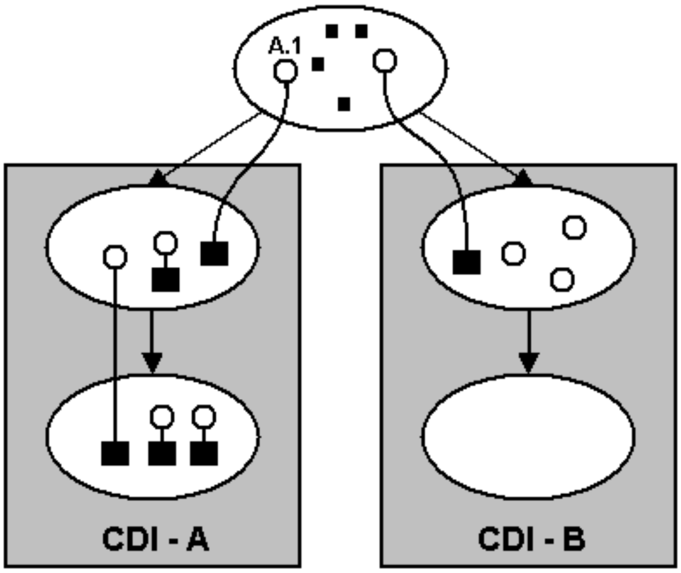


Figure 7.9: Containment Domains

Validate Access

$\textsc{ValidateAccess} : D \times T \rightarrow B$

As stated above, an access is validated successfully if the CDI of the m-type is the same as the CDI of the m-domain, **and**, the containment level of the m-type is less than, or equal to, the containment level of the m-domain. All m-domains may access objects with a zero m-type.

Legal Domains

$\textsc{LegalDomains} : D \rightarrow P(D)$

M-Domains are allowed to create subjects anywhere in the same CDI. Recall that this policy is to ensure data integrity between CDIs, not to restrict information flow.

Subjects in the zero m-domain may create subjects with CDI identifiers that are not in use, creating new data structures. This requires that the policy keep a list of free CDIs.

Legal Types

$\textsc{LegalTypes} : D \rightarrow P(T)$

To maintain correct layering, subjects may only assign m-types with the same CDI and with an equal, or lower, containment level. Any subject may assign the zero m-type.

Other Security Policies

A number of additional security policies can be expressed using the containment model with slight modifications. For example, if user sessions are started inside a non-root node, and subjects are not allowed to assign the zero m-type, confinement [Lam73] can be enforced. Providing system defined components for controlled access between confinement m-domains, results in a model similar to the MIT model [ML97].

7.5 A Formal Model

A faulty access control mechanism is catastrophic, therefore, implementations of access control mechanisms must be verified. Furthermore, it should be possible to formally reason about security. Both of these require a formal model of the access control mechanism. This section presents such a model, in a style similar to that presented by Robert Grimm and Brian Bershad in [GB97a].

7.5.1 Terminology

- **Capability** A token which conveys to the holder the discretionary right to perform certain operations on a particular object. C is the set of all capabilities.
- **Component-Instance** A unique name associated with an m-domain and an m-type. I denotes the set of all instances in a system.
- **M-Domain** A label associated with a subject or a component-instance, that determines the permissions of an executing thread. D is the set of all m-domains.
- **Object** A memory segment associated with an m-type. O denotes the set of all objects in a system.
- **Operation** An action performed on a passive entity by an active entity. M denotes the set of all operations in a system.
- **Passive Entity** An object or a component-instance.
- **Protection Domain** A set of capabilities. Each thread is associated with exactly one protection domain for discretionary access control.
- **Subject** An active entity, i.e. a thread, associated with an m-domain. S denotes the set of all subjects in a system.
- **Thread** A single, sequential flow of control. Equivalent to a subject from an access control perspective. Associated with a m-domain for mandatory access control, and a protection domain for discretionary access control.
- **M-Type** A label associated with an object or component-instance, that determines who may access the entity. T is the set of all m-types.
- **User** Any user who interacts directly with a computer system and thus has threads of control executing some code on their behalf, unit of accountability in a system.

This list was adapted from the security glossary published in [GB97a], which in turn cites [CSI] as a source of definitions.

7.5.2 Access Modes

An access mode *am* defines the set of legal operations a subject can perform on an object or component-instance. Six operations are defined.

$$am \subseteq M$$

read Execute load instructions on a memory segment.

write Execute store instructions on a memory segment.

execute Jump, or branch, to an address in a memory segment.

destroy Invalidate a memory segment.

pdx_execute Jump, or branch, to a specified address in a memory segment, and modify the discretionary protection domain.

invoke Modify the m-domain associated with a subject for the duration of a method call.

An `owner` access mode consists of read, write, execute and destroy permissions.

> **Rule 1** *A subject can only perform operations on an object or component-instance that are described by the corresponding access mode.*

7.5.3 Discretionary Access Control

Discretionary access control is performed using password capabilities [APW86]. Each subject is associated with exactly one protection domain *pd*, which comprises a set of capabilities. A function ProtectionDomain returns the protection domain of a subject. A protection domain can be modified.

$$pd \subseteq C$$
$$\textsc{ProtectionDomain} : S \rightarrow P(C)$$

A capability is associated with exactly one object, and conveys an access mode to the holder. A capability whose access mode includes `pdx_execute` is also associated with a protection domain, and a set of valid entry points. The rules below formalise discretionary access control.

E; The set of function entry points.
$\textsc{AccessMode} : C \rightarrow P(M)$; access mode of a capability.
$\textsc{CapabilitySet} : O \rightarrow P(C)$; set of capabilities associated with an object.
$\textsc{EntryPoints} : C \rightarrow P(E)$; valid `pdx_execute` entry points.
$\textsc{PdxDomain} : C \rightarrow P(C)$; protection domain associated with a PDX capability.
$\textsc{ProtObject} : P(C) \rightarrow O$; the object in which a protection domain is stored.

Rule 2 *Each subject is associated with exactly one protection domain, and each object is associated with a set of capabilities.*

Rule 3 *A capability is associated with exactly one object.*

Rule 4 *A subject s can* at most *perform the operations m on an object o for which they possess an appropriate capability, i.e.*

$$\exists c \in \textsc{ProtectionDomain}(s) \bullet c \in \textsc{CapabilitySet}(o) \wedge m \in \textsc{AccessMode}(c)$$

Rule 5 *A subject s_p in protection domain pd_p, can create a subject s_c in a protection domain pd_c iff it holds a capability to the object in which pd_c is stored, i.e.*

$$\exists c \in pd_p \bullet c \in \textsc{CapabilitySet}(\textsc{ProtObject}(pd_c))$$

A subject can create another subject in its own protection domain without any validation.

Rule 6 *A subject may add a capability to an object for which it possesses a capability with owner access mode.*

Rule 7 *A subject s_c may invoke the entry point address e in a protection domain pd_{amp} if (o is the segment in which e resides):*

$$\exists c \in \textsc{ProtectionDomain}(s_c)\bullet$$
$$c \in \textsc{CapabilitySet}(o) \wedge pdx\_execute \in \textsc{AccessMode}(c)$$
$$\wedge e \in \textsc{EntryPoints}(c) \wedge pd_{amp} \subseteq \textsc{ProtectionDomain}(s_c) \cup \textsc{PdxDomain}(c)$$

At most in rule four is due to the fact that access operations must be successfully validated by both discretionary and mandatory access control to be allowed.

7.5.4 Mandatory Access Control

A modified version of DTE, described in Section 7.3, is used for mandatory access control. DTE uses domains and types to restrict accesses. As mandatory access control mechanism has already been described in detail, these rules are presented without annotation.

Rule 8 *Each subject is associated with exactly one m-domain, each object is associated with exactly one m-type and each component-instance is associated with exactly one m-domain **and** one m-type.*

Rule 9 *An object may not be the passive entity of a invoke operation, and a component instance can be the passive entity only of invoke operations.*

Three functions, defined by a system administrator, control the legal accesses in a system. Their semantics are discussed below.

$$\textsc{ValidateAccess} : D \times T \to P(M)$$
$$\textsc{LegalDomains} : D \to P(D)$$
$$\textsc{LegalTypes} : D \to P(T)$$

Rule 10 *A subject in m-domain d, may* at most *perform an operation m on a passive entity with m-type t if:*

$$m \in \textsc{ValidateAccess}(d, t)$$

Rule 11 *A subject in m-domain d_p may create a subject in m-domain d_c iff:*

$$d_c \in \textsc{LegalDomains}(d_p)$$

Rule 12 *A subject in m-domain d_p may create an object with m-type t iff:*

$$t \in \textsc{LegalTypes}(d_p)$$

Rule 13 *A subject in m-domain d_p may create a component-instance i with m-type t_i and m-domain d_i iff:*

$$d_i \in \textsc{LegalDomains}(d_p) \wedge t_i \in \textsc{LegalTypes}(d_p)$$

Rule 14 *A validated invoke operation results in the domain associated with the subject to be changed to the domain associated with the component-instance on which the operation was performed.*

At most in rule ten is again due to the fact that access operations must be successfully validated by both discretionary and mandatory access control to be allowed.

