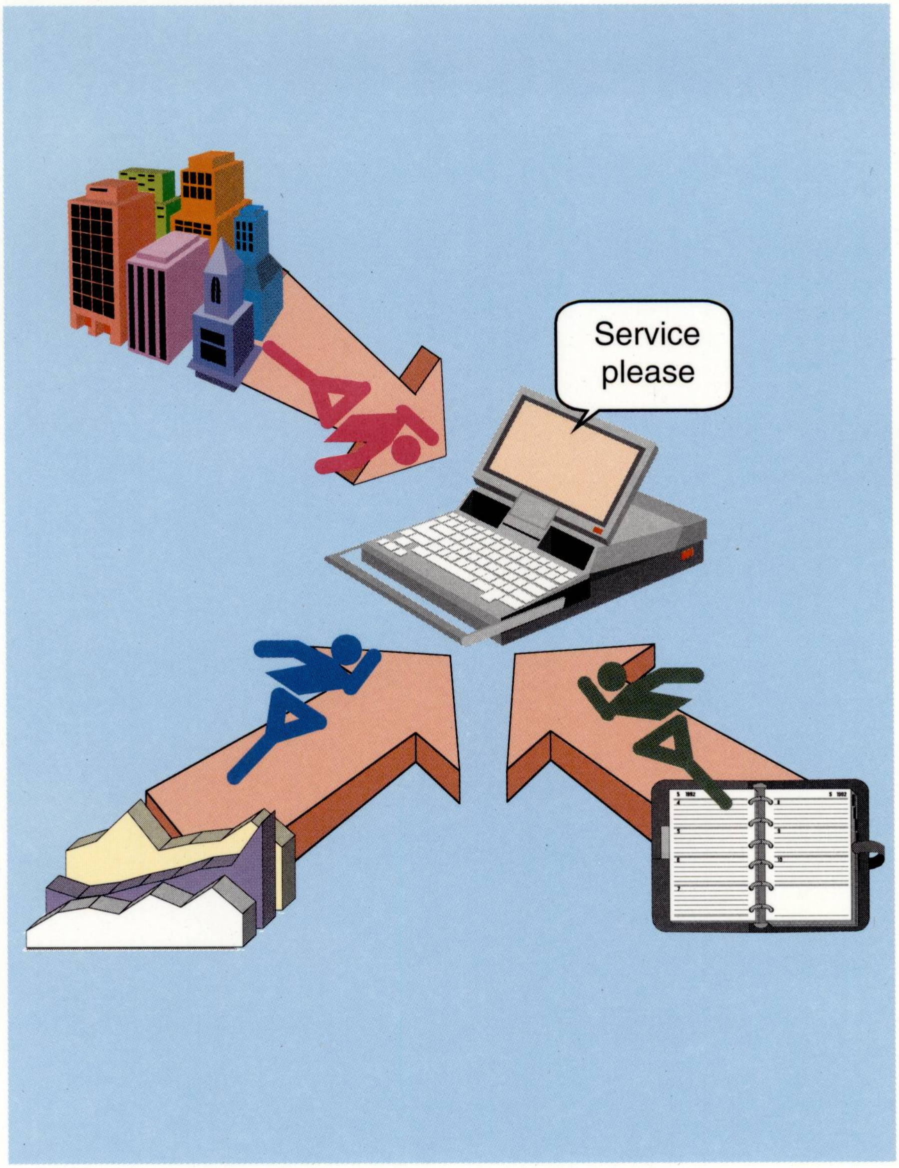
Volume 12 Issue 1 *May 1997 Edition*



ICL

The **Systems** *Journal*

ICL Systems Journal

All correspondence and papers to be considered for publication should be addressed to the Editor.

The views expressed in the papers are those of the authors and do not necessarily represent ICL policy.

Published twice a year by Research and Advanced Technology, ICL, Bracknell.

1997 subscription rates (including postage & packing):

	UK and Europe	Rest of World
Annual subscription	£72	$120
Single issues	£43	$72

ISSN 1364-310X

ICL Systems Journal

Volume 12 Issue 1

Contents

Front cover: Mobile Agents in action! See the paper, "Mobile Agents—The new paradigm in computing," in this issue.

Editorial

The ICL Systems Journal normally avoids having issues devoted to a single theme, except in exceptional circumstances. However, the practice of having a small number of papers on a related theme has been followed frequently, particularly when new fields of importance are emerging.

In this issue, papers on three important areas are being published, namely, Java, Mobile Agents and Constraint Programming. The first two of these are, of course, related within the context of distributed computing and the Editor intends that papers on significant applications of these new technologies will follow in future issues. Constraint Programming, however, is not new and applications of it have been described in earlier issues. What makes a further set of papers timely is the development of a powerful new software platform, ECL^{i}PS^e, at IC-Parc, a research institute at Imperial College in London, of which ICL is one of the sponsors. This platform enables the technology of Constraint Programming to be applied to a wide variety of difficult problems with much greater ease than hitherto.

The Editor is aware that a number of people in the field of computing perceive the intellectual hurdle of Constraint Programming to be difficult to surmount. Consequently, two introductory papers have been invited from Owen Evans and Mark Wallace to provide readers with "a lift over this hurdle". These papers are followed by a comprehensive description of ECL^{i}PS^e, also written by Mark Wallace, together with some of his colleagues at IC-Parc. A paper on its application will follow in the next issue.

