Easing the creation and maintenance of software systems through the use of domain machines

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/12694](https://dspace.lboro.ac.uk/2134/12694)

Publisher: © Ian A. Coutts

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
Easing the Creation and Maintenance of Software Systems through the use of Domain Machines

Ian A Coutts

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy of Loughborough University

August 2003

© Ian A Coutts 2003
Synopsis

Current approaches to the realisation of software systems employ elegant domain abstractions to handle system complexity. However, we do not preserve these abstractions within the systems we create: rather, we defer to the language imposed by the underlying architecture of computer hardware. Consequently, the intended purpose of much of the software which comprises these systems becomes lost. This results in software systems which require an in-depth knowledge of the specific implementation in order to support subsequent change.

It is the use of computers in "real world" systems whose scale and complexity is orders of magnitude greater than was originally anticipated by the inventors of software languages that has contributed to the problem. In order to address this problem, research into the application of novel techniques for large scale system realisation is needed.

This research investigates the creation and use of domain machines which directly execute domain abstractions, thereby preserving the structure and intent of the original system design. The author proposes a framework for the realisation of domain machines which when populated can provide separate specialised support for the particular characteristics of a software system design.

The proposed framework is applied to two separate and very different case studies that are both: on a scale with systems in industry and commerce; and are tested through integration with an industrial/commercial software environment.

The research proves that adopting the framework is an improvement on traditional approaches to large scale system construction and evolution, and also that the framework is not specific to a single application domain.
## Contents

Chapter 1 - Introduction: Computer Systems .......................................................... 1  
  1.1 Composition of Computer Systems ................................................................. 1  
  1.2 Creating Computer Systems ............................................................................. 4  
  1.3 Modifying Computer Systems .......................................................................... 6  
  1.4 Research Requirements ..................................................................................... 9  

Chapter 2 - Implementing Software Systems ............................................................ 11  
  2.1 The Development of Programming Languages .................................................. 11  
     2.1.1 First Generation Imperative Languages .................................................... 11  
     2.1.2 Second Generation Imperative Languages ............................................... 12  
     2.1.3 Third Generation Imperative Languages .................................................. 13  
     2.1.4 Object Oriented Languages ...................................................................... 14  
     2.1.5 Imperative Programming Languages ....................................................... 15  
     2.1.6 Functional and Logic Programming ............................................................ 15  
     2.1.7 Domain Specific Languages ...................................................................... 16  
  2.2 Building Complex Software Systems ................................................................ 17  
     2.2.1 Component Interaction .............................................................................. 17  
     2.2.2 Component Behaviour .............................................................................. 21  
  2.3 Current Support for Software System Creation and Change ............................. 24  

Chapter 3 - Domain Machines .................................................................................. 26  
  3.1 The Problem ...................................................................................................... 26  
  3.2 Mind the Gap ..................................................................................................... 27  
  3.3 Creating Domain Machines .............................................................................. 28  
  3.4 Benefits of the Approach ................................................................................. 30  
  3.5 Proposed Framework for Domain Machine Creation ......................................... 32  
  3.6 Experimental Work ............................................................................................ 34  

Chapter 4 - Manufacturing Control Systems ............................................................ 35  
  4.1 Domain Concepts .............................................................................................. 35  
  4.2 Characteristics of Distributed Control Systems ................................................. 36  
  4.3 Support for the Implementation of Component Actions ..................................... 37  
  4.4 Supporting Component Interaction .................................................................... 40  
  4.5 Supporting Component Behaviour ..................................................................... 42  
  4.6 Separating Manufacturing Concepts from the Computer System ..................... 45  
  4.7 Creating a Prototype Distributed Control Machine ............................................ 47  

Chapter 5 - Supporting Component Interaction ....................................................... 49  
  5.1 Definition of Interaction Constructs .................................................................. 49  
  5.2 An Approach to the Realisation of Interaction Constructs ................................. 53  
  5.3 The Distribution Infrastructure Interface .......................................................... 55  
  5.4 The Process Control Interface .......................................................................... 59  
  5.5 The Interaction Service Interface ...................................................................... 60  
  5.6 The Interaction Construct Layer ....................................................................... 63  
  5.7 Concluding Remarks .......................................................................................... 65  

Chapter 6 - Supporting Component Behaviour ....................................................... 66  
  6.1 Definition of Behaviour Constructs .................................................................. 66  
  6.2 An Approach to the Realisation of Behaviour Constructs ................................... 72  
  6.3 Validation of Behavioural Models ..................................................................... 74  
  6.4 Execution of Behavioural Models ...................................................................... 75  
  6.5 Concluding Remarks .......................................................................................... 79
List of Figures

Figure 1. The Composition of Computer Systems ........................................... 2
Figure 2. Computer System Software and Computing Machine ............................ 3
Figure 3. Computer System Creation .......................................................... 6
Figure 4. Computer System Modification ...................................................... 8
Figure 5. Raising the Level of Infrastructure ............................................... 29
Figure 6. Domain Machine Implementation Issues ......................................... 31
Figure 7. Framework For Domain Machine Creation ....................................... 33
Figure 8. A Typical Manufacturing System .................................................. 35
Figure 9. Characteristics of Software Systems ............................................. 38
Figure 10. Proof tree used to perform reverse([a,b,c],RevList) ......................... 47
Figure 11. Framework for a Manufacturing Control Domain Machine ................. 48
Figure 12. Component Interaction Constructs ............................................. 50
Figure 13. Simple Synchronisation Extra Logical Predicates ............................ 51
Figure 14. Support for Component Interaction within the Manufacturing Control Domain Machine ................................................................. 54
Figure 15. Distribution Infrastructure Interface Services .................................. 56
Figure 16. CIM-BIOSYS Management and Interaction Service Types ................. 57
Figure 17. Orbix Bind Function .................................................................... 59
Figure 18. Process Control Service Functions ............................................... 60
Figure 19. General Management Functions of the Interaction Service Interface .... 62
Figure 20. Interaction Service Functions of the Interaction Service Interface ....... 62
Figure 21. Response Service Functions of the Interaction Service Interface ......... 63
Figure 22. Declarative Behaviour Constructs ................................................ 69
Figure 23. Time Interval Constructs ............................................................. 69
Figure 24. Support for Component Behaviour within the Manufacturing Control Domain Machine ................................................................. 74
Figure 25. Solution to the Dining Philosophers problem .................................. 80
Figure 26. Conventional and Proposed Methods of System Implementation ......... 82
Figure 27. UML (Version 0.8) Concurrent Object Message Diagram .................. 85
Figure 28. UML (Version 0.8) State Diagram for the Component assembly .......... 86
Figure 29. Interpreted Petri Net for the Component print ................................ 87
Figure 30. Proposed Change to Assembly System ......................................... 98
Figure 31. UML State Diagram for New Assembly Component ....................... 99
Figure 32. Relationship between Banking Information and Process Support Systems and the Real World of Banking ................................................... 110
Figure 33. Framework for a Banking Domain Machine .................................... 117
Figure 34. Support for Banking Domain Process Models .................................. 120
Figure 35. Support for Banking Information Transactions ................................ 129
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrency</td>
<td>The simultaneous execution of system component actions.</td>
</tr>
<tr>
<td>components</td>
<td>The parts of a computer system that are combined to comprise a complex whole.</td>
</tr>
<tr>
<td>component actions</td>
<td>The discrete operations which do not involve other system components.</td>
</tr>
<tr>
<td>component behaviour</td>
<td>The manner in which a system’s components act and interact.icates.</td>
</tr>
<tr>
<td>component interactions</td>
<td>The invocation of component actions by other system components.</td>
</tr>
<tr>
<td>computer infrastructure</td>
<td>A combination of computer hardware and operating system programs.</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language: a programming language whose constructs are based on the principle concepts within a particular application domain.</td>
</tr>
<tr>
<td>declarative language</td>
<td>A programming approach which concentrates on the essentials of a problem with little regard to operational detail.</td>
</tr>
<tr>
<td>distributed system</td>
<td>A system comprising physically remote components which operate concurrently.</td>
</tr>
<tr>
<td>distribution infrastructure</td>
<td>An infrastructure to support component interaction within distributed systems.</td>
</tr>
<tr>
<td>embedded DSL</td>
<td>A DSL in which a domain-specific vocabulary is introduced into an existing host programming language.</td>
</tr>
<tr>
<td>imperative language</td>
<td>A programming language that depends heavily upon the assignment statement and read/write memory for accomplishing individual tasks.</td>
</tr>
<tr>
<td>integrated DSL</td>
<td>A DSL which combines domain-specific vocabulary, and a domain-independent way of composing complex things from simpler ones.</td>
</tr>
<tr>
<td>marshalling</td>
<td>The process of taking a collection of programming abstractions and assembling them into a form suitable for communication.</td>
</tr>
<tr>
<td>software system</td>
<td>A system comprising high level programs, encoded using programming language constructs which are automatically transformed into computer readable form.</td>
</tr>
<tr>
<td>transaction</td>
<td>A activity involving two parties that reciprocally affect or influence each other.</td>
</tr>
</tbody>
</table>
Chapter 1 - Introduction: Computer Systems

Computer systems created to support human activity require enormous quantities of human effort to construct and maintain [Brooks95], efforts which too frequently result in spectacular failures [Hatton97, Lions96, Leveson93].

"£1.5bn squandered on government IT: The cost of cancelled or over budget government IT projects has now topped £1.5bn in the last 6 years" [Computing03].

This chapter identifies characteristics of computer systems, the issues concerning their creation and modification, and highlights key problem areas which provide the rationale for the research reported in this thesis.

1.1 Composition of Computer Systems

When considering the issues concerned with the creation and modification of computer systems it is helpful to define their composition and boundaries. Jackson draws an important line between computer systems and the real world in which they exist

"The real world provides the subject matter for the system: it contains the engines to be controlled, the employees to be paid, the customers and suppliers whose transactions are to be accounted for. The system itself consists of computer and manual procedures and hardware; we think of it having a definite boundary - the system boundary - across which inputs and outputs flow between the real world and the system" [Jackson83].

Essentially computer systems are machines made up of hardware and software which exist within and interface to the real world. The hardware typically comprises computer processors and associated digital electronics, peripherals such as data storage devices, communication networks to enable individual computers to transmit and receive data and occasionally special devices such as robotic manipulators. The types of computers within any system may vary from general purpose machines, e.g. personal computers or mainframes, to dedicated control devices used for a single application, e.g. car assembly. Although these various computers are seemingly diverse in construction and application they are generally based on the concepts
introduced in the late 1940s by John von Neumann and his colleagues. These concepts are embodied within a device called the von Neumann machine, which is characterised by a large uniform store of memory cells and a processing unit with local cells, called registers. The processing unit is capable of loading data from memory to registers, performing arithmetic or logical operations on registers, and storing the values of registers back into memory.

Computer system software consists of programs, which are defined as precise representations of algorithms written in computer readable languages [Harel87a]. An algorithm being a sequence of computational steps (not necessarily executed by a computer) which transform some input values to output values [Cormen90]. A program consists of a sequence of instructions to perform arithmetic or logical operations and an additional set of control instructions, which can affect the next instruction to be executed, possibly depending on the content of some register. The programs which comprise the computer system software are used to control the operation of its underlying hardware.

A computer system's software can be conceptually separated into the programs concerned with the "housekeeping operations" of the hardware, commonly referred to as the computer operating system and the programs which define given tasks or applications. This separation is depicted in Figure 1.

Figure 1. The Composition of Computer Systems
The success of the computer is primarily due to its flexibility, as it can be programmed via its software to perform many different tasks. Such tasks being encoded into programs which are subsequently executed by the computer. When constructing programs to perform a task, the operating system programs and computer hardware are often viewed as subordinate to the undertaking in hand and can thus be considered as part of the computer infrastructure. This infrastructure may be viewed as a relatively static element of a computer system as, although change does occur, it is relatively infrequent compared to the change exhibited by task/application programs within a computer system. The programs which utilise the computer infrastructure can then be considered to define the computer system. This decomposition is depicted in Figure 2.

![Programs which define a Computer System](image)

*Figure 2. Computer System Software and Computing Machine*

This collection of programs which define the computer system is usually complex, as Brooks observes:

"Digital computers are themselves more complex than most things people build. They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do” [Brooks87].

For what might be considered a relatively mundane task, complex software may be required to decompose the task into the set of bit operations required in order for a computer to execute it. As Harel observes
"A computer can directly execute only a small number of extremely trivial operations, like flipping, zeroing, or testing a bit" [Harel87a].

To realise the sophisticated computer systems we desire today from such trivial operations is therefore a complex task which leads Brooks to observe

"The complexity of software is an essential property, not an accidental one" [Brooks87].

1.2 Creating Computer Systems

Many different definitions exist for the individual stages which transform a requirement into a system. Booch identifies design as a key phase

"In every engineering discipline, design encompasses the disciplined approach we use to invent a solution for some problem, thus providing a path from requirements to implementation" [Booch94].

The design process must consider the requirements for a system and create an abstract solution to satisfy them. The abstract nature of solutions within the design phase provides the means for handling the inherent complexity of computer systems. As Shaw states

"We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalised, idealised model of the object" [Shaw81].

The design process also adopts the technique of decomposition to complement the abstractions provided. Decomposition provides a means of

"not thinking about everything at once, but structuring a complex topic as a number of simpler topics that you consider separately" [Jackson95].

Using these techniques the design process handles system complexity and provides a path from requirements to implementation. During the design stage of the computer system life cycle, system components are identified and the relationships between components are defined to facilitate their integration. The means adopted
to realise the computer system is determined within the implementation stage of the life cycle, which transforms a set of abstract solutions into a set of programs for subsequent execution by the computer infrastructure.

The transformation achieved during system implementation is typically broken down into a number of distinct smaller transformations. Due to the availability of high level programming languages, described by Ben-Ari as

"an abstraction mechanism. It enables a programmer to specify a computation abstractly, and to let a program (usually called an assembler, compiler or interpreter) implement the specification in the detailed form needed for execution on a computer" [Ben-Ari96].

The transformation from system design to a computer system can be achieved by encoding the design within a programming language and then automatically transforming it into computer readable form. Such confidence is placed in programming languages that the high level programs, considered as the software system, are often considered to define the computer system, rather than the computer programs generated from them. As the task of transformation is performed automatically using a computer and consumes relatively little human effort, the encoding of the design into a programming language then becomes the main task within the system implementation stage. This process is depicted in Figure 3.

To create the software system, the abstract solutions held within the system design are transformed into the expressions and statements offered by the programming languages adopted. In addition, to integrate the distinct system components as described within the system design, facilities offered by the computer infrastructure, such as communication networks and protocols, are used.

The process of encoding the system design into a suite of high level programs for subsequent transformation and execution is a human-intensive and complex task. It is this task which has troubled the many research communities involved with computer systems’ creation during recent years and is generally known as the “software crisis”. It is a task which demands detailed computer systems knowledge but is often undertaken by available inexperienced developers in a rapidly growing industry with a large shortage of experienced staff. For many reasons, it is during this process
Introduction: Computer Systems

System Design

Solution in terms of the problem domain

Informal Transformation

Software System

Solution encoded using programming language constructs

Formal Transformation

Computer System

Solution encoded using bit/byte operations

Operating System

Processor & Peripherals

Operating System

Processor & Peripherals

Operating System

Processor & Peripherals

Communication Network

Computer Infrastructure

Figure 3. Computer System Creation

of transforming design into an implemented software system that many of the abstractions and decompositions used within the design to handle system complexity are lost.

1.3 Modifying Computer Systems

Once a computer system has been created and is operational the requirement for maintenance arises. Pressman defines maintainability as

"Maintainability of software is the ease with which software can be understood, corrected, adapted, and/or enhanced" [Pressman92].
Fenton [Fenton96] classifies three types of maintenance:

1. Corrective - this is fault finding and fixing;
2. Adaptive - modifying software to interface with a changing environment, and;
3. Perfective - adding new functionality to a working piece of software.

Corrective maintenance can be considered as part of the system creation process and is often a tiresome but finite task. The other two types, however, assume that computer systems are not replaced wholesale when a design or implementation change is required but evolve in some fashion. As Booch observes

"Because a large software system is a capital investment, we cannot afford to scrap an existing system every time its requirements change. Planned or not, large systems tend to evolve over time” [Booch94].

The study of the dynamics of computer systems has resulted in the derivation of a number of “laws” based on observation, these are known as Lehman’s laws [Lehman85]. The first law states that system requirements will always change and so a system must evolve if it is to remain useful. The second law states that as a system is changed its structure is degraded. The relationship between a system design change and a change to the system itself is depicted in Figure 4. The figure shows how change results in a further loss of design integrity.

As its name implies the main strength of software is flexibility, therefore implementing a design change within a software system should not pose any difficulty. This however is not the case, as Cox observes

"There was a time when the virtue of software over physical media like paper and pencil was in its very responsiveness... Although this may be to some extent true for small projects (program building), it is not (and has never been) true for ambitious undertakings (system building). In fact, software systems are usually the least responsive element in many organisations today. The organisation as a whole is able to adapt more fluidly than the software upon which it has grown dependent” [Cox87].

Software, although general purpose enough to handle many of the issues related to implementing a system design, does not directly support the abstractions and decompositions employed within a design. The resulting system components are
New system design must be mapped onto original system software, in order to create new system software, resulting in further loss of design integrity.

"If you take a typical COBOL, FORTRAN or BASIC program, and try to read it seeking the original intention of its designers, you cannot. The semantics, the meaning, of the specification is lost irretrievably in the process of converting the specification to design and then code" [Graham94].
Such programming languages were devised for the creation of general purpose computer programs not systems, as Cox explains

"The primary difference between programs and systems is in their ability to adapt to change. Programs are generally small, lightweight targets, and they can be relatively nimble and adapt as fast as conditions change. But large software systems are anything but responsive. They are large, complex, and brittle, easily disrupted by the slightest impact from outside" [Cox87].

This phenomenon of small, seemingly localised change causing unforeseen effects throughout a software system has been termed the "ripple effect" [Bennett96], and is additional evidence of the failure of a software system to retain the flexibility inherent in individual computer programs.

1.4 Research Requirements

Our current approach to the realisation of software systems, outlined within this chapter, fails to retain the abstractions and decompositions essential to handle their complexity. Important system characteristics are lost within a maze of programming language constructs required to define the actions of individual programs. This loss occurs because the encoding of system characteristics into a suitable software representation is designated as a programming task and, as such, is influenced by individual preferences and idioms.

This loss of abstraction and decomposition within the software systems we build causes serious problems, particularly when modifying computer systems to meet the inevitable changes in system requirements. The loss of abstraction and decomposition within software systems provides the rationale for the research reported in this thesis. The research aims to investigate a means to:

• ease the creation of software systems, and;
• ease the modification of software systems.

This is to be accomplished by aligning the abstractions and decompositions used to implement software systems with those described within their design.

It is the author’s contention that the problem described in this chapter is an engineering problem. It is the use of computers within "real world" systems, whose scale and complexity are orders of magnitude greater than was ever anticipated by
those responsible for the original design of hardware and software components, that has contributed to the problem. It can be argued that the solution to the problem lies in applied software systems engineering research. The research described in this thesis is based on building and testing novel proof of concept implementations that are both on a scale with those in industry and commerce, and interfaced and tested in an industrial or commercial environment. Previous and parallel work in computer science and software engineering has of course influenced the work. The following chapter describes some of the work which has influenced contemporary software systems implementation.
Chapter 2 - Implementing Software Systems

In the previous chapter the author identified the need to ease the creation and modification of software systems. The study presented in this chapter covers the evolution of research aimed at addressing this need. The broad range of research that has been conducted in this field and its relevance to the work described in this thesis is classified as follows:

- the development of programming languages, and;
- the building of complex software systems.

The study covers a long time frame as the manner in which programming languages have evolved, usually in response to issues raised during their application to real world problems, is of direct relevance to the research conducted in this thesis.

2.1 The Development of Programming Languages

Software systems are defined using programming languages which provide abstractions to overcome the complexities involved with the construction of executable computer programs. The following section briefly outlines the evolution of programming languages.

2.1.1 First Generation Imperative Languages

The first programming languages were developed in the 1950s, in an attempt to overcome the complexity associated with producing the many instructions required to program a computer to perform a specific task. Initially pseudo-code interpreters [Wilkes51] were developed: pseudo-codes are instruction codes that are different to those provided by the computer, they implement a virtual computer with its own set of data types and operations and so provide facilities considered suitable to the task in hand. The cost of such a facility was speed of operation as each high level operation must be interpreted and converted to the native instructions of the computer before it could be executed. During the late 1950s program language compilers were introduced to overcome this run-time inefficiency. These languages, subsequently classified as first generation programming languages, allowed the programmer to use a mathematical notation to produce a computer program [Backus78a]. Two of the most widely known first generation programming languages are IBM’s Mathematical FORmula TRANslating System (FORTRAN) [IBM56] and the COmmon
Business Oriented Language (COBOL) [US61], a language intended for commercial record-based applications. These first generation programming languages were divided into two parts: a “declarative” part, which described the data areas, their lengths, and their initial values; and an “imperative” part, which contained the commands to be executed during the running of the program and comprised computational statements, control flow statements, and input/output statements. Although easing the process of computer program creation, the structure of first generation programming languages is based on the hardware architecture of early electronic computers, and therefore their use still required a good understanding of the target computer on which the program was to be executed.

2.1.2 Second Generation Imperative Languages

The advent of second generation programming languages such as Algol 60 [Naur78], soon followed. These provided higher level structured control statements and by hierarchically structuring the control flow, eliminated much of the need for hardware influenced control instructions such as GOTOs. These languages were also free-format languages with machine independent lexical conventions and so could be expressed using context-free grammars [Chomsky59] such as Backus Naur Form (BNF) [Backus63].

Following the advent of second generation programming languages, greater emphasis was placed on the process of creating software, and by the early 1970s the term “software engineering” had been coined [Naur68]. At this time a “top-down” methodology emerged for constructing programs by progressing from a statement of the program’s objectives through successively more precise intermediate stages to final code [Dijkstra72, Wirth71]. This methodology still underpins many of today’s implemented systems. The success of the top-down method lies in the decomposition of complex problems into smaller, more independent ones. The main drawback to this method, not fully appreciated at the time, is that the final program does not preserve the series of abstractions through which it is created, which in turn increases the complexity of the task of modifying programs developed in this manner [Shaw80].
2.1.3 Third Generation Imperative Languages

Third generation programming languages, introduced in the 1970s, such as Pascal [ISO82] and C [Kernighan78] combined practical engineering principles with the technical achievements of second generation programming languages. The result was simple, efficient programming languages suitable for general purpose application. Many of these languages are still in use today. During this period the importance of organising programs into modules was recognised. These modules are larger entities that encapsulate both data and functionality, which lead to programming language support for abstract data types [Hoare72]. Language facilities were developed based on data type abstraction [Dahl68], ideas about strategies for defining modules [Parnas72] and concerns over program organisation [Dijkstra68a, Wulf73]. When adopting a data type abstraction, the functional properties and operations of a data structure are specified and then implemented in terms of existing language constructs and other data types. However for a data type abstraction to be used effectively, its specification must express all the information needed by the programmer who uses it [Shaw80]. If the programmer must understand its implementation in order to put it to use, then much of the power of the abstraction has been lost.

The late 1970s and early 1980s saw more improvements to third generation programming languages [Wirth77, Liskov77] which culminated in programming languages such as Ada [ANSI83] which support such concepts as data abstraction and concurrent programming. Essentially these programming languages can be seen as the culmination and fulfilment of the evolutionary process that began with FORTRAN [Mac1ennan87]. During this time issues such as heterogeneity, the ability for the same programming abstractions to be executable on different computer infrastructures, were addressed. So, as standardised programming languages have been used for system construction, problems associated with the heterogeneity of computer infrastructures have become less common. However, some differences in language implementation, underlying machine architecture or operating system may still pose problems [Sommerville96].

More recent developments in language design have evolved around a more comprehensive view of the programming process, and can be broken down into three main...
areas: object oriented programming, typified by languages such as Smalltalk [Goldburg83], C++ [Stroustrup91] and Java [Gosling99]; functional programming, typified by languages such as Miranda [Turner86]; and logic programming, typified by languages such as Prolog [Clocksin81].

2.1.4 Object Oriented Languages

Object oriented development consists of object oriented analysis, design and programming and attempts to ensure an object oriented strategy is used throughout the development process [Booch94]. A system based on objects is one whereby a computation is represented by a series of entities which interact to achieve a desired effect. An object is defined as an encapsulation of a set of operations or methods which can be invoked externally, and a state which remembers the effect of the methods [Blair91]. The intended role of an object oriented programming language is to make object oriented design easier to implement by supporting the direct realisation of objects identified during the design process by providing object classes and inheritance. However, an object oriented design may also be implemented using programming languages such as C or Pascal which do not directly support objects, but support abstract data types. Many popular object oriented languages, for example C++, whilst supporting the object paradigm do not necessarily impose strict adherence to it. This can lead to the rapid loss of the original design integrity when software is modified and may negate the advantages offered by the object oriented approach.

During the late 1990s Java [Arnold00] became a popular object oriented language. The language can be viewed as a distillation of a number of object oriented languages and arguably offers the best features of its competitors within a single language. However, one important feature in which it does differ from many other languages is the specification of a virtual machine language. The Java Virtual Machine (JVM) [Lindholm96] ensures that Java programs can be developed in isolation from the intended target computer infrastructure. It is the role of the JVM to ensure that the partially compiled programs (expressed in Java byte code) execute correctly within the target environment. Although the adoption of run-time code interpretation may be viewed as inefficient, the power of a platform-independent "virtual computer" has provided the Java programming language with an advantage
over other object oriented languages. Within the last few years JVMs have been deployed within web browsers, mobile phones, large web servers and many other diverse computing applications, thereby providing a known consistent target computing platform for the software system development process.

2.1.5 Imperative Programming Languages

Although increasingly abstract, the languages discussed so far carry the mark of the underlying computer on which they are to be executed - the von Neumann architecture. They are imperative languages and depend heavily upon the assignment statement and a read/write memory for accomplishing a programming task. Imperative languages are described by Backus in the following way

"Von Neumann programming languages use variables to imitate the computers' storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing and arithmetic" [Backus78b].