7.5.5 Combined

Rule 15 *A subject, s, can perform an operation, m, on an object, o, iff Rule 4 and Rule 10 are* ***both*** *satisfied, i.e.*

$$(\exists c \in \textsc{ProtectionDomain}(s) \bullet c \in \textsc{CapabilitySet}(o) \wedge m \in \textsc{AccessMode}(c)) \;\wedge$$
$$(m \in \textsc{ValidateAccess}(\ \textsc{Domain}(s),\ \textsc{Type}(o)))$$

Chapter 8

Access Control Implementation

> *Security is of paramount importance in any system, but even more so in an extensible system where any user can add to the overall functionality.*
>
> **Tim Wilkinson and Kevin Murray, 1994 [WM95]**

8.1 Overview

This chapter presents an implementation of the access control model, described in Chapter 7, on the Mungi operating system. For complete mediation, the following five operations must always be validated.

- A subject performing a read, write, execute or destroy operation on an object.
- A subject performing an invoke operation on a component-instance.
- Assigning a domain label to a subject on creation.
- Assigning a type label to an object on creation.
- Assigning domain and type labels to component-instance on use.

Security requirements differ from site to site. An implementation of an access control model should therefore allow policies to be easily changed by an authorised administrator.

This requires the separation of mechanism and policy, which, fortunately, Chapter 7 has already done. Seven functions were required to describe the validations; each can be clearly classified as providing either mechanism or policy.

Mechanism	**Policy**
Domain	LegalDomains
Type	LegalTypes
InstanceCreator	ValidateAccess
InstanceDomain	

For flexibility, the policy functions are implemented in user-space PDX objects called *security policy objects*, or just *policy objects* when the context is clear. When one of the policy functions is required, Mungi locates the policy object and performs a `PdxCall()` for the appropriate function. Despite the excellent performance of `PdxCall()` compared to other systems' cross-domain calls, performing a `PdxCall()` on every validation would introduce a significant overhead. Therefore, a mandatory validation cache is implemented.

As Mungi already implements discretionary access control using password capabilities [HEV$^+$98], this chapter is only concerned with mandatory access control. Section 8.2 describes how policy objects are constructed and how Mungi locates and invokes a policy object, Section 8.3 describes the in-kernel implementation of the mechanism functions, and Section 8.4 contains some concluding remarks.

8.2 Policy Objects

Policy objects allow system-wide security policies to be implemented outside the kernel. Policy objects are implemented as standard PDX objects rather than components, as the latter would make MCS part of the trusted computing base (TCB). This section describes how policy objects are constructed, and how Mungi locates and invokes policy functions. Appendix A contains three sample policy object implementations.

8.2.1 Policy Object Interface

A policy object must implement four functions, corresponding to the three policy functions introduced in Chapter 7, plus a new function, DomainStack, required by the implementation.

$$\begin{aligned}&\textsc{LegalDomains} : D \to P(D)\\&\textsc{LegalTypes} : D \to P(T)\\&\textsc{ValidateAccess} : D \times T \to P(M)\\&\textsc{DomainStack} : D \to T\end{aligned}$$

DomainStack is required because a thread's stack object is created by Mungi. Mungi contains no knowledge of the security policy, therefore, it doesn't know what type label it should assign to the stack object. As this is a matter of policy it is added to the policy object. When a new thread is created, its domain label, provided by the policy aware client, is passed to the policy object which returns the type label for the thread's stack.

When the policy object is registered as a PDX object using `ObjCrePdx()`, the functions *must* appear in this order. This allows the kernel to easily obtain the entry point corresponding to a policy function once the object descriptor has been retrieved. How the kernel locates the appropriate object descriptor is described in Section 8.2.3.

Interface functions are called using `PdxCall()`, whose signature:

```
int PdxCall( pdx_t proc, cap_t param, cap_t *ret, void *pd );
```

allows a 128 bit argument and result. As up to two labels may be passed to a policy object function, and additional bits are required for other arguments, domain and type labels are defined as 60 bit values. A field, **Ext**, in the standard argument structure, indicates that additional information immediately follows the standard argument on the stack. This facility can be used to increase the size of labels if 60 bits proves insufficient. The structure of labels is specified by the security policy. The following sections describe the argument and return value formats for each function. All result descriptors contain the arguments passed to the function, an indication of whether the validation was successful, and a field identifying the function invoked.

Legal Domains

Conceptually, LegalDomains is passed a domain label and returns the set of domain labels that subjects executing in the parameter domain may assign. As this set may contain up to 2^{61} elements, it is impractical to implement the function this way. Instead, the policy object expects two domain labels to be passed, and returns whether a subject in the first domain may legally assign the second. As well as being implementable, this function results in simpler kernel code. Stated formally, the policy object implements the function (note that B is the boolean set defined in Section 7.3):

$$\textsc{LegalDomains}' : D \times D' \rightarrow B$$

$$\textsc{LegalDomains}' = d' \in \textsc{LegalDomains}(d)$$

Figure 8.1 shows the argument (upper) and result (lower) formats, with the most significant bit on the left. Result fields is explained in Section 8.2.4.

Active Domain$_{(60)}$	**Pad**$_{(4)}$	**New Domain**$_{(60)}$	**Pad**$_{(1)}$	**Ext**$_{(3)}$

Active Domain$_{(60)}$	**Pad**$_{(4)}$	**New Domain**$_{(60)}$	**Pad**$_{(1)}$	**R**$_{(1)}$	**Fn**$_{(2)}$

Active Domain is the domain label of the subject performing the operation.

New Domain is the domain label that the subject is attempting to assign.

Pad bits must all be set (only for this function).

R is set iff the validation was successful.

Fn identifies the interface function returning this result. For LegalDomainsfn is 0.

Ext indicates the format of additional information following the argument on the stack.

Figure 8.1: LegalDomains Argument / Result Formats

Legal Types

Conceptually, LegalTypes is passed a domain label and returns the set of types that subjects executing in the parameter domain may assign. Again, practical concerns result in the policy object providing the function:

$$\begin{array}{|l} \textsc{LegalTypes}' : D \times T \rightarrow B \\ \hline \textsc{LegalTypes}' = t \in \textsc{LegalTypes}(d) \end{array}$$

Figure 8.2 shows the argument and result formats for LegalTypes.

Domain$_{(60)}$	**Pad**$_{(4)}$	**Type**$_{(60)}$	**Pad**$_{(1)}$	**Ext**$_{(3)}$

Domain$_{(60)}$	**Pad**$_{(4)}$	**Type**$_{(60)}$	**Pad**$_{(1)}$	**R**$_{(1)}$	**Fn**$_{(2)}$

Domain is the domain label of the subject performing the operation.

Type is the type label that the subject is attempting to assign.

R is set iff the validation was successful.

Fn identifies the interface function returning this result. For LegalTypesfn is 1.

Ext indicates the format of additional information following the argument on the stack.

Figure 8.2: LegalTypes Argument / Result Formats

Validate Access

Conceptually, ValidateAccess is passed a domain label and a type label, and returns the set of legal operations. Again, practical concerns result in the policy object providing the function:

$$\begin{array}{|l} \textsc{ValidateAccess}' : D \times T \times M \rightarrow B \\ \hline \textsc{ValidateAccess}' = m \in \textsc{ValidateAccess}(d, t) \end{array}$$

Figure 8.3 shows the argument and result formats for ValidateAccess.

DomainStack

DomainStack is passed a domain label and returns the type label that Mungi should assign to the stack object of a thread being created in this domain. Figure 8.4 shows the argument and result formats for DomainStack.

Domain$_{(60)}$	**AC**$_{(4)}$	**Type**$_{(60)}$	**AC2**$_{(1)}$	**Ext**$_{(3)}$

Domain$_{(60)}$	**AC**$_{(4)}$	**Type**$_{(60)}$	**AC2**$_{(1)}$	**R**$_{(1)}$	**Fn**$_{(2)}$

Domain is the domain label of the subject performing the operation.

Type is the type label of the object the subject is attempting to access.

AC & AC2 is a bit-field indicating the access mode being requested. From the least-significant-bit, read, write, execute, destroy and invoke.

R is set iff the validation was successful.

Fn identifies the interface function returning this result. For ValidateAccessfn is 2.

Ext indicates the format of additional information following the argument on the stack.

Figure 8.3: ValidateAccess Argument / Result Formats

Domain$_{(60)}$	**Pad**$_{(4)}$	**Pad**$_{(61)}$	**Ext**$_{(3)}$

Domain$_{(60)}$	**Pad**$_{(4)}$	**Type**$_{(60)}$	**Pad**$_{(2)}$	**Fn**$_{(2)}$

Domain the domain label being assigned to the new thread.

Type is the type Mungi should assign to the stack object.

Fn identifies the interface function returning this result. For DomainStack fn is 3.