V.A.J. Maller

JAVA™ – an overview

Nic Holt

High Performance Systems, ICL, Manchester, UK

Abstract

The origins of Java can be traced back to the early 1990s when Sun engineers were investigating the requirements for cheap consumer devices capable of communicating with each other. In 1994, Sun recognised the potential of coupling Java technology with the World-Wide Web, paving the way for a new era of "Network Computing". Since then Java has become a cornerstone of Internet distributed computing and all major IT industry suppliers have embraced Java technology in some form or other. This paper identifies the significant elements of Java technology, outlines the positioning of major IT suppliers and discusses its potential influence on new business models based on distributed computing.

1. Introduction

Java has become inextricably associated with the rapid evolution of Internet computing sparked by the introduction of the World-Wide Web and widely available Internet services. This vision is one of universally accessible online information services supported by a global network infrastructure. Although the consumer market is likely to be more significant in the longer term, the immediate focus is on the potential business benefits of applying these technologies within organisations—the so-called Intranet. In the longer term, the distinction between Intranets and the Internet will become blurred. The emergence of commercial-strength secure distribution infrastructure overlaying the Internet will enable new models of inter-organisational collaboration and commerce, as well as intra-organisational operation.

The question of the moment is whether the convergence of several technologies heralds a revolution in distributed computing, or whether the benefits of Java, relative to steady evolution of current approaches, are being over-hyped. There are major commercial issues at stake. In particular, the prevailing "WinTel" (Microsoft Windows/Intel) dominance of the (enterprise) desktop is under threat—a response to the massive perceived management costs of PCs, widely quoted as around $4,000 per PC per annum [Forrester, 1995].

2. Background

The origins of Java can be traced back to the early 1990s when Sun engineers started to investigate the requirements for cheap consumer devices with embedded processors, capable of communicating with each other. The short product life-cycle of the consumer market dictated that it should be possible to introduce upgraded products—including, significantly, the processors embedded within them—whilst maintaining previous investment in software and applications. Over the next two years a new language, "Oak" (later to be renamed Java), as well as an operating system (a precursor of JavaOS) and custom microprocessors (the PicoJava architecture) were developed. In 1994, Sun recognised the potential of coupling the nascent Java technology with the burgeoning World-Wide Web, heralding the dawn of a new era of "Network Computing". Since then almost all major IT industry suppliers have embraced Java technology in some form or other.

3. What is Java?

Java is a modern object-oriented programming language. The term "object-oriented" means that a program is written in terms of real-world objects and the actions that can be performed on them. This is rapidly becoming the preferred industry approach and is embodied in languages such as C^{++} and Visual Basic, as well as the distributed object architectures of Microsoft ActiveX and OMG CORBA. The emergence of self-contained objects offers the prospect of software "building-blocks" or components which can be simply integrated into sophisticated applications. Microsoft ActiveX, OpenDoc and Java Beans are examples of component software technologies.

As a programming language, Java has several features which offer improvements over alternative languages. These include "safety features" which ensure more reliable programs, extensive libraries of useful services, a component-based approach to application construction and a natural fit with networked computing. Nonetheless, even with the benefits of Java it still requires skill and experience to write good software.

Java's main distinguishing feature, however, is not in the language itself but in the means by which a Java program is executed on a computer. Normally, a program is issued in a (binary) form which is specific to the processor on which it is intended to run and, in some cases, specific to the combination of processor and operating system. Application vendors must make a separate investment in each target platform for each software product. This results in a huge barrier to entry into volume markets such as the desktop for alternatives to the Windows/Intel architecture.

In contrast, Java is compiled to a machine-independent form known as Java byte-code. This can be thought of as the instruction set for an imaginary machine. A Java program is executed by a piece of software known as the Java VM (Virtual Machine), which interprets the byte-code form of the program. To achieve program portability, it is only necessary to ensure that the Java VM itself is available on each computer system—a once-off porting activity which can be undertaken independently.

None of this approach is new. In the late 1970s, the University of California, San Diego, developed the UCSD Pascal system which made high-level programming feasible on personal computers. In the UCSD system, Pascal was compiled to a portable, interpreted form known as p-code. The limitations of Pascal, in particular its limited I/O features, eventually led to its demise at the hands of the C language.

With aims similar to Java, the Royal Signals and Radar Establishment at Malvern (now DRA, Malvern) developed the Architecturally Neutral Distribution Format (ANDF) during the early 1990s, as a means of distributing software written in any source language in a machine independent form. Although technically sound, the concerns of software vendors that it would enable reverse engineering of programs distributed in ANDF prevented its widespread adoption.

What distinguishes Java from its predecessors is the completeness of the "vision" promoted by SUN, including the language itself, a rich set of standard libraries, development tools, execution environments, inherent distribution & networking capabilities and, perhaps most importantly, a credible case.

4. Other Elements of Java

Java Beans is the framework for developing platform-independent Java software components which can be independently developed, delivered and deployed—the equivalent of Microsoft OLE or ActiveX Controls. Javasoft have published a full standard for Java Beans and support is provided in version 1.1 of the Java Development Kit. Several of the leading industry suppliers have adopted Java Beans as their standard technical framework for networked applications.

Java Remote Method Invocation (RMI) is a mechanism which allows one Java application component transparently to invoke a method in another component which is executing on a remote platform. RMI provides a richer set of facilities than, for example, CORBA, but is inherently not as scalable. Javasoft's positioning of RMI is for use within bounded groups of platforms (or between components in different processes on the same platform), using CORBA IIOP for wide-scale distribution.