To provide an alternative means of computation, efforts focused on providing a means of creating computer programs that were independent of the underlying hardware architecture. This initiative gave rise to functional and logic programming.

2.1.6 Functional and Logic Programming

The central concept of functional programming is of applying a function to its arguments. It is an attempt to provide a more mathematical way of specifying computation. In a functional language, each expression denotes a pure value, which may be a primitive data element such as an integer, a function or a higher order function. Thus each expression can be thought of as a description of a static mathematical entity rather than a dynamic computation. Functional programming languages possess a high level of abstraction and so suppress many of the details found in imperative programming languages. The lack of assignment operations allows functional programs to be evaluated in many different orders, which makes them ideal for programming parallel computers. In addition, functional programs are more amenable to mathematical proof and analysis than imperative programs.

Logic programming also departs radically from the mainstream of computer languages. Rather than being derived via a set of abstractions from the von Neumann
machine and instruction set, it is derived from mathematical logic, which has no direct relation to one machine model or another. When using a logic programming language, explicit instructions for operation are not given, but rather the knowledge about a problem and the assumptions about it are stated explicitly as logical axioms (well-formed formulae which are taken to be true without proof in the construction of a theory). A logic program is a set of axioms, or rules, defining relations between things, its computation is a deduction of consequences of the program. In essence such a declarative programming approach concentrates on the essentials of a problem without too much involvement with operational detail.

Functional and logic programming languages provide a paradigm that is not modelled on the underlying hardware architecture on which the programs are executed. Also they have much to offer in enabling domain abstractions to be captured within a software system. Their properties of mathematical proof and analysis have led to their use for simulation and modelling, particularly within critical systems. However, general adoption of these languages has been poor due to the level of intellect required for their serious application and the lack of industrial-strength commercially available tools appropriate for building large systems. Other initiatives tasked with enabling the capture of domain concepts have focused on the generation of languages which are specific to particular domains. These initiatives have found a number of niche applications.

2.1.7 Domain Specific Languages

Domain specific languages (DSL) are programming languages whose constructs are based on the principle concepts within a particular domain and have proved themselves to be an effective mechanism for supporting some of the requirements for computer system evolution. They have achieved this through reduced development and maintenance time, systematic re-use and easier verification [DSL]. Common examples such as Lex and Yacc (for program lexing and parsing), VHDL (for hardware description) and HTML (for document mark-up) have demonstrated the effectiveness of DSLs. Ongoing research within the DSL community is investigating embedded and lightweight DSLs [Hudak97], deriving languages (for combining services on the Web) [Cardelli99], and DSL compiler/interpreter techniques. One area of research typified by the Compose project has defined a method and tools for
Implementing Software Systems

DSL development (SPRINT) [COMPOSE]. This work aims to support the creation of DSLs, which by their very nature depend on their domain of application.

2.2 Building Complex Software Systems

The previous section provided a brief overview of the development of programming languages which provide an abstracted means of creating computer programs. Within this section the means by which programming languages are used to implement the characteristics of software systems is discussed.

The actions performed by a system component can be viewed as the set of functions or services it provides to the rest of a computer system. The nature of these discrete services is akin to the functionality provided by individual computer programs, in that they are bounded, well defined and do not rely on interaction with other system components. The task of providing this functionality has been labelled “programming in the small” [DeRemer75] and is the problem the programming languages previously described were designed to solve. Component actions can therefore be represented as algorithms (which solve a given problem, perform a computation or provide a particular service) that can subsequently be transformed into a computer executable form [Harel87a].

Programming languages prove satisfactory for the implementation of component actions, but when we are required to create systems of interacting software components, labelled by DeRemer as “programming in the large” the abstractions provided by programming languages prove inadequate. As Wegner observes

"Programming in the large is inherently interactive and cannot be expressed by or reduced to programming in the small" [Wegner97].

Put simply, interacting concurrent processes cannot be expressed by sequential algorithms [Milner93] and so a distinction can be drawn between the provision of standalone functionality and integrated systems [Sloman87]. The following sections describe programming abstractions produced to support the additional characteristics required to define software systems.

2.2.1 Component Interaction

In order for computer programs to become components of a distributed software system they must interact in some way. Therefore, facilities for their interaction
must exist within the programming language representation of system components. These facilities must include communication, the exchange of data between components, and marshalling, the process of taking a collection of programming abstractions and assembling them into a form suitable for communication.

Distributed computer communications emerged with the introduction of high speed Local Area Networks (LANs) at the beginning of the 1970s. Initially the facilities offered by communication networks were designed into separate application programs such as terminal handlers and remote job entry handlers. This situation was quickly deemed unsuitable and research focused on providing computer communication facilities within existing operating systems [McQuillan77, Poncet76, Pouzin73, Cypser78, Wecker80, Metcalfe76]. An example of this early research is the extension to the Unix operating system [Ritchie74] to incorporate Inter Process Communication (IPC). This was initially achieved in the late 1970s with the BSD4 version of Unix developed at the University of California at Berkeley [Leffler89] and has subsequently been provided for many other computer operating systems. The IPC primitives in the BSD4 version of Unix are provided as operating system calls accessible via various programming languages and are implemented as a software layer over a number of communication protocols. The IPC operations are based on socket pairs, an abstracted communication channel, one socket belonging to each of a pair of communicating processes.

ISO Open Systems Interconnection: Following the rapid development of communication networks and protocols the International Standards Organisation (ISO) formed Subcommittee 16 to develop an architecture for open systems interconnection. The architecture was intended to serve as a framework for the definition of standards [IEEE83, Zimmerman80, ISO83]. This model, known as the ISO OSI reference model, has become an important means of classifying computer communication protocols. The 1980s saw initiatives to standardise protocol suites which populated the ISO OSI model. Initiatives such as Manufacturing Automation Protocol (MAP) and Technical Office Protocol (TOP) [Struckey89] attempted to specify networks and protocols for various application domains, as well as identifying high level services applicable to those domains. Of a more lasting influence, however, were efforts to simplify the use of the IPC mechanisms previously developed and produce facilities such as network wide shared file systems [SUN89].
Remote Procedure Calls: An example of these efforts is the Remote Procedure Calling mechanism (RPC) [Xerox81, Birrell84], designed to ease the production of client/server systems. In such systems clients invoke service operations by sending request messages to servers, which perform the requested operation and send a reply message back to the client. Meanwhile the client waits for the reply message before continuing its execution, even when no result is expected. RPC mechanisms integrate this client/server approach with conventional procedural programming languages. This enables clients to communicate with servers by calling procedures in a similar way to the conventional use of procedure calls in high level programming languages. RPC is modelled on the local procedure call, but the called procedure is executed in a different system component usually on a different host computer. RPCs fall into two classes:

1. the RPC mechanism is integrated with a particular programming language that includes a notation for defining interfaces [Shrivastava91, Liskov88], and;
2. a special-purpose interface definition language is used for describing the interfaces between clients and servers [SUN90, ANSA89, Jones86].

The 1980s saw a number of breakthroughs in the area of constructing distributed software systems; much of this work was influenced by theoretical advances [Hoare78, Milner80, Milner89] and the adoption of the ISO OSI reference model and its associated communication standards. New constructs were added to existing programming languages [Kramer83, Weston88] and new programming languages devised [Andrews82]; both approaches integrate communication protocols and programming languages, rather than using existing languages to utilise the communication facilities offered by a computer system infrastructure.

An example of this approach is the CONIC [Sloman85] programming language, an extension to Pascal which provides message-passing between separate software modules. CONIC systems are created by linking together modules to form interconnected components, each module possesses ports (which facilitate interaction via messages passing) and communication primitives to either send a message to an exit port, or receive a message from an entry port. CONIC’s underlying communication system supports a topology of interconnected independent LANs.
Open Distributed Processing: In 1987 work began on an ISO reference model for Open Distributed Processing (RM-ODP) [Brenner87, Hutchison91] which continued the OSI drive for openness within the development of distributed applications. The reference model for ODP was conceived with a much wider scope than that of OSI, and describes a system from five viewpoints [Linington92, ISO92):

- Enterprise - which focuses on purpose, scope and policies for the system;
- Information - which focuses on semantics of information and information processing;
- Computation - which focuses on functional decomposition into objects which interact at interfaces;
- Engineering - which focuses on mechanisms and functions required to support distributed interaction, and;
- Technology - which focuses on the choice of technology adopted.

Much of the research conducted in the 1980s addressed the problems of component distribution in a proprietary fashion with little emphasis on interworking with other solutions. An important concept reflected in the RM-ODP is the provision of an open infrastructure to support distributed systems, this can be thought of as a distribution infrastructure which complements an existing computer infrastructure.

Distribution Infrastructures: To date, a number of distribution infrastructures have been devised to support a technology independent means for system components to interact, examples include: Distributed Computing Environment (DCE) [OSF92]; Common Object Request Broker Architecture (CORBA) [OMG94]; Computer Integrated Manufacturing - Building Integrated Open SYStems (CIM-BIOSYS) [Gascoigne92]; and the Advanced Network Systems Architecture (ANSA) [ANSA93]. Of these infrastructures CORBA, based on the object oriented paradigm, has gained considerable multi-vendor backing. Within CORBA, objects use a Object Request Broker (ORB) to make and receive requests and to respond to other distributed objects. An object references another object by specifying an operation name, the target object reference (each CORBA object has a reference which uniquely identifies the object) and zero or more parameters. The outcome of a request is either a result or an exception. An object's interface is specified by defining a set of operation signatures which define the type of requests that can be made on an object satisfying that interface. OMG has defined an Interface Definition Lan-
guage (IDL), modelled on earlier RPC implementations, which is used to describe the interfaces that client objects call and server objects provide. Although considered an open approach, CORBA only establishes an object oriented enclave within a distributed system. The current lack of support for asynchronous message passing restricts system component implementation strategies [Sommerville96].

**Summary of Component Interaction:** Component interaction is facilitated via message passing. A message is sent by a source component via a physical connection, supported by the computer infrastructure, and received by the connected destination component. This is unidirectional interaction; bidirectional interaction permits a reply to be sent in response to a message received. Such interactions can also be dressed up as procedure calls or object method invocations, in order to provide a programming language abstraction. This approach is often adopted within client/server type system configurations.

### 2.2.2 Component Behaviour

The previous sections have described how programming languages facilitate the implementation of component actions and how the communication mechanisms provided by the computer infrastructure enable system components to interact. In order to integrate separate components to form a system, some means is required to implement the relationship between component actions and interactions. This is the implementation of component behaviour, the manner in which a component acts and reacts to external stimuli. In order to determine the behaviour of a system component, its possible states and state transitions which occur in response to events must be specified. In addition, the behaviour of a system component may be determined by the state of the components with which it interacts. As state information is usually distributed in space and time, synchronisation of behaviour is required to ensure that interacting system components share a consistent view of each of their states.

**Modelling Behaviour:** The specification and modelling of deterministic behaviour is a mature discipline. Alan Turing and Alonzo Church conducted pioneering research in the 1930s before the introduction of the computer. Among the many advances made was a means of describing the states and possible state transitions of a theoretical computation, both diagrammatically and mathematically. This work
gave rise to the theory of finite state machines [Harel87a]. Many behavioural specification notations now exist, examples include: Communicating Sequential Processes (CSP) [Hoare85], an algebra for concurrent interacting processes; Calculus of Communicating Systems (CCS) [Milner89], a theory developed on the semantics of indivisible interaction between system components; Petri nets [Petri73], a mathematical and graphical notation for modelling discrete event based systems; LOTOS [ISO89], a process algebra based on CCS which has been widely applied to the specification of communication protocols [Milner89]; Estelle [ISO86], a formal description technique based on an extended state transition model; and Statecharts [Harel87b], a simple but highly expressive notation for state transition diagrams which has been used within many software design methods [Booch94].

Although notations have been adopted to describe system component behaviour within the design stage of system creation, during system implementation they are invariably transformed, using ill-defined methods, into an algorithmic programming language representation, poorly suited for the task of expressing component behaviour [Wegner97, Milner93, Manna92]. As no explicit support for system-wide behaviour exists in today’s programming languages, the use of algorithmic control constructs, procedural decomposition and interaction mechanisms are adopted. This results in the semantics of component behaviour becoming hidden within computational based abstractions.

Synchronisation: Component behaviour must not only be specified, it must also synchronise component actions and interactions. According to Sloman [Sloman87], interacting components within a system use synchronisation mechanisms to coordinate access to shared resources or to ensure that events occur in some specific order. Bacon [Bacon93] sites two reasons for synchronisation mechanisms:

1. two or more components may need to co-operate in order to achieve a given task (this implies that the operating mechanism must provide facilities for identifying co-operating components and synchronisation of behaviour with each other), and;

2. two or more components may need to compete for access to shared services or resources (the main implication is that the synchronisation mechanism must provide facilities to allow one component to wait for a resource to become available and another component to signal the release of a resource).
Semaphores and Monitors: Early work in this area centred on techniques for concurrent programming. This is concerned with producing several sequential processes whose execution sequences are interlaced. The sequential programs are not independent, they must communicate with each other in order to synchronise or exchange data. Semaphores [Dijkstra68b, Dijkstra68c] were introduced as a simple yet sufficiently powerful means to synchronise tasks capable of sharing common memory within a single computer. This work developed into the concept of monitors [Brinch Hansen73, Hoare74]. A monitor is an abstract data type which provides routines guaranteed to be executed in mutual exclusion. A monitor can represent any resource that is shared by multiple tasks. All synchronisation and communication is achieved within the monitor and so potential concurrency errors are limited to the programming of the monitor itself, not the process it serves. The interface to a monitor is similar to that of an operating system in that a process calls the monitor to request or receive a service, and synchronisation among processes is automatically ensured. The major disadvantage of the monitor is that it is a centralised facility. Buhr [Buhr95] observes that communication among tasks using monitors requires the creation of a potentially large number of passive monitor objects, which must be properly managed within a system. Therefore, monitors are best used to serialise access to passive objects and resources, that do not have their own thread of execution.

Guarded Commands: Both the semaphore and the monitor are centralised facilities, whilst this is natural within a single computer, they are difficult to implement within a distributed system, as any synchronisation must be based on the sending and receiving of messages using the interaction facilities described within the previous section. There may indeed be no implicit synchronisation between sending and receiving a message, they may “pass” each other in transit, hence each component of a distributed system must decide for itself when to cease and resume execution of tasks. Guarded commands [Dijkstra75, Liskov83] were proposed as an alternative to monitors and provide for the selective execution of a statement depending on the value of a Boolean guard. Essentially, guarded commands can be used to define localised schedulers which block execution while a process is not in a state to execute selected actions. The concept of guarded commands has been combined with interaction capabilities within languages such as SR [Andrews82] and CSP.
Implementing Software Systems

[Hoare78] to provide guarded interaction language constructs. This concept of synchronised interaction, incorporated within CSP, has been used as the basis for the rendezvous mechanism of Ada. A rendezvous involves the transfer of information at a predetermined point within two interacting processes. The essence of any rendezvous is that the party which arrives earlier is required to wait.

**Summary of Component Behaviour:** Implementing component behaviour involves the coordination of actions and interactions of two or more software components with respect to time. Synchronisation is used to prevent interference when components access a shared resource, i.e. mutual exclusion, or make sure some actions performed by separate components are executed in a particular order. It should be noted that the approach to implementing component behaviour has been similar to that of implementing component interaction, in that extra constructs and data abstractions have been added to existing programming languages.

### 2.3 Current Support for Software System Creation and Change

This chapter has reviewed program language developments used to implement the different characteristics of software system components. It is clear that much has been achieved to facilitate the creation and modification of software systems.

Programming languages have developed to a point where high level abstractions remove the need for intimate knowledge of the underlying computing machine. The standardisation of such languages ensures portable solutions across many computer infrastructures [Sommerville96]. A number of distribution infrastructures exist which enable the construction of distributed interacting system components using heterogeneous computing platforms, and a number of synchronisation techniques have emerged to allow system components to cooperate and share common system resources.

However, although tackling different aspects of the system implementation problem, one underlying theme emerges. System components are being constructed using programming techniques and therefore systems are being constructed as large computer programs. Problems inherent with this approach have been identified regularly over the past three decades [DeRemer75, Sloman87, Milner93, Wegner97].
Although extensions and improvements have been made to programming languages and computer infrastructures over recent decades to tackle the problems associated with software system creation and change, they have only treated the symptoms of the problem and not the cause. Domain abstractions have been devised to unambiguously convey the essential characteristics of a system, but the construction of user defined type hierarchies supported by modern programming paradigms does not guarantee clarity of the original domain concepts within the implemented system. Such approaches introduce problems and complexities of their own as Brooks observes

"at some point the elaboration of a high level language creates a tool mastery burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs" [Brooks87].

The adoption of evermore elaborate computational constructs tends to obfuscate the intended role of software components, making system creation needlessly complex and subsequent modification unnecessarily difficult to achieve.

It is the author’s contention that a successful approach to software system creation may be provided through explicit support for separate domain characteristics. Domain-specific techniques have found success in niche areas (e.g. writing device drivers or program parsing). However, here we are concerned with software systems that support some activity within business and commerce. Our domain is typically a manufacturing production line, a retail stock control system or a foreign exchange desk in an investment bank, i.e. large-scale distributed systems.
Chapter 3 - Domain Machines

The previous two chapters have outlined 1) the problems of computer system creation and modification, and 2) approaches and techniques to aid the solution of these problems. This chapter introduces the author’s hypothesis that the use of a domain machine to enable direct execution of the domain abstractions within a design model will ease the creation and evolution of large-scale distributed systems.

3.1 The Problem

The previous two chapters have identified the following points:

- with the use of a suitable notation, the characteristics of a software system design are communicated using decomposition and abstraction;
- high-level computational abstractions, in the form of programming languages, have evolved to overcome the complexities associated with implementing computer programs;
- during the system implementation process the decompositions and abstractions held within a design are transformed into the abstractions offered by high level programming languages, and;
- programming language extensions and user-defined abstract data types are adopted to aid the implementation of system characteristics.

This approach to system creation is producing systems which are difficult to change. The requirement to adopt elaborate programming language extensions impedes the system creation/modification process by placing an intellectual burden on the system implementor. The characteristics of system components become hidden within a maze of computational abstractions, abstract data types and programmer-devised idioms. The semantics and integrity of the system design become lost within the implemented system.

Clearly a disparity exists between the form of a system design and the form required for its implementation. This disparity is currently negotiated using an intellectually intensive programming task, which fails to preserve a clear expression of system characteristics. This disparity provides the focus for the research reported within this thesis.
3.2 Mind the Gap

Obviously, the disparity between a system and its design is not new, indeed a number of research initiatives, under the umbrella title of operational approaches to software development [Zave84] have investigated concepts such as executable specifications, automatic program transformation and program generation. The operational approach involves formulating a system specification to provide an implementation independent solution to a given system requirement, and provided with a suitable interpreter, the execution of the specification can be simulated. The operational specification is later subjected to transformations that preserve system features but map the mechanisms adopted onto those of a high-level programming language, a process called system transformation and realisation. Automating the transformation and realisation process has become the major goal of research in this area, and progress has been made in a variety of application domains. Examples of operational approaches to system development include: the Jackson System Development Method [Jackson83] for constructing data processing systems; the Gist project [Balzer82] centred on the construction of artificial intelligence and database systems; the PAISLey project [Zave82] which focuses on real time embedded systems; and PROTOB [Brun086] aimed at constructing communication protocols and manufacturing control systems. These approaches have attempted to provide a formalism to transform a system design into the abstractions provided by existing programming languages. The operational approach also underpins research conducted into “automatic code generation”, where formal specifications are transformed into existing programming languages such as Ada [Sacha92], Pascal [Antoy87] and C [Valenzano90].

To overcome the disparity between a system and its design, these approaches usually constrain either the application domain or the abstractions and decompositions available for system design. Such approaches are limited by their reliance on existing implementation techniques and do not address the issue of preserving a clear representation of system characteristics within an implemented system. Therefore, the systems produced using such an approach still manifest themselves as large computer programs.
More recent research has attempted to support a "multiparadigm" approach to system creation [IEEE86]. The rationale behind this approach is described by Zave:

"Different aspects of a system require different approaches. But programmers are confined to their language’s one paradigm. Multiparadigm programming lets you match the paradigm to the problem" [Zave89].

Here the recognition has been made that different characteristics of software systems require different types of abstractions to support them. This leaves the question of how to combine the various paradigms at a conceptual level.

The term "software architecture" [Perry92, Garlan93] has been adopted to describe the issues related to supporting a conceptual view of software systems whilst also providing the means for their realisation. Research in this area has lead to a range of development environments for distributed software systems [Magee94, Luckham95, Garlan95] which propose and support various software system decompositions. Much of the support provided by this research manifests itself as either extensions to existing programming languages or new system programming languages. The emphasis has moved away from providing one all-purpose programming language towards the creation of multiple languages to perform a well defined but limited role within a software system.

It is this concept of separate specialised support for the separate characteristics of a software system design that underpins the approach to system creation proposed within this thesis.

**3.3 Creating Domain Machines**

The author proposes an approach to system creation which obviates the requirement to transform a system design into a single programming language representation by raising the "level" of computer infrastructure to a point where it supports the decompositions and abstractions contained within a system design.

The approach requires that an infrastructure supports the execution of component parts of a design, rather than providing a single facility for the execution of complete computer programs. As the infrastructure will be a relatively static and subordinate part of a system, the conceptual framework required to compose the separate
components of a system may be supplied by the system design. To implement a system, a design may be translated into a form whose execution is supported by the infrastructure, but the requirement to transform the design specifications into programming language constructs is removed. Therefore, the clear expression of system characteristics held within a design is retained within the implemented system. It is recognised that the component parts of a design that are required to be executed by the infrastructure could vary widely from one domain to another. It is therefore proposed that the additional raised level of infrastructure takes the form of a domain specific machine (domain machine). Figure 5 contrasts existing and proposed approaches to system implementation.
However, increasing the level of infrastructural support for a computer system requires a number of system implementation issues to be resolved:

- The infrastructure must support the decompositions and abstractions adopted within system design notations. Although a different syntax may be adopted for ease of execution, the underlying semantics must remain the same. Currently there is no agreed way to represent a system design, many different notations exist and it would not be feasible to support all notations. Many of the design notations are graphical in nature, although means are being produced to execute graphical languages they are still a subject for research in their own right [Burnett95].

- To transform design declarations into the realisation of what they describe, some formalism is required. The infrastructure may provide separate formalisms to support transformations of various design abstractions so long as the decompositions held within the design are not violated.

- System designs contain constructs which describe the essence of system component characteristics, but although such abstractions provide a powerful tool with which to design a system, they do not contain the means to achieve what they describe. This must be achieved via the computational abstractions and computer system infrastructure currently available.

These issues are superimposed onto the proposed domain machine in Figure 6.

Handling the above implementation issues within a domain machine removes them from the implementation process of individual systems, i.e. the solutions to the respective implementation problems are encapsulated within the domain machine. Realising an infrastructure capable of resolving the above implementation issues forms the main thrust of the work described within this thesis.

Given this, the objective of the research is to investigate the resolution of these three issues via the creation of prototypical domain machines, and to evaluate these machines through their application to real and significant software system creation and maintenance problems taken directly from industry.

3.4 Benefits of the Approach

Using the proposed approach to software system creation involves the translation of a system design into an executable form suitable for “embedding” within the
Same system design expressed using a different syntax, thereby retaining original design abstractions and decomposition

Separate formalisms which map design abstractions onto facilities offered by the computer infrastructure

Means to make it all happen i.e. read system design, apply correct formalisms and utilise the computer infrastructure in order to execute the system

**Figure 6. Domain Machine Implementation Issues**

domain machine. The translation process should be a simple mapping of design constructs, and therefore this process can be automated. A change required to an existing computer system based on this approach, can be facilitated via direct manipulation of the system design.

The mechanisms used to embed a particular design will reside within the domain machine. Any operational aspects of the system's run-time configuration will be provided as configuration options supported by the machine. This will include the choice of computer hardware, computer operating systems, communication networks, network protocols and distribution of system components. The design of a system component is directly executed by facilities provided by the domain machine and issues related to the operational aspects of the system execution are supported as configuration options, this obviates the problematic informal transformation stage, from design to software system, that was introduced in Chapter 1.
To summarise:
• the proposed approach eases the creation of software systems by eliminating the requirement to transform the decompositions and abstractions held within a system design into the computational abstractions offered by high-level programming languages, and;
• The task of subsequent system change is eased by preserving within the system the abstractions and decompositions adopted for its design, thereby retaining the design semantics.

3.5 Proposed Framework for Domain Machine Creation

Having identified our overall hypothesis of raising the level of infrastructure to ease system creation and change this section proposes a generic framework for the implementation of domain machines which fulfil this role. The structure of this framework feeds off successful layered models of the past, e.g. ISO OSI [ISO83], and tackles the three implementation issues identified in Section 3.2.

Central to this framework is the concept of a host language which itself provides infrastructural facilities within the domain machine. In his paper “The Next 700 Programming Languages” Landin [Landin66] stated that we can logically separate a language into two parts:

1. a domain-specific vocabulary, and;
2. a domain-independent way of composing complex things from simpler ones.

With these two language aspects in mind, there are two strategies we can adopt in realising a domain machine, which Elliott [Elliott99] defines as integrated and embedded. In the integrated approach, the domain-specific language combines both the language aspects. In the embedded approach the domain-specific vocabulary is introduced into an existing host programming language. While these two strategies may be similar in spirit, Elliott states that the pragmatics of implementing them vary considerably.

Using an integrated approach it is possible to create the perfectly suited domain language and associated machine. However this advantage comes with the heavy burden of having to construct most aspects of the language and machine from scratch. The main advantage of the embedded approach is that it utilises an existing language and its infrastructure and with it a great deal of existing language implemen-
tation knowledge and tools. Given the author’s requirement to provide an approach to the construction of domain machines that can be applied to real world applications (not just to the design of software engineering tools), the utilisation of proven languages and infrastructures was chosen.