Ext indicates the format of additional information following the argument on the stack.

Figure 8.4: DomainStack Argument / Result Formats

8.2.2 Multiple Validations

A single Mungi operation often requires multiple validations. For example, an `ObjCreate()` system call must validate the client's access to the passed `objinfo` structure, as well as validating the assigned type label. Policy objects, therefore, provide a fifth entry point that accepts a list of validation requests, reducing the number of invocations that such operations must perform. A validation request is simply the argument to the appropriate policy function, as shown in the previous section, with the **Fn** field set. This function, whose implementation is shown in Figure 8.5, passes each validation request to the appropriate function, and replaces the validation request with the result descriptor.

```
/* Definitions provided by man_security.h */
typedef struct {
        unsigned domain : 60;
        unsigned access : 4;
        unsigned dot  : 60;
        unsigned access2 : 1;
        unsigned result : 1;
        unsigned fn    : 2;
} po_valdesc_t;

typedef struct {
        int count;
        po_valdesc_t vals[1];
} po_mvaldesc_t;

/* Implemented in policy object */
typedef po_valdesc_t (*po_func)(po_valdesc_t);
po_valdesc_t pofn[4] = { po_legaldomains, po_legaltypes, po_validateaccess, po_domainstack };

cap_t po_multival( po_mvaldesc_t *vals )
{
        int i;
        cap_t res;
        i = vals->count;
        for( i = 0; i < count; i++ )
                (vals->vals[i]) = pofn[(vals->vals[i]).fn](vals->vals[i]);
        res.addr = vals;
        res.passwd = 0x0;
        return res;
}
```

Figure 8.5: Multiple Validations Entry Point

8.2.3 Registration

This section describes how the kernel locates a policy object. Once the policy object is located, the kernel can easily find the entry point corresponding to a particular policy function from its object descriptor, stored in the Mungi object table.

At system configuration time, the address of an object, called a *P-List*, is registered with the system. This value cannot be changed. The P-List is an array containing the address of each policy object in the system, with a `NULL` address indicating the end of the list.

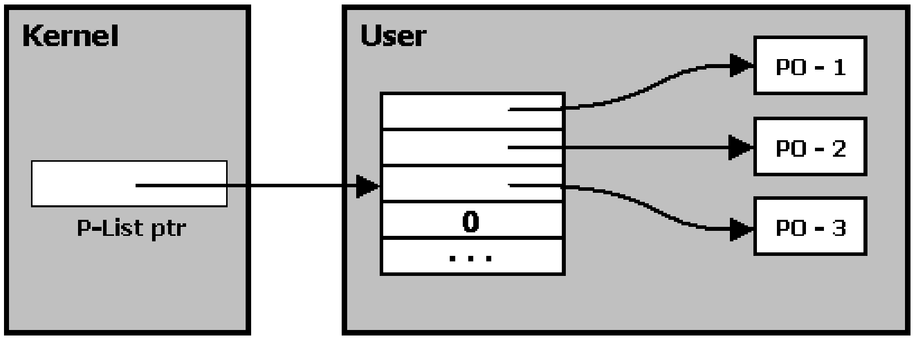


Figure 8.6: Policy Object Registration

When the kernel needs to call a policy function, it reads the first address in the list and retrieves the corresponding object descriptor. If this policy object successfully validates the operation, the kernel reads the second address in the list and repeats the validation request. For a successful validation, each policy object in the list must allow the access. For simplicity, the remainder of this chapter assumes a list with a single entry.

Any subject with write access to the P-List may dynamically add or remove policy objects.

This scheme can be extended by allowing entries in the P-List to contain addresses of further P-Lists. As each P-List would have its own permissions, various administrative structures could be supported. An investigation of security administration structures is beyond the scope of this thesis. Administrative-RBAC 97 [SBC+97] contains an interesting discussion of administration structures in the context of role-based access control.

8.2.4 Asynchronous Validation

A policy function is invoked via a `PdxCall()` from the kernel. Unlike a client, it is not practical for the kernel to block while the `PdxCall()` is in progress. This section describes how the kernel performs this asynchronous invocation.

An asynchronous procedure call has two parts, issuing the request, and handling reply. These two parts are described separately, below. Between the execution of these two parts Mungi must continue to handle incoming system calls.

Issuing a Validation Request

All validation requests to policy objects are issued using a single function.

```
int aval_issue( char fn, long d, long dot, char m, void *cdata,
                mthread_t *cthread, syscall_t *lsysmsg,
                mthread_t **lthread );
```

- **fn** indicates the policy object function to be invoked.
- **d** is the first parameter to the policy object function; always a domain.
- **dot** is the second parameter; can be a domain, a type, or empty.
- **m** is the access mode being requested; only used by ValidateAccess.
- **cdata** contains the data that must be stored for the caller to be restarted when the validation returns. If the call is issued from a system call handler, this points to the `syscall` global variable, which contains all the system call parameters. If the call is issued from the page fault handler, this is the fault address.
- **cthread** points to the descriptor of the client thread being served.
- **lsysmsg** is an out parameter used to issue the `PdxCall()`, described below.
- **lthread** is an out parameter used to issue the `PdxCall()`, described below.

An implementation of this routine is shown in Figure 8.7 and described in detail below. List numbers for the description correspond to numbers in the comments in Figure 8.7.

```
/* Definitions from man_security.h */
#define GET_FUNCTION_ID(x) (x&0x03)
typedef enum {MVAL_CACHE_HIT, MVAL_PROT, MVAL_PENDING} mval_status_t;
int mvcache_check( char fn, long d, long dot, char m, po_result_t *res );
extern void *plptr;

int aval_issue( char fn, long d, long dot, char m, void *cdata, mthread_t *cthread,
            syscall_t *lsysmsg, mthread_t **lthread )
{
    object_t *obj;
    char cache_hit;
    po_result_t res;

    /* check the parameters are valid */
    assert( GET_FUNCTION_ID(fn) >= 0 && GET_FUNCTION_ID(fn) <= 3 );
    assert( (m & ~0x65) == 0 );

    /* 1. if there are no policy objects specified then return OK */
    if( plptr == NULL || *((void**)plptr) == NULL )
        return MVAL_CACHE_HIT;

    /* 2. check mandatory validation cache */
    cache_hit = mvcache_check( fn, d, dot, m, &res );
    if( cache_hit == 1 ) {
        if( res.r == 1 )
            return MVAL_CACHE_HIT;
        return MVAL_PROT;
    }

    /* 3. add entry to the pending validation queue */
    pv_add( fn, d, dot, m, cdata, cthread );
    fn = fn & 0x3;

    /* 4. get the object descriptor for the policy object */
    obj = object_find( *((void**)plptr) );
    if( obj == NULL )
        return MVAL_PROT;

    /* 5. issue the pdx call */
    lsysmsg->syscall.number = SYS_PDX_CALL;
    lsysmsg->syscall.data.pdx.call.proc = ((obj->info).pdx.entry)[fn-1];
    lsysmsg->syscall.data.pdx.call.param = aval_construct_param( d, dot, m );
    lsysmsg->syscall.data.pdx.call.pd = PD_EMPTY;
    *lthread = (mthread_t*)-1;

    /* return pending status */
    return MVAL_PENDING;
}
```

Figure 8.7: Invoking a Policy Object Function

1. As a validation is the conjunction of the policy object results, and one, if there are no policy objects defined then the validation succeeds.

2. A cache of recent validations is maintained by the kernel. Before issuing a call, the function checks whether a matching validation can be found. Both successful and unsuccessful validations are stored in the cache, so the function has to check the **R** bit in the result. If a matching validation *is* found, then the function returns control to the caller who may continue synchronously. To prevent denial-of-service attacks, the cache limits the number of unsuccessful validations stored. The current implementation sets this limit to one entry.

3. If the validation is not in the cache, a request must be issued to the policy object. To allow the kernel to match validation replies to client threads, and continue the requested operation, an entry is added to the *pending validation (PV) queue*.

```
typedef struct {
    /* info required to match up to returning calls */
    char  fn;
    long  d;
    long  dot;
    char  m;

    /* info required to resume the original call */
    mthread_t *cthread;
    union {
        syscall_t sysmsg;
        long fault_addr;
    } cdata;
} pv_entry_t;
```

   The MSB of **fn** indicates whether the `cdata` is for a page fault or a system call handler. This bit must be set appropriately in the **fn** argument by the caller. Once the entry has been constructed, the bit is cleared.

4. Using the P-List pointer, `plptr`, the address of the policy object is located, its descriptor retrieved, and the appropriate entry point determined.