JavaOS™, is a highly compact operating system designed to run Java applications directly on microprocessors in anything from network computers to pagers.

HotJava Views is a groupware products incorporated into Javasoft's Web Browser. The development model of Views is similar to Apple when the Macintosh was released: provide useful but limited capability and leave the door open for ISVs to add value.

5. Java Performance

The penalty for adopting an interpretive approach to program execution is that it is inefficient—around 20 times slower than compiling the program directly to the instruction set of the real machine. There are several ways in which the inefficiencies of interpretation can be mitigated and the performance penalty of 20 reduced to a range of 2–5. The technique currently in vogue is known as JIT (Just In Time) compilation, which involves compiling individual functions to Java byte-code as they are first used.

Recent development of dynamic translation (load-time or runtime compilation) techniques has resulted in Java execution rates which, perhaps surprisingly, are faster than those for statically compiled C^{++} on the same hardware; this is because, at runtime, it is possible to determine actual data access patterns and conditional branch decisions and, hence, generate code optimised for the circumstances in which it is being executed. It is likely that dynamic translation will have a profound impact in a wider arena: it significantly alters the design assumptions on which many modern processor designs are based—in particular, the effectiveness of brute force, global, static compiler optimisations.

If portability is not a key requirement (in particular, for servers), the original Java program itself can be compiled directly to machine code, in which case there is no penalty.

An alternative approach, also being adopted, is to produce silicon which actually has the JVM as its native instruction set. Sun Microelectronics have designed such a chip, called PicoJava, which will shortly achieve first silicon. Early simulation results indicate that this will cause a 10 fold increase in performance over an interpreter given similar processor clock-speeds. The implications of a new architecture for the industry should not be underestimated, given the mindshare that Java enjoys.

6. Distributing Java Programs

The portability of Java programs in byte-code form has a dramatic benefit: it is possible to download software across the network (Internet or Intranet), as it is required, from a remote, centrally administered server, thus reducing cost of ownership. Furthermore, exactly the same copy of the program can be downloaded to any type of client. The server requires no knowledge of the type of the client system and, apart from the generic Java VM itself (and a Web browser or its equivalent) no special software is required on the client system to enable it to execute the downloaded program. This allows a diversity of platforms to be supported by a common server (see Figure 1).

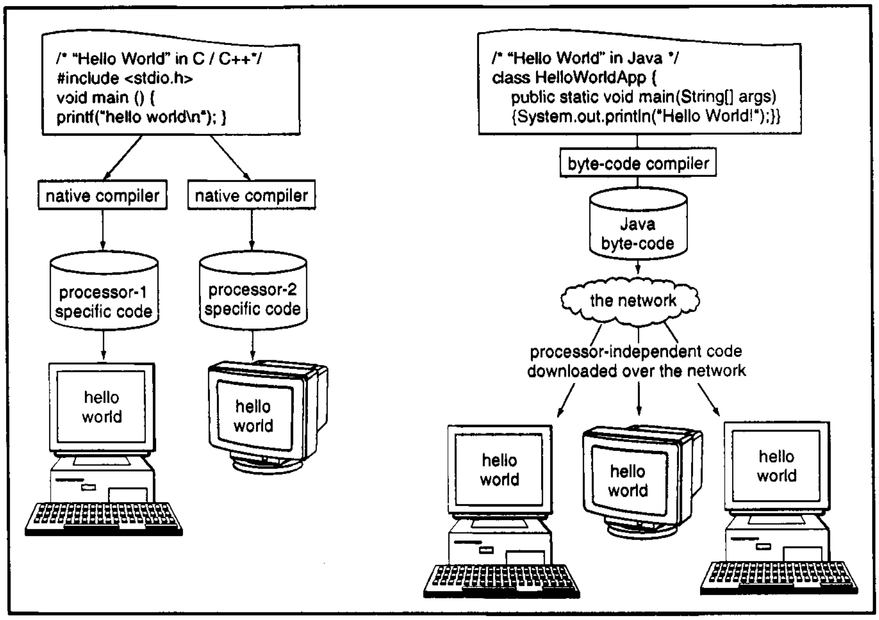

/* "Hello World" in C / C++*/
#include <stdio.h>
void main () {
printf("hello world\n"); }
native compiler
native compiler
processor-1 specific code
processor-2 specific code
hello world
hello world
/* "Hello World" in Java */
class HelloWorldApp {
public static void main(String[] args)
{System.out.println("Hello World!");}}
byte-code compiler
Java byte-code
the network
processor-independent code downloaded over the network
hello world
hello world
hello world

Figure 1

7. Application of Java to Networked Information Services

7.1 Desktop Systems

The major short-term opportunity for Java is on the desktop within the Enterprise. The approach is to download the client elements of Enterprise applications and groupware, on demand, from centrally administered servers.

- Support costs are drastically reduced and the same version of the application is used by everyone.
- Workstation specifications can be relaxed, allowing the use of "thin clients" which don't have the burden of hundreds of megabytes of operating system software.
- The global network provides secure access not only to resources within the Enterprise ("the Intranet") but also to external services such as those required to operate business relationships with suppliers, customers and collaborators.

This approach is viable for most desktop users, other than "power users", and studies undertaken by Sun suggest that it is appropriate for up to 90% of Enterprise employees.