In order to marry a selected host language with an existing computer infrastructure, the framework contains a lower layer, the role of which is to abstract any implementation requirements of the underlying computer infrastructure as cleanly as possible. The intent is to provide a means to implement the domain-specific aspects of the machine without regard to any given computer infrastructure. The combination of host language/environment, domain-specific language extensions and imperative application programming interface provides us with the framework shown in Figure 7.

![Figure 7. Framework For Domain Machine Creation](image)

The framework tackles the three implementation issues raised in Section 3.2 by:

1. Utilising domain-specific language extensions to support system design decompositions and abstractions;
2. Coupling the host language/environment to the computational abstractions provided by the computer infrastructure in order to provide the means of executing a system design, and;

3. Harnessing and extending the formalisms provided by the host language/infrastructure to transform various design abstractions into the underlying computational abstractions provided by the computer infrastructure.

3.6 Experimental Work

Having proposed the framework the research is aimed at proving two things:

1. That using the framework to raise the level of infrastructure to a point where design models can be directly executed is an improvement over traditional approaches to system construction and evolution, and;

2. That the framework is not specific to a single domain. It is beyond the scope of this thesis to prove that the framework is completely generic in nature.

In order to prove the two points above two separate but related experiments have been conducted. Both experiments involve the design and implementation of major software applications, these applications were developed within a university laboratory and deployed at the premises of sponsoring commercial companies.

The first experiment used the framework to build a manufacturing control domain machine and prove the benefits of direct design model execution over traditional approaches to software system realisation and modification. The second experiment used the framework to build a banking domain machine using different implementation technology to prove the general nature of the framework itself.
Chapter 4 - Manufacturing Control Systems

This chapter details the approach used to create a prototype manufacturing domain machine. Concepts and characteristics of the manufacturing control domain are discussed and attributes of manufacturing component systems are identified. These are defined as action, interaction and behaviour. The chapter continues by specifying the technology to be used in populating the proposed general framework for domain machines.

4.1 Domain Concepts

A modern manufacturing facility is a complex system consisting of many activities and components. In a Computer Integrated Manufacturing (CIM) plant many of the processes leading to the manufacture of a product are integrated and controlled by computer. There is a strong interdependence between all manufacturing functions, and information is passed back and forth between them. Every function and sub-function must be operating as an integral part of a whole system. In general within a CIM plant this complexity is handled through the adoption of a system hierarchy, as depicted in Figure 8.

![Figure 8. A Typical Manufacturing System](image-url)
The control of a manufacturing system requires the processing and exchange of a large amount of data. Usually, the flow of information between the components of a system is of such magnitude that real-time control is impossible [Jones89]. Such a system can only be controlled effectively if its activities operate in an independent manner. However, this is often contrary to the operational mode of a manufacturing plant where machine tools, material handling systems, robots and measuring machines have to be synchronised to obtain a linear flow operation [Brussel94]. The activities on the shop floor are the most critical ones with regard to synchronisation: here material and data are processed simultaneously and in real time. Typically when a hierarchical manufacturing control system is designed, a level of responsibility is assigned to each tier. The task of each activity can then be specified, the cooperation with external entities is defined and a centralised control strategy is conceived. In a CIM system these distributed control systems are realised through networks of computers.

4.2 Characteristics of Distributed Control Systems

Peterson suggests, a computer system can be considered as a set of interacting components, integrated to comprise the complex whole, and Peterson states that

"Each component may itself be a system, but its behaviour can be described independently of other components of the system, except for well-defined interactions with other components" [Peterson81].

Thus the behaviour of an integrated system component can be considered as the manner in which it acts and reacts to external stimuli generated by other system components or by the real world. The execution of component actions may, from a system point of view, be defined in isolation from other system components. However, as the actions performed by system components may occur simultaneously, i.e. a system may exhibit concurrency, and as the system components interact, it becomes necessary for synchronisation to occur. This leads Peterson to observe

"The timing of actions of different components may be very complex and the resulting interactions between components difficult to describe" [Peterson81].
An extra level of complexity appears when components are required to reside within different computers, or even in different geographical locations. This is distribution, defined by Harel as

"a special kind of concurrency, where the concurrent components are physically remote" [Harel87a].

Therefore computer systems can be considered as comprising a set of interacting components whose concurrent actions may be distributed and whose collective behaviour synchronises such actions to create a complex whole.

As previously stated, due to the availability of programming languages, a computer system may be defined by the software system from which it is created. The software system therefore must implement the actions, interactions and behaviour required of the computer system, where actions, interactions and behaviour can be considered as:

- Actions - discrete operations which do not involve other system components;
- Interactions - the invocation of component actions by other system components, and;
- Behaviour - the manner in which a system's components act and interact.

The software system may be visualised as one large program in which the characteristics described by the system design are realised using program language constructs. This view of a software system is depicted in Figure 9.

4.3 Support for the Implementation of Component Actions

Chapter 2 described how programming languages have evolved primarily to ease the creation of isolated functionality (component actions). As programming languages are well suited to the task of implementing the design of component actions, little advantage would result in the pursuit of a new means of transforming design abstractions for discrete operations into computer programs. Therefore this section discusses how programming languages provide declarative abstractions to support the implementation of component actions.

Using a programming language to implement the design of component actions involves the composition of algorithms which describe a process or rules for computation. Algorithms are usually specified by a nesting of expressions or a sequence
Software system may be considered as one large program

Within each component actions, interactions and behaviour are described using program language constructs. This requires system issues such as distribution, concurrency and synchronisation to be described by the same means.

Figure 9. Characteristics of Software Systems

of statements [Fischer93]. An expression is a nest of function calls or operators (a syntactic variant of functions) used to denote an operation, which, when applied to a number of arguments, produces a result. A statement is an expression that does not return a value and therefore cannot be used as the argument of a function. Statements (control statements, procedure calls and assignments) are executed for their side-effects, which usually alter the program’s memory, e.g. assigning a value to a variable, or produce some effect within the computer infrastructure, e.g. storage of data on a physical device. Declarations can be used to reduce the apparent nesting depth of expressions. A local variable declaration allows the programmer to define a name for the result of a sub-expression. A subroutine declaration permits a programmer to isolate and name some part of the program code. The defined names can then be used to build routines with a lower apparent degree of nesting and therefore a greater degree of abstraction.

As discussed in Chapter 2, some programming languages provide the facility to express an algorithm as a declaration of axioms. This removes the need for the programmer to specify the computational process required to implement a particular component action. Programming languages such as Haskell [Hudak92], Miranda [Turner86] and Prolog [Clocksin81] provide such a facility and therefore fulfil the requirement to represent a component action by a declarative means. Such declara-
tions retain the semantics of component design, and are automatically transformed into a form which can be executed by a computer infrastructure.

As an example of the ability to implement algorithms as a declaration of axioms, consider the implementation of the oldest known non-trivial algorithm devised by Euclid. This algorithm, which computes the greatest common divisor (GCD) of two integers, can be expressed by the following axioms.

The GCD of two positive integers \( a \) and \( b \) is -

- if \( a \) is equal to \( b \) the GCD of \( (a - b) \) and \( b \).
- if \( a \) is greater than \( b \) the GCD of \( a \) and \( (b - a) \).
- if \( b \) is greater than \( a \)

These axioms can be expressed using the functional programming language Miranda as

\[
gcd a b = \begin{cases} 
a, & \text{if } a = b \\
gcd (a-b) b, & \text{if } a > b \\
gcd a (b-a), & \text{if } a < b \\
\end{cases}
\]

in the logic programming language Prolog as

\[
gcd(A,A,A). \\
gcd(A,B,D) :- (A>B), R is A-B, gcd(B,R,D). \\
gcd(A,B,D) :- (A<B), gcd(B,A,D).
\]

and imperative language C as

\[
int gcd(int a,int b) 
{ 
    \text{if}(a==b) \text{return}(a); 
    \text{if}(b>a) \text{return}(gcd(a,b-a)); 
    \text{if}(a>b) \text{return}(gcd(a-b,b)); 
}
\]

In the above examples, a process of computation is not specified, but a declaration of the three axioms is required to implement the GCD algorithm. If a statement of axioms cannot describe the desired algorithm, most programming languages provide a highly abstracted means to describe desired actions with little consideration of the underlying computer infrastructure adopted. As an example, consider an algorithm expressed using the Prolog programming language that uses the facilities provided by the computer infrastructure to represent the characters contained within a text file. Axioms can be used to express how the end of a file is represented

The end of a file is represented by the value \(-1\)

\[
\text{endOfFile}(-1).
\]

Clauses can be defined to obtain the next character from a file

\[
\text{Unify C with the next character, if the end of the file has not been reached}
\]

\[
\text{nextChar(C)} :- \text{get0(C)}, \text{not(endOfFile(C))}.
\]
The contents of a text file can then be represented as a Prolog list using the following declarations:

*The head of the list is the next character that can be obtained from the file, and the tail of the list is the remaining characters within the file.*

\[
\text{file}([C|Cs]) :- \text{nextChar}(C), \text{file}(Cs).
\]

*A file may contain no characters*

\[
\text{file}([]).
\]

The above declarations may be used in conjunction with existing facilities which open and close files, to unify a variable with the contents of a file stored within the computer infrastructure:

\[
?- \text{see(filename), file(CharacterList), seen}.
\]

As the above examples show, programming languages can support the transformation of declarative expressions of components' actions into the programs required to execute them using a computer infrastructure. Therefore, such programming languages should be adopted to support the implementation of component actions within the prototype system infrastructure.

### 4.4 Supporting Component Interaction

Many different design notations exist for describing the breakdown of software systems into interacting components. However, they can be grouped into two broad categories, structured design and object oriented design [Sommerville96].

Structured design methods [Martin90, DeMarco79, Yourden79] model interaction using Data Flow Diagrams (DFDs), which define a network of functional processes connected to one another by "pipelines" of data. Using the data flow diagram notation a system component is represented by a process and usually depicted graphically by a circle or rectangle with rounded edges. The flow describes the movement of data from one part of the system to another and is represented graphically by an arrow into and out of a process. Flows are named, the name represents the meaning of the message that moves along the flow, and also indicates the direction of information flow using an arrowhead.

Object oriented design methods [Booch94, Rumbaugh91, Booch95] depict the breakdown of a system into interacting components using some form of object message diagram. An object message diagram depicts objects interacting through links.
with other objects. The existence of a link allows objects to send messages to each other, messages usually consist of three elements: a synchronisation symbol, denoting the direction of the message; an operation invocation or event dispatch; and an optional sequence number to convey the relative ordering of messages.

These notations essentially adopt one abstraction, the message, in order to describe system component interaction. The message abstraction is used to convey the interacting participants, the instigator and target of the interaction and the meaning conveyed. Issues such as message transportation, marshalling between local and system-wide representations, buffering and dispatch strategies, communication network protocols and system component location are omitted. When creating a system however, component interactions must be realised and these issues resolved.

Within Chapter 2 the author described the emergence of distribution infrastructures, which support system component interaction across distributed heterogeneous computing platforms. These infrastructures provide services to facilitate component interaction, irrespective of the component’s physical location, or computer infrastructure adopted, thereby providing a level of distribution transparency [ISO92] and some degree of system configuration capability [Kramer85]. To achieve this, distribution infrastructures introduce abstractions and tools to facilitate access to the services they provide. These are typically based on the elaboration of existing programming languages, and/or a set of abstract data types.

As an example, consider OMG’s CORBA specification, which defines a set of services to allow system components to be implemented as distributed objects. Within the CORBA specification, all that is required for an object to interact with another object is an object reference [Schmidt95]. The distribution infrastructure (named the ORB) is responsible for automating other common communication activities such as locating the target object, activating it, delivering the message and returning any response. An object’s interface is described using an Interface Definition Language (IDL), the syntax of which is based to some extent on the high-level programming language C++. A compiler transforms the IDL into client-side “stubs” and server-side “skeletons”, which can be composed using a variety of programming languages. The component implementor then populates the stubs and skeletons with the additional code required to realise the interacting system components.
Although this technology has a role to play in the creation of distributed computer systems, the concepts introduced such as object references, interface inheritance, object adaptors and IDL compilers are far removed from the simple declarative message abstraction adopted by system design notations. Such technology addresses how to perform component interaction and not how to automatically realise declarations of desired component interactions. In this way the CORBA specification is typical of contemporary distribution infrastructures. As a consequence even with the latest distribution technologies the simple declarative constructs held within a system design are not readily mapped onto the means by which they may be achieved.

Therefore, the requirement placed on a prototype system infrastructure is to provide the means to: declare system component interactions; and to map them automatically onto the services provided by distribution infrastructures such as those specified by CORBA.

4.5 Supporting Component Behaviour

Many alternative notations exist for describing the behaviour of systems and system components. However a few primary constructs are fundamental to all behavioural modelling techniques [Chow96]; these are state, event and operation. The behaviour of a component can be described as a succession of states and state transitions. The state of a component may be considered as an abstraction of the relevant information necessary to describe its actions and interactions [Peterson81]. Events describe occurrences which may cause a change in the state of a component such as interactions with other system components or the passage of time. Operations typically reflect some component action or interaction conducted during the transition from one state to another.

A simple model of behaviour, based on the work conducted by Turing in the 1930s is the Finite State Machine (FSM) [Davis88]. A FSM is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an input, the machine may generate an output and change state. Both the output and the new state are purely functions of the current state and the input. Such an abstract machine contains a complete description of what a component will do without describing how it will do it. This simple means of modelling behaviour has been modified and incorporated into the notations used to design the behaviour of system
components. Typical design notations used to describe component behaviour include State Transition Diagrams [Yourden79, Martin90, Booch94, Booch95] and Petri nets [Petri73, Peterson81].

The main constructs of a state transition diagram are states, depicted graphically by rectangles, and state transitions, which are depicted by arrows. Transitions may possess an associated condition which, if satisfied, causes the component to change state. As part of the change of state the component may typically perform one or more actions, and interact with another system component. An extension to this notation, called statecharts, has been proposed by Harel [Harel87b] and has been incorporated within a number of system design notations.

A Petri net is composed of four parts, a set of places, a set of transitions, an input function and an output function. The input and output functions relate transitions to places. The input function is a mapping from a given transition to a collection of places, known as the input places of the transition. The output function maps a given transition to a collection of places, known as the output places of the transition. A graphical representation of Petri nets has been defined, with the aim of providing an intuitive description of behaviour. A place can be represented by a circle and a transition by a bar. Places and transitions are connected by directed arcs. The translation between graphical and formal representations is well established [Peterson81]. In addition to its defined structure a Petri net is marked, the marking is an assignment of tokens to the places of a Petri net. The tokens are used to define the state of the system described by the net, therefore the number and position of tokens may change during the execution of a Petri net.

Petri nets can be abbreviated or extended. Abbreviations are simplified representations used to lighten the graphical representation, but have the same expressive power as ordinary Petri nets. Extensions are Petri nets to which functioning rules have been added, enabling a greater number of applications to be treated. A number of extensions exist to describe the functioning of systems whose evolution is considered by external events and/or time, examples are synchronised Petri nets, timed Petri nets, interpreted Petri nets and stochastic Petri nets [David94]. Petri nets are a generic representation and can be used to model classical state transition diagrams and their extensions [Peterson81].
Such formalisms offer a clear and concise representation of the behaviour of a system component, by providing a history of actions and interactions occurring during a component's execution. This clarity is achieved by ignoring issues such as: mechanisms for notification of event occurrences, demultiplexing events, how to suspend and resume activity whilst synchronisation occurs, how state is mapped onto the execution status of a component and how objects shared between actions and interactions are realised and managed. In order to realise the behaviour described by these abstractions within a system the behaviour must be encoded using constructs provided by programming languages and utilise the services offered by the computer infrastructure.

As a simple example, consider the representation of component state. To describe the state of a component during execution we not only need a copy of the program which defines the component, but also a copy of the execution stack, the value of global and static variables, the current values of the program counter and other hardware registers. To avoid this complexity and provide a clear declaration of state, a single abstract data type may be used. This approach is adopted by the "State" design pattern [Gamma95], a laudable attempt to capture and disseminate a sound method of state representation using the object oriented paradigm. The State pattern defines an abstract data type which uses the techniques of polymorphism and inheritance to provide an explicit representation of state and so avoids large conditional programming statements. The logic that determines the state transitions does not reside in monolithic if or switch control statements, but instead is partitioned between state subclasses.

The State pattern is an application of a well structured and documented approach to what may be considered the simplest behavioural concept, and one ideally suited to computers as they are ostensibly state transition systems. However, even this approach seems to have deviated from a simple declarative notion of component state towards an elaborate, albeit elegant, programming language structure which requires programming knowledge to understand and implement, but may demand a great deal more knowledge to identify and modify within existing systems.

Clear system design representations of component behaviour when encoded within programming languages and interfaced to computer infrastructures become lost
within many other implementation issues and programming abstractions. Therefore, the requirement placed on the domain machine to be built and tested in this work, must be to retain and execute the descriptions of component behaviour expressed within a system design.

4.6 Separating Manufacturing Concepts from the Computer System

The previous sections have stated that a prototype system infrastructure must support the separation between a description of action, interaction and behaviour of a system component and the means by which they are realised. This section outlines the solution adopted by the author to achieve this separation.

Chapter 2 described recent advances in computer programming languages which aim to provide a means of computation that does not depend on any particular computer hardware architecture. One such programming model is logic programming, where the separation of declaration and realisation is a fundamental tenet [Sterling94]. A logic program can be thought of as a description of a solution rather than a prescription of a process that results in the computation of a solution. Logic programs comprise a set of axioms, which define relationships between objects, and not a description of a computational process. For this reason, logic programming languages are described as declarative rather than imperative. In an imperative language, the programmer specifies a sequence of steps that will eventually produce the result as a side effect of some computational process. Whereas in declarative languages control is provided by standard deduction methods, built into the language interpreter, which provide the same effect as executing an imperative program [Kowalski79].

Prolog (PROgramming in LOGic) is a deterministic language based on a computational model of logic programming used for expressing predicates, relations and axioms. Prolog is declarative as its programs aim to model a particular problem area, avoiding procedural aspects of achieving a given task. When using Prolog, the programmer does not specify an algorithm in the same way as in a conventional programming language [Clocksin81]. The programmer may state facts or make queries about those facts, a query is a request to prove a theorem and so is also often referred to as a goal. Prolog programs comprise objects (constants, variables, structures, lists), predicates, operators, functions and rules. The rules are expressed as
Horn clauses, a machine manipulable logic with a well-defined syntax and semantics [Davis85]. Prolog uses an interpreter to support these syntactic and semantic rules, which can be modified using in-built predicates of the language. In order to provide a practical programming language, Prolog supports extra-logical predicates to provide access to the underlying computer infrastructure.

Prolog programs possess both declarative and procedural semantics. When using Prolog a programmer defines predicates and writes logical formulas involving those predicates, in this way a database of facts and relationships is constructed which contains a declarative model. The declarative semantics define which goals can be considered true according to a given program, but make no reference to the sequencing of goals within a program. However, the sequencing information is relevant to the procedural semantics which define exactly how the Prolog language will execute any particular goal. As an example of this separation, consider the task of reversing the elements contained within a list. The axioms which describe the rules required to perform such a computation are

*The reverse of an empty list is an empty list*

*To reverse a non empty list, recursively reverse the tail of the list, and then append the first element at the end of the reversed tail*

These axioms can be expressed in Prolog as

\[
\text{reverse}([],[]).
\text{reverse}([\text{Head},\text{Tail}],\text{RevList}):-
\text{reverse}(\text{Tail},\text{RevTail}),\text{append}(\text{RevTail},[\text{Head}],\text{RevList}).
\]

A query or request to prove a theorem using the rules and relationships held within the database would be expressed as

\[
\text{?- reverse([a,b,c],RevList)}.
\]

Such a query would result in the Prolog interpreter, unifying the variable RevList with the reversed list, and indicating a successful completion of the goal

\[
\text{RevList} = [c,b,a] ?
\text{Yes}
\]

Figure 10 depicts the proof tree used to prove the theorem using the rules and relationships held within the Prolog database. The consequence of this is to perform a computation based on the procedural semantics imposed on those rules and relationships. These different semantics provide a separate view of what is required to reverse a list and how it is achieved.
The separation of declarative and procedural semantics provided by the Prolog programming language is aligned with the general requirements of the prototype domain machine as it provides a means to instantiate, understand and execute declarative models. Due to this alignment the author selected the Prolog programming language to underpin the implementation of the prototype domain machine, essentially performing the role of a substrate onto which the support identified in this chapter for system action, interaction and behaviour could be added.

4.7 Creating a Prototype Distributed Control Machine

The framework used to implement the prototype domain machine is illustrated in Figure 11. Within the infrastructure, the Sicstus Prolog programming environment [SICStus97] provides support for the declaration and execution of components actions. This version of Prolog was chosen for the following reasons:

- conformance to the ISO Prolog programming language standard;
- support for heterogeneous computing platforms;
- provision of a mixed programming language facility, which allows Prolog programs to be integrated with imperative languages such as C, and;
- mature support for the facilities offered by the computer infrastructure.

The mixed language programming interface also offered the means to integrate the distribution infrastructure with the prototype system infrastructure. Two distribution infrastructures Orbix [IONA95] and CIM-BIOSYS [Gascoigne94] were used to provide alternative means of achieving system component interaction within a distributed computer environment. These infrastructures represent both commercial and research technologies and were both included in an attempt to provide a general
Manufacturing Control Systems

Figure 11. Framework for a Manufacturing Control Domain Machine

level of component interaction support not specifically tailored to one distribution infrastructure.

With these programming tools and environments in place, two main support facilities had to be created in order to realise the prototype domain machine:

1. the definition and realisation of a set of interaction constructs to describe and execute system component interaction using the services offered by a distribution infrastructure, and;

2. the definition and realisation of a set of behaviour constructs to describe and execute the evolution of component state, coupled to the execution of component actions and interactions.

The following two chapters detail the approach taken to providing these two support facilities.
Chapter 5 - Supporting Component Interaction

This chapter describes the definition and realisation of the declarative constructs offered by the prototype domain machine to support system component interaction. The constructs allow declarations of component interactions to be automatically mapped onto the services provided by either of two distributed computing infrastructures, CIM-BIOSYS and Orbix. The constructs obviate any requirement to produce an imperative programming language representation of component interaction, allowing simple design declarations to define and execute system-wide component interaction.

5.1 Definition of Interaction Constructs

This section details the interaction constructs defined, and discusses their syntax, semantics, and relationship to existing design notations used to describe component interaction.

Within the previous chapter the intention to employ the Prolog programming language to underpin the creation of a system infrastructure was outlined. When defining a set of interaction constructs, the author chose to enunciate them as Prolog extra logical predicates. Although extra logical predicates lie outside of the logic programming model of Prolog, they do achieve a side-effect in the course of being satisfied as a logical goal. There are three types of extra logical predicates: predicates concerned with input and output; predicates for accessing and manipulating the program; and predicates for interfacing with the computer infrastructure. Extra logical predicates contribute significantly to the practicality of Prolog as a programming language as such side-effects are precluded within the pure, i.e. logic based, part of the language. By expressing component interaction constructs as extra logical predicates the following facilities were offered by the Prolog programming language:

- the separate declarative and procedural semantics of Prolog could be applied to the interaction constructs;
- Prolog's built-in database and associated extra logical predicates for database modification could be used to create and manipulate interaction constructs, and;
- the built-in logic interpreter and theorem proving mechanisms could be used to define and validate the syntax of interaction constructs and facilitate their execution.

The constructs defined to support component interaction are shown in Figure 12.

<table>
<thead>
<tr>
<th>Construct Description</th>
<th>Predicate Form</th>
<th>Operator Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>To declare a message to be sent to a system component -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to(+message,+component)</td>
<td>in predicate form, or</td>
<td>in operator form.</td>
</tr>
<tr>
<td>+message to +component</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To declare the arrival of a message from a system component -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>from(?message,?component)</td>
<td>in predicate form, or</td>
<td>in operator form.</td>
</tr>
<tr>
<td>?message from ?component</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To declare the arrival of a message from an unspecified system component -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>from(?message)</td>
<td>in predicate form, or</td>
<td>in operator form.</td>
</tr>
<tr>
<td>?message from ?component</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key**

+ item must be identified

? item may or may not be identified

component identified using a constant or variable

constant represents a specific item

message either a constant, a variable or a structure

structure a single item which consists of a collection of other items

variable denotes a definite but yet unidentified item

---

**Figure 12. Component Interaction Constructs**

Each construct has two forms, predicate and operator, and although they possess different syntax their semantics are identical.

The declarative semantics of the interaction constructs are the same as the extra logical predicates defined by the Prolog language. As an illustration of this consider the following examples -

**A declaration of two messages to be sent to a component named tom**

    hello to tom
    result(dick,56,passed) to tom

**A declaration of two messages received from a component named dick**

    hi from dick
    status(failed) from dick

The declarative semantics of Prolog determine if such goals are considered true. In addition, as these constructs are implemented as extra logical predicates, the facilities provided by Prolog for theorem proving such as unification are also available.

For example the declaration

    Message from fred
if proved, unifies the message instigated by the component *fred* with the Prolog variable *Message*. The declaration

```
hello from Anybody
```

if proved, unifies the variable *Anybody* with the name of the system component which instigated the message *hello*, and the declaration

```
Message from Anybody
```

if proved, unifies the variables *Message* and *Anybody* with the message and name of its instigator.

To provide some simple form of component synchronisation, the additional extra logical predicates shown in Figure 13 were created to suspend component activity in

```
Wait for goal to be proved true - if goal fails it is re-evaluated when some change occurs to the program due to an event occurrence -
```

```
waitFor(+goal)
waitFor +goal
```

in predicate form, or

in operator form.

```
Wait a maximum period of time for goal to be proved true -
```

```
secondWaitFor(+number,+goal)
+number secondWaitFor +goal
```

in predicate form, or

in operator form

```
Wait a particular period of time -
```

```
secondWait(+number)
+number secondWait
```

in predicate form, or

in operator form

```
Key
```

```
+ item must be identified
?
item may or may not be identified
goal any valid Prolog goal
number integer value
```

**Figure 13. Simple Synchronisation Extra Logical Predicates**

between message occurrences, or for predefined periods of time. This allows simple component synchronisation issues to be defined in the following way

```
20 secondWaitFor Message from fred
```

this declaration suspends execution for 20 seconds or until a message is received from the component *fred*, in which case the variable *Message* is unified with the received message. The declaration

```
pardon to fred, waitFor Message from fred
```
Supporting Component Interaction

declares a synchronised two way message exchange. Using the facilities provided by the Prolog language this declaration could be generalised by the following declaration

\[
\text{twoWay(Component, Message, Reply)} :\text{-} \quad \text{Message to Component,}
\]

\[
\text{waitFor Reply from Component.}
\]

This facility allows for the declaration of additional interaction constructs provided by some design notations. Once a new construct has been declared, it can be used in an identical fashion to the interaction constructs provided by the system infrastructure.