5. A `PdxCall()` is then issued. A brief description of the Mungi system call handler is required to understand this code. System calls are invoked by a client sending an IPC to the Mungi kernel. This IPC contains a `syscall_t` structure that contains

a system call identifier and the system call arguments. A thread in Mungi, shown in Figure 8.8, waits for and dispatches each system call to an appropriate handler.

```
/* system call jump table */
static void (*(msys_call[SYSCALLS_MAX]))(void);

static syscall_t sysmsg;
static mthread_t *thread;

void syscall_loop( void )
{
        for( ; ; ) {
                msys_call[sysmsg.syscall.number & (SYSCALLS_MAX−1)];
        }
}
```

Figure 8.8: Mungi System Call Loop

Every system call handler must wait for the next system call before returning, and fill in the `sysmsg` and `thread` parameters. Two functions are supplied for this.

```
syscall_wait();
syscall_return();
```

`syscall_wait()` simply waits for the next system call, sets `thread` to the sender of the IPC, and `sysmsg` to the contents of the IPC. `syscall_return()` sends an IPC containing `sysmsg` (which the handler will have modified) to `thread`, before waiting for the next system call.

`aval_issue()`, therefore, can simulate a `SYS_PDX_CALL` system call by filling in the `sysmsg` and `thread` variables appropriately. When it returns `MVAL_PENDING` to a handler, the handler should return immediately, without waiting for a system call. Control will then return to the `syscall_loop()` function. As `aval_issue()` has setup the `sysmsg` and `thread` variables, the PDX system call handler will be invoked. On completion, this handler will wait for the next system call.

When `aval_issue()` returns `MVAL_PENDING` to the page fault handler, it must explicitly call the PDX system call handler, with an indication telling it not to wait for the next system call.

Handling Replies

A PDX system call, as issued by `aval_issue()`, results in a new thread being created. The descriptor of this thread contains the `thread` variable, which identifies the client waiting for the result, and a flag indicating that it is a PDX thread. Threads terminate by sending an IPC to Mungi, containing the `SYS_THREAD_RETURN` system call identifier, and a 128 bit result argument.

This invokes the `sys_thread_return()` system call handler, which is primarily responsible for cleaning up thread resources. Before cleaning up, however, the handler checks if the terminating thread is a PDX thread. If so, it sends an IPC containing the result to the client thread, whose identity is stored in the PDX thread's thread descriptor. A client identity of `-1` indicates that this is a policy function invocation performed by the kernel, and an IPC should not be sent. Instead, a handler function is called, with the validation result passed as an argument. The implementation of this handler is shown in Figure 8.9, and is described in detail below.

1. Using **fn** and the returned arguments, a matching entry for the validation is located in the PV queue. This entry is removed from the queue and returned.
2. Regardless of whether the validation was successful, the result is added to the validation cache. If the reply is for a multiple validation, all results are cached.
3. If the original caller was a system call handler, then the system call must be restarted. As system calls perform all validations before modifying any state, they can simply be re-dispatched. The `sysmsg` and `thread` values are set up using the values stored in the PV entry, and `aval_return()` returns. `sys_thread_return()` cleans up the PDX thread, and returns to the dispatcher without waiting for the next system call. Therefore, the halted system call is immediately restarted. When the handler re-executes `aval_issue()`, it will find the validation in the mandatory cache, and so can continue synchronously.
4. As the Mungi version on which the prototype was implemented does not swap virtual memory to disk, the page fault handler is extremely simple. It consists of a validation, followed by a mapping. Therefore, `aval_return()` completes the page fault handling itself, and then waits for the next system call.

   A Mungi version with a more complex page fault handler would require an IPC, containing the matching PV entry, to be sent from the system call thread to the page fault handler. Using the data in the PV entry, the page fault handler can continue

```
/* Definitions from man_security.h */
#define GET_IS_PAGEFAULT(x) (x&0x0080)
pv_entry_t pv_find_and_remove( po_result_t result );
int mvcache_check( char fn, long d, long dot, char m, po_result_t *res );
void aval_complete_pagefault( pv_entry_t user, po_result_t res );

/* Implemented in man_security.c */
void aval_return( cap_t result )
{
    pv_entry_t pve;
    po_mvaldesc_t *pmv = result.addr;
    po_result_t *pres = (po_result_t*)&result;

    /* 1. find and remove the matching entry from the queue */
    pve = pv_find_and_remove( *pres );

    /* 2. add the returned entry to the cache */
    if( IS_MULTICALL(pmv) != 0 ) {
        for( i = 0; i < pmv->count; i++ )
            mvcache_add( pmv->vals[i] );
    } else {
        mvcache_add( *pres );
    }

    /* 3. continue the system call */
    if( GET_IS_PAGEFAULT(pve.fn) == 0 ) {
        sysmsg = pve.cdata.sysmsg;
        thread = pve.cthread;
        caller = pve.cthread->l4id;
        return;
    }

    /* 4. otherwise, complete the page fault */
    aval_complete_pagefault( pve, *pres );
    syscall_wait();
}
```

Figure 8.9: Policy Object Result Handler

processing the fault. Since page fault handlers that swap virtual memory to disk must be able to handle asynchronous behaviour anyway, asynchronous validations should not require any structural modifications.

8.3 Kernel Mechanics

Complete mediation requires a validation to be performed whenever a subject performs an operation on an object. In Mungi, subjects operate on objects either directly, i.e. by accessing a memory location, or via the system call interface. A validation must also be performed whenever security attributes, i.e. domain and type labels, are assigned. This section describes the code added to the Mungi kernel to perform these validations.

8.3.1 Domain Assignment

Domain labels are assigned whenever a new subject is created. Mungi must validate such assignments using the LegalDomains policy function. Subjects are created by the `ThreadCreate()` system call, and by a method invocation. This section describes the code added to `ThreadCreate()`; method invocation is described in Section 8.3.4.

A domain label is added to the `threadinfo_t` structure, which is part of the thread descriptor. This structure is passed as an argument to the `ThreadCreate()` system call, and can also be read using `ThreadInfo()`. The added member defined as:

```
unsigned long   domain;
```

Given this `domain` member, the implementation of Domain $: S \rightarrow D$, is trivial. S is implemented as `mthread_t*`, and the Domain function as:

```
DOMAIN(s)   ((s->info).domain)
```

Mungi validates the assigned domain by adding the following code to the `ThreadInfo()` system call handler *after* the validation of the `threadinfo` parameter.

```
/* Defined in man_security.h */
#define aval_issue_legal_domains(d,dot,cdata,cthread) \
        aval_issue(1,d,dot,0,cdata,cthread,&sysmsg,&thread)

/* Implemented in sys_thread_create */
if( info != NULL && info->domain != DOMAIN(thread) ) {
        r = aval_issue_legal_domains( DOMAIN(thread), info->thread, &sysmsg, thread );
        if( r != MVAL_CACHE_HIT ) {
                if( r == MVAL_PENDING )
                        return;
                else
                        goto error;
        }
}
```

As it cannot violate access control (though resource utilisation is another question) a thread may create threads in its own domain without requiring a validation.

8.3.2 Type Assignment

Type assignments must be validated using the LegalTypes function whenever a new object is created. Objects are created using the `ObjCreate()` system call. Although threads can operate on other threads using system calls such as `ThreadDelete()`, these system calls all have static access control policies, based on a thread hierarchy, defined in the Mungi API [HVGR99]. Therefore, threads are not objects in the access control sense.

Validation of type assignments is almost identical to validation of domain assignments. A type member is added to the `objinfo_t` structure, which is part of the object descriptor. This structure is passed as an argument to the `ObjCreate()` system call, and can also be read using `ObjInfo()`. The added member defined as :

```
unsigned long   type;
```

As the operation involved is known statically, the Type function is split in two, as shown below. ObjectType is described in this section, InstanceType in Section 8.3.4.

$$\textsc{ObjectType} : O \rightarrow T$$
$$\textsc{InstanceType} : I \rightarrow T$$

O is implemented as `object_t*`, and $\textsc{ObjectType} : O \rightarrow T$, as:

```
OBJECT_TYPE(o) ((o->info).type)
```

Validation of type assignment is implemented by adding the following code to the `ObjCreate()` system call handler *after* the validation of the `objinfo` argument.

```
#define aval_issue_legal_types(d,dot,cdata,cthd)     aval_issue(2,d,dot,0,cdata,cthd,&sysmsg,&thread)

r = aval_issue_legal_types( DOMAIN(thread), type, &sysmsg, thread );
if( r != MVAL_CACHE_HIT ) {
        if( r == MVAL_PENDING ) {
                return;
        } else {
                addr = NULL;
                goto error;
        }
}
```

8.3.3 Object Accesses

Object accesses require both discretionary and mandatory access control restrictions to be enforced. Mungi already implements capability based discretionary access control using the `validate_obj()` function. Therefore, to add mandatory access control, a call to $\textsc{DteValidate}$ must be added wherever `validate_obj()` is called. This includes the page fault handler, and any system call handler that is passed a pointer. A standard discretionary validation is shown below.

```
obj = validate_obj( address, APD(thread), access, NULL);
if (obj == NULL) {
    ret = ST_PROT;
    goto out;
}
```