7.2 Consumer Devices

A second major opportunity for Java is its use in consumer devices such as set-top boxes, portable devices, such as PDAs, and public devices such as kiosks. A Java-based approach enables such devices to provide universal access to information and services anywhere on the global network. This will be one of the driving forces speeding the introduction of a global, commercial-strength security model.

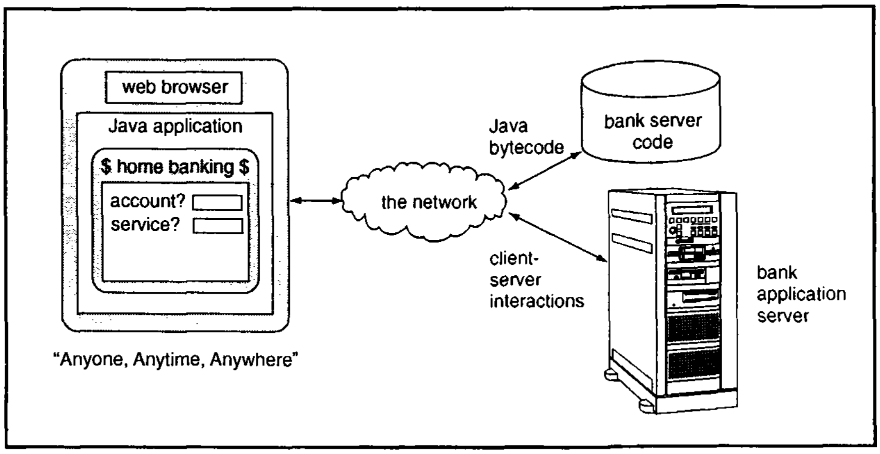


Figure 2

7.3 Server Systems

A longer term opportunity for Java is in server systems (see Figure 2). The issue of portability is less critical for such systems—there may only be a handful of credible server platforms by the end of the decade (in marked contrast to the probable increase in diversity of embedded processors for client systems) and porting server applications to these few is probably commercially viable. The positive features of Java as a programming language may enable it to supplant C^{++} as the programming language of choice for the software engineering of large-scale server applications. In such situations, Java programs would be compiled directly to the machine code of the server processor, ensuring no loss of efficiency. An additional benefit of Java (as indeed of other Object Oriented approaches) is that by enabling the "encapsulation" of legacy services and information as componentised objects, it greatly simplifies the integration of such systems into an evolving IT service provision.

There are now several Application Development Environments available for large-scale software development, many of them re-engineered from C^{++} tools. Java class libraries are being specified and implemented, offering standardised interfaces to a wide range of services. Together with either native (platform specific) compilers and/or dynamic compilation techniques, these constitute a comprehensive set of tools for developing efficient, maintainable large-scale applications.

It is interesting to note that some industry analysts (e.g. David Coursey) are suggesting that, with increasing commoditisation of server platforms, developing a comprehensive Java Server Environment is one of the few opportunities for Sun to create a defensible business.

8. How is the Industry Responding?

Most of the industry is adopting a highly positive response towards Java and the distributed computing model with which it is associated.

- Sun have established JavaSoft to develop and sell Java technology, arguably to support the repositioning of their predominately hardware business towards the server market. They are developing a new range of hardware processors which execute Java byte-codes directly and this may be particularly useful in low-cost terminal devices.
- There is a risk that Sun could, by ownership of the Java standards, become a latter-day Microsoft, having proprietary ownership of pervasive de facto standards. This risk is balanced by the establishment of an industry review process, which will be open to Java licensees. There has also been a recent move by Sun to submit Java to ISO for (independent) standardisation—a move fiercely contested by the rest of the industry, since Sun wishes to retain control of Java standards and thus appears to be trying to use ISO to rubber-stamp its own interests.
- Novell has established close links with JavaSoft as the basis for a repositioning around a product line known as IntraNetware, based entirely on the technologies described above. Novell aims to become *the* one-stop shop for Intranet technology, building on their Netware channel.
- Oracle and Informix have both announced frameworks based on networked computing and Java (Network Computer Architecture and Universal Web Architecture respectively). Oracle are also enhancing their applications to allow, for example, an Oracle Manufacturing system to handle stock-level queries and ordering from any Java-enabled client.
- Although not a vendor, the Object Management Group (OMG) is an industry standards body which has developed specifications for distributed object-oriented computing, under the brand of CORBA. The CORBA interoperability protocol (IIOP) is the chosen basis for all the above distributed computing initiatives and products. The CORBA security specifications (CORBASEC and SECIIOP) are likely to become the basis for secure distributed Java-based computing. The availability

of client support for CORBA IIOP in the next release of the JDK from JavaSoft, will place CORBA-based security products in an important position in the growing distributed computing market.

- Netscape has a strong link with JavaSoft and has been buying into Internet technology companies (e.g. Visigenic, a producer of CORBA distributed ORB technology). Netscape has just announced Communicator, a Java-enabled groupware suite with in-built CORBA IIOP support.
- Corel, who now owns the WordPerfect Office products, have re-engineered them in Java to produce the Java Office Application suite; initial product versions of these are now available.
- IBM has established a new Network Computing Products Division to exploit the opportunities provided by Java and related technologies and has announced a Network Computer product. Lotus has produced Java-enabled versions of Notes (with considerable help from JavaSoft).
- Sun, Netscape, IBM and Novell are jointly sponsoring a high profile "Java Education World Tour".
- Even Microsoft has embraced Java (reluctantly at first) and indeed its strategic software development tools support Java (the J^{++} tools) equally alongside Visual C^{++} and Visual Basic. Microsoft is making moves which suggest that it wants to wrest control of the Java language from Sun, including the release of a Windows-specific native compiler for Java. There are rumours that the company will produce a compiler from Visual Basic to Java byte-code, which, like Microsoft C and C^{++}, could be partly interpreted and downloadable over the network into Windows client systems.