The procedural semantics of the interaction constructs are the same as regular Prolog extra logical predicates. Therefore they do not generate alternative solutions for Prolog’s automatic theorem proving process, as the side-effects generated by the constructs, i.e. component interaction, are only performed once during the proof of a given theorem.

This thesis is founded on the principle of preserving the design abstractions used to describe system component interaction within an implemented computer system. Given this, the interaction constructs defined must posses the same semantics as those adopted within system design notations. Variation in syntax is inevitable due to the range of design notations in existence. However, if only syntax variations exist then a simple translation processes may be defined to provide equivalence between different design notations.

As discussed within the previous chapter the three underlying concepts communicated by design notations for interaction are meaning, participants and direction. The interaction constructs defined provide explicit support for each of these concepts. The constructs can in fact be considered in the following form

\[ \text{meaning direction peer-participant} \]

This leaves only the identity of the declaring component as implicit within a declaration of a component interaction.
5.2 An Approach to the Realisation of Interaction Constructs

This section details the solution adopted to create the interaction constructs defined within the previous section. The constructs utilise the system component interaction services provided by a distribution infrastructure.

In order to realise a mapping between declarative interaction constructs and the varied imperative abstractions provided by distribution infrastructures a number of issues were addressed, namely:

- how to access the interaction services via the imperative programming abstractions offered, e.g. Application Programming Interfaces (APIs), abstract data types, pseudo code precompilers, prescribed code templates and operating system services;
- how to manage the interaction service usage, i.e. whether particular services were synchronous or asynchronous, required single or multiple execution threads, consideration of any additional operational requirements such as heartbeats or watchdog timers, and handling of any service configuration information issued upon component invocation or during component execution;
- how to provide interaction services not offered by a particular distribution infrastructure, e.g. association control when using broadcast systems or timing constraints imposed on outstanding service requests;
- how to marshal between the weakly typed messages defined within the interaction constructs and the data type requirements of the interaction services provided by distribution infrastructures, and;
- how to package the solution in a form suitable for execution by a computer infrastructure.

To address this range of issues and retain some implementation flexibility, the author used a layered machine architecture. This approach organises a solution into a series of layers each of which provides a set of services [Sommerville96]. The services of one layer are used to implement the services of the next. Examples of this approach can be seen in the ISO OSI reference model of network protocols [Zimmerman80] and the X windows programming tool kit [Scheifler86]. This approach was adopted not only as a means of decomposing implementation prob-
Supporting Component Interaction

problems, but also as a means of providing the level of portability required to handle various distribution and computer infrastructures.

The layers of services constructed to address the issues highlighted above are illustrated in Figure 14, and a short description of the role of each layer follows.

![Diagram of Manufacturing Control Domain Machine with layers labeled: Interaction Constructs, Sicstus Prolog Environment, Distribution Infrastructure Interface, Process Control Interface, Computer Infrastructure]

Figure 14. Support for Component Interaction within the Manufacturing Control Domain Machine

The Distribution Infrastructure Interface layer provides a set of imperative interaction services which are independent of the distribution infrastructure adopted. It addresses issues associated with access to interaction services and provision of services not offered by particular distribution infrastructures.

The Process Control Interface layer provides a programming model for the execution of computational processes which schedule their activity around events such as message receipts and timing constraints. This layer removes the requirement to handle the specifics of a computer infrastructure and provides a simple and consistent process execution facility.

The Interaction Service Interface layer provides a simple imperative programming model for the construction of system components. The layer utilises the services provided by the Distribution Infrastructure Interface and Process Control Interface to offer a programming model which is independent of both distribution
and computer infrastructures. Essentially this layer provides an implementation tool kit for the construction of system components using conventional programming methods, and therefore can be used as an aid to the construction of portable system components.

The **Interaction Construct** layer combines the facilities offered by the Interaction Service Interface with Prolog. This layer converts the imperative based programming model of component interaction offered by the Interaction Service Interface into a form appropriate for declarative logic programming. The layer also contains the realisation of the interaction constructs implemented as Prolog extra logical predicates, and integrates the constructs with the resident language interpreter and automatic theorem proving mechanisms.

The composition of each of these layers and the services offered are described within the following four sections.

**5.3 The Distribution Infrastructure Interface**

The Distribution Infrastructure Interface comprises a layer of software which provides a neutral imperative programming interface to a distribution infrastructure. It is packaged as an ANSI C [ISO90] run-time library in order to provide a collection of easily accessible services. Its primary role is to map neutral component interaction service requests onto the services offered by various distribution infrastructures. With this aim in mind, the services offered relate to simple concepts of component association and asynchronous message passing, and do not impose any particular programming model. The layer therefore avoids issues relating to synchronisation of system-wide component activity.

The services offered by this interface relate to initialisation and termination of distribution infrastructure service availability, execution of distribution infrastructure service requests, handling of responses to distribution infrastructure service requests and handling of any commands issued by the distribution infrastructure itself. The services offered by the interface are accessed by populating a predefined C abstract data type with service request information, and passing a reference to this data type as an argument to a predefined C function. The abstract data type and functions
Supporting Component Interaction

relating to services offered by the Distribution Infrastructure Interface are shown in

Figure 15.

Abridged abstract data type used to hold service information

```
struct st_dii_req{
    long id;               /* request_id */
    int request_type;     /* request_type */
    char *who_i_am;       /* who_i_am */
    unsigned char *data;  /* data */
    char *who_they_are;   /* who_they_are */
    int data_len;         /* data_len */
    ...
    void (*response_func)(int,int,struct st_dii_req *);
    int expedite;         /* expedite */
    int kill_flag;        /* kill_flag */
    int timeout;          /* timeout */
};
```

Process initialisation information in the form of command line arguments

```
void dii_proc_args(int *argc,char **argv);
```

Initialise service link

```
int dii_init_link(void *unique_handle);
```

Terminate service link

```
void dii_term_link(void *unique_handle);
```

Obtain component system identifier

```
char *dii_get_instance_name();
```

Register function to handle commands from distribution infrastructure

```
int dii_reg_async_service_func(int service_type,
    void (*user_function) (char*,unsigned char*,int));
```

Deliver service request to distribution infrastructure

```
int dii_service_request(struct st_dii_req *service_info);
```

Register function to handle termination request from distribution infrastructure

```
int dii_reg_term_func(void (**user_function)());
```

Register function which handles service errors

```
int dii_reg_error_func(void (**user_function)(char *));
```

Figure 15. Distribution Infrastructure Interface Services

Two versions of the Distribution Infrastructure Interface have been produced to
date, these facilitate access to the interaction services of the two distribution infra­
structures CIM-BIOSYS and Orbix. The different natures of these two distribution
infrastructures aided the definition of a neutral set of services for system component
interaction.

The CIM-BIOSYS integration infrastructure [Gascoigne92] was created at the MSI
Research Institute at Loughborough University to provide a consistent set of serv­
ices to enable the integration of system components irrespective of physical location
and computer infrastructure used. The range of services offered provide for file
access, database access and system configuration as well as direct system compo-
To utilise the interaction services offered by CIM-BIOSYS a number of operational requirements were encountered:

- upon initialisation a system component must process configuration information issued by CIM-BIOSYS, this includes a system-wide component name together with a number of infrastructure communication parameters;
- system components must communicate with the CIM-BIOSYS infrastructure using the BSD sockets inter-process communication facility;
- an abstract data type, realised as a C structure, must be used to interact with CIM-BIOSYS. This data type comprises two parts, a header structure which contains operational management information, and an optional service structure, which contains integration service requests and responses. The various management and service types pertinent to component interaction are shown in Figure 16, and;
- once a communication channel is established between a system component and CIM-BIOSYS, the component is required to issue a "heartbeat" header structure to CIM-BIOSYS approximately once every five seconds. An associated acknowledgement is issued by CIM-BIOSYS upon its receipt.

### Figure 16. CIM-BIOSYS Management and Interaction Service Types

The Distribution Infrastructure Interface created to utilise the CIM-BIOSYS infrastructure presents the imperative service interface defined in Figure 15 and hides the operational details discussed above from the service user.

In contrast to CIM-BIOSYS, the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) is a specification of the interfaces and services provided by a distributed object oriented computing infrastructure,
known as an Object Request Broker (ORB) [Vinoski93]. The intention of the specification is to allow various vendors to provide programming tool kits based on the interface defined by the OMG. In order to create a Distribution Infrastructure Interface for a distribution infrastructure based on the CORBA specification, the author used the Orbix programming tool kit [IONA95]. The selection of this particular product was based solely on the grounds of availability and conformance to the CORBA specification.

Orbix is a programming tool kit, and is well suited to the construction of an imperative interface to the underlying services provided by an ORB. However, as the interaction service interface devised by the author was not based on the CORBA specification, a number of implementation issues had to be resolved:

• as a client/server [Orfali96] architecture is assumed within the CORBA specification, modifications were required to support the general peer to peer architecture supported by the domain machine;

• within the CORBA specification, component interaction is performed by synchronised remote method invocation. As the Distribution Infrastructure Interface excludes the synchronisation issues associated with inter-system component activity, the tight connection between instigation and receipt of messages is broken, thereby removing any implicit coupling of separate system component activity;

• within the CORBA specification component interaction is facilitated by the use of object references, an abstract data type managed by the component itself. Issues relating to the declaration and binding of object references to other system components were handled within the Distribution Infrastructure Interface. The format of the Orbix bind function is shown in Figure 17, and;

• within Orbix, component interface declarations are described using IDL and compiled into C++ programming constructs. These constructs were then mapped into the simpler data types and function call conventions of C by the Distribution Infrastructure Interface.

The Distribution Infrastructure Interface created, which utilises Iona’s Orbix product, presents the imperative service interface defined in Figure 15, and hides both the implementation and operational details discussed above from the interaction service user.
To establish an association with the server which provides the implementation of the object interface required

```c
static Interface_ptr _bind(  
    const char* markerserver,  
    const char* host,  
    const CORBA::Context&,  
    CORBA::Environment& env =  
    CORBA::default_environment);
```

Key

- `markerserver` is object_name `:` server_name

Figure 17. Orbix Bind Function

5.4 The Process Control Interface

The Process Control Interface comprises a layer of software which provides two general purpose programming models for the execution of processes required to schedule their activity around event notifications. Again it is packaged as an ANSI C runtime library. The layer's primary role is to obviate any involvement with the variety of programming models and facilities for process execution management offered by computer infrastructures and associated programming tool kits. With this aim in mind, the two programming models are based on the concepts of interrupt and callback.

The interrupt-based programming model provides a mechanism for registering service user-defined functions which are invoked when particular computer system events occur. Such events then interrupt the ongoing thread of process execution and call the associated service user function. Events may be associated with the passage of time or the arrival of a message from the computer infrastructure.

The callback-based programming model also provides the means for registering service user functions to be invoked upon computer events, but assumes that the main thread of execution is controlled by a "process manager". The role of the manager is to suspend process execution until some noteworthy event occurs, then yield control to the associated service user function and regain control upon the function's completion. If a hybrid programming model is required, an additional facility is provided to allow the manager to be polled by the executing process.

Both programming models provide a facility for service user data to be passed as an argument to functions invoked upon event occurrences. The only requirement being
that any computer storage considerations are handled by the service user. The functions which provide access to the Process Control Interface are shown in Figure 18.

```
Call function timer function every period milliseconds. The timer function may
deregister itself or the function timer_dereg_func can be called

int timer_reg_func(void *unique_handle, int period,
int (*timer_function)(void *),
void *timer_data);
int timer_dereg_func(void *unique_handle,
int (*timer_function)(void *),
int timer_id);

Call function input function when data can be read from file descriptor fd.
fd_dereg_func can be called to deregister the function

int fd_reg_func(void *unique_handle, int fd,
void (*input_function)(void *),
void *user_data);
int fd_dereg_func(void *unique_handle, int fd);

Initialise and terminate general event manager
int pc_init(void *unique_handle);
int pc_term(void *unique_handle);

Process manager control services
int pc_manage_start(void *unique_handle);
int pc_manage_stop(void *unique_handle);
int pc_manage_once(void *unique_handle);
int pc_sync_sleep(void *unique_handle, int period);
int pc_sync_pause(void *unique_handle);
```

Figure 18. Process Control Service Functions

Five versions of the Process Control Interface have been produced, three of which are built on the Xintrinsics [Nye92], Xview [Heller90] and Orbix programming tool kits and two on the Unix [Kernighan84] select and signal operating system facilities.

5.5 The Interaction Service Interface

The Interaction Service Interface comprises a layer of software which provides three simple imperative programming models for the construction of interacting system components, again packaged as an ANSI C runtime library. The layer’s primary role is to obviate any involvement with the intricacies of various distribution infrastructures and computer infrastructures. With this aim in mind the layer utilises the programming models of the Process Control Interface and the interaction facilities offered by the Distribution Infrastructure Interface. The software within this layer presents both a simplified set of imperative interaction services alongside a more comprehensive imperative interaction facility. The simple concepts of compo-
component association and asynchronous message passing of the Distribution Infrastructure Interface are retained and issues such as the synchronisation of system-wide component activities are avoided. However, the programming models adopted from the Process Control Interface do provide the means to structure and manage intra-component activities, albeit at a level of abstraction related to the underlying facilities of the computer infrastructure.

When using the services of the Interaction Service Interface a number of programming models may be adopted. In addition to the interrupt and callback models already provided by the Process Control Interface, this layer introduces a simple blocking request model. When using this programming model a single thread of execution control is pursued. Requests for interaction services, block until an appropriate response is received from the distribution infrastructure or a timeout condition is encountered.

As well as a choice of the programming model adopted the interaction services provided may be accessed via different types of service functions. The most general functions require a predefined abstract data type to be created, and initialised with all relevant information regarding the service request. This data structure is then passed as a single argument to a general purpose service function. A simpler service interface is also offered by specialised functions, to which only a small number of more pertinent arguments are passed. The remaining service information defaults to predefined values. The range of services offered by the Interaction Service Interface can be grouped into general service management, issuing interaction service requests and handling incoming interaction service requests.

The functions associated with general service management are shown in Figure 19. These functions mainly relate to the “housekeeping” duties required when utilising the services offered by a distribution and computer infrastructure in tandem. An additional service function is provided to identify the current system component, as frequently, one-to-many relationships exist between executable binary images and the system components they represent.

A selection of the functions which provide access to interaction services is shown in Figure 20. Some of these functions relate to a particular programming model, while others provide different levels of interaction service access. When an interaction
Supporting Component Interaction

This function sets up buffers, opens I/O streams etc. required for a component to interact with the distribution infrastructure. It is also used to process any arguments passed to the component by the distribution infrastructure.

```c
int isi_init(void *unique_handle, int *argc_ptr, char **argv);
```

Obtains component's system wide name

```c
char *isi_get_instance_name();
```

This function closes down the I/O streams, buffers etc. initiated by isi_init and informs the distribution infrastructure of component's pending termination.

```c
void isi_term(void *unique_handle);
```

Register function term_func to be called when component receives terminate command from distribution infrastructure.

```c
int isi_reg_receive_terminate_func(void *(*term_func)(void *), void *user_data);
```

Figure 19. General Management Functions of the Interaction Service Interface

Send "connect with component" request to the distribution infrastructure.

```c
int isi_connect(char *who, void (*response_func)(int, int, void *), void *user_data);
int isi_init_connect(char *who, unsigned char *data, int data_len, void (*response_func)(int, int, void *), void *user_data);
int isi_st_connect(struct isi_req *service_info);
```

Send "release connection with component" request to the distribution infrastructure.

```c
int isi_release(char *who, void (*response_func)(int, int, void *), void *user_data);
int isi_st_release(struct isi_req *service_info);
```

Send "message to component" request to the distribution infrastructure.

```c
int isi_send_data(char *who, unsigned char *message, int message_len, void(*response_func)(int, int, void *), void *user_data);
int isi_st_send_data(struct isi_req *);
```

Figure 20. Interaction Service Functions of the Interaction Service Interface

service request is issued by a component, the response from the underlying infrastructure is handled in a manner which is dependent on the type of function used. For blocking requests a simple success indicator is returned, and if appropriate a global error code indicator is initialised. When using the interrupt or callback programming models, a response handling function may be elected to be invoked when the relevant response is received. Associated service user information may also be registered, and this is also passed to the function when invoked. In addition, the
response function is passed a variety of arguments upon invocation which depend on the interaction service type requested. Two of the arguments are success and error code indicators.

Incoming interaction service requests may be ignored by a system component or handled using the functions shown in Figure 21. These functions allow the service user to elect functions to handle various types of incoming interaction service requests. Associated user information may also be registered in the same manner as interaction service response functions. In addition to this facility the Interaction Service Interface also provides a service which allows the user to obtain relatively static interaction information such as a list of peer components with which an association currently exists.

Due to its intended role, only one version of the Interaction Service Interface has been created by the author. This single interface can utilise both of the Distribution Infrastructure Interfaces and all five Process Control Interfaces and so offers a flexible means to provide system component interaction.

5.6 The Interaction Construct Layer

The Interaction Construct Layer comprises software which integrates the imperative interaction services provided by the Interaction Service Interface with the Sicstus
Prolog programming environment to provide a concrete implementation of the interaction constructs. The layer comprises both a Prolog part and an ANSI C part, which reflects the layer's role as a bridge between the declarative and imperative programming paradigms.

The C programming language part of the interface performs three main roles: providing access to the services provided by the Interaction Service Interface; provision of extra interaction and process control functionality not supplied by the Interaction Service Interface; and the integration of these facilities with the Sicstus Prolog environment.

In order to allow Prolog extra logical predicates to utilise the C functions provided by the Interaction Service Interface, the interaction construct layer converts Prolog objects (structure, atoms, etc.) into corresponding native C data types and vice versa. In addition the automatic management of component associations is provided as the interaction constructs possess no notion of component association.

To handle the occurrence of asynchronous events within the Prolog environment, such as the arrival of a message from another system component, externally generated events are enunciated as entries in the Prolog database and therefore in the Prolog program. This approach allows the Prolog database to represent an event queue which may be managed using the standard extra logical predicates provided for program modification. To enable events to appear within the prolog database, a C function was created which supports asynchronous database access. When an external event occurs, any ongoing Prolog execution is interrupted and the event inserted, as a Prolog fact into the database. In addition a process scheduling capability was added to allow the Prolog environment to suspend its execution until the occurrence of a pre-specified event generated from either the distribution or computer infrastructure.

The Prolog programming language part of the interaction construct layer performs three main roles: it implements the interaction constructs, it provides a link between these constructs and the C part of the layer and also, integrates the interaction constructs with the resident logic interpreter provided by the Prolog language.

Facilities provided by Sicstus Prolog were used to create a Prolog-to-C function call interface which serviced all interactions initiated by the interaction constructs. Inter-
actions initiated by other system components and any internal timing constraints, manifest themselves as facts instantiated within the Prolog database. A facility to examine and manage these facts is provided. This was created using the existing extra logical predicates of the Prolog language.

The interaction constructs are defined as Prolog extra logical predicates, which utilise the Prolog-to-C function call interface to initiate interactions and the asynchronous database access facility to observe externally generated interactions. Additional operators required to enhance the syntax of the interaction constructs are declared to the resident logic based interpreter through use of the meta logic facilities provided by the Prolog language.

The interaction construct layer described within this section along with the three imperative interface layers previously described, were combined with the core Sicstus Prolog environment to produce an interaction facility. This facility forms a major part of the manufacturing control domain machine.

5.7 Concluding Remarks

The facility described in this chapter provides an environment which supports the declaration, manipulation and execution of design-based abstractions of system component action and interaction.

To illustrate this facility, consider a system component which echoes any messages received back to their originator. This is a simple task to describe, as it is not complicated by component actions or behaviour. However, this simple task is not trivial to implement within a distributed heterogeneous computing environment. Using the interaction constructs to describe such a system component involves the following declaration

```prolog
  echo :- waitFor Message from Somebody, Message to Somebody, echo.
```

This declaration also serves as the realisation of the system component when used in conjunction with the facility described in this chapter. There is no longer a need to convert the description into a target programming language and to have to deal with the problems introduced by distribution and computer infrastructures. The system component consists of a simple declaration of its interactions with other system components.
Chapter 6 - Supporting Component Behaviour

This chapter describes the definition and realisation of the declarative constructs which are provided by the manufacturing control domain machine to support the implementation of component behaviour. These constructs enable the design abstractions that describe desired component behaviour to form part of an implemented system. This removes the requirement to produce a programming language representation of component behaviour.

6.1 Definition of Behaviour Constructs

This section defines the behaviour constructs and discusses their syntax, semantics and relationship to existing design notations used to describe component behaviour.

To enable a domain machine to provide constructs that can both represent and execute design abstractions of component behaviour, a number of requirements must be met:

- the evolution of component state must remain explicit during system execution;
- the instigation of component actions and interactions must also remain explicit and must be bound to the evolution of a component state, and;
- a well-defined translation process must exist between system design abstractions and those provided by the computer infrastructure.

As discussed within chapter 4, when describing behaviour, the underlying concepts communicated by design notations are state, event and operation. Petri nets represent a general theory for state/event systems and extensions to Petri nets provide a powerful notation for modelling operations and the occurrence of externally generated events. Previous research into Petri net theory has shown how they can be used to model other behavioural notations, such as state diagrams [Murata89] and marked graphs [Peterson81].

Therefore, to provide a general representation of component behaviour, the definition of behaviour constructs is based on the constructs provided by Petri net theory. Petri net theory dates from the early 1960s [Milner89] and was the first general theory for systems comprising interacting concurrent components. Petri net theory is a generalisation of the theory of automata, in that Petri nets represent the independent occurrence of several state transitions.
This thesis is founded on the principle that system design abstractions should be preserved within an implemented system. This principle requires that the behaviour constructs provided by the prototype system infrastructure must possess the same semantics as those adopted by system design notations. As with the interaction constructs described within the previous chapter, variations of syntax are inevitable, but a simple translation process from design notation to behaviour construct must exist which retains the semantics of a component's design. To provide this facility, the semantics of the behaviour constructs are based on the semantics of interpreted Petri nets.

As discussed in chapter 4, a Petri net is defined by a set of places, a set of transitions, an input function and an output function. Tokens populate the places of a Petri net and are used to define its state. A Petri net is executed by the firing of transitions and is controlled by the number and distribution of tokens within the Petri net. A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. A transition may fire only if each of its input places is populated by as many tokens as there are arcs from the place to the transition. Given this condition the transition is said to be enabled. The tokens in the input places which enable a transition are referred to as its enabling tokens. A transition fires by removing all of its enabling tokens from its input places and then depositing into each of its output places, one token for each arc from the transition to the place. Firing a transition will, in general, change the marking of the Petri net.

To facilitate the modelling of systems whose evolution is determined by the occurrence of externally generated events and the passage of time, various extensions to Petri nets have been proposed [David94]. One class of extension allows an element of computation to be incorporated, and is often referred to as an interpreted or predicate/action Petri net [Murata89]. This class of Petri net allows the declaration of events, conditions and operations to be associated with transitions. These additions were incorporated into the behaviour constructs to allow the execution of system component action and interaction to be integrated with the evolution of component state. These additional constructs modify the execution of a Petri net in the following way. In order for a transition to fire, it must be enabled (as previously discussed), any associated event must have occurred and any associated condition must be true. Upon firing a transition any associated operations are performed. The suc-
cess or failure of operations has no bearing upon the firing of the transition. After the operations are performed, the marking of the net is updated as previously described. To ease the construction of interpreted Petri nets the use of variables is usually supported. The variables can be used to determine the value of conditions and their values can be manipulated by events and operations. However, it follows that the state of the Petri net is a function of its marking and of the values associated with its variables.

To provide a concrete representation of interpreted Petri nets within the prototype system infrastructure, its constructs were enunciated as Prolog structures (single items which consist of a collection of other items), with the aim of instantiating them as facts within a Prolog database. The constructs which comprise a Petri net model could then be created and manipulated using Prolog extra logical predicates.

To provide support for interpreted Petri net variables, a “store” construct was adopted. This construct was created to avoid confusion with the variable support offered by the Prolog language which possesses value-oriented semantics. The variables defined within interpreted Petri nets are based on those provided by most imperative programming languages, which posses variable-oriented semantics. When using value-oriented semantics, identifiers can only refer to a value and no updating is allowed. Whereas, when adopting variable oriented semantics identifiers usually refer to memory storage locations of a computer which can change their values during execution of a program. As a syntactic device to distinguish setting and obtaining the value of a store, the character ‘@’ was used to identify the value held within a store. Hence, assignment of a value to a store takes the form

\[
\text{storeID is value}
\]

and the value associated with a store is obtained using

\[
@\text{storeID}
\]

Apart from this syntactic device, the syntax rules of Prolog are assumed for store manipulation. For example

\[
\text{counter is } @\text{counter + 1}
\]

increments by one, the value held within the store \text{counter} and

\[
\text{buffer from fred, } @\text{buffer to tom}
\]

assigns the message from the component \text{fred} to the store \text{buffer} and then sends the contents of store \text{buffer} to the system component \text{tom}. Prolog variables can also
be used to pass items between events, conditions and operations within a single transition construct, but they do not have any bearing on the state of the Petri net once the transition has been fired.