`validate_obj` returns a pointer to the object descriptor on success, or `NULL` on failure. Mandatory access control is implemented by adding the following *after* every discretionary validation in a system call handler (not the page fault handler).

```
#define aval_issue_validate_sc(d,dot,m,cdata,cthd) aval_issue(3,d,dot,m,cdata,cthd,&sysmsg,&thread)

r = aval_issue_validate_sc( DOMAIN(thread), type, access, &sysmsg, thread );
if( r != MVAL_CACHE_HIT ) {
    if( r == MVAL_PENDING )
        return;
    else
        goto error;
}
```

The following validation code is added to the page fault handler.

```
#define aval_issue_validate_pf(d,dot,m,cdata,cthd,s,t) aval_issue(0x83,d,dot,m,cdata,cthd,s,t)

r = aval_issue_validate_pf( DOMAIN(thread), type, access, fault_addr, faulter_id, &lsm, &lthrd );
if( r != MVAL_CACHE_HIT ) {
    if( r == MVAL_PENDING ) {
            _sys_pdx_call( lsm, lthrd );
            continue;
        } else {
            obj = NULL;
        }
}
```

As the page fault handler does not return to the system call dispatcher, it must explicitly invoke the `PdxCall()` system call handler. This function, however, retrieves the system call parameters from the `sysmsg` and `thread` global variables, which the pager cannot interfere with. Therefore, the original `PdxCall()` system call handler is modified to use local versions of `sysmsg` and `thread`, passed to the handler as arguments. The handler is renamed `_sys_pdx_call()`. A two line function that calls `_sys_pdx_call()` with the global `sysmsg` and `thread`, and then waits for the next system call, is installed as the new `PdxCall()` system call handler. Since `_sys_pdx_call()` is now thread-safe, and does not wait for the next IPC, it can be called directly by the page fault handler.

8.3.4 Invoke Operation

An invoke operation requires three validations, the domain of the instance must be validated, the type of the instance must be validated and the invoking thread must be validated as having `invoke` permissions on the instance.

Section 7.3.1 introduced two mappings required for validating the instance properties, and the separation of Type in Section 8.3.2 created a third.

$$\textsc{InstanceCreator} : I \rightarrow D;$$
$$\textsc{InstanceDomain} : I \rightarrow D;$$
$$\textsc{InstanceType} : I \rightarrow T;$$

Component-instances have type `void*`, and point to the corresponding component-instance header (defined in Section 5.3.2). Unlike threads and objects, creation of a new component-instance does not require kernel intervention, therefore, it cannot be used to store their properties. A component-instance's domain and type are therefore stored as the first two words of the component-instance header. This memory can be modified by the component implementation, therefore, these values must be validated before each use. InstanceDomain and InstanceType are implemented as macros.

```
INSTANCE_DOMAIN(i)    ((unsigned long*)i)[0])
INSTANCE_TYPE(i)      ((unsigned long*)i)[1])
```

Validation of a component-instance's domain and type labels requires the domain of the subject that created the component-instance. As it is used to validate access, it must be stored in a location that the kernel knows cannot be accessed by others, i.e. it must be stored by the kernel. Rather than store the creator of each component-instance, Mungi records the domain in which each *object* was created. A component-instance's creator is assumed to be the creator of the object in which the instance's data is located. By defining LegalDomains carefully, it can be guaranteed that this approximation never violates security. The implementation of InstanceCreator is shown below.

```
int instance_creator( void *instance, long *domain )
{
        object_t *obj;

        /* get the descriptor for the containing object */
        obj = object_find( instance );
        if( obj == NULL )
                return −1;

        /* return its domain */
        *domain = (obj−>info).domain;
        return 0;
}
```

`invoke` is implemented using Mungi's `PdxCall()` system call. `PdxCall()` is also used for other operations, such as static method calls, and so a method of signaling an `invoke` operation to the kernel is required. Furthermore, the kernel must be able to retrieve the instance. Although the kernel knows that the instance is always the first word of an invocation, it does not know whether this is contained in the actual `param` argument, or in a parameter buffer (see Section 5.3.3).

A subject passes this information to the kernel using the least significant two bits of the `proc` parameter (the first parameter to `PdxCall()`). This parameter is the address of the method to invoke, therefore, it must be word aligned making the lower two bits redundant. A set LSB indicates an `invoke` operation, and a set second-LSB indicates that the `param` argument is a pointer to a parameter buffer.

As it requires three validations, the `invoke` operation uses the `po_multival()` entry point. `aval_issue_multival()` is similar to `aval_issue()`, however, it accepts a variable number of validation descriptors as arguments. A `MVAL_CACHE_HIT` is returned only if *every* validation request is found in the cache. Validations that are not found in the cache are added to a list stored in a Mungi object, and a capability to this object is added to the policy object's protection domain. Figure 8.10 shows the validation code for an `invoke` operation. `aval_construct_param()` is used to format the validation requests.

To allow the `domain` member of the `threadinfo` structure in the descriptor of the PDX thread to be set correctly, the signature of `thread_pdx_create()` is modified to accept the `target_domain`. The `thread_pdx_create()` sets the domain of the thread immediately after allocating a descriptor.

```
/* get the flags */
is_invoke = (unsigned long)proc & 0x01;
is_indirect = (unsigned long)proc & 0x02;
proc = (unsigned long)proc & ~(0x3);

/* validate an invocation */
if( is_invoke != 0 ) {

        /* get the instance */
        if( is_indirect )
                instance = *((void**)param.address);
        else
                instance = (void*)param.address;

        /* get the info about the instance */
        r = instance_creator( instance, &creator );
        if( r != 0 )
            return ST_INV_INSTANCE;
        idomain = INSTANCE_DOMAIN( instance );
        itype = INSTANCE_TYPE( instance );

        /* now do all three validations */
    domain_validation = aval_construct_param( creator, idomain, 0 );
        domain_validation.fn = MVAL_LEGAL_DOMAINS;
        type_validation = aval_construct_param( creator, itype, 0 );
        type_validation.fn = MVAL_LEGAL_TYPES;
        access_validation = aval_construct_param( idomain, itype, MVAL_AC_INVOKE );
        access_validation.fn = MVAL_VALIDATE_ACCESS;
        r = aval_issue_multival( 3, domain_validation, type_validation, access_validation );
        if( r != MVAL_CACHE_HIT ) {
            if( r == MVAL_PENDING )
                        return -1;
            else
                        return ST_PROT;
        }
        target_domain = idomain;
} else {
        target_domain = DOMAIN(thread);
}
```

Figure 8.10: Invoke Validation

8.4 Concluding Remarks

The implementation of the access control mechanism is extremely simple. This flows from two key features of the Mungi design.

- A 'pure' single-address-space leads to a single object type, and hence a single validation routine. This contrasts to traditional systems which have many kinds of objects, e.g. files, processes, semaphores, memory, etc.
- Building protection in at the base of the operating system.

Together, these two features mean that Mungi's protection mechanisms are small and well structured, a critical characteristic of effective security [SS75].

Currently, domain and type labels are static properties. Since the `invoke` operation requires that each object descriptor store the domain of its creator, and thread descriptors contain a reference to their parent's descriptor (and hence domain), all label assignments could be validated on-demand. This would allow domain and type labels to be modified using `ThreadInfo()` and `ObjInfo()`. The benefits and subtleties associated with dynamic labels is a topic for future research.

Chapter 9

Access Control Performance

Protection is not tolerated if the costs are prohibitive.

Jeroen Vochteloo, 1998 [Voc98]

9.1 Overview

This chapter evaluates the performance overhead introduced by the mandatory access control (MAC) implementation described in Chapter 8. Section 9.2 shows its effect on micro-benchmark results, and Section 9.3 shows its effect on end-to-end performance. All results were obtained on a hardware platform developed at UNSW, featuring a 100MHz MIPS R4600 processor and 64MB of RAM, rated at 1.9 SPECint-95.

9.2 Micro-benchmarks

Table 9.1 shows the performance of all Mungi operations that are affected by the mandatory access control implementation. For each operation, it shows the number of validations requested, and the operation's performance without MAC, with MAC but no registered policy objects, with MAC and a policy object but no in-kernel validation cache, and with the complete implementation. A policy object implementing Denning's Lattice (shown in Appendix A) is used for the latter two results. As policy decisions for Denning's Lattice can be evaluated by a simple integer comparison, they impose a negligible policy overhead, allowing the overhead of the mechanism to be clearly observed.

System call results are the average of a large number of iterations, therefore, they represent performance under favourable cache conditions. Page fault overhead was measured by executing a benchmark that caused a defined number of page faults. The presented results represent the total performance, less the performance of the benchmark without MAC, divided by the number of page faults. All results are shown in μs. A number of observations regarding the results of Table 9.1 follow.