  Microsoft would like to hide Java components inside ActiveX wrappers; JavaSoft would like to do the opposite. The battleground is over control of Enterprise (and open) distribution architecture and protocols. Microsoft, intent on expanding from the desktop to the enterprise server, is pushing Distributed COM and "the rest" are using CORBA IIOP as distributed networking protocols. A key deciding issue here could be the credibility of the distributed security features of each offering.
- Sun has licensed ICA—a Citrix Systems Inc. protocol enabling ultrathin clients to execute Windows applications on the server—for incorporation into the Java Virtual Machine. ICA-based clients with WinFrame servers offer a possible alternative strategy to Java in some circumstances.
- Many software development tool vendors are hastily converting their existing C^{++} and Visual toolsets to become Java toolsets. Leading toolsets

include Symantec Visual Cafe Pro.

- Java is seen as an opportunity by vendors of well-established software products to re-vitalise them in the guise of "Internet-enabled" products.

9. How Will Java Change the Future?

Java and its associated technologies will speed the globalization of information systems. The ability to download software, as it is required, will break down barriers within and between organisations, allowing rapid access to information or services anywhere on the global information network. Early adoption will be in applications where security is non-critical. However, the extent to which electronic commerce is able to benefit will depend upon the establishment of a global infrastructure of trusted security services.

Proliferation of "Thin Client" systems, NCs and PDAs, all running downloaded Java software, will truly enable access by "anyone, anytime, anywhere". Simplified client systems and software will greatly reduce the total cost of ownership of desktop systems (estimated reduction by a factor of 4). The move away from "fat clients" will substantially loosen the grip of Microsoft on the desktop. Potential beneficiaries include not only the (mainly software) vendors identified in the previous section, but also suppliers of simplified hardware platform technologies, such as Acorn (ARM processors).

Network- (or server-) centric distributed computing will lead to a new generation of client-server architectures in which the revitalised role of the server as the trusted guardian of shared resources including information, services and software will be crucial.

Java-based Intranet systems will enable organisations to exploit their information assets more effectively to achieve integration and co-ordination within the organisation.

It will be easier to integrate business processes between different organisations, as well as between individuals and organisations. This is a consequence of the elimination of the need to pre-install or configure software before doing so: it will be automatically downloaded as required.

Since software will generally not be permanently installed on a client system, new software charging models will be utilised such as pay-by-use, or server-session-based, etc. This may lead to new distribution and channel models for client software.

Java portability ("write once, run anywhere") will reduce software packaging and distribution costs. Its rich set of libraries, component-based software integration model ("Java Beans") and safety features will increase development productivity.

Portable software reduces customer "lock-in" and will affect the structure of competition in the industry.

A component-based software architecture provides opportunities for niche software vendors to develop specialist components which can be readily integrated into other applications and end-user solutions.

The ability to manage client software from a central server opens new opportunities for service providers to offer fully serviced IT facilities to small businesses—the IT equivalent of fully serviced office suites.

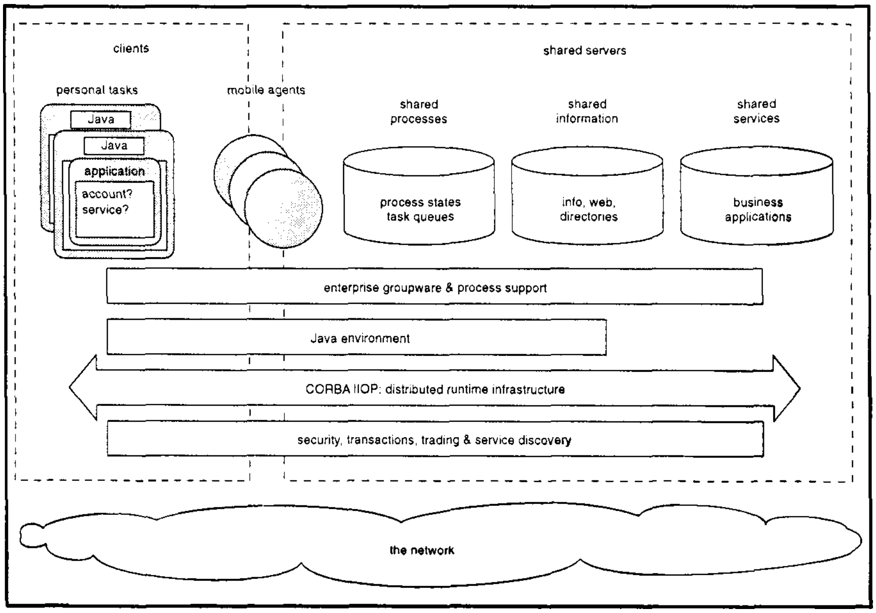


Figure 3 A Vision of the "Networked Organisation", showing the positioning of Java, and the potential exploitation of related technologies & services.

10. Java Opportunities

10.1 Networked Business Solutions

Java will undoubtedly bring about fundamental changes in the nature and profile of IT provision within the arena of medium to large organisations. This will affect not just product businesses, but also the way in which solutions and services are integrated, deployed and maintained.