The constructs to declare the behaviour of a system component are shown in Figure 22. These constructs possess the same semantics as those defined for interpreted Petri nets. To allow component behaviour to be determined by the passage of time, a number of time interval constructs were defined and enunciated as Prolog extra logical predicates, these are shown in Figure 23.
To illustrate constructs used to model system component behaviour consider the following examples. The declaration of a transition from a state idle to a state busy upon receipt of the message go from any system component can be expressed as

```
place(idle,1).
place(busy,0).
transition(t, go from, true, []).
input(t, [idle]).
output(t, [busy]).
```

To describe interactions instigated by a system component, outgoing messages may be included as actions. The following model describes switching between the two states idle and busy upon successive occurrences of the message pressed being received from the component switch.

```
place(idle,1).
place(busy,0).
transition(t1, pressed from switch, true, [run to machine]).
transition(t2, pressed from switch, true, [stop to machine]).
input(t1, [idle]).
input(t2, [busy]).
output(t1, [busy]).
output(t2, [idle]).
```

Stores can be used to hold values required by more than one transition. The following model describes the choice of state evolution, depending upon the value held within the store identified by the label x.

```
place(p0,1).
place(p1,0).
place(p2,0).
place(p3,0).
transition(t0, noEvent, true, {compute(x)}).
transition(t1, noEvent, @x>5, []).
transition(t2, noEvent, @x<=5, []).
input(t0, [p0]).
input(t1, [p1]).
input(t2, [p1]).
output(t0, [p1]).
output(t1, [p2]).
output(t2, [p3]).
store(x, 0).
```
Multiple input places and the binding of stores to events can also be used. The following model describes a build operation which depends on the availability of a tool and the receipt of a part number received as a message from another system component.

The interval constructs can be used to express a component state that changes after the passage of a pre-specified period of time. The following model describes the transition from a state wait to a state start after a token has been present in a state wait for 5 seconds.
Interval constructs can also combine stores and actions. The following model describes pausing for a period specified by a message sent by another system component before returning back to the state running.

```
place(running,1).
place(pause,0).
transition(t1,t from, @t > 0, [stop, @t second interval delay]).
transition(t2, noEvent, delay elapsed, [go]).
input(t1, [running]).
input(t2, [pause]).
output(t1, [pause]).
output(t2, [running]).
store(t, 0).
```

This section has proposed the syntax and semantics of constructs capable of describing the behaviour of system components. The following sections detail the approach adopted to support the instantiation and execution of these constructs within a prototype system infrastructure.

### 6.2 An Approach to the Realisation of Behaviour Constructs

This section outlines the solution adopted to realise the behaviour constructs defined within the previous section. The constructs are used to compose a model which describes the evolution of component state, instigating associated actions and interactions. As previously described, the prototype domain machine supports the creation of algorithms to perform component actions and automatically executes declarations of component interactions. Therefore the role of the behaviour constructs is to integrate these actions and interactions, binding them to the evolution of the state of a system component.

To provide a facility capable of instantiating models of component behaviour and executing what they describe within an implemented software system, a number of issues were resolved:
• how to validate a behaviour model for correct syntax and semantics, i.e. to ensure that a model adheres to the rules used to define interpreted Petri nets;

• how to manipulate the model during its execution. Unlike most high-level programs, the model is not a static entity, as it must not only be interpreted but also updated. In this regard the model is akin to the machine code programs executed by a computer, as it contains separate control and data partitions. The control is represented by the Petri net structure, and data is represented by the net marking and the values held within stores;

• how to enable the evolution of the behaviour model to be driven by the occurrence of externally generated events. These events must determine the state of a system component with consideration to the passage of time and interactions with other system components, and;

• how to bind, the execution of actions and interactions described within the model, to the evolution of the state of a system component.

To address these issues and thereby provide the required model execution facility, a virtual machine [Abelson85] was constructed to form part of the banking domain machine. The virtual machine was augmented with a model validation facility to ensure that models submitted for execution possessed correct syntax and semantics. In addition as part of the domain machine, the virtual machine could invoke individual component actions and interactions via the resident Prolog interpreter. This was possible as the interaction constructs described within the previous chapter were implemented as Prolog extra logical predicates and individual component actions could be defined using existing Prolog predicates.

The relationship between the virtual machine and declarations of component action, interaction and behaviour is shown in Figure 24.

As the virtual machine is required to reside within the domain machine, the machine was implemented using the Prolog programming language. This choice was influenced by the following considerations:

• the behaviour constructs were enunciated as Prolog structures and therefore could easily be handled using the Prolog language;

• the database provided by Prolog could be used to store behaviour models and Prolog extra logical predicates used to manipulate the model;
6.3 Validation of Behavioural Models

To provide some degree of robustness for a system component which directly executes a design model, a model validation facility was implemented. Here, the intention is to validate a design before commencing its execution in order to avoid runtime errors caused by incorrect syntax or invalid semantics. To achieve this, the
model validation facility was created in two parts: a construct syntax checker, and a Petri net structure validator.

The construct syntax checker reads a behavioural model from the resident computer file system and breaks it down into a sequence of words. This process is commonly referred to as Lexical Analysis and was implemented using the Lexical Analyser Generator (LEX) software tool [Levine92]. When provided with a specification of valid words, LEX produces a C programming language function capable of returning the next valid word from a pre-specified input file. The C function produced was integrated into the Sicstus Prolog environment using the Sicstus C-to-Prolog language interface. The sequence of words is checked for correct syntax using the Prolog grammar rule formalism [Clocksin81], commonly referred to as the Definite Clause Grammar (DCG) facility. From a specification of the behaviour construct grammar, Prolog generates clauses which parse sequences of words and ascertain if the constructs therein conform to the grammar specified.

The syntax checking process tests for the valid formation of behaviour constructs, but does not check whether the constructs define a valid interpreted Petri net. This process is performed by the Petri net structure validator which uses the set of well established rules that govern the structure of Petri nets [Peterson81]. An example of one of these rules specifies that a transition must be connected to at least one input place or one output place. Such rules can be readily expressed as Prolog clauses, for example the following declarations

\[
\begin{align*}
\text{connected}(\text{Transition}) & : - \quad \text{input}(\text{Transition}, \{\text{Place}_{1}\}) , \\
& \quad \text{place}(\text{Place}_{1}) . \\
\text{connected}(\text{Transition}) & : - \quad \text{output}(\text{Transition}, \{\text{Place}_{1}\}) , \\
& \quad \text{place}(\text{Place}_{1}) .
\end{align*}
\]

express the rule just described and this expression can be used to form part of the model validation facility. To implement the Petri net structure validator, Petri net structure rules were expressed as Prolog clauses. Prolog functions were produced which applied the rule set to the interpreted Petri net model being validated.

6.4 Execution of Behavioural Models

To produce the behavioural model execution facility as described earlier in the chapter, a number of implementation issues were resolved. During system execution the virtual machine must:
Supporting Component Behaviour

• be able to access the behavioural model in order to execute the state changes, actions and interactions that the model describes. The virtual machine must also be able to modify the representation of the component state held within the model;
• embody the execution semantics of interpreted Petri nets, as defined earlier in the chapter;
• resolve any non-deterministic behaviour specified within a Petri net model. To illustrate this point consider the following Petri net. Both transitions a and b are enabled and therefore may be fired. However, if transition a is fired, transition b is no longer enabled and so cannot be fired. Therefore, a choice as to which transition to fire must be made. Within the implementation of the virtual machine such non-determinism must be resolved by some clearly defined policy;
• schedule its activity around the occurrences of events specified within the model, and;
• implement the time interval constructs described earlier in the chapter.

To provide access to the behavioural model after it has been validated, the model is loaded into the resident Prolog database. The pattern matching and unification facilities offered by the Prolog language can then be used to access any of the constructs which comprise the model. To enable modification of constructs containing representations of state, the extra logical predicates for database manipulation provided by Prolog can be used. Using these facilities the virtual machine was provided with random access to the behaviour model constructs.

To provide the machine with the appropriate execution semantics, the rules defining when particular transitions are enabled were encoded as Prolog clauses. As an example, consider the following declaration

```
```
The predicate `transitionEnabled` can be used to ascertain if a particular transition referred to by its label is enabled, or to search all existing transition constructs within the database and unify the label of an enabled transition with a Prolog variable. Consider the following model:

```
place(p1, 1).
place(p2, 0).
place(p3, 0).
transition(a, noEvent, true, []).
transition(b, noEvent, true, []).
transition(c, noEvent, true, []).
input(a, [p1]). output(a, [p2]).
input(b, [p1]). output(b, [p3]).
input(c, [p3]). output(c, [p1]).
```

the above declarations support the following Prolog queries. To establish if transition `a` is enabled
```
| ?- transitionEnabled(a).
Yes
```
to establish that transition `c` is enabled
```
| ?- transitionEnabled(c).
No
```
and to use the backtracking and unification facilities of Prolog to find all enabled transitions.
```
| ?- transitionEnabled(T).
T = a ?;
T = b ?;
No
```

An equivalent set of Prolog clauses were constructed to describe the occurrence of events, to evaluate conditional expressions, perform operations and store manipulations. Using this technique, the execution semantics of interpreted Petri nets were expressed as Prolog clauses and incorporated into the virtual machine.

To define a policy for resolving non-deterministic behaviour within the virtual machine, the author chose to adopt an approach similar to that used by the Prolog language itself to resolve logical non-determinism [Davis85]. The approach
resolves any conflict by relying on the order in which the transition constructs appear within the Prolog database. To illustrate the approach, consider the following situation

Given this situation, the transition a will fire and the transition b will not. However, to redress the apparent imbalance imposed by this strategy, the transition a is removed from its current position within the database and re-inserted at the bottom of the list of transitions. Therefore, the next time such a conflict exists the transition b will fire and the transition a will not. If the non-deterministic behaviour described by a model is an essential property of the system design, the behaviour must be broken down and implemented as two concurrent components, and two separate messages must be sent to these components. This would then allow the vagaries of computer networks, hardware and operating systems to determine the outcome.

To allow the machine to schedule activity around occurrences of events specified within a behavioural model, an interface to the Process Control Layer was constructed to suspend execution until the occurrence of an external event. After such an event the virtual machine triggers any appropriate Petri net transitions, returning back to the Process Control Layer when no further activity is possible.

An interface to the Process Control Layer was also produced to implement the time-interval constructs defined in this chapter. These constructs were realised as Prolog extra logical predicates, in the same manner as the interaction constructs described within the previous chapter.

To provide the means to initiate model execution, a driver loop was constructed to successively determine which transitions are enabled, fire any transitions possible, perform any operations specified and then modify the state of the behaviour model.
If no transitions can be fired due to the absence of external events, the driver loop suspends execution until an occurrence of an event initiated by the external environment occurs.

6.5 Concluding Remarks

This chapter has detailed the definition and realisation of declarative constructs intended to describe and execute the behaviour of system components. Execution of the constructs within an implemented system is supported by a virtual machine embedded within the manufacturing control domain machine.

During system execution one instance of the virtual machine is created for each model of component behaviour. A virtual machine is invoked when a message is sent to an inactive system component or by explicit instruction from the distribution or computer infrastructure. Once instantiated the virtual machine validates the relevant behavioural model and then proceeds with its execution. If well established models are present within a system, the model validation process can be omitted to benefit runtime performance. Once execution has commenced, a system component may be interrupted via the domain machine for either model development or model debugging purposes. Other system components will execute in ignorance of such intervention. This facility, which exists due to the presence of the virtual machine, allows the design of a component's behaviour to be developed within an implemented system, as no distinction exists between the design and its implemented form.

As an illustration of the behavioural model execution facility, consider the Dining Philosophers problem proposed by Dijkstra [Dijkstra71]. The problem requires synchronisation of concurrent activities, a typical problem when information and resources are shared. Five philosophers are seated at a large round table laden with chinese food. Each philosopher either eats or meditates. Between each philosopher is one chopstick. However, two chopsticks are required to eat the food. The problem is that if all philosophers pick up the chopstick on their left and wait for the chopstick on their right, they will wait forever and starve (deadlock).

The behaviour required to fulfil the synchronisation problem is shown as an interpreted Petri net in Figure 25. In order to implement a system component which
Supporting Component Behaviour

Figure 25. Solution to the Dining Philosophers problem

exhibits this behaviour, sending and receiving messages such as *eat* and *finished* to and from each of the philosophers (also implemented as separate system components), a number of implementation issues must be resolved. These issues include:

- how to represent the state of the system at any given point during its execution;
- how to despatch and schedule system component activity around the many different interactions within the system, and;
- how to resolve the non-deterministic behaviour described by the Petri net model.

However, if the prototype system infrastructure were used within the implemented software system, the interpreted Petri net model which describes the desired behaviour could be translated into the behaviour constructs defined within this chapter as follows.
place(m1,1).
place(m2,1).
place(m3,1).
place(m4,1).
place(m5,1).
place(e1,0).
place(e2,0).
place(e3,0).
place(e4,0).
place(e5,0).
place(c1,1).
place(c2,1).
place(c3,1).
place(c4,1).
place(c5,1).

transition(s1,noEvent,true,[eat to p1]).
transition(f1,finished from p1,true,[]).
transition(s2,noEvent,true,[eat to p2]).
transition(f2,finished from p2,true,[]).
transition(s3,noEvent,true,[eat to p3]).
transition(f3,finished from p3,true,[]).
transition(s4,noEvent,true,[eat to p4]).
transition(f4,finished from p4,true,[]).
transition(s5,noEvent,true,[eat to p5]).
transition(f5,finished from p5,true,[]).

input(s1,[m1,c5,c1]).
inptu(f1,[e1]).
input(s2,[m2,c1,c2]).
inptu(f2,[e2]).
input(s3,[m3,c2,c3]).
inptu(f3,[e3]).
input(s4,[m4,c3,c4]).
inptu(f4,[e4]).
inptu(s5,[m5,c4,c5]).
inptu(f5,[e5]).

output(s1,[e1]).
output(f1,[m1,c5,c1]).
output(s2,[e2]).
output(f2,[m2,c1,c2]).
output(s3,[e3]).
output(f3,[m3,c2,c3]).
output(s4,[e4]).
output(f4,[m4,c3,c4]).
output(s5,[e5]).
output(f5,[m5,c4,c5]).

The behaviour described within the model can be directly executed using the domain machine. Therefore any requirement to realise the design using a programming language representation is removed as the design can be embedded within the implemented system.

The following chapter describes an application of the author's manufacturing control domain machine to a shop floor control system built within the university laboratory and installed and tested within a collaborating company's manufacturing facility.
Chapter 7 - A Manufacturing Control System Implementation

This chapter describes the implementation of a distributed software system used to control a production line within an electronics manufacturing plant. The chapter provides details of the system design and implementation. This is followed by a description of a typical system change and a demonstration of the advantages of the author's proposed approach over a system implemented using contemporary design and implementation techniques involving "design patterns" realised using C++. 

Figure 26 repeats the diagram introduced in chapter 3 as Figure 5. It shows the system design in the top left-hand corner, and both the conventional approach to system implementation and the author's proposed new approach to implementing the
design. The following section describes the design of a manufacturing control system. This is followed by section 7.3 where the conventional approach to implementation of the system in C++ is described. Section 7.4 then describes how the same design is implemented using the author's proposed approach.

Having created two systems of the same design, based on the conventional and the new approach we can now test the impact of making a required change to such a manufacturing control system and compare the results of such a change in order to quantify the benefits of the author's approach over existing conventional techniques. Section 7.5 details this process and the results are reported and discussed in section 7.6.

When describing a software system of any realistic size, a problem of presentation inevitably occurs [Dijkstra72]. Therefore, only examples of system components which reflect the pertinent issues addressed within this thesis, have been included.

7.1 A Software System Design

The software system described within this chapter controls the various Printed Circuit Board (PCB) assembly operations of a manufacturer of computer hardware. Technical innovation within the area of PCB production has lead to the advent of surface mount technology and specialist computer controlled machinery and auxiliary equipment to assemble surface mount PCBs.

When procuring such technology the main priority for selection typically concerns production performance issues such as throughput, yield, quality and product variety. Only secondary consideration is given to other classes of requirements such as how readily the separate computer controlled devices can be integrated to form a system. Previous collaborative research and development initiatives between the MSI Research Institute and the computer hardware manufacturer has sought to address some of the integration issues. This collaboration has resulted in the installation of the CIM-BIOSYS integration infrastructure [Gascoigne92] at the company to support a software control system [Zhang92] which functions as part of an assembly system. The example presented in this chapter is based on a subsequent study, Model Driven CIM [Weston98], conducted by the MSI Research Institute to iden-
ify how inevitable changes within the physical manufacturing system could be accommodated by the existing software system.

The assembly system requires the coordinated operation of a set of automated machines, a solder paste screen printer, a component placement machine and a reflow soldering machine. These machines are physically linked together with an automated conveyer and buffering facilities to provide for the flow of PCB products through the assembly system. Batches of PCBs flow through each machine and are inspected by human operators at two inspection points. In general terms the system comprises a heterogeneous set of automated machines, connected to a computer network, and human operators equipped with computer terminals. Each part of the system operates as an independent entity waiting for the correct combination of commands or internal conditions before performing a pre-defined function. This situation gives rise to a distribution of system behaviour.

A variety of designs for a software system to control the surface mount assembly line were proposed by the collaborative research study. These were typically based on the concept of creating a system of distributed components which execute in parallel and interact through message passing. The interacting software components which comprise one such design are shown in Figure 27. The notation used is the Unified Modelling Language version 0.8 [Booch95]. Figure 27 is an UML concurrent object message diagram which details the software system components required to control the surface mount assembly line. This diagram describes the components and their interactions in terms of message passing. This system decomposition does not detail the actions performed by each component or their behaviour.

The behaviour of the system components may be defined using UML state diagrams. Such diagrams define how a component responds to events, such as incoming messages, and therefore provides a clear expression of component behaviour. Figure 28 is a state diagram for the assembly component identified in Figure 27.

By examination of both Figure 27 and Figure 28 it is evident that the component assembly receives batch orders from the component scheduler and controls the print and populate components. Upon receipt of the appropriate message, assembly sends a message to print which includes a batch size indication and initialises a
count of how many boards have been assembled. Assembly will then progress through a series of defined states receiving and sending messages to its associated components. Upon completion of a particular batch of PCBs, a message containing the number of completed PCBs is sent to the scheduler. The state diagram contains system design details such as the ability of associated components to handle batches of PCBs (print) or just single PCBs (populate).

As discussed within previous chapters, many design notations exist. As an alternative to UML consider the interpreted Petri net shown in Figure 29 which defines the behaviour of the component print. Although a different notation, the figure still clearly defines the behaviour of the print component as it interacts with other sys-
A Manufacturing Control System Implementation

Figure 28. UML (Version 0.8) State Diagram for the Component assembly

tem components and progresses through a number of defined states. The intention is not to compare various design notations, but to show that although different notations exist, each is best suited to describing a particular aspect of a system design. The clear concept communicated by a design is usually lost when transformed into a form suitable for execution by a computer infrastructure.

7.2 A Conventional Approach to System Implementation

In order to implement the system design using an existing computer infrastructure, the actions, interactions and behaviour of each system component must be realised. As discussed within previous chapters, the primary tool available for this purpose is the programming language. A high-level program can be considered as a description
of a process, which when executed, is intended to implement the behaviour described within the component’s design. During the implementation of a system component, the abstractions held within a design are mapped onto those offered by the programming language. Within this programming process, consideration is given to any constraints imposed by the computer infrastructure.

To discuss the issues regarding implementation of a system component, let us consider implementing the behaviour of the system component print (see Figure 29), using the computer infrastructure and distribution infrastructure already installed on the assembly line. For the purpose of the following discussion the C programming
A Manufacturing Control System Implementation

language has been used, mainly due to its applicability and popularity for “system programming” tasks. It is important to stress that program segments are used to illustrate how the issues encountered during implementation may be resolved, and are not proposed as either a definitive or ideal solution. Many different programming languages could be used, but the requirement to address the issues discussed below would remain.

Upon invocation the component is passed a number parameters from its environment, i.e. the computer infrastructure. Three of these parameters are system configuration information and relate to the identity of the component, the communication channel to facilitate interaction with the distribution infrastructure (in this case CIM-BIOSYS) and a system version indicator. This information must then be retrieved from the data structure char *argv[] supplied by the computer infrastructure.

```c
while(--argc > 0 && ++argv -- '-')
    switch(*((argv)+1))
    { case 'S' :  strncpy(my_socket_name,*argv+2,ARRAY_SIZE);  
                  my_socket_name[ARRAY_SIZE-1] = '\0';      
                  break;
    case 'A' :  strncpy(my_name,*argv+2,ARRAY_SIZE);       
                  my_name[ARRAY_SIZE-1] = '\0';             
                  break;
    case 'V' :  versionID = atoi((argv)+2);               
                  break;
    default :   fprintf(stderr,"... %s
", (*((argv)+1)); 
                  break;
    }
}
```

To facilitate interaction with CIM-BIOSYS and thereby other system components, a communication channel to CIM-BIOSYS must be created. In this case BSD sockets are employed. An IPC_ACK_IND heartbeat packet is issued by the component to inform CIM-BIOSYS of the component’s readiness to form part of the executing system.

```c
cbs_sckt_addr.sa_family = 1;
sprintf(cbs_sckt_addr.sa_data,"_%.4s",SCKT_ADDR_LEN,"cbs");
cbs_addr_len = 6;
sock_id = socket(AF_UNIX,SOCK_DGRAM,PF_UNSPEC);
my_sckt_addr.sa_family = 1;
my_name_len = strlen(my_socket_name)+1;
my_name_len += 2;
sprintf(my_sckt_addr.sa_data,"%.4s",SCKT_ADDR_LEN,my_socket_name);
bind(sock_id,&my_sckt_addr,my_name_len);
FD_SET(sock_id,&ev_inputfds);
send_command(IPC_ACK_IND,0);
printf("Component %.4s Running\n",ARRAY_SIZE,my_name);
```
A Manufacturing Control System Implementation

The function send_command has been defined as a general utility which sends management information from the component to CIM-BIOSYS. This function encodes the abstract data type struct s_ipc_head with relevant information and sends it to CIM-BIOSYS.

```c
void send_command(cmd, arg)
{
    struct s_ipc_head ipc_head;
    int bytes, n;
    strcpy(ipc_head.sender, my_name);
    strcpy(ipc_head.target, "cbs");
    strcpy(ipc_head.str_argl, my_socket_name);
    ipc_head.sender_proc = APP_PROC;
    ipc_head.cmd = cmd;
    ipc_head.argl = arg;
    for(bytes = 0; bytes < (ipc_head.argl+IPC_PCKT_OHEAD); bytes += n)
    {
        n = sendto(sock_id, (char *)(&ipc_head + bytes),
                    sizeof(ipc_head) - bytes, 0,
                    &cbs_sckt_addr, cbs_addr_len);
    }
}
```

To allow component activity to be scheduled around its interactions with other system components, the component must suspend its execution until a communication is received from CIM-BIOSYS, or until four seconds have elapsed. In this case the component must send an IPC_ACK_IND heartbeat packet to CIM-BIOSYS. Once either of these events occurs, the component must execute appropriate actions or interactions and then suspend its activity.