Operation	# Vals.	No MAC	No PO	No Cache	Complete
PdxCall	3	29	30	68	30
ObjCreate	2	30	29	68	31
ObjCrePdx	2	46	46	81	48
ObjDelete	1	44	44	87	44
ObjInfo	1	14	13	50	14
ObjPasswd	1	43	44	79	44
ObjResize	1	27	27	63	28
ApdInsert	1	13	13	47	14
ApdGet	1	14	13	51	14
ThreadCreate	2	92	89	126	91
ThreadInfo	1	17	17	52	16
SemCreate	1	48	46	87	48
SemDelete	1	28	28	63	30
SemSignal	1	24	25	59	24
SemWait	1	24	25	60	26
Page Fault	1	N/A	0	41	0.6

Table 9.1: Micro-benchmark Access Control Overhead (μs)

- There is no relationship between the number of validations requested, and the *mechanism* overhead. Obviously, if the policy overhead was comparable to the mechanism overhead, validations would exhibit such a relationship.
- Mandatory access control does not impose any overhead if it is not used.
- An uncached validation request imposes an overhead of approximately 38μs. This cost is dominated by the required `PdxCall()`, however, an additional 10μs overhead is being added by the kernel processing. Although time constraints prevent a more

detailed analysis here, it is believed that this cost can be reduced by optimising the validation cache, the pending validation queue, and the system call dispatcher. Currently, an uncached validation results in an overhead between 42 and 300%.

- A validation cache reduces this overhead to around 1μs for *repeated calls.*

9.3 Macro-benchmarks

This section presents results from three macro-benchmarks: the OO1 and Andrew benchmarks introduced in Chapter 6, and a simplified version of the Jigsaw benchmark [FKJ$^+$90].

OO1 is modified to assign a domain, D_C, to the client thread, a domain, D_{DC}, and type, T_{DC}, to the database component-instance, and a type, T_{DD}, to the actual database data. D_C may access T_{DC} but not T_{DD}. D_{DC} may access T_{DD}.

Jigsaw is a memory-intensive benchmark. An $n \times n$ grid of 1kB 'tiles' is generated, and all four edges of each tile are assigned an identifier. Identifiers of adjacent edges are the same; border edges wrap around. The grid is then randomly scrambled and placed in a linked list. The benchmark then takes the first tile and performs a linear search for the tiles matching each edge. As each edge is found, the match is recorded in both tiles. When all four edges are matched, the next tile is taken. This continues until all edges of all tiles have been matched. The implementation stores 128 tiles in a single Mungi object, with a randomly assigned type label. The client has read permission to all types.

For the Andrew benchmark, each component-instance is assigned a different type label, and all files within a single directory are assigned the same domain label. Mungi objects containing the actual file data are assigned a 'public' type, available to any domain. To investigate the relative performance of memory-intensive and `invoke` intensive programs, the Andrew benchmark is executed three times: with all validations, with `PdxCall()` validations disabled, and with page fault validations disabled (`PdxCall()` enabled).

Table 9.2 shows the performance of each benchmark without mandatory access control (MAC) implemented, with MAC implemented without a validation cache, and with the complete MAC implementation. Results are the average of 10 executions, and are shown in milli-seconds. Values in brackets show the access control overhead as a percentage of the performance without access control. Despite its random nature, results for the Jigsaw benchmark exhibited a standard deviation of only 3.8, or 1.3%.

Benchmark	No MAC	MAC, wo/Cache	MAC, w/Cache
OO1	187.8	411 (218%)	187.8 (0%)
$\text{Jigsaw}_{56\times56}$	374	489 (30%)	375 (0.3%)
Andrew	672	923 (37%)	674 (0.3%)
Andrew (no `invoke` vals.)	672	720 (7.2%)	674 (0.3%)
Andrew (no PF vals.)	672	879 (31%)	673 (0.2%)

Table 9.2: Macro-benchmark Access Control Overhead (ms)

Table 9.2 shows that end-to-end performance depends on the behaviour of the validation cache. With a validation cache, mandatory access control imposes an overhead of less than 0.3%, while without a validation cache, all benchmarks show an unacceptable overhead.

The results also show that memory-intensive programs, e.g. the Jigsaw benchmark and Andrew benchmark without `PdxCall()` validations, are less dependent on cache performance than `invoke`-intensive programs, e.g. the OO1 benchmark and Andrew benchmark without page fault validations. This is because page mappings, and hence their validations, are cached by the memory-management unit (MMU). Jigsaw performs worse than the Andrew benchmark (30% vs. 7.2%) because it traverses a randomly arranged linked list, therefore, it exhibits less spatial locality leading to more page faults.

Therefore, to determine the average end-to-end overhead imposed by mandatory access control, a 'typical' cache hit rate, especially for `invoke` operations, must be determined. Unfortunately, there are no results regarding protection cache hit rates publicly available, and time constraints prevent a complete investigation here. Instead, a brief discussion of the issues involved is presented below.

Performance of mandatory access control depends principally on the validation cache hit-rate. In general, there are three causes of cache misses.

- **Capacity** cache misses occur when the cache becomes full and entries must be discarded; a capacity miss occurs when a discarded entry is requested.
- **Cold** cache misses occur when an entry that has never been in the cache is requested.
- **Conflict** cache misses occur when two entries correspond to the same cache location. Even though the cache is not full, an entry must be discarded. A conflict occurs when the discarded entry is requested.

As the mandatory validation cache is implemented in software, a highly associative cache can be used so that conflict misses should not be an issue. Therefore, the cache performance depends on the number of component-instances with different type labels invoked (capacity misses), and how often these component-instances are changing (cold misses).

Existing component-based software, e.g. Apache, Netscape and Visual Basic, typically contain a set of core-components, and a set of independent extensible modules. Assuming that all core-components execute in the same protection domain, and each extensible module executes in its own protection domain, these programs would involve at most ten domains. Furthermore, while extensible software is inherently dynamic, component changes occur at a much lower rate than the average time between invoke operations.

9.4 Conclusion

Benchmarks show that the overhead of mandatory access control depends primarily on the validation cache hit rate, especially for invoke-intensive software. Existing software, including the evaluated benchmarks, demonstrate a cache hit rate approaching 100%, resulting in an end-to-end performance overhead of less that 0.3%.

Although these results are encouraging, current extensible software has been developed on monolithic operating system providing rudimentary protection mechanisms. How representative such software is of extensible services developed from scratch on a secure, extensible, operating system, is unknown. Therefore, a concrete evaluation requires workloads from a mature, truely extensible, multi-user, operating system providing mandatory access control. Unfortunately, no such system exist, yet.

Part III

Conclusion

Chapter 10

Conclusion

An extensible operating system need not be a compromise of performance, flexibility and safety.

Tim Wilkinson and Kevin Murray, 1994 [WM95]

Chapter 3 identified a set of requirements for extensible systems. This list describes how each of these requirements is satisfied by the architecture presented in this thesis.

- *Add services dynamically.* Component-classes become available to clients, and other components, the moment they are registered with the Component Locator.
- *Add services polymorphically.* MCS interfaces are inherently polymorphic, allowing clients to use new component-classes without modifying their code.
- *Be late-binding.* Interface objects determine the component-class of a CICAP dynamically using the Component Locator. Dynamic linking and loading are provided by `PdxCall()`, making MCS truely late binding.
- *Deploy updated versions.* MCS provides support for major and minor version changes. Clients transparently migrate to new minor versions, while multiple major versions can coexist. A field in the instance header, **VL**, indicates the instance's data format, so that new versions do not have to maintain binary compatibility.
- *Impose as few restrictions on programmers as possible.* Components naturally support a modular design. Apart from this, components do not impose any restrictions. In particular, components can be written in any 'unsafe' language.

- *Module extension.* Obviously, a new component-instance may invoke methods on an existing component-instance if it possesses its CICAP.
- *Perform fine-grained customisation.* Delegation allows component-instances to be customised at interface granularity.
- *Provide a semantic model, or standard, for extension construction.* Chapter 4 provides a detailed semantic model for component construction.
- *Provide acceptable performance.* Chapters 6 and 9 demonstrated that MCS provides the performance required for the development of secure system services. Benchmark results show that MCS performance is derived from three, equally important, factors: the excellent performance exhibited by Mungi, a lightweight component-model, and use of the single-address-space to allow bulk data sharing.
- *Transparent interposition.* Transparent interpositioning is a specific form of delegation, in which the overriding component-instance forwards requests to the component-instance being delegated, after performing some additional processing.
- *Execute in an amplified protection domain.* Components execute in an amplified protection domain that is the union of the protection context and a subset of the client's protection domain. As the protection context contains only those objects required for the service, and the client subset contains only the objects being operated on, components satisfy the principle of least privilege.
- *Store privileged data.* Privileged data can be stored in the component-instances, which cannot be directly accessed by clients.
- *Perform discretionary access control.* CICAPs provide discretionary access control for component-instances. A client holding an owner CICAP may register additional CICAPs with access to a subset of the component's interfaces.
- *Mechanisms allowing authorized extensions to perform privileged system operations.* All resources in Mungi are mapped into the single-address-space and can be addressed from every protection domain. Therefore, almost any system operation can be performed by a sufficiently privileged extension.
- *Adequate access control.* Chapter 7 presented a new access control model capable of providing effective security in extensible systems.
- *A minimal trusted computing base (TCB).* By presenting a pure single-address-space, and controlling access to this address space using password capabilities, Mungi provides uniform protection and a minimal TCB.