Several software vendors are developing downloadable Java versions of Groupware client applications; this immediately enables anyone with a Java-enabled client to use Groupware services. The general approach is to evolve Groupware products into Java components ("Beans"), which can be flexibly integrated into business solutions, either by software vendors or by third party solutions builders.

Such products are initially targeted at the Intranet market, eventually providing a comprehensive framework for the operation of an organisation's core business processes—both internally and in its interactions with other organisations (and individuals) over the Internet. Longer term evolution of groupware products will enable the collaboration of geographically dispersed participants into virtual groupings for social, educational or community purposes.

10.2 Java-based Services

Java-based computing offers the opportunity to provide innovative services over the Internet to anyone who has a Java-enabled Web Browser. ISPs, for example, could host Java-based services for external services customers (e.g. the First Direct Home Banking service). Equally, they could offer "fully serviced virtual offices" to small businesses, based on downloadable Java applications, removing software administration burdens from unskilled users. Networked services for small businesses could include simple accounting, stock management, virtual purchasing groups (combining the buying power of numerous small retailers to achieve wholesale price reductions) and product and service directories.

10.2 Consumer Devices

Perhaps the most exciting opportunity for Java is that originally envisaged by SUN at its genesis—the routine embedding of programmable devices into consumer products. Products ranging from electronic and white goods to mobile phones and set-top boxes already incorporate microprocessors. Indeed, it is widely reckoned that over half the lines of code written each year are for such purposes; but, typically, the software is non-portable and hard-wired into the devices.

During the 1980s, advances in technology made it possible for general purpose computers to abandon the machine hall for workers' desks and, more recently, people's homes. Trends in technology and costs continue inexorably and the next market era will see general purpose computing devices embedded in consumer products, connectable to "the network". Just as the size of the desktop market long ago outstripped that for central DP systems, it is likely that the consumer market will dwarf both of these (see Figure 4).

An early example of such devices is the Java-enabled set-top box, providing interactive access to a wide-range of information-based services, as well as media broadcasts. Another example is the Personal Digital Assistant (PDA) which combines the convenience of a personal organiser, the freedom to rove of a mobile phone, together with the programmability and application availability of a PC to deliver networked services "anytime, anywhere".

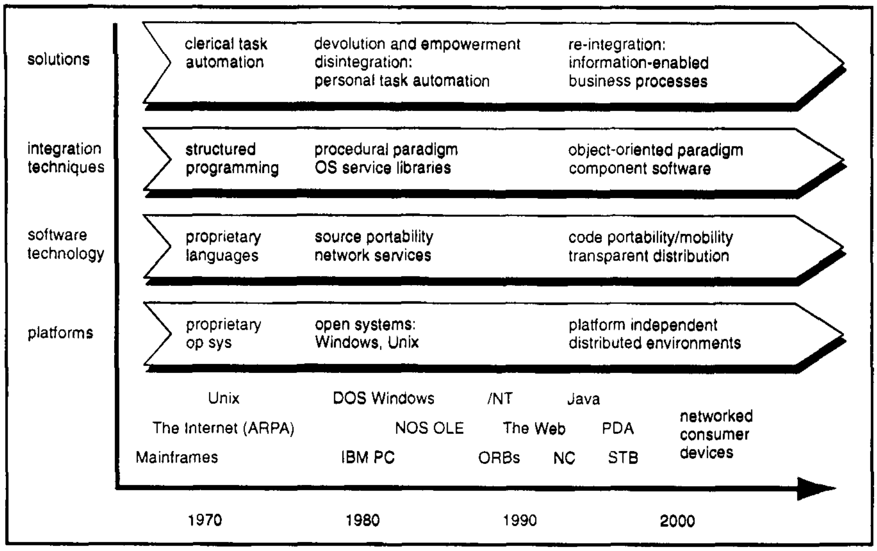

solutions
clerical task automation
devolution and empowerment disintegration: personal task automation
re-integration: information-enabled business processes
integration techniques
structured programming
procedural paradigm OS service libraries
object-oriented paradigm component software
software technology
proprietary languages
source portability network services
code portability/mobility transparent distribution
platforms
proprietary op sys
open systems: Windows, Unix
platform independent distributed environments
Unix
DOS Windows
/NT
Java
The Internet (ARPA)
NOS OLE
The Web
PDA
networked consumer devices
Mainframes
IBM PC
ORBs
NC
STB
1970
1980
1990
2000

Figure 4

10.4 Software Engineering

The Java phenomenon exemplifies a movement towards object oriented methods, tools and technologies, which has been gathering momentum over the last few years and is now rapidly gaining widespread adoption. One of the key factors has been the natural synergy between O-O approaches and distributed computing. O-O methods and approaches have a natural synergy with the tools and technologies being developed under the Java banner. As noted above, this could well pave the way for Java supplanting C^{++} as the preferred language for large software developments.

11. Conclusion

Successive generations of computing devices have seen an exponential trend in volumes and total market value. The next major growth area is likely to be in networked consumer devices, with volumes predicted in billions. Java paves the way for development of this new market by providing inherent platform independence and distribution capability. Network-centered computing will create new products for consumers (whether home or business) and new opportunities for information-based service suppliers.

Acknowledgements

The author would like to acknowledge assistance in the preparation of this paper from Simon Kenyon, who is an ICL Distinguished Engineer and Java consultant for ICL itcentre.