The function defined below implements the suspension of execution using the Unix operating system call select. The function returns when the component has been
struct s_ebs_mess *wait_for_event()
{
  int result, waiting, addr_len, fd_tab_size = 64;
  fd_set ifds;
  struct timeval interval;
  struct s_ebs_mess *cbs_mess_ptr;
  struct s_ipc_head *ipc_head_ptr;
  struct sockaddr sock_addr_ptr;
  interval.tv_usec = 0;
  interval.tv_sec = 4;
  waiting = TRUE;
  while (waiting == TRUE)
  {
    memcpy((void *)&ifds, (void *)&ev_inputfds, sizeof(fd_set));
    result = select(fd_tab_size, &ifds, NULL, NULL, &interval);
    if(result == 0) send_command(IPC_ACK_IND, 0);
    if((result > 0) && (FD_ISSET(sock_id, &ifds))
    {
      recvfrom(sock_id, buffer, S_LEN, 0, &sock_addr_ptr, &addr_len);
      ipc_head_ptr = (struct s_ipc_head *)buffer;
      switch(ipc_head_ptr->cmd)
      {
        case IPC_ABORT_IND:
          send_command(IPC_ABORT_IND, 0);
          aborting = TRUE;
          break;
        case IPC_TERM_REQ:
          send_command(IPC_TERM_REQ, 0);
          break;
        case IPC_TERM_CMD:
          cbs_mess_ptr = (struct s_cbs_mess *)
            (buffer+sizeof(struct s_ipc_head));
          break;
        case IPC_DATA:
          wait = FALSE;
          break;
        default
          break;
      }
    }
    return(cbs_mess_ptr);
  }
}

To handle the events described by the component's design the function
process_event has been defined. The function must filter out responses to previ­
ous interaction requests and badly formed data packets. Within the program, the dis­
tinction between events which have some meaning in the design and events
concerned with the execution of a computer process, is drawn by declaration of sep­
arate functions. This distinction is purely arbitrary and only reflects the desire of the
implementor to retain design semantics within the implemented system. However,
there is no guarantee that this distinction will be communicated to a third party who
may be attempting to understand or modify the program. An alternative strategy
would be to make all computer based events visible within the component's design.
However, this removes the power of the design abstractions adopted.
In order to register the occurrence of an event defined by the design of the component, the function `process_event` manipulates globally defined variables to determine the subsequent operations and interactions to be undertaken. The values associated with these variables can be seen as contributing to a definition of the state of the component at a given moment in time.

```c
void process_event(cbs_mess_ptr)
    struct s_cbs_mess *cbs_mess_ptr;
    char *data;
    int len;
    if(cbs_mess_ptr->check_bytes[0]=='x' &&
        cbs_mess_ptr->check_bytes[1]=='z')
    { if(cbs_mess_ptr->cmd > CBS_RESP_IND)
        { if(cbs_mess_ptr->status < 0)
            { fprintf(stderr,"Failure to ... (error %d)\n",
                cbs_mess_ptr->int_arg1);
            } else
                { cbs_mess_ptr->cmd -= CBS_RESP_IND;
                    switch(cbs_mess_ptr->cmd)
                    { case CBS_EST_LINK : connections++;
                        break;
                    case CBS_SEND_APP : break;
                    default : break;
                    }
                } else
                { if(cbs_mess_ptr->cmd == CBS_SEND_APP)
                    { data = (char *)cbs_mess_ptr->data;
                        len = (char *)cbs_mess_ptr->data_len;
                        if(sscanf(data,"print(%d)\n", &batchSize) == 1)
                            { todo += batchSize;
                                input_buffer += batchSize;
                                if(made == -1)made = 0;
                            }
                        if(!strncmp("moveComplete", data, len)) moveComplete++;
                        if(!strncmp("printComplete", data, len)) printComplete++;
                        if(!strncmp("inspect("Failed")", data, len)) inspect_Failed++;
                        if(!strncmp("inspect("OK")", data, len)) inspect_OK++;
                    }
                }
        } else
            { if(cbs_mess_ptr->check_bytes[0]=='x' &
                cbs_mess_ptr->check_bytes[1]=='z')
                { if(cbs_mess_ptr->cmd > CBS_RESP_IND)
                    { if(cbs_mess_ptr->status < 0)
                        { fprintf(stderr,"Failure to ... (error %d)\n",
                            cbs_mess_ptr->int_arg1);
                        } else
                            { cbs_mess_ptr->cmd -= CBS_RESP_IND;
                                switch(cbs_mess_ptr->cmd)
                                { case CBS_EST_LINK : connections++;
                                    break;
                                case CBS_SEND_APP : break;
                                default : break;
                                }
                            } else
                            { if(cbs_mess_ptr->cmd == CBS_SEND_APP)
                                { data = (char *)cbs_mess_ptr->data;
                                    len = (char *)cbs_mess_ptr->data_len;
                                    if(sscanf(data,"print(%d)\n", &batchSize) == 1)
                                        { todo += batchSize;
                                            input_buffer += batchSize;
                                            if(made == -1)made = 0;
                                        }
                                    if(!strncmp("moveComplete", data, len)) moveComplete++;
                                    if(!strncmp("printComplete", data, len)) printComplete++;
                                    if(!strncmp("inspect("Failed")", data, len)) inspect_Failed++;
                                    if(!strncmp("inspect("OK")", data, len)) inspect_OK++;
                                }
                            }
                        }
                    }
                }
            }
    }
}
```

To allow the component to interact with other system components, both a `struct s_cbs_mess` and a `struct s_ipc_head` abstract data type must be populated with relevant data. The data types are then concatenated and sent to CIM-BIOSYS using BSD sockets as previously discussed. This sequence of statements can form the
body of a procedure send_message which maps onto one of the abstractions held within the component design.

```c
void send_message(who, message)
char *who, *message;
{
    struct s_cbs_mess cbs_mess;
    struct s_ipc_head ipc_head;
    char buffer [S_LEN + 2];
    int bytes, n;
    cbs_mess.cmd = CBS_SEND_APP;
    cbs_mess.status = 0;
    cbs_mess.check_bytes[0] = 'x';
    cbs_mess.check_bytes[1] = 'z';
    cbs_mess.seq_no = id++;
    cbs_mess.exp_flag = FALSE;
    cbs_mess.permit = 0x7FFF;
    cbs_mess.int_arg1 = 0;
    cbs_mess.int_arg2 = 0;
    cbs_mess.int_arg3 = 0;
    gethostname(cbs_mess.str_arg1,NAME_LEN);
    strcpy(cbs_mess.str_arg2,who);
    strcpy(cbs_mess.str_arg3,my_name);
    cbs_mess.data_len = strlen(message) + 1;
    memcpy(cbs_mess.data, message, cbs_mess.data_len);
    strcpy(ipc_head.sender, my_name);
    strcpy(ipc_head.target,"cbs");
    strcpy(ipc_head.arg1,my_socket_name);
    ipc_head.sender_proc = APP_PROC;
    ipc_head.cmd = IPC_DATA;
    ipc_head.arg1 = CBS_MESS_OHEAD+cbs_mess.data_len;
    memcpy(buffer, (char *) &ipc_head,IPC_PCKT_OHEAD);
    memcpy((char *)buffer + IPC_PCKT_OHEAD , (char *) &cbs_mess,ipc_head.arg1);
    for (bytes = 0; bytes < (ipc_head.arg1+IPC_PCKT_OHEAD)-bytes ; bytes += n)
    { n = sendto(sock_id,buffer + bytes,
                 (ipc_head.arg1+IPC_PCKT_OHEAD) - bytes,
                 0,&cbs_sckt_addr,cbs_addr_len);
    }
}
```

A similar function can be declared to form associations between two components wishing to interact, this function has been omitted for brevity.

Once the ability to interact with other system components and to schedule activity around those interactions has been established, it only remains to define the behaviour of the component print within the program. This entails: inspecting the global variables used to register the occurrence of events, manipulating additional variables used to define the state of the component, waiting for the arrival of messages from other components and sending messages when appropriate. The encoding of these details is a relatively straightforward process, although as the representation of the component’s state is dispersed among a number of variables, coupling between seemingly separate conditions can occur. For example, in the following program segment the correct ordering of two of the if statements which test the variables
inspect\_Failed and input\_buffer is essential to the component fulfilling its design. If their order is reversed and a PCB fails the inspection process, then a deadlock will occur.

```c
while(running == TRUE)
    { if(made >= toDo)
        { running = FALSE;
            break;
        }
    if(inspect\_Failed && (inspecting == TRUE))
        { inspecting = FALSE;
            input\_buffer++;
            inspect\_Failed--;
        }
    if((input\_buffer > 0) && (printer\_free == TRUE))
        { send\_message("printer\_Feed", "move\_To\_Printer");
            printer\_free = FALSE;
            input\_buffer--;
            loading++;
        }
    if(move\_Complete && (loading > 0))
        { send\_message("printer", "print\_PCB");
            loading--;
            printing++;
            move\_Complete--;
        }
    if(print\_Complete && (printing > 0) && (inspecting == FALSE))
        { send\_message("printer\_Check", "inspect\_PCB");
            printing--;
            printer\_free = TRUE;
            inspecting = TRUE;
            print\_Complete--;
        }
    if(inspect\_OK && (inspecting == TRUE))
        { inspecting = FALSE;
            made++;
            sprintf(tmp, "printed\$(\#d", made); send\_message("assembly", tmp);
            inspect\_OK--;
        }
    cbs\_mess\_ptr = wait\_for\_event();
    if(aborting == TRUE) break;
    process\_event(cbs\_mess\_ptr);
}
```

Finally when the component has been instructed to terminate by the computer infrastructure a number of tidying up operations should be executed which are illustrated by the following statements

```c
send\_command(IPC\_ABORT\_IND, 0);
unlink(my\_sckt\_addr.sa\_data);
printf("Component \$s terminating\n", 256, my\_name);
return(0);
```

This example indicates some of the problems inherent with the implementation of software systems. During the implementation process the abstractions held within the design are transformed to those suitable for execution by a computer infrastructure, resulting in the creation of computer programs. Due to the availability of high-

---

93
A Manufacturing Control System Implementation

level programming languages, the resulting programs are intelligible to the human reader. However, as programming languages are primarily based on the architecture of the underlying hardware upon which programs are executed, many of the key system design abstractions become lost within a maze of program statements, abstract data types and implementation idioms. Understanding the connectivity between one part of a program and another may be difficult. Understanding the connectivity between one part of a program and another part of a different program with which it interacts, can be a complex problem to master.

As the programs executing within a system define its behaviour, they are the artifacts that must be understood and maintained. Due to the nature of programming languages and computer infrastructures these artifacts do not contain a clear expression of system behaviour.

7.3 The Proposed Approach to System Implementation

The main task when implementing a component design using the approach proposed in the thesis is the translation of design constructs into those offered by the author's manufacturing domain machine. In the case of the component print this is a simple task as the design is expressed as an interpreted Petri net. The following model is a textural representation of the design shown in Figure 29

```
input (startBatch, []).
input (fillInput, []).
input (load, [inputBuffer, printerFree]).
input (loaded, [loading]).
input (printed, [printing, inspectionFree]).
input (inspectionFailed, [inspecting]).
input (inspectionOK, [inspecting]).
input (finished, [outputBuffer]).

output (startBatch, [inputBuffer]).
output (fillInput, [inputBuffer]).
output (load, [loading]).
output (loaded, [printing]).
output (printed, [printerFree, inspecting]).
output (inspectionFailed, [inputBuffer, inspectionFree]).
output (inspectionOK, [outputBuffer, inspectionFree]).
output (finished, [terminate]).

place (inputBuffer, 0).
place (loading, 0).
place (printing, 0).
place (printerFree, 1).
place (inspecting, 0).
place (inspectionFree, 1).
place (outputBuffer, 0).
place (terminate, 0).
```
transition(startBatch, print(bSize) from true, [n is @n+bSize-1, todo is @todo+bSize]).
transition(fillInput, noEvent, [n > 0, [n is @n - 1]].
transition(load, noEvent, true, [moveToPrinter to printerFeed]).
transition(loaded, moveComplete from true, [printPCB to printer]).
transition(printed, printComplete from true, [inspectPCB to printCheck]).
transition(inspectionFailed, inspect("Failed") from true, []).
transition(inspectionOK, inspect("OK") from true, [made is @made+1, printed(@made) to assembly]).
transition(finished, noEvent, [made >= @todo, [10 secondWait]].

store(bSize, 0)
store(n, 0)
store(todo, 0)
store(made, 0)

This model retains the concepts of component behaviour and interaction expressed within the design. It can also be directly executed by the author's domain machine using the same computer and distribution infrastructure as the C program discussed in the previous section.

However, if we examine the UML state diagram for the component assembly it is evident that some translation from the state transition notation used to that of an interpreted Petri net is required. The distinct states of the component are represented by the UML construct state, which maps onto the behaviour construct place in the following manner.

The state transitions specified using UML relate two states or one state to itself, these constructs map onto the behaviour constructs in the following way:

\[
\text{transition}(t_1, \text{from } \text{stateA to stateB})
\]

\[
\text{input}(t_1, \text{[stateA]}).
\]

\[
\text{output}(t_1, \text{[stateB]}).
\]

\[
\text{transition}(t_2, \text{from stateC to stateC})
\]

\[
\text{input}(t_2, \text{[stateC]}).
\]

\[
\text{output}(t_2, \text{[stateC]}).
\]
and the transitions specified by UML can possess event indications, guard conditions, event triggers and operations. These constructs map as follows:

\[
\text{event}(\text{arguments}) \quad \text{transition}(t, \text{event}(\text{arguments}), \text{condition}, [\text{sendEvent}(\text{arguments}) \text{ to target, operation}(\text{arguments})]).
\]

Therefore, the translation between the notation of UML state diagrams and the constructs supported by the domain machine is relatively straightforward. As a system design is often produced using a computer based tool this translation process can easily be automated. To provide an example automated translation process, the author used Paradigm Plus [Platinum96] to create the designs contained within this chapter. Paradigm Plus provides a utility for generating customised reports from existing designs using a proprietary scripting language. The following script extract illustrates how the transition constructs can be generated from a UML state diagram.

```plaintext
printTransitions:
foreach initiator
    foreach initiator.event.state
        print "transition("owner$",noEvent,true,[])."
    next
next
foreach state
    foreach state.event.state
        print transition(""event.name","","event.event","",print event.guard,"","event.operation,")."
    next
return
```

In addition facilities exist within the scripting language to create designs, thus enabling existing system designs expressed using the interaction and behaviour constructs to be translated into UML notation and depicted using its graphical syntax.

Once the mapping between a particular design notation and the author's constructs is established, only the mechanistic exercise of translation remains in order to implement the behaviour and interactions described within the design. This is illus-
trated by the following model of the component assembly generated by the Paradigm Plus tool

input(1, [waitingOnBatch]).
input(2, [printOnly]).
input(3, [printOnly]).
input(4, [printMake]).
input(5, [printMake]).
input(6, [printMake]).
input(7, [makeOnly]).
input(8, [makeOnly]).

place (waitingOnBatch, 1, 0).
place (printOnly, 0, 0).
place (printMake, 0, 0).
place (makeOnly, 0, 0).

store (bSize, 0).
store (made, 0).
store (amount, 0).

transition(1, start(bSize) from, true, [print(bSize) to print, made is 0]).
transition(2, printed(amount) from, @amount < @bSize, [popPCB to populate]).
transition(3, printed(amount) from, @amount >= @bSize, [popPCB to populate]).
transition(4, printed(amount) from, @amount < @bSize, [popPCB to populate]).
transition(5, populated(amount) from, true, [made is @made + @amount]).
transition(6, printed(amount) from, @amount >= @bSize, [popPCB to populate]).
transition(7, populated(amount) from, true, [made is @made + @amount]).
transition(8, noEvent, @made >= @bSize, [finished(@made) to scheduler]).

which is a representation of the state diagram (shown in Figure 29) expressed using the constructs supported by the domain machine.

7.4 Comparison of Conventional and Proposed Implementations

This section examines how the approaches to system implementation outlined above accommodate change. Many types of change can occur within a software system, but they can be broadly classified into two types:

1. Implementation - a change in the way a system is implemented, and;
2. Functional - a change in the functionality of the system.

Changing the way in which a system is implemented would not normally involve a change to the design of the system. Such a change may involve part of the computer infrastructure such as operating systems, computer hardware or communication networks, or a change of the programming language used to define a particular system component. Although sometimes involved, this type of change is well accommodated by the use of standardised operating system interfaces, such as POSIX [IEEE95], adoption of distribution infrastructures, such as CORBA [OMG94],
standardised programming languages, such as ANSI C and good program design techniques.

Of greater impact is the ability to change the functionality of a system or of individual system components. This usually entails a change in the system design followed by a modification of the system components to reflect the design change. For standalone programs this type of change may be relatively straightforward. However, this is not the case with software systems, where a change within one component may have unforeseen ramifications elsewhere within the system. Although abstractions adopted within the system design may help in visualising changes to an existing system, realising these changes within the implemented system involves understanding and modifying the programming language representation of system components and ensuring any changes made agree with the new system design.

As an illustration of this problem consider the following change to the system described in this chapter. As surface mount technology progresses, a superPrinter is produced which can not only print solder paste onto a PCB but is equipped with a computer controlled vision system capable of inspecting the printed PCB. The superPrinter also possesses the capability of reworking, i.e. washing off and reprinting, misprinted PCBs which fail the inspection process. Such a device could replace the existing printing facility within the system.

A Change to the System Design

To accommodate such a device the software system described earlier would also require modification. Changing the system design is a relatively straightforward process and may result in a design change such as that shown in Figure 30.

```
A1 printPCB
A2 printComplete
A3 inspectPCB
A4 inspect(status)
A5 rework
```

**Figure 30. Proposed Change to Assembly System**
In the new system design the component print has been removed, as the task of coordinating three separate software components no longer exists. However, the task of handling batch requests and monitoring the state of the printing process remains, and these tasks can be subsumed within the component assembly. This system modification leads to a change in the design of the assembly component as shown in Figure 31.

Figure 31. UML State Diagram for New Assembly Component
It involves the addition of two new states, inspectOnly and inspectMake, five additional state transitions and three additional component interactions. The overall effect of such a change may even be considered a simplification of the overall system design.

A Change to the Conventional Implementation of assembly

This section discusses how the design changes made to the component assembly can be accommodated within the existing software system. Rather than use the unstructured program segments discussed earlier for the component print, a more realistic and well-structured program will be considered. The following program has been written in the object oriented programming language C++ [Stroustrup91], and has been structured by adopting two design patterns [Gamma95].

The object oriented paradigm supports abstraction by providing facilities to create abstract data types with associated sets of operations. These operations, termed methods, are defined for and characterise the behaviour of the abstract data type [Wiener88]. Design patterns aim to capture solutions to specific problems in object oriented software design. Design patterns detail solutions that have developed and evolved over time, in that they reflect the experience and knowledge gained as developers have struggled for greater reuse and flexibility in their software.

The design patterns adopted are State and Singleton. The State pattern has been used to provide a clear representation of the state of the assembly component, in an attempt to retain some of the semantics of the component’s design. A justification for this approach is provided by Gamma:

"Like long procedures, large conditional statements are undesirable. They're monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend. The state pattern offers a better way to structure state-specific code. The logic that determines the state transitions doesn't reside in monolithic if or switch statements but instead is partitioned between the State subclasses. Encapsulating each state transition and action in a class elevates the idea of an execution state to full object status. That imposes structure on the code and makes its intent clearer" [Gamma95].
The Singleton pattern has been adopted to ensure that the assembly component can only be in one state at any given moment in time. This is achieved by ensuring that only one object representing the component's state, can exist at any given time.

The program itself consists of a class `Assembly`

```cpp
class Assembly
{
public:
    Assembly();
    void Init(int* argc, char** argv);
    void Run();
    void DispatchEvent(char*);
    void Start(int);
    void Printed(int);
    void Populated(int);
protected:
    friend class State;
    void Send(char*, char*);
    void ChangeState(State*);
private:
    EventHandler *_eventHandler;
    State *_state;
};
```

with each of its methods defined as follows

```cpp
Assembly::Assembly()
{ 
    _eventHandler = new EventHandler;
    _state = Waiting::Instance();
}

void Assembly::Init(int* argc, char** argv)
{ 
    _eventHandler->Init(this, argc, argv);
    _eventHandler->Association("print");
    _eventHandler->Association("populate");
}

void Assembly::Run()
{ 
    _eventHandler->Dispatch(this);
}

void Assembly::DispatchEvent(char* event)
{ 
    int anAmount;
    if (sscanf(event, "start (%d)", &anAmount) == 1) 
    { 
        Start(anAmount);
    }
    if (sscanf(event, "printed (%d)", &anAmount) == 1) 
    { 
        Printed(anAmount);
    }
    if (sscanf(event, "populated (%d)", &anAmount) == 1) 
    { 
        Populated(anAmount);
    }
}

void Assembly::ChangeState(State *s)
{ 
    _state = s;
}
```
void Assembly::Send(char* who, char* message)  
{  
  _eventHandler->Send(who, message);  
}

void Assembly::Start(int BSize)  
{  
  _state->Start(this, BSize);  
}

void Assembly::Printed(int Amount)  
{  
  _state->Printed(this, Amount);  
}

void Assembly::Populated(int Amount)  
{  
  _state->Populated(this, Amount);  
}

To enable the component to schedule its activity around the occurrence of events identified within the design, the class EventHandler has been created, and is defined as follows:

class EventHandler
{
  public:
    ~EventHandler();
    void Init(Assembly*, int*, char**);
    void Association(char* who);
    void Send(char*, char*);
    void Dispatch(Assembly*);
};

the methods for EventHandler implement much of what was discussed earlier in this chapter for the component print. For brevity, only an annotated method prototype is included:

void EventHandler::Init(Assembly* a, int* argc, char** argv)  
{  
  // Initialise link with Distribution Infrastructure  
};  

EventHandler::~EventHandler()  
{  
  // Terminate link with Distribution Infrastructure  
};

void EventHandler::Association(char* who)  
{  
  // Encode connection service request  
  // Send service request to Distribution Infrastructure  
};

void EventHandler::Send(char* who, char* message)  
{  
  // Encode message into service request  
  // Send service request to Distribution Infrastructure  
};

void EventHandler::Dispatch(Assembly* a)  
{  
  // Wait for a component level event to occur  
  // Call Assembly method DispatchEvent(char*)  
}
To enable a clear representation of the component’s state at any given time, the class **State** has been defined

```cpp
class State {
public:
    virtual ~State();
    virtual void Start(Assembly*, int);
    virtual void Printed(Assembly*, int);
    virtual void Populated(Assembly*, int);
protected:
    static int BatchSize, AmountMade;
    void ChangeState(Assembly*, State*);
    void Send(Assembly*, char*, char*);
    static State* Setlnstance(State*);
    static State* _instance;
};
```

and the data items and methods for this class are defined as follows

```cpp
State* State::_instance = 0;
int State::BatchSize;
int State::AmountMade;

State* State::Setlnstance(State* s)
{ if(_instance) delete(_instance);
    _instance = s;
    return(_instance);
}

void State::Start(Assembly* a, int BSize);
void State::Printed(Assembly* a, int Amount);
void State::Populated(Assembly* a, int Amount);

void State::ChangeState(Assembly* a, State* s)
{ a->ChangeState(s);
}

void State::Send(Assembly* a, char* who, char* message)
{ a->Send(who, message);
}
```

The class **State** provides the default behaviour for each state of the component assembly. To define the behaviour of the component for each of its particular states, four specialisations (or sub-types) are defined

```cpp
class Waiting : public State {
public:
    static State* Instance();
    virtual void Start(Assembly*, int);
};

class PrintOnly : public State {
public:
    static State* Instance();
    virtual void Printed(Assembly*, int);
};
```
A Manufacturing Control System Implementation

class PrintMake : public State
{
public:
  static State* Instance();
  virtual void Printed(Assembly*, int);
  virtual void Populated(Assembly*, int);
};

class MakeOnly : public State
{
public:
  static State* Instance();
  virtual void Populated(Assembly*, int);
};

each class then defines the behaviour of the component assembly when in each of its
four states

State* Waiting::Instance()
{  return( SetInstance( new Waiting()));
}

void Waiting::Start(Assembly* a, int BSize)
{  State::BatchSize = BSize;
   State::AmountMade = 0;
   ofstream buffer;
   buffer << "print(" << BSize << ")" << '\0';
   Send(a,"print",buffer.str());
   ChangeState(a,PrintOnly::Instance());
}

State* PrintOnly::Instance()
{  return( SetInstance( new PrintOnly()));
}

void PrintOnly::Printed(Assembly* a, int Amount)
{  Send(a,"populate","popPCB");
   if(Amount >= State::BatchSize)
      { ChangeState(a,MakeOnly::Instance());
   }  else
      { ChangeState(a,PrintMake::Instance());
   }
}

State* PrintMake::Instance()
{  return( SetInstance( new PrintMake()));
}

void PrintMake::Printed(Assembly* a, int Amount)
{  Send(a,"populate","popPCB");
   if(Amount >= State::BatchSize)
      { ChangeState(a,MakeOnly::Instance());
   }
}

State* MakeOnly::Instance()
{  return( SetInstance( new MakeOnly()));
}
void MakeOnly::Populated(Assembly* a, int Amount)
{
    ofstream buffer;
    State::AmountMade += Amount;
    if (State::AmountMade >= State::BatchSize)
    { buffer << "finished(\n State::AmountMade << ") << '\0';
      Send(a,"scheduler",buffer.str());
      ChangeState(a,Waiting::Instance());
    }
}

To execute the behaviour defined by the above program all that remains is for the component to create an assembly object upon invocation, initialise it and call the method Run

main(int argc,char *argv[])
{
    Assembly a;
    a.Init(&argc,argv);
    a.Run();
}

To modify the above program to reflect the suggested changes to the component’s design the following prerequisite knowledge is required:

• an understanding of the particular object oriented concepts contained within the C++ programming language as well as its syntax. For example, understanding the role of the virtual destructor defined within the class State;

• an understanding of particular design patterns adopted. Although considered a laudable attempt to communicate good program design practices, they require some prerequisite implementation knowledge. For example, the State pattern relies on the object oriented concept of polymorphism, in order to change the behaviour of the component when in various states, and;

• an understanding of all the issues discussed in the previous implementation section (7.2).

The change itself would require the creation of two new state subclasses to represent the new states expressed within the design, and modification of three of the existing subclasses to reflect the new state transitions. Both the Assembly class and State class would require modification to handle the new events declared within the design. Although by no means prohibitive these requirements must be understood and resolved.
A Change to the Design-Based Representation of assembly

If the approach to system implementation proposed in this thesis were adopted, none of the issues encountered using the programming language based approach would arise. The suggested change to the design could be automatically translated, using the means previously described, into constructs supported by the domain machine. This would result in the following model being produced

\[
\begin{align*}
\text{input} (1, \{\text{waitingOnBatch}\}) & \quad \text{output} (1, \{\text{printOnly}\}). \\
\text{input} (2, \{\text{printOnly}\}) & \quad \text{output} (2, \{\text{inspectOnly}\}). \\
\text{input} (3, \{\text{inspectOnly}\}) & \quad \text{output} (3, \{\text{makeOnly}\}). \\
\text{input} (4, \{\text{inspectOnly}\}) & \quad \text{output} (4, \{\text{printMake}\}). \\
\text{input} (5, \{\text{inspectOnly}\}) & \quad \text{output} (5, \{\text{printOnly}\}). \\
\text{input} (6, \{\text{printMake}\}) & \quad \text{output} (6, \{\text{inspectMake}\}). \\
\text{input} (7, \{\text{printMake}\}) & \quad \text{output} (7, \{\text{printMake}\}). \\
\text{input} (8, \{\text{inspectMake}\}) & \quad \text{output} (8, \{\text{printMake}\}). \\
\text{input} (9, \{\text{inspectMake}\}) & \quad \text{output} (9, \{\text{printMake}\}). \\
\text{input} (10, \{\text{inspectMake}\}) & \quad \text{output} (10, \{\text{makeOnly}\}). \\
\text{input} (11, \{\text{inspectMake}\}) & \quad \text{output} (11, \{\text{inspectMake}\}). \\
\text{input} (12, \{\text{makeOnly}\}) & \quad \text{output} (12, \{\text{makeOnly}\}). \\
\text{input} (13, \{\text{makeOnly}\}). & \quad \text{output} (13, \{\text{waitingOnBatch}\}). \\
\end{align*}
\]

\[
\begin{align*}
\text{place} (\text{waitingOnBatch}, 1, 0). \\
\text{place} (\text{printOnly}, 0, 0). \\
\text{place} (\text{inspectOnly}, 0, 0). \\
\text{place} (\text{printMake}, 0, 0). \\
\text{place} (\text{inspectMake}, 0, 0). \\
\text{place} (\text{makeOnly}, 0, 0). \\
\text{store} (\text{bSize}, 0). \\
\text{store} (\text{made}, 0). \\
\text{store} (\text{amount}, 0). \\
\text{store} (\text{printed}, 0); \\
\end{align*}
\]

\[
\begin{align*}
\text{transition}(1, \text{start(bSize)} \text{ from}, \text{true}, \{\text{printPCB to superprint, made is 0}\}). \\
\text{transition}(2, \text{printComplete from}, \text{true}, \{\text{inspectPCB to superprint}\}). \\
\text{transition}(3, \text{inspect("OK")} \text{ from}, \text{@bSize <= 1, \{popPCB to populate\}}). \\
\text{transition}(4, \text{inspect("OK")} \text{ from}, \text{@bSize > 1,} \\
\quad \{\text{popPCB to populate, printPCB to superprint, printed is @printed + 1}\}). \\
\text{transition}(5, \text{inspect("Failed") from}, \text{true, \{reworkPCB to superprint\}}). \\
\text{transition}(6, \text{printComplete from}, \text{true}, \{\text{inspectPCB to superprint}\}). \\
\text{transition}(7, \text{populated(amount) from}, \text{true, \{made is @made + @amount\}}). \\
\text{transition}(8, \text{inspect("OK") from}, \text{@printed < @bSize,} \\
\quad \{\text{popPCB to populate, printPCB to superprint, printed is @printed + 1}\}). \\
\text{transition}(9, \text{inspect("Failed") from}, \text{true, \{reworkPCB to superprint\}}). \\
\text{transition}(10, \text{inspect("OK") from}, \text{@printed >= @bSize,} \\
\quad \{\text{popPCB to populate}\}). \\
\text{transition}(11, \text{populated(amount) from}, \text{true, \{made is @made + @amount\}}). \\
\text{transition}(12, \text{populated(amount) from}, \text{true, \{made is @made + @amount\}}). \\
\text{transition}(13, \text{noEvent}, \text{@made >= @bSize, \{finished(@made) to scheduler\}}). \\
\end{align*}
\]

The new design could then be executed directly using the domain machine. Therefore, the removal of an intermediary representation of the component's design, expressed in an imperative programming language, would ease modification of the behaviour of the system component.
7.5 Results and Discussion

The discussion within this chapter is based on a software system which controls the operation of a PCB assembly line. The issues concerning the creation and modification of system components would inevitably be more involved within even larger systems.