Above all, this thesis aimed to bring a large number of existing ideas together to produce a cohesive base for a safe, effective, and efficient extensible system. Where existing ideas proved inadequate new solutions were engineered. Although the principle contribution of this thesis is the whole rather than its parts, a few key points are stated below.

- Separating interfaces from component-classes results in a lightweight model.
- A component model must focus on the goals, rather than the traditional techniques, of component-oriented development.
- A common motivation based on flexibility leads to a natural synergy between extensible systems and single-address-space systems, that is demonstrated in Chapter 6.
- Software policies are required to enforce least privilege in extensible systems. Allowing services to assign security attributes in a system controlled manner, allows the specification and enforcement of such policies.
- As extensible systems represent an entirely new system model, new protection models that complement extensibility are required.

MCS proves that a component model, implemented using a trusted path mechanism, can provide a safe, effective and efficient architecture for system extensibility.

Part IV

Appendicies

Appendix A

Example Policy Objects

This appendix provides some example policy object implementations.

A.1 Standard Definitions

```
/*****************************************************************************
 * std_policyobject.h
 *
 * Standard definitions for policy objects
 *****************************************************************************/
#ifndef __STD_POLICYOBJECT_H__
#define __STD_POLICYOBJECT_H__

/* Validation descriptor */
typedef struct {
    unsigned long domain : 60;
    unsigned long access : 4;
    unsigned long dot : 60;
    unsigned long access2 : 1;
    unsigned long r     :  1;
    unsigned long fn    :  2;
} po_valdesc_t;
#define GET_ACCESS_MODE(v) (v.access<<1 | v.access2)

/* Access modes */
#define MAN_AM_READ 1
#define MAN_AM_WRITE 2
#define MAN_AM_EXECUTE 4
#define MAN_AM_DESTROY 8
#define MAN_AM_INVOKE 16

/* Policy functions */
#define MAN_PF_LEGALDOMAINS 0
#define MAN_PF_LEGALTYPES 1
#define MAN_PF_VALIDATEACCESS 2
#define MAN_PF_DOMAINSTACK 3

#endif /* __STD_POLICYOBJECT_H__ */
```

A.2 Denning's Lattice

A.2.1 Header

```
/*****************************************************************************
 *
 * spo_denning.h
 *
 * A policy object that implements the Denning's Lattice information flow model
 *
 *****************************************************************************/
#ifndef _SPO_DENNING_H_
#define _SPO_DENNING_H_

#include "mungi/man_security.h"

#define GET_CLEARANCE(label) (label&0x0000000000FFFFFF)
#define GET_COMPARTMENT(label) (label&0x0FFFFFFFFF000000)

po_valdesc_t spo_dl_legal_domains( po_valdesc_t );
po_valdesc_t spo_dl_legal_types( po_valdesc_t );
po_valdesc_t spo_dl_validate_access( po_valdesc_t );
po_valdesc_t spo_dl_domain_stack( po_valdesc_t );

#endif /* _SPO_DENNING_H_ */
```

A.2.2 Implementation

```
#include "spo_denning.h"

po_valdesc_t spo_dl_legal_domains( po_valdesc_t val )
{
    if( GET_CLEARANCE(val.domain) >= GET_CLEARANCE(val.dot) &&
        !((~GET_COMPARTMENT(val.domain))&GET_COMPARTMENT(val.dot)) )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_dl_legal_types( po_valdesc_t val )
{
    if( GET_CLEARANCE(val.domain) >= GET_CLEARANCE(val.dot) &&
        !((~GET_COMPARTMENT(val.domain))&GET_COMPARTMENT(val.dot)) )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_dl_validate_access( po_valdesc_t val )
{
    if( GET_CLEARANCE(val.domain) >= GET_CLEARANCE(val.dot) &&
        !((~GET_COMPARTMENT(val.domain))&GET_COMPARTMENT(val.dot)) )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_dl_domain_stack( po_valdesc_t )
{
    val.dot = val.domain;
    return val;
}
```

A.3 Chinese Wall

A.3.1 Header

```
/*****************************************************************************
 *
 * spo_chinese_wall.h
 *
 * A policy object that implements the Chinese Wall model.
 *
 *****************************************************************************/
#ifndef _SPO_CHINESE_WALL_H_
#define _SPO_CHINESE_WALL_H_

#include "mungi/std_policyobject.h"

#define DOM_SUPERUSER 0
#define TYP_PUBLIC 0

po_valdesc_t spo_cw_legal_domains( po_valdesc_t );
po_valdesc_t spo_cw_legal_types( po_valdesc_t );
po_valdesc_t spo_cw_validate_access( po_valdesc_t );
po_valdesc_t spo_cw_domain_stack( po_valdesc_t );

#endif /* _SPO_CHINESE_WALL_H_ */
```

A.3.2 Implementation

```
#include "spo_chinese_wall.h"
#include "avl.h"

po_valdesc_t spo_cw_legal_domains( po_valdesc_t val )
{
    if( val.domain == val.dot )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_cw_legal_types( po_valdesc_t val )
{
    /* users can only create unclassed objects, superuser can create any */
    if( val.domain == DOM_SUPERUSER || val.dot == TYP_PUBLIC )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_cw_validate_access( po_valdesc_t val )
{
    int already_exists = 0;

    /* entries are timed out of the avl tree */
    if( val.dot != TYP_PUBLIC )
        already_exists = avl_add( val.domain, val.dot );

    if( already_exists == 0 )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_cw_domain_stack( po_valdesc_t val )
{
    /* entries are stored in the public area. Protected by discretionary AC only */
    val.dot = 0;
    return val;
}
```

A.4 Bell LaPadula

A.4.1 Header

```
/******************************************************************************
 *
 * spo_bell_lapadula.h
 *
 * A policy object that implements the Bell LaPadula information flow model.
 *
 *****************************************************************************/
#ifndef _SPO_BELL_LAPADULA_H_
#define _SPO_BELL_LAPADULA_H_

#include "mungi/std_policyobject.h"

po_valdesc_t spo_bl_legal_domains( po_valdesc_t );
po_valdesc_t spo_bl_legal_types( po_valdesc_t );
po_valdesc_t spo_bl_validate_access( po_valdesc_t );
po_valdesc_t spo_bl_domain_stack( po_valdesc_t );

#endif /* _SPO_BELL_LAPADULA_H_ */
```

A.4.2 Implementation

```
#include "spo_bell_lapadula.h"

po_valdesc_t spo_bl_legal_domains( po_valdesc_t val )
{
    if( val.dot == val.domain )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_bl_legal_types( po_valdesc_t val )
{
    if( val.dot >= val.domain )
        val.r = 1;
    else
        val.r = 0;
    return val;
}

po_valdesc_t spo_bl_validate_access( po_valdesc_t val )
{
   char am;

   ma = GET_ACCESS_MODE(val);
   if( val.dot >= val.domain && ma == MAN_AM_WRITE )
       val.r = 1;
   else if( val.dot <= val.domain && ma == MAN_AM_READ )
       val.r = 1;
   else if( val.dot == val.domain )
       val.r = 1;
   else
       val.r = 0;
   return val;
}

po_valdesc_t spo_bl_domain_stack( po_valdesc_t val )
{
    /* as the client needs RW access, the stack must be at the same level */
    val.dot = val.domain;
    return val;
}
```

Bibliography

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[AC96] Martin Adabi and Luca Cardelli. *A Theory of Objects*. Springer Verlag, Berlin, 1996.

[AGS83] Stanley R. Ames, Morrie Gasser, and Roger R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, July 1983.

[APW86] M. Anderson, R.D. Pose, and C.S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, 1986.

[BALL89] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on OS Principles*, pages 102–113. ACM, December 1989.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA90, ACM SIGPLAN Notices, 25(10)*, pages 303–311, 1990.

[BK85] William Boebert and Robert Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, pages 18–27, 1985.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[BSP+95] Brian Bershad, Stephen Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995. ACM.

[BSS+95a] Lee Badger, Daniel Sterne, David Sherman, Kenneth Walker, and Sheila Haghighat. A domain and type enforcement unix prototype. In *Proceedings of the Fifth USENIX Unix Security Symposium*, pages 127–140, June 1995.

[BSS+95b] Lee Badger, Daniel Sterne, David Sherman, Kenneth Walker, and Sheila Haghighat. Practical domain and type enforcement for unix. In *Proceedings of the Symposium on Security and Privacy*, pages 66–77, May 1995.

[Bub99] Adam Buble. Comparing CORBA implementations. Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, 1999.

[Che94] David Cheriton. Low and high risk operating system architectures, (panel). In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, November 1994.

[CJ75] Ellis Cohen and David Jefferson. Protection in the HYDRA operating system. In *Proceedings of the 5th ACM Symposium on OS Principles*, pages 141–159. ACM, 1975.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[CM98] K. Mani Chandy and Jayadev Misra. *Parallel Program Design - A Foundation.* Addison-Wesley, Reading, MA, 1998.