Java and Java-based trademarks and logos are trademarks of Sun Microsystems, Inc.

References

Computing Strategies, Forrester, vol. 12, no. 3, January, 1995

A vast amount of documentation is also available on the World-Wide Web. Useful starting points include:

JavaSoft White Papers & Java Specifications

http://www.javasoft.com

Novell strategic alliance with Sun

http://www.novell.com/strategy/sunqa13.html

The Oracle Network Computing Architecture

http://www.oracle.com/nca/index.html

Informix Universal Web Architecture

http://www.informix.com/

Corel Java Office Suite

http://206.116.221.27/

IBM Network Computing Products Division

http://www.internet.ibm.com/

Netscape Intranet client-server software

http://www.netscape.com/comprod/announce/index.html

Microsoft Java Strategy White Paper
http://www.microsoft.com/ntserver/JAVAWP.EXE

Biography

Nic Holt is a Systems Architect in ICL High Performance Systems. He has worked for ICL in a wide variety of technical roles since 1972, and is currently developing a technical architecture to support the evolution of the business systems of a large UK Government customer; the eventual system will be one of the largest in the world. Nic is an ICL Distinguished Engineer, a Fellow of the BCS and a Chartered Engineer. He has strong links with the research community, is a member of the EPSRC IT & Computer Science Panel, and was appointed a Visiting Professor at Glasgow in 1990.

Mobile Agents—The new paradigm in computing

L. L. Thomsen and B. Thomsen

Research & Advanced Technology, ICL, Bracknell, Berkshire, UK

Abstract

The emergence of agent based systems is signalling the beginning of one of the most important paradigm shifts in computing since object oriented methods and client/server based distributed systems. This paradigm shift will obviously require technology development, but of equal importance, or perhaps of even more importance, it will also require substantial education and methodology development. It is not hard to predict that agent technology is an important emerging technology, since it is already beginning to send shock waves through the computer industry. In the current hype, however, it is particularly difficult to distinguish publicity from reality and to get a clear impression of what agents are all about. In this paper we will try to give a comprehensive overview of what mobile agents are, what they may be used for and what the technical issues are.

1. Introduction

The importance of agent technology is not hard to predict, since shock waves are already being felt throughout the computer industry. A steadily increasing number of research projects, prototypes and even products containing, or claiming to contain, agent technology are being announced almost daily, and autonomous and mobile agents are beginning to appear on the internet. The emergence of these agent based systems is signalling the beginning of one of the most important paradigm shifts in computing since the introduction of object oriented methods and client/server based distributed systems.

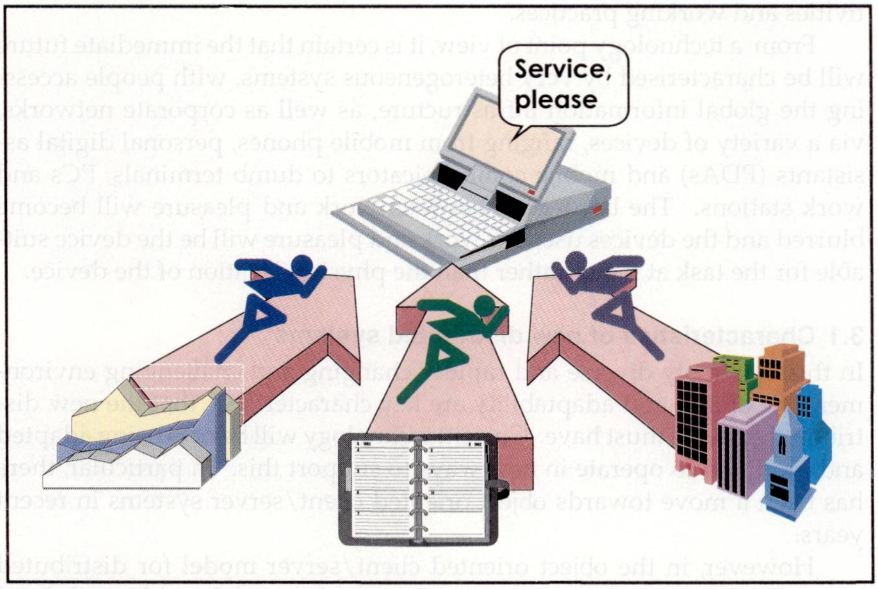
It seems inevitable in computing that whenever a new concept is proposed a hype is created and promises are made of a cure for all ills. So it is not surprising that with agents the current hype is making it particularly difficult to distinguish publicity from reality and to get a clear impression of what agents are all about.

Based on our experience of the Facile project at ECRC [Thomsen et. al. 1996a], [Thomsen et. al. 1996b], in particular the development of the Mobile Service Agent [Thomsen et. al., 1995a] application and Eclipse Agents [Bonnet et. al., 1996], together with our continuing survey of other research projects and emerging products, we will try to give a comprehensive overview of what mobile agents are, what they may be used for and what the technical issues are.

2. Mobile Agents—what they really are

Beyond the confusion created by the current hype there is to some extent an understandable confusion about agents, since there are, at least, two categories: intelligent agents and mobile agents. Historically, the two types of agent have been proposed and studied in rather different research communities. Intelligent agents have been studied quite extensively for several decades in the AI community, whereas mobile agents are more recent, mainly coming from distributed computing and to some extent from the programming language research communities.

Intelligent agents have been mainly studied in the context of robotics, planning and scheduling and machine learning to enable computer systems to act on partial and/or inconsistent information; e.g. letting robots learn about their environment or letting agents assist users by learning their preferences. Simple intelligent agents are making their way into products in the fields of handwriting recognition and information filtering, but the more spectacular dreams of human-like intelligent agents have so far failed to materialise.



Mobile agents, in contrast, need not be "intelligent"; in fact most mobile agents do not satisfy the AI community definition of "agent-hood". Put bluntly, mobile agents are just fragments of programs and their execution state that travel around a network of computers. Downloading an agent is often no different from downloading an application from an FTP site, except that the agent can take advantage of the client's infrastructure for agents without involving the user. The infrastructure (i.e., the client's program that receives and executes agents) allows the user to download and run an agent without having to take the steps of creating source directories, configuring, compiling and installing binary program images. This simplicity makes it much easier to download applications for temporary use; the user can retrieve the application with little effort, examine it and then throw it away or save it for later use. The catch-phrase here is "software on demand" — plug in an empty computer and the software you need comes to you. When you have finished, you can throw away the software so that you do not need to buy a bigger hard disk!.

Clearly, mobile agents can be used for more than just a convenient way of distributing lightweight applications but we shall return to this later.

3. Mobile Agents—what they can be used for

As the trend towards ever greater integration of computing and communication systems grows, an increasing number of people will expect to have access to information and computing power anywhere and at any time. This greater integration will facilitate the development of new leisure activities and working practices.

From a technology point of view, it is certain that the immediate future will be characterised by very heterogeneous systems, with people accessing the global information infrastructure, as well as corporate networks, via a variety of devices, ranging from mobile phones, personal digital assistants (PDAs) and mobile communicators to dumb terminals, PCs and work stations. The borderline between work and pleasure will become blurred and the devices used for work and pleasure will be the device suitable for the task at hand, rather than the physical location of the device.

3.1 Characteristics of new distributed systems

In this extremely diverse and rapidly changing and challenging environment, flexibility and adaptability are key characteristics that the new distributed systems must have. Current technology will be or is being adapted and modified to operate in new ways to support this. In particular, there has been a move towards object oriented client/server systems in recent years.

However, in the object oriented client/server model for distributed computing, communicating systems have to interact through predefined interfaces supported by preprogrammed methods. Each interface has implications for how the resources at each system can be accessed, what information can be exchanged, how control flow in an application can be distributed over the network, and what connectivity must be maintained between interacting systems.

Although some of these limitations in client/server interfaces are being addressed via the notion of interface brokers, many systems restrict communications to simple data. These restrictions constrain how applications can use the distributed system infrastructure. As systems move towards richer data formats and communication protocols, more possibilities for innovative applications emerge. The contrast between the available services, based on FTP, versus those based on the WWW dramatically shows this principle in action.

In developing new distributed system infrastructures, the next step towards increased flexibility is to communicate programs, not just data. For example, there are now proposals for including programs in the MPEG-4 video transmission standard [ISO, 1996], as well as in the virtual reality standard, VRML [Bell et. al., 1995]. PostScript also falls into this category, but the most spectacular example is probably the use of Java applets [Campione and Walrath, 1996] for transmitting executable contents on the WWW. In this respect it is also worth mentioning TeleScript [White, 1994] and Java servlets [Campione and Walrath, 1996] in the field of specialised servers.

However, each of the above mentioned systems has a very specific use for transmissible programs. By generalising the principle, transmissions can contain pieces of program code that can travel around a network and be executed at different nodes. Such program "fragments" can carry communication links with them, their own execution state, data, local procedures and other types of information. These travelling programs are called "mobile agents". By using communicating agents within a distributed system, applications can essentially program the network to suit their particular needs. The resulting communications interface is so rich that the above mentioned restrictions on how clients and servers can interact effectively disappear. Hence mobile agents can serve a variety of purposes in distributed systems.

3.2 Some uses of Mobile Agents

Many people have already experienced—maybe unknowingly—the first instance of mobile agents through the use of Java applets [Campione and Walrath, 1996] and Javascripts [Kent and Kent, 1996] to add active contents to HTML documents on the internet. TeleScript [White, 1994] and Safe-TCL [Gallo, 1994] have been proposed for similar use in adding active contents to e-mails. To some extent these "all singing and dancing" e-mails and home pages are just the vanguard of novel applications based on the mobile agent principle.

More generally, mobile agents can be used to distribute interfaces to servers, since an executing agent can communicate repeatedly without intervention from the user, allowing the construction of dynamic services, such as watching the fluctuation of share prices, and only notifying the user when certain thresholds are reached.

Mobile Agents can act as local representatives for remote services, provide interactive access to data they accompany, and carry out tasks for users temporarily disconnected from the network. Agents also provide a means for the set of software available to a user to change dynamically according to the user's current needs and interests. This idea forms the basis for the "slim" client or Networked Computer.

In a "wireless" network, agents can dramatically reduce the need for communication. For example, consider a service that is normally accessed by first completing a variable series of forms. This interface could be implemented by sending each form to the client, receiving and processing the completed form and then sending the appropriate next one. On a "wireless" network it would be much less expensive to receive one agent with all the forms and the logic for using them followed by one transmission to the server. Even in wired networks the reduced need for communication can be of importance. Since networks are now spanning the Earth one has to take into account that one's communicating partner may be on the other side of the globe and that the upper limit to the speed of communication is dictated by the speed of light.

By reducing communication needs, agents can also mask poor "