If the change process took place according to good software engineering practice, the UML design diagrams would first be modified regardless of how the system was implemented. The design process is required in order to formally represent the new system and ensure it meets the new system requirements.

Having completed the design modification, the proposed new approach only requires the automated translation process to be initiated in order to create the new executable design model. This executable model included at the end of the previous section contains a total of 26 lines of automatically created code.

Table 1 provides a quantitative representation of the work involved to make the required change to the conventional C++ system implementation. This gives an impression of the scale of the modification required in a conventional system implementation, relative to the very simple change made to the system design. In this case a total of 108 lines of code were added or modified.

What the quantitative analysis does not embrace is the degree of “craft” required to implement a system design. This is due to the absence of a formalism to map the semantics of the system design onto those offered by a programming language. It
becomes difficult to differentiate the semantics of the system from those of the programming language.

Although programming languages and practices are becoming more sophisticated, they remain weak in their ability to retain system design decompositions and abstractions within implemented systems. Frequently a burden is placed on the system implementor, who has to master elaborate programming language extensions and implementation techniques which do little to retain system design decompositions and abstractions.
Chapter 8 - Banking Information Systems

Within the previous four chapters the author’s approach has been applied to an application in the manufacturing domain and compared with contemporary software design and build techniques to establish the benefits of the approach. The manufacturing application is an example of control software, i.e. a piece of software whose role is to control and monitor process machines in a manufacturing production facility. In order to demonstrate the general nature of the approach Chapter 8 and Chapter 9 describe how the author’s proposed framework is applied in a different domain and to a different style of application. The new domain is banking, and the nature of the application is an information and business process support system. The business process used in the work is foreign exchange dealing.

This chapter discusses the concepts and characteristics of the banking domain. These are then defined in terms of process and information models and the capability to interact with and manipulate such models. The chapter continues by describing how the author’s proposed general framework for domain machines can be applied to the banking domain, and which technologies can be used to populate the framework.

8.1 Banking Domain Concepts

The banking IT system investigated in conjunction with this work is typical of the systems operating in the City of London in the late 1990s, it provided support for “Front Office” trading and “Back Office” settlement. The original IT implementation comprised a packaged system which provided general banking services such as loans, and retail banking. Such vendor supplied packaged systems are always a compromise, aiming to meet most of the requirements of most of the banks. Over a number of years the central system was continually modified to support better the bank’s particular processes.

The original system has been supplemented by additional systems in the front office such as those which support derivatives trading, foreign exchange dealing and the input of real-time global financial information. In the back office a global bank-to-bank fund transfer system, a derivatives settlement system and a group risk consolidation system are typical major additions. These systems are implemented on a
range of computer platforms using various solutions from bespoke client-server applications to customised spreadsheets. All necessary integration between these large grain components of the system are implemented through bespoke links.

The bank’s business support software systems typically comprise a combination of configured or modified vendor packages integrated with in-house applications built to support the operations that are unique to the bank. In broad terms, these systems are required to:

• enable data entry;
• store information;
• provide various views of, and generate various reports from, stored information, and;
• model and support financial transactions.

In general, these information and business support systems do not control business processes, they provide information and guidance in order to facilitate the control of business processes by people [Jackson95]. Figure 32 illustrates this point, highlight-
which directly drives real world machines). A foreign exchange dealer or derivatives trader will do deals over the phone while using information provided by their support IT system. Dependant on the level of support provided the dealers may then be able to confirm and settle deals using their IT systems, but initiation of deals, and control of the dealing process is in the hands of the dealer.

To enable an information and business support system to provide relevant information and provide process support we must produce a model of the real world of banking and embody that model within the system. In effect, the system becomes a simulation of the real world, and derives its information directly from its model, and only indirectly from the real world itself.

What is required to implement such systems is:
• an environment which supports a domain process model that is aligned to real world business concepts and processes, and;
• a set of supporting services which allow for information transactions with the domain model. Where transactions are defined as a communicative action or activity involving two parties that reciprocally affect or influence each other.

These characteristics are significantly different from those of the previously considered domain of manufacturing control systems, where the key characteristics of action, interaction and behaviour were used to create models that directly controlled a manufacturing process. The following two sections describe how the author’s proposed domain machine framework may be populated to support the two key banking characteristics, namely a banking process model and the means to transact with such a model.

8.2 Supporting Domain Process Models

The creation of consistent business models can be eased through the use of standards. In chapters 2 and 3 the Object Management Group (OMG) has been introduced and their support for component interaction via CORBA has been used in this thesis to support component interaction. Continuing work within the OMG has focused on the creation of a number of standard high-level component models, each representing the needs of an important computing market. The CORBA Finance specification targets financial services and accounting [OMG98].
8.2.1 CORBA Finance - Currency Specification

The purpose of the currency specification is to propose a standard for objects encompassing the concepts of currency and money. The currency specification uses value types as defined in the Objects-by-Value specification [OMG94]. These value types have minimal behaviour and contain mostly accessor methods. The primary purpose of these objects is to encapsulate data. These value types are used as parameters or return types to the business utilities.

There are abstractions for Currency, Money and ExchangeRate, plus abstractions to manipulate these including CurrencyBook, MoneyCalculator, MoneyFormatter and ExchangeRateManager. The following points, taken from the OMG Finance Specification, briefly describe these abstractions and their relationships:

- the Currency object contains operations for setting and accessing the attributes of a particular currency;
- the Money object contains operations for setting and accessing the amount and type of currency for a particular instance of money;
- the ExchangeRate operations will support conversions of money from one currency to money of another in addition to setting and accessing the attributes of a particular exchange rate;
- the CurrencyBook maintains a group of currencies. It is used by the MoneyFormatter to retrieve the currency symbol and by the MoneyCalculator to retrieve the base currency when converting to base currency;
- the ExchangeRateManager maintains exchange rates. It is used by the MoneyCalculator to retrieve an ExchangeRate to exchange Money;
- the MoneyCalculator is a utility used for performing money arithmetic. It supports a standard set of operations for arithmetic calculations and additional state operations to support rounding rules, precision settings, and conversion rules. These state settings are saved on a per-client basis;
- the MoneyCalculator uses the CurrencyBook to retrieve the base currency and the ExchangeRateManager to retrieve an appropriate ExchangeRate. The MoneyCalculator takes Money as parameters. There is a MoneyFormatter class utility used for parsing and formatting money into strings. The formatter is dependent upon state settings and therefore the identifier is used for all operations to iden-
tfy the application client. The MoneyFormatter takes Money as parameters. The MoneyFormatter uses Currency to retrieve the symbol.

8.2.2 The OMG's Interface Definition Language

In order to specify the services provided by a system component, OMG’s Interface Definition Language (IDL) is used. An IDL interface describes the functionality that will be provided by an object in an implementation independent fashion. This description is needed to develop client software which makes use of objects which support the interface. An interface typically specifies the attributes and operations, and includes the parameters of each operation. In order to achieve this, IDL possesses its own programming language independent data types and supports the creation of new data types, for example the following IDL declaration describes Money as offered by the currency specification.

```idl
value Money {
    CBO::Ddecimal getValue() raises(FbcException);
    void setValue(in CBO::DDecimal amount) raises(FbcException);
    wstring getCurrencyMnemonic() raises(FbcException);
    void setCurrencyMnemonic(in wstring mnemonic) raises(FbcException);
};
```

Money refers to a specific amount of a particular currency. The abstraction for Money must clearly support editing and reading this amount as well as the knowledge of its currency.

The Money value is supported with a decimal abstraction that is defined as a common object in the OMG's Common Business Object (CBO) module, a specification of objects that are common across all their defined vertical domains including CORBA Finance. The money definition above implies that currency is the type while money is the instance of the type, i.e. 3 pounds is an instance of the currency pounds. To support this, the money abstraction defined in the specification supports access to its unique currency identifier.

The author proposes that the constructs required by the banking domain can be realised by the adoption of the industry agreed standard business components specified by the OMG, and expressed in an implementation neutral form. Within a banking domain machine these constructs must be coupled with and translated onto the underlying computer infrastructure, in order to enable transactions with the domain process model.
8.3 Supporting Transactions with the Domain Process Model

During the 1990s World Wide Web (WWW) technologies have been established as the dominant means of building distributed information systems. When considering the new domain it was natural to adopt this popular technology to support transactions with the domain process model. At the heart of WWW technologies the Hypertext Transfer Protocol (HTTP) enables the invocation of information requests and delivery of the appropriate information responses. In a standard Web-based application the information contained within HTTP transactions is modelled using the Hypertext Markup Language (HTML) which is designed for and limited to describing the composition of Hypertext pages.

The success of the World Wide Web led to further exploitation of HTTP as a means to transfer application-specific information in a range of formats. In response, the Extensible Markup Language (XML) has been defined in order to provide a means to describe and structure application information.

Given its wide acceptance and suitability for supporting information transactions, HTTP was used as a means of providing a device-independent request/response protocol, while XML was used as the domain-independent data structuring language. The following two subsections provide details of both HTTP and XML.

8.3.1 Hypertext Transfer Protocol

HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which (through the extension of its request methods, error codes and headers) can be used for many tasks beyond its use for hypertext. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. However, practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods and headers that indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI), as a location (URL) or name (URN), for indicating the resource to which a method is to be applied. Messages are passed in a format similar to that used by Internet mail as defined by the Multipurpose Internet Mail Extensions (MIME). In this way, HTTP
allows basic hypermedia access to resources that are available from a diverse range of applications. It can be seen that HTTP mirrors the author's requirement for the separation of process execution from information transactions.

Operationally the HTTP protocol is a request/response protocol. A client sends a request to a server in the form of a request method, URI, and protocol version. This is followed by a MIME-like message containing request modifiers, client information, and possible body content. The server responds with a status line, including the message's protocol version and a success or error code. This is followed by a MIME-like message containing server information, entity meta-information, and possible entity-body content. Most HTTP communication is initiated by a user agent and consists of a request to be applied to a resource on some origin server. The synchronous nature of HTTP information flow provides a natural means to implement the reciprocal transactions within a domain process model.

8.3.2 Extensible Markup Language

XML is essentially a set of rules for designing text formats that let you structure your data. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML is extensible, platform-independent, and it supports internationalisation and localisation (i.e. fully Unicode-compliant). Like HTML, XML makes use of tags (words bracketed by '<' and '>') and attributes (of the form name="value"). While HTML specifies what each tag and attribute means, and often how the text between them will appear in a web browser, XML uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it. This allows XML to be used as the means to structure information content within banking information systems.

Like HTML, XML files are text files that people shouldn't have to read, but may when the need arises. Less like HTML, the rules for XML files are strict. A forgotten tag, or an attribute without quotes makes an XML file unusable, while in HTML such practice is tolerated and is often explicitly allowed. This makes XML suitable for automated parsing and generation, and ideal as a means for information flow between process executing within a domain machine. Since XML is a text format and it uses tags to delimit the data, XML files are nearly always larger than comparable binary formats.
8.4 Separating Banking Concepts from the Computer System

The author’s proposed framework for domain machines requires a substrate which is used as a foundation for supporting the services provided by the machine. This role was fulfilled by Sicstus Prolog in the author’s manufacturing domain machine. When considering the banking domain, a substrate was required to provide support for OMG-based standard models, and both the XML and HTTP transaction protocols. Developments in World Wide Web technology provided the logical solution to the search for a suitable substrate.

The increased level of sophistication of many Web sites built since the late 1990s, has led to the improved support for server-side functionality. These Web sites typically employ a “three-tier” architecture which separates the issues of data storage, business logic and presentation. Within this architecture each tier can be designed with regard to its own particular role, which has led to a number of tailored software products becoming available to ease the implementation process within each tier.

Application Servers are used to execute the business logic within the middle tier of a three-tier system. They provide support for scalable processing through the provision of numerous middleware facilities to handle such issues as distribution, application clustering and resource abstraction. Essentially they provide a platform-neutral environment in which to implement business logic, without any of the distractions associated with managing the access to the underlying computing infrastructure.

Application servers in the middle tier can be coupled with Web servers in the top tier, which allows the processes that encode business logic to be bound to the protocols of the World Wide Web. This combination of an Application Server and Web Server provides a suitable substrate which enables support for executing declarations of domain process models and the transactions which interact with these models.

8.5 Creating a Banking Domain Machine

As described throughout this thesis what is required within a software system is an approach that enables:
1. the support of design models which possess well-defined computational semantics, and;

2. a well-defined mapping from the model to the underlying computer infrastructure.

As such it is possible to apply the author's proposed framework as detailed in Chapter 3 to this new application domain. Figure 33 describes the technologies used in applying the framework to the banking domain and can be compared with Figure 11 in chapter 4 where the same general framework is populated with technologies suitable for implementing a manufacturing control system.

Within the author's banking domain machine the Tomcat [Tomcat] application server and Apache [Apache] Web server were used to provide the substrate. The OMG Currency Specification, HTTP and XML are bound to the Java programming language [Arnold00] in order to provide access to the domain-specific constructs, and Java itself was used to implement any addition functionality required within the domain machine. Details of these bindings are included in the following chapter.

The following chapter details the approach taken in implementing the banking domain machine and describes a "demonstrator" foreign exchange dealing applica-
tion originally implemented at Loughborough University and integrated within the information systems of a prestigious merchant bank in the City of London.
Chapter 9 - Banking System Implementation

This chapter describes the implementation of a foreign exchange dealing system that provides a further demonstration of how the author's approach can be successfully applied. The system is fully integrated with a distributed information system within a merchant bank in the City of London. The work was completed as part of a three year EPSRC funded research project supported by the bank.

Both the nature of the software and the application domain are very different from the manufacturing control system described earlier in the thesis. This further implementation is provided to demonstrate the general nature of the approach and no attempt is made to generate quantitative data regarding system change.

The following section details how the banking domain machine provides services for supporting process models, and how these services are connected to the underlying computer infrastructure. Section 9.2 provides similar detail for the provision of transaction services and their implementation within the banking domain machine. These two sections are analogous to the descriptions in chapters 5 and 6 describing the implementation of the manufacturing domain machine. The final section in the chapter deals with the implementation of the foreign exchange dealing system. The implementation is achieved through the execution of a foreign exchange process model and related transaction model using a banking domain machine.

9.1 Supporting Banking Domain Process Models

The approach adopted to support banking domain process models is depicted in Figure 34. The problem is broken down into three main concerns:

1. providing an environment which supports real-world domain concepts. This environment is supplied by using declarative financial process constructs;
2. realising these concepts. This is achieved by using imperative implementations of OMG currency objects, and;
3. mapping the OMG currency objects onto the underlying computer infrastructure. This mapping is achieved by using the bridge object design pattern.
9.1.1 Financial Process Constructs

The OMG Currency Specification [OMG98] was used in order to provide supporting constructs for a foreign exchange process model. The concept of Money, which is central to the currency specification, is used here to illustrate how the currency model was realised in the demonstration system.

Within the OMG Currency Specification the concept of Money is supported by a class hierarchy that provides a range of concepts from a fixed price (typically the value associated with an enacted transaction) to a fluctuating spot price (typically a share price from a real-time financial price feed such as Reuters). The issues involved with how such values are obtained from the existing IT system, must not be allowed to taint the pure business considerations that relate to this hierarchy. Some means of instantiating a monetary value must exist, but knowledge of the mechanisms employed to achieve this should be hidden. When implementing concepts such as Money, the software developer is then free to consider how these classes should support the services detailed in the OMG specification. The issues associated with actually providing for the storage and retrieval of the information which underpins the Money abstraction can then be addressed within a separate class hierarchy. As such, these issues which may be very complex when interfacing
with legacy IT systems, could be resolved by a separate developer who has relevant experience.

The constructs specified by the OMG provide the means of constructing an executable financial domain process model, but do not provide the means by which it can be executed by the domain machine as part of an externally defined transaction. To achieve this with the banking domain machine the “Command” design pattern was employed. This design pattern allows a request to be issued without any knowledge of the requested operation or the request receiver. Gamma [Gamma95] defines the command pattern as follows:

*Command - Encapsulate a request as an object, thereby letting you parameterize clients with different request, queue or log requests, and support undoable operations.*

The Command pattern decouples execution from invocation by encapsulating the implementation of functionality within an object, rather than within a function or method call. The object provides a known “execute” interface which is used by its client to invoke the functionality. The consequence of this approach is that commands become objects and can therefore be added and replaced at execution time without impact upon the calling process.

Within the author’s domain machine an abstract class named Banking Process was created to define the command interface required for all executable financial processes. Rather than adopt the pure interface approach used within the OMG currency specification an abstract class was employed within the domain machine. This allows the BankingProcess class to define and implement common financial process functionality such as access to runtime input/output parameters and the ability to resolve CORBA-defined object references. The interface is shown below

```java
package BankingProcess;

public abstract class BankingProcessModel{
    public virtual void execute();
    public string param(String s);
    public org.omg.CORBA.Object getObjectRef(String s);
}
```

In order to execute a particular financial process within the domain machine, the process must inherit from the class BankingProcessModel and define its process
within an implementation of the `execute()` method. Due to the use of inheritance, the defined process may use the methods already defined within `BankingProcessModel` which offer common financial process functionality without qualification.

### 9.1.2 Implementation of OMG Currency Objects

The OMG Specification provides the interface abstractions that concepts such as `Money` must fulfil. Within an implemented IT system these interfaces must be supported by real objects written using a concrete programming language. Within the author's banking domain machine the Java programming language and development kit (JDK) were used, primarily due to Java's adherence to the object oriented paradigm and it's support for heterogeneous distributed computing environments.

The OMG Currency specification was bound to the Java programming language using the IDL-to-Java compiler 'idltojava' supplied by Sun Microsystems. This process transforms the IDL declarations into Java interface declarations. The `Money` interface presented in the previous chapter is transformed by idltojava into the following Java code.

```java
package FbcCurrency;
public interface Money extends org.omg.CORBA.Object {
    void init(String currencyMnemonic, CBO.DDecimal theValue)
        throws FbcCurrency.FbcException;
    String getCurrencyMnemonic() throws FbcCurrency.FbcException;
    void setCurrencyMnemonic(String currencyMnemonic)
        throws FbcCurrency.FbcException;
    CBO.DDecimal getValue() throws FbcCurrency.FbcException;
    void setValue(CBO.DDecimal amount) throws FbcCurrency.FbcException;
}
```

A number of Java classes were then constructed to support this interface. The implementation comprises a class hierarchy which supports abstract notions of monetary values that relate to specific concepts such as spot prices (from a price feed) or prices associated with a particular financial contract. Each member of the class hierarchy implements the `Money` interface declared above, but behaves in accordance with the financial concept it implements. For example the price paid for a bond is fixed, but the spot price of a particular share changes with time. Hence, the continued invocation of the method `setValue(CBO.DDecimal amount)` will result in an exception being thrown for an object of type `Price`, but not for an object of type `SpotPrice`. 

---

122
In line with the design considerations described in the previous section the concept of a monetary value is implemented separately from how monetary values are mapped onto the underlying computer infrastructure. This is in line with the OMG value-based semantics for money. In essence, objects that support the Money interface are intended to be passed as parameters between business utilities. Therefore the implemented class hierarchy contains a number of relatively simple Java classes which contain no mechanisms to relate them to the underlying computer infrastructure within the Bank.

9.1.3 The Currency Bridge

In order to support the abstractions specified by the OMG Currency Specification and retain some flexibility as to how they were mapped onto the underlying computer infrastructure, the service abstractions (relating to the banking world) and the means of linking them to the computer infrastructure must be separate within the banking domain machine design. Some clearly defined means to distinguish these concepts had to be employed.

As in the previous manufacturing implementation, design patterns [Gamma95] have been adopted to provide a means to tackle different aspects of the design problem. One particular design pattern, the Bridge, enables the separation of the classification of domain abstractions from the details associated with a particular implementation.

The following provides a definition of the Bridge pattern:

\[ \text{Bridge - De-couple an abstraction from its implementation so that the two can vary independently.} \]

When adopting an Object Oriented approach to software design, if an abstraction can have several possible implementations, it is usual to accommodate each one by using class inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend and reuse abstractions and implemenations independently.

When using the Bridge design pattern two separate class hierarchies were constructed. One supports the well-defined abstractions required to support financial
services, while a separate class hierarchy reflects the implementation considerations for various financial IT systems. This allows the separate considerations that relate to the two different disciplines of business and technology to be handled using appropriate techniques, and so ensures that neither compromises the other. Such an approach also structures a solution that can aid future migration. Modifications can be handled within one class hierarchy without impacting upon the financial services supported by the other hierarchy. This flexibility is analogous to that described when using two different distribution infrastructure interfaces (ORBIX and CIM-BIOSYS) in the manufacturing controls system domain machine.

The bridge is created at runtime and is dependent on the context of the application. This context information is passed when the client binds a financial service to its underlying computer infrastructure, and only then can the required bridges be instantiated. The issue of which bridges to instantiate may be considered a "configuration issue" and best encapsulated within a separate class. To help structure the system architecture in this area, object design patterns are again employed, through use of the "Abstract Factory" pattern.

The intent of the Abstract Factory pattern is to:

\textit{Provide an interface for creating families of related or dependant objects without specifying their concrete classes [Gamma95].}

To illustrate the design decisions described above, consider the following design fragment which is concerned with the coupling of objects which support the Money interface defined within the OMG Currency Specification. The goal is to provide some means of instantiating a monetary value, whilst hiding the mechanisms employed to achieve it.

Within the author's banking domain machine the task of relating simple monetary classes to the underlying computer infrastructure falls to the MonetaryValueInitialiser bridge. A Java class MonetaryValueInitialiser provides an implementation independent abstraction which couples particular instances of Java classes (which implement the Money interface) with the underlying computer infrastructure. In order to achieve this MonetaryValueInitialiser retains a reference to an object which implements methods defined by the implementation-specific interface MonetaryValueGG. To illustrate this, consider the following abridged code segments.
The interface MonetaryValueGG comprises method declarations that are supported by various classes which map objects, which support the Money interface, onto the actual underlying IT system, as shown below.

```java
package Bridge;
import FbcCurrency.*;

public interface MonetaryValueGG{
    public void getMonetaryValue(SecurityNumber sn, Money monetaryValue);
    public void getPrice(SecurityNumber sn, Money monetaryValue);
}
```

An object of class MonetaryValueInitialiser maintains a reference to an object that implements the MonetaryValueGG interface. Through this reference, methods are invoked to initialise objects that support the Money interface as shown below.

```java
package Bridge;
import FbcCurrency.*;

public class MonetaryValueInitialiser{
    MonetaryValueInitialiser(MonetaryValueGG theImplementation){
        _Impl = theImplementation;
    }
    public Money initialise(SecurityNumber sn, Money monetaryValue){
        if(_Impl != null){
            _Impl.getMonetaryValue(sn,monetaryValue);
        }
        return monetaryValue;
    }
    protected boolean bridged(){
        if(_Impl == null){
            return false;
        }else{
            return true;
        }
    }
    protected MonetaryValueGG theImpl(){
        return _Impl;
    }
    private MonetaryValueGG _Impl = null;
}
```

The abstraction MonetaryValueInitialiser can then be refined through inheritance without regard to any implementation issues. The following class PriceInitialiser defines an abstraction that relates to mapping fixed monetary values (represented by the Price abstraction) to the underlying IT system. However the logic contained within PriceInitialiser is de-coupled from the mechanisms required to actually
instantiate its monetary value. In the following declaration an exception is thrown if an attempt is made to update an instantiated price.

```java
package Bridge;
import FbcCurrency.*;
import Money.*;

public class PriceInitialiser extends MonetaryValueInitialiser{

    PriceInitialiser(MonetaryValueGG theImplementation){
        super(theImplementation);
    }

    public Price initialise(SecurityNumber sn, Price aPrice)
        throws PriceException{
        try{
            aPrice.getValue();
            throw new PriceException();
        }catch(FbcException fbc){
            if(bridged()){
                theImpl().getPrice(sn, aPrice);
            }
        }
        return aPrice;
    }
}
```

Conversely the class hierarchy which implements the MonetaryValueGG interface can be designed without regard to the semantics of classes such as Price. Objects can be instantiated using the methods defined by the Money interface only. This allows the class hierarchy to focus on the particular requirements associated with access to information stored in various underlying computer infrastructure. This will include issues such as interpreting database schema, accessing database tables, screen scraping, interposition, remote access methods, etc.