[COD98] *The CODA Filesystem Manual*, October 1998.

[COM95] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification, ver 0.9*, October 1995.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[CSI] Center for Secure Information Systems, George Mason University. *Security Glossary.* http://www.isse.gmu.edu/~csis/glossary/merged_glossary.html.

[CU99] Object Management Group (OMG). *Charles University Response to the ORBOS Benchmark RFI*, October 1999. Technical Report: document bench/98-10-04.

[Cus93] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the Symposium on Security and Privacy*, pages 184–194, April 1987.

[Den76] Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–242, 1976.

[Dig92] Digital Equipment Corp., Maynard, MA, USA. *Alpha Architecture Handbook*, 1992.

[DoD85] United States Department of Defense (DoD). *Department of Defense Trusted Computer System Evaluation Criteria*, 1985. Technical Report: DOD 5200.28-STD.

[DVH66] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–154, March 1966.

[ES90] Margaret Ellis and Bjarn Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[Fab74] R.S. Fabry. Capability-based addressing. *Communications of the ACM*, 17:403–412, 1974.

[FCDR95] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95*, pages 426–438. ACM, 1995.

[FKJ+90] David Finkel, Robert E. Kinicki, Aju John, Bradford Nichols, and Somesh Rao. Developing benchmarks to measure the performance of the Mach operating system. In *Proc. of the USENIX Mach Workshop*, 1990.

[GB97a] Robert Grimm and Brian Bershad. Access control in extensible systems. Technical Report UW-CSE-97-11-01, University of Washington, Seattle, USA, 1997.

[GB97b] Robert Grimm and Brian Bershad. Security for extensible systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 62–66. ACM, May 1997.

[GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the Second ACM workshop on Role-based access control*, 1997.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, December 1997.

[GR89] Adele Goldberg and Kenneth Rubin. *Smalltalk - 80: The Language. Revised Edition.* Addison Wesley, Reading, MA, 1989.

[GSB+99] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HLP+00] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation.* USENIX/IEEE/ACM, 2000.

[HLR98] Gernot Heiser, Fondy Lam, and Stephen Russell. Resource management in the Mungi single-address-space operating system. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 417–428, Perth, Australia, February 1998. Springer-Verlag. Also available as UNSW-CSE-TR-9705 from http://www.cse.unsw.edu.au/school/research/tr.html.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17:549–57, 1974.

[Hol92] Ian M. Holland. Specifying reusable components using contracts. In *Proceedings of the 6th European Conference on Object Oriented Programming (ECOOP)*, 1992.

[HSH81] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 341–348. ACM/IEEE, May 1981.

[HVGR99] Gernot Heiser, Jerry Vochteloo, Emmanuel Galanos, and Stephen Russell. The Mungi kernel API, version 1.2. Technical Report UNSW-CSE-TR-DRAFT, University of NSW, University of NSW, Sydney 2052, Australia, September 1999. Available from: http://www.cse.unsw.edu.au/home/~disy.

[IA-00] Intel Corporation. *IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture*, January 2000.
URL http://developer.intel.com/design/ia-64/index.htm.

[IBM94] IBM. *The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developer's perspective*, 1994. White Paper.

[Jae98] Trent Jaeger. *Access control in configurable systems.* http://www.research.ibm.com/sawmill/index.html, September 1998.

[Jav97] *JavaBeans 1.0 Specification*, July 1997. http://java.sun.com/beans.

[JLI98] Trent Jaeger, Jochen Liedtke, and Nayeem Islam. Operating system protection for fine-grained programs. In *USENIX Security Symposium*, January 1998.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KL92] Gregor Kiczales and John Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92, ACM SIGPLAN Notices, 27(10)*, pages 435–451, 1992.

[Lad97] Mark Ladue. When Java was one: Threats from hostile byte code. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[Lam74] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[Lam93] John Lamping. Typing the specialisation interface. In *OOPSLA '93*, page 201, 1993.

[Lau00] William Lau. Distributed mungi. BE (Hons) thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, 2000. Available at: http://www.cse.unsw.edu.au/home/~disy.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[LFGS99] David Levine, Sergio Flores-Gaitan, and Douglas Schmidt. An empirical evaluation of OS support for real-time CORBA object request brokers. In *Real-time Technology and Applications Symposium (RTAS)*, June 1999.

[Lie92] Jochen Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992. Springer Verlag.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[Lie96] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Computer Science. Addison Wesley, may 1989. ISBN 0-201-06196-1.

[LS97] Emil C. Lupu and Morris Sloman. Reconciling role-based management and role-based access control. In *Proceedings of the Second ACM workshop on Role-based access control*, 1997.

[LSM+95] Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, Jeff Turner, and John Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, United Stated National Security Agency (NSA), 1995.

[Mey90] Bertrand Meyer. *Eiffel - The Language*. Prentice Hall, 1990.

[MF97] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley & Sons Inc., New York, USA, 1997.

[Min95] Spencer E. Minear. Providing policy control over object operations in a Mach-based system. In *Proceedings of the 5th USENIX UNIX Security Conference*, 1995.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralised model for information flow control. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 129–142, St. Malo, France, October 1997.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299, 1986.

[Nel91] Greg Nelson. *Programming in Modula-3*. Prentice Hall, 1991.

[Obe94] Oberon Microsystems. *Oberon/F Users Guide*, 1994. Available from: http://www.oberon.ch.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide*. John Wiley & Sons Inc., New York, 1996.

[OLW98] John K. Ousterhout, J. Y. Levy, and Brent B. Welch. The safe-TCL security model. In *Proceedings of the 23rd USENIX Annual Technical Conference*, 1998.

[OMG98] Object Management Group (OMG). *CORBAservices: Common Object Services Specification*, December 1998.

[OMG99a] Distributed Systems Research Group, Charles University. *CORBA Comparison Project: Project Extension Final Report*, August 1999. http://nenya.ms.mff.cuni.cz.

[OMG99b] Object Management Group (OMG). *C Language Mapping Specification*, June 1999.

[OMG99c] Object Management Group (OMG). *The Common Object Request Broker: Specification, ver 2.3.1*, October 1999.

[OMG99d] Object Management Group (OMG). *CORBA Components: Joint Revised Submission*, October 1999. Technical Report: orbos/99-02-05.

[Org93] Elliott I. Organick. *A programmer's view of the Intel 432 System*. McGraw-Hill, 1993.

[OSD94] Radical operating system structures for extensibility: A panel session. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 195–199, November 1994.

[PB96] Przemyslaw Pardyak and Brian Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[RA85] John Rosenberg and David Abramson. MONADS-PC—a capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, volume 1, pages 222–31. IEEE, 1985.

[RS97] Paul Roe and Clemens Szyperski. Lightweight parametric polymorphism. *Lecture Notes in Computer Science*, No. 1204:pg. 140–154, 1997.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21:91–118, January 1991.

[San93] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[San95] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the First ACM workshop on Role-based access control*, 1995.

[SBC+97] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of the Second ACM workshop on Role-based access control*, 1997.

[SCC97] Secure Computing Corporation. *DTOS General System Security and Assurability Assessment Reports*, June 1997. Technical report MD A904-93-C-4209 CDRL A011, (http://www.securecomputing.com/randt/HTML/dtos.html).

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, October 1996. USENIX Assoc.

[Sin97] Pradeep Sinha. *Distributed Operating Systems*. IEEE Computer Society Press, 1997.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86*, pages 38–45. ACM, November 1986.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, USA, 1996.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[Ste87] Lynn A. Stein. Delegation is inheritance. In *OOPSLA87, ACM SIGPLAN Notices, 22(12)*, pages 138–146, October 1987.

[Sun85] Sun Microsystems, Mountain View, CA, USA. *Remote Procedure Call Protocol Specification*, 1985.

[Sut97] Jeff Sutherland. Why I love the OMG. *Homepage_Journal*, 1997.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York and Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[TOP99] Jonathon Tidswell, Geoffrey Outhred, and John Potter. Dynamic rights: Safe extensible access control. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 113–120, October 1999.

[Ude94] Jon Udell. Componentware. *BYTE Magazine*, 19(5):46–56, May 1994.

[Val95] Ray Valdes. Interoperable objects. *Dr. Dobb's Journal*, January 1995.

[VERH96] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 161–165, Seattle, WA, USA, October 1996. IEEE.

[Voc98] Jerry Vochteloo. *Design, Implementation and Performance of Protection in the Mungi Single-Address-Space Operating System*. Phd thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, July 1998. Available from http://www.cse.unsw.edu.au/‘disy/papers/.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 203–216, Asheville, NC, USA, December 1993. ACM.

[WM95] Tim Wilkinson and Kevin Murray. Extensible, flexible and secure services in Angel, a single address space operating system. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 755–758, Brisbane, Australia, April 1995. IEEE.

[WMR+95] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Single address space operating systems. Technical Report UNSW-CSE-TR-9504, University of NSW, University of NSW, Sydney 2052, Australia, November 1995.