An example of a class that implements the MonetaryValueGG interface is shown below.
public class CBSBondGG extends CBSGG implements MonetaryValueGG{
    public void getMonetaryValue(SecurityNumber sn, Money monetaryValue){
        try{
            currency = retrieveCurrency(sn);
            value = retrieveCurrentValue(sn);
            monetaryValue.init(currency, value);
        } catch(FbcException fbe){...
    }
    public void getPrice(SecurityNumber sn, Money monetaryValue){
        try{
            currency = retrieveTranCurrency(sn);
            value = retrieveTranPrice(sn);
            monetaryValue.init(currency, value);
        } catch(FbcException fbe){...
    }
}

Once the two separate hierarchies associated with 1) the banking model, and 2) the links to the underlying computer infrastructure are implemented, the issue of which particular class to reference with a given MonetaryValueInitialiser, must be resolved. To aid clarity to the design and to externalise such decisions within a separate abstraction, the Abstract Factory pattern has been adopted. A class which implements the factory interface can construct the association between the two sides of the bridge at runtime, given some contextual information on which to base the decision. In this particular case a security number (a banking domain concept used to identify a particular financial instrument) is used. The factory must therefore support an interface such as the following.

public interface MoneyInitialiserFactoryIF{
    public MonetaryValueInitialiser createMonetaryValueInitialiser(SecurityNumber sn);
    public PriceInitialiser createPriceInitialiser(SecurityNumber sn);
}

How these runtime decisions are resolved can be encoded using different strategies within different class declarations. For example the following class is designed to determine what kind of security is associated with the security number, and create different bridges accordingly.
package Bridge;

public class MoneyInitialiserFactory implements MoneyInitialiserFactoryIF{
    private MoneyInitialiserFactory(){
    }

    public PriceInitialiser createPriceInitialiser(SecurityNumber aSecurityNumber)
    PriceInitialiser pi;
    switch(securityType(aSecurityNumber))
    case BOND :  pi = new PriceInitialiser(new CBSBondGG()); break;
    case FX :  pi = new PriceInitialiser(new DBSGG()); break;
    case CL :  pi = new PriceInitialiser(new CBSCLGG()); break;
    }
    return pi;
}

public static MoneyInitialiserFactory instance(){
    if(_SingletonFactory == null){
        _SingletonFactory = new MoneyInitialiserFactory();
    }
    return _SingletonFactory;
}

private static MoneyInitialiserFactory _SingletonFactory = null;

Having structured the system as described in this example, a model built to use the banking domain constructs, can call a factory method to create and bind a finance abstraction (such as Money) to the relevant implementation in the underlying computer infrastructure using a call such as:

    PriceInitialiser pvi =
    MoneyInitialiserFactory.instance().createPriceInitialiser(sn);

A Price object can then be instantiated by the client without any regard of how it related to the underlying computer infrastructure as follows

    Price val = pvi.initialise(sn,new Price());

Hence the domain machine de-couples the financial process model from the underlying computer infrastructure allowing the design, implementation and maintenance of these models to take place independently.
9.2 Support for Banking Information Transactions

The previous section discussed how the author's banking domain machine provides support for domain models. This section details the means for interacting with such models.

The approach adopted is depicted in Figure 35, again the problem was broken down into three separate concerns:

1. providing the means to express transactions with a financial process model. This is enabled through the use of declarative information transaction constructs;
2. binding these transaction constructs with the execution of process models. This is achieved by providing specific transaction/process binding services, and;
3. mapping these services onto the transaction protocol of the underlying computer infrastructure. This is achieved by providing specific services based on HTTP and XML.
9.2.1 Information Transaction Constructs

To enable interaction with the models which sit above the banking domain machine some form of transactional protocol must be put in place. As with the constructs which support the model, the definition and implementation of such a protocol are separate concerns.

Within the previous chapter HTTP and XML were identified as a suitable means for supporting information transactions, HTTP to be used as a means of providing a device-independent request/response protocol, while XML is to be used as the domain-independent data structuring language. What is missing within the financial domain is an agreement on how to describe financial transactions using such techniques (something analogous to the OMG Currency Specification). It is likely that this situation will be resolved in due course, as more financial institutions use Web-based services to trade. However, this situation lends support to the author’s contention that the definition and the implementation of services must be separate concerns. When standards do arrive, the means to implement them could already be in place.

Given the absence of a standard to describe such a transaction, the author adopted the following simple syntax to describe a financial transaction as a tuple.

```plaintext
Transaction <Name> {
    Request http://<host>/<path>?<parameter1>, ... ,<parameterN>
    Process <package>.<classname>
    Response <Document Type Definition>
}
```

The tuple is made up of: a request, which comprises an Input URL Process; a process, which comprises a Java Command object, and; a response, which comprises a DTD Output Format, (where DTD is the Document Type Definition that describes the structure of an XML document). Using this definition a transaction can be considered as a parameterised URL request which invokes a Java-encoded financial process which in turn generates an XML response whose format is described by the DTD, e.g.

```plaintext
Transaction <Name> {
    Request http://www.bigbank.com/withdraw?amount,account
    Process bigbank.personal.withdraw
    Response http://www.bigbank.com/withdraw.dtd
}
```
9.2.2 HTTP/XML Transaction Support

In order to provide an implementation of the HTTP component of the domain machine the Java Servlet API was used to provide a language binding to the HTTP protocol. A servlet is a Web component, that is managed by a container, typically a Web application server, that generates dynamic content. Like other Java-based components, servlets are platform-independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java-enabled Web server. All servlet containers must support HTTP as a protocol for requests and responses.

The Java Servlet API contains a central abstraction which is named Servlet. All servlets implement this interface either directly or, more commonly, by extending a class that implements the interface. The two classes in the servlet API that implement the Servlet interface are GenericServlet and HttpServlet. For most purposes, HttpServlet is extended during the implementation of servlets.

The HttpServlet abstract subclass adds additional methods beyond the basic Servlet interface which are automatically called by the service method in the HttpServlet class to aid in processing HTTP-based requests. Methods are defined for HTTP GET, POST, PUT, DELETE, HEAD, OPTIONS and TRACE requests. As discussed within the previous section the servlet API was used to bind the HTTP protocols onto the Java Programming Language.

Within the Domain machine the class BankingDomainMachine inherits from HttpServlet and overrides the doGet method as follows:

```java
public class BankingDomainMachine extends HttpServlet {
    
    public void doGet(HttpServletRequest request, 
                      HttpServletResponse response) 
        throws ServletException, 
               java.io.IOException {
    }
```

This method then uses the servlet API to extract all relevant request information from the HTTP request and a standard output stream from the HTTP response. This information is forwarded to the domain machine Transaction/Process Binding service.
Using this approach all knowledge of HTTP protocol is contained within one domain machine component. If the protocol used to define financial transaction input required changing then only this domain machine component would require modification, no other domain machine components would be impacted.

To support the generation of particular XML content the DTD definition named in the transaction declaration is used as a "template holder" for the results produced by the execution of a financial process. In order to provide a binding between XML and the Java programming language the XML for Java (or XML4J) [XML4J] toolkit was adopted. The toolkit was used to generate Document Object Model (DOM) XML trees in memory, based on the DTD defined for a given transaction.

XML4J provides a Java API which allows the construction of validated DOM trees, the intention is to ensure that any XML that is generated will be "well formed" and readily parsed by any recipient. When constructing an internal DOM tree, the content model defined by the DTD is used to ensure the validity of the in-memory representation. This ensures that any new leaf or node that is attached is done so in accordance with the DTD. Once constructed XML4J provides a simple Java method print which converts the internal representation into a XML document.

In order to maintain the separation of output transaction protocols from financial process execution, an object BusinessProcessResult was implemented. Upon construction, the BusinessProcessResult object is passed a DTD which it reads, validates and uses to validate any subsequent additions to its internal structure. The BusinessProcessResult object provides a simplified means of storing "tree" structured data, the intention being to decouple the final representation of the transaction response from the business process that generated it. The interface is shown below.

```java
package BankingProcess;

public class BankingProcessResult{

  BankingProcessResult(String pathToDTD);
  
  public boolean addDataElement(String placeHolder, String value);
  public boolean addNode(String placeHolder, String value);
  public boolean setNode(String placeHolder, String value);

  public String extract();
  public void extract(PrintWriter out);
}
```
Given this simplified interface, tree structured data can be added to a BusinessProcessResult object using a placeholder argument (used to define its position in the tree) and a value argument. For example the following method invocations add three “employees” to a BusinessProcessResult and additionally add the “name” and “email address” of one of the “employees”.

```java
addDataNode(".department.employee.id","J.D");
addDataNode(".department.employee.id","B.S");
addDataNode(".department.employee.id","A.M");
setNode(".department.employee.id","J.D");
addDataElement(".department.employee.id.name","John Doe");
addDataElement(".department.employee.id.email","John.Doe@a.com");
```

The BusinessProcessResult also provides a method `extract` which produces a validated XML document from its current internal data structure.

### 9.2.3 Transaction/Process Binding Services

The primary role of this service is to provide the “link” between receiving a transaction request, executing the appropriate financial process model and then ensuring the correct XML encoding of any resultant information is returned to the transaction initiator. In this sense the service is very much like the activity scheduler within the manufacturing control domain machine described in chapter 6, the main difference being its implementation language - Java rather than Prolog.

During initiation of the domain machine execution the binding service loads and parses the transaction models made available to the domain machine. Rather than use a separate grammar specification and parser technology, the facilities already offered by the Java programming language were used to validate the expressions contained within the transaction models. Once validated, the business process models that are declared with the transaction models are loaded into the domain machine using the resident Java Classloader facility.

In order to execute a financial process the binding service first ensures a BusinessProcessParameter object has been created by the underlying HTTP transaction service and then it creates a new BankingProcessResult object. The service then invokes the execute method defined by the financial process with the command design pattern described in section 9.1.1. Upon completion the binding service then invokes the extract method on the BankingProcessResult object associated with the executed process model.
9.3 An Example System

The application used to test the author’s approach provides support for foreign exchange spot deal trading. The system comprises a basic Forex spot deal application which is integrated with the collaborating bank’s existing IT systems. The application is capable of: taking input from a “price feed” for Forex rates; handling four currencies (Sterling, Dollar, Yen and Euro); allowing the user to input the source currency and amount, and allowing the user to input the target currency. Screen inputs are checked by ensuring that input of source currency, amount and target currency are mandatory fields (a simple error message is generated if they are not entered). The appropriate exchange rate, determined by the source and target currencies, is extracted from the “price feed”. The target amount is calculated and displayed using the exchange rate.

The following section provides an example design fragment of the protocol devised for the author’s banking domain machine and is centred on foreign exchange spot deal trading services.

9.3.1 Supporting Forex Transactions

Given the absence of international standards in the financial transactions arena, the transaction protocol supported by the banking domain machine has been based on the current systems within the collaborating bank.

Essentially the protocol models a two-way flow of information, inputs to the business domain machine take the form of parameterised HTTP GET requests, while outputs take the form of an XML document. XML could have been used for modelling inputs. However, HTTP was chosen, as the input requirements are relatively simple and HTTP allows requests to be easily implemented within HTML pages without the use of any additional tools.
The inputs are constructed using the following HTTP parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Possible Values</th>
<th>Service Provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>AllCurrencyMn, FxCalc, FxCommit, FxSearch</td>
<td>Lists available currency mnemonics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Calculates other amount given deal currency, deal amount and other currency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Checks deal details and posts information to the Central Banking System</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Searches database of all deals that match given criteria</td>
</tr>
<tr>
<td>BuyOrSell</td>
<td>Buy, Sell</td>
<td>Deal currency to be bought</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deal currency to be sold</td>
</tr>
<tr>
<td>DealCurrency</td>
<td>Currency Mnemonic</td>
<td>Indicates deal currency</td>
</tr>
<tr>
<td>DealAmount</td>
<td>Number</td>
<td>Quantity of deal currency</td>
</tr>
<tr>
<td>OtherCurrency</td>
<td>Currency Mnemonic</td>
<td>Indicates other currency</td>
</tr>
<tr>
<td>OtherAmount</td>
<td>Number</td>
<td>Quantity of other currency</td>
</tr>
<tr>
<td>Output</td>
<td>XML, Text</td>
<td>Response encoded using XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Response encoded as a space delimited string</td>
</tr>
</tbody>
</table>

**Table 2: HTTP Query Parameters for Forex Spot Trading**

The output of a transaction is modelled by a DTD (document type definition) which describes the format of the XML output generated by an input request, as shown below.

```xml
<!DOCTYPE message [
  <!ELEMENT message (currency* | fxContract* | errorMessage)>]
<!ATTLIST message context (AllCurrencyMn | FxCalculated | FxCommitted | FxSearched) #REQUIRED>
<!ATTLIST message status (OK | ERROR) #REQUIRED>
<!ELEMENT errorMessage (#PCDATA)>  
<!ELEMENT fxContract (buyOrSell, dealMoney, otherMoney, exchangeRate, spotDate)>  
<!ELEMENT buyOrSell (#PCDATA)>  
<!ELEMENT dealMoney (currency, amount)>  
<!ELEMENT otherMoney (currency, amount)>  
<!ELEMENT currency (#PCDATA)>  
<!ELEMENT amount (#PCDATA)>  
<!ELEMENT exchangeRate (#PCDATA)>  
<!ELEMENT spotDate (#PCDATA)> ]>
```

These two transactional components model the inputs and outputs to the banking information model held within the banking domain machine, in a way which is independent of how they are implemented. For example, the following input is a request...
to the banking domain machine to perform a Forex spot calculation. The calculation involves converting a set amount of Euros into Japanese Yen.

\[
\text{http://domainmachine/Forex?context=Fxcalc\&buyorsell=sell\&dealcurrency=EUR\&dealamount=100,000\&othercurrency=JPY}
\]

This input request may be generated by any application capable of issuing HTTP GET requests or may be generated via a browser using the following HTML:

```
<html>
<head>
<title>Simple Forex</title>
</head>
<body>
<h1>Simple Forex Machine</h1>
<form method="get" action="http://domainmachine/Forex">
    <tt>Deal Amount : </tt>
    <input type="text" name="dealamount">
    <select name="dealcurrency">
        <option selected>GBP</option>
        <option>USD</option>
        <option>EUR</option>
        <option>JPY</option>
    </select>
    <br>
    <tt>Other Amount : </tt>
    <input type="text" name="otheramount" value="">
    <select name="othercurrency">
        <option selected>GBP</option>
        <option>USD</option>
        <option>EUR</option>
        <option>JPY</option>
    </select>
    <br>
    <input type="radio" name="buyorsell" value="buy" checked>Buy
    <input type="radio" name="buyorsell" value="sell">Sell
    <input type="submit" value="Calculate">
    <input type="submit" value="Commit">
    <input type="hidden" name="context" value="FxCalc" value">
</form>
</body>
</html>
```

The banking domain machine will respond by generating and returning the following XML document. This document may be parsed by an application or simply displayed by a browser.

```
<?xml version="1.0"?>
<!DOCTYPE ForexResponse SYSTEM "http://domainmachine/Forex.dtd">
<message context="FxCalculated" status="OK">
    <fxContract>
        <buyOrSell>sell</buyOrSell>
        <dealMoney>
            <currency>EUR</currency>
            <amount>1,000.00</amount>
        </dealMoney>
        <otherMoney>
            <currency>JPY</currency>
            <amount>127,700.00</amount>
        </otherMoney>
        <exchangeRate>127.7000</exchangeRate>
        <spotDate>28-2-2000</spotDate>
    </fxContract>
</message>
```

Some of the transactions provided are simply requests for information, others may modify the state of the banking domain machine or even cause the machine to inter-
act with other banking systems, i.e. in order to execute a Forex deal the banking machine could send a deal request to an external trading system.

9.3.2 Forex Process Model

The following model describes the process required for one part of the transaction protocol described in the previous section when Context = FxCalc.

```java
class ForexContract extends BankingProcessModel {
    void execute(BankingProcessResult result, String buyOrSell, String dealCurrency, String dealValue, String otherCurrency) {
        CurrencyBook cbRef = CurrencyBookHelper.narrow(getObjectRef("Currency Book"));
        if(!cbRef.containsCurrency(dealCurrency)) {
            throw new BPMException("Currency " + dealCurrency + " not held in Currency Book");
        }
        if(!cbRef.containsCurrency(otherCurrency)) {
            throw new BPMException("Currency " + otherCurrency + " not held in Currency Book");
        }
        MoneyFormatter mfRef = MoneyFormatterHelper.narrow(getObjectRef("Money Formatter");
        mfRef.setFormattingString(aCorbaId(), "#,##0.00#");
        Money dealMoney = mfRef.parseForCurrency(aCorbaId(), dealValue, dealCurrency);
        ExchangeRateManager rfRef = ExchangeRateManagerHelper.narrow(getObjectRef("FX Rate Feed"));
        ExchangeRate theRate = rfRef.getExchangeRateForRateType("Spot", dealCurrency, otherCurrency);
        Money otherMoney = theRate.exchange(dealMoney);
        IntlTradingIF itRef = IntlTradingIFHelper.narrow(getObjectRef("IntlTrading"));
        Calendar now = Calendar.getInstance();
        DTime spotDate = itRef.getSpotDate(Calendar.getInstance(), dealCurrency, otherCurrency, 2);
        result.addDataElement(".buyOrSell", buyOrSell);
        result.addDataElement(".dealMoney.currency", dealMoney.getCurrencyMnemonic());
        result.addDataElement(".otherMoney.currency", otherMoney.getCurrencyMnemonic());
        result.addDataElement(".dealMoney.amount", dealMoney.getValue());
        result.addDataElement(".otherMoney.amount", otherMoney.getValue());
        result.addDataElement(".exchangeRate", theRate);
        result.addDataElement(".spotDate", spotDate);
    }
}
```

The process essentially performs the following steps during execution:

- obtain a reference to CurrencyBook;
• check if $\text{dealCurrency}$ is valid;
• check if $\text{otherCurrency}$ is valid;
• obtain a reference to $\text{MoneyFormatter}$;
• attempt to convert $\text{dealCurrency}$ and $\text{dealValue}$ into $\text{Money}$;
• obtain a reference to $\text{ExchangeRateManager}$;
• attempt to obtain $\text{ExchangeRate}$;
• convert the $\text{Money}$ into $\text{Money}$ of $\text{otherCurrency}$;
• obtain reference to $\text{IntlTrading}$;
• obtain the relevant $\text{spotDate}$, and;
• populate $\text{BankingProcessResult}$.

Much of the logic within the method is concerned with obtaining remote object references to components of the OMG specification. Once a reference has successfully been obtained the desired remote method request can be invoked. What is not encoded within this method is any knowledge of how the model is implemented, thus the separation of model from model interaction is achieved.

### 9.4 Concluding Remarks

In this chapter, the author’s framework for domain machines, based on the concept of using an underlying substrate to support technologies that are capable of linking to both a domain model and an underlying general purpose computing infrastructure, has been re-tested. The application domain here is very different from the original manufacturing domain, and the technologies used in the new domain machine are very different. The WWW-based technologies used in this experiment have been positioned within the author’s framework, forming a system with the same “ease of change” properties of as those of the previous manufacturing experiment.

As such, the framework could be used as a template, where different domains may demand different technologies, but the domain machine concept and author’s proposed framework remain the same. It is likely that many different domains will fall into general classifications that may be catered for by a set of standard domain machines, i.e. manufacturing control systems, information systems, real-time control systems.
Chapter 10 - Conclusions

The research described in this thesis has identified and addressed a key problem with the creation and modification of applied, industrial-strength software systems - namely the disparity between a system's design and implementation. This disparity is currently negotiated using an intellectually intensive, costly "one way" programming task, which fails to preserve a clear expression of design concepts within the implemented system.

10.1 Summary of the Research Approach

The approach adopted to address the problem has been to raise the "level" of system infrastructure in such a way as to retain fundamental system design concepts within the system's implementation. Such an infrastructure provides separate specialised support for the different characteristics of a software system design and in doing so eliminates the requirement to transform a system design into the computational abstractions offered by high-level programming languages.

In order to achieve this "raised level" of infrastructure the following three research issues were identified:

1. the infrastructure must support the decompositions and abstractions adopted within system design notations;
2. some formalism is required to transform design declarations into the realisation of what they describe, and;
3. the means to execute such design constructs using contemporary computational abstractions and currently available computer infrastructure must exist.

To address these research issues the author proposed "A Framework for the Realisation of Domain Machines". The framework adopted an "embedded" approach to domain machine creation whereby domain concepts were introduced into a "host" computing environment. The host environment essentially acts as a substrate which provides domain-independent execution facilities plus the capability to add domain-specific facilities. A specific domain machine is realised through the selection of an appropriate software technology substrate and the use of further relevant software technologies to populate the framework. Such a machine addresses the research issues above by:
Conclusions

- utilising domain-specific language extensions to support the system design decompositions and abstractions of the domain;
- coupling the host environment to the computational abstractions provided by the underlying computer infrastructure in order to provide the means of executing a system design, and;
- harnessing and extending the formalisms provided by the host infrastructure to transform various design abstractions into the underlying computational abstractions provided by the underlying computer infrastructure.

The feasibility of the proposed framework was examined through the construction of two distinct domain machines, the first within a computer-controlled manufacturing facility and the second within a commercial IT environment. The research had two objectives:

1. to investigate whether the use of the framework to raise the level of infrastructure to a point where design models can be directly executed is an improvement over traditional approaches to system construction and evolution, and;
2. to prove that the framework is not specific to a single application domain.

10.2 Conclusions

The first experiment detailed within this thesis examined the application of the domain machine framework to a manufacturing control system. The services offered by the domain machine were aligned to fundamental concepts of the domain, namely action, interaction and behaviour. Separate formalisms were employed to map these concepts onto the services provided by existing computer system technology. Chapters 4, 5 and 6 described the construction of the domain machine and chapter 7 detailed a comparison of two manufacturing control systems, one constructed using a contemporary software design approach and the other built using the domain machine.

This experiment showed both qualitatively and quantitatively the relative ease by which the domain machine implementation of the manufacturing control system could be constructed and modified and therefore its superiority over the traditional approach.

The second experiment detailed within the thesis showed how the domain machine framework was applied to a very different application domain, that of banking infor-
Conclusions

mation systems. Here the fundamental concepts were concerned with financial transactions and the processes required to support them. Here information was the primary concern of the domain rather than control. Chapters 8 and 9 described how the same framework that was used for the manufacturing control system was populated with different technologies in order to support the construction of a foreign exchange trading support system.

This experiment showed that the framework was not specific to one application domain. Whilst falling short of proving the framework is truly generic in nature, the disparity of the two domains examined in this thesis goes some way to showing how generally applicable the domain machine framework is to the construction of software systems.

The major contributions of this thesis are:

• evidence that it is possible to improve upon conventional programming techniques for the implementation and evolution of software systems;
• the introduction of a framework for the creation of domain machines, and;
• the specification of two sets of technologies to populate the framework, one suitable for manufacturing control systems, and a second suitable for financial IT systems.

10.3 Final Commentary

Domain machines are used within a number of computer applications today, for example a Web browser is a content presentation domain machine which executes declarative representations of pages using HTML. Within operating systems themselves domain machines are used to scan convert font descriptions for portrayal on raster display devices, such machines execute "hinting" programs that describe how best to quantise the shape of a given character. The parser generators used by the author to build the domain machines described within this thesis are themselves domain machines, which are programmed using declarative descriptions of particular grammars.

However, one common theme pervades such domain machines - their domain is bounded, well understood, formally described and usually a computer application itself. One has to look a lot harder to find examples of domain machines applied to human-based activities within industry or commerce, where the domain in question
is difficult to formalise within a single language. This is seen as one of the major drawbacks of the domain machine approach, if you can’t formally describe the domain, how can you build a machine to execute domain process descriptions. The search is often for a mathematically correct and complete domain language and if this can not be found then the domain machine can not be built.

What is required is a more pragmatic approach to domain machine creation, this requires a more engineering-like approach rather than a scientific approach. Here, partial and “best fit” solutions are adopted and used to create a general approach and an accompanying set of tools for the application of domain machines within the real world.

10.4 Issues for Further Investigation

Whatever the merits of the approach to software system creation and modification described within this thesis, the language we use to define the implemented system must change from a nesting of expressions that merely abstract the architecture of computer hardware, to one that naturally supports concepts of the application domain.

At the interface between an executable design model and a domain machine, the lowest level of the model comprises a set of fundamental domain-specific elements. Future work must determine what level of granularity and what degree of “richness” is required within the set of domain-specific fundamental elements, and what relationships are required between these elements such that they are capable of describing the processes in the domain. When complete, the set of elemental domain abstractions will have effectively created a language with which to express domain process models. Research to define and create these domain specific languages is required to enable the full exploitation of the domain machine concept.

The attempt to ease the process of software creation and modification described in this thesis has focused entirely on the implementation phase of the system life cycle. The rationale for this approach was derived from the disproportionate amount of human effort expended in a process that could be and should be assigned to the computer infrastructure. Raising the level of the computer infrastructure, as
described in this thesis, would allow software engineers to focus on the formulation of viable system designs rather than on their means of realisation.
References


Apache http://httpd.apache.org/


<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Title</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
EPSRC97  EPSRC, IT & Computer Science, SEBPC “Handling Legacy Financial IT: A Road to True System Evolution”. Grant Ref No: GR/L48300.


Gamma95  E. Gamma et al., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, Reading Mass. USA, 1995.


Grellier97  Grellier T., Quote from T. Grellier M.D. of MMT Computing UK, supplier of IT and IT staff to the Finance Industry.


Hudak97  Hudak P., "Keynote Address: The Promise of Domain-Specific Languages", IC on Domain Specific Languages, Santa Barbara 1997.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Author(s)</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>McLean97</td>
<td>McLean M.</td>
</tr>
</tbody>
</table>


Tomcat  http://jakarta.apache.org/tomcat/


<table>
<thead>
<tr>
<th>Ref</th>
<th>Author(s)</th>
<th>Title/Description</th>
<th>Source/Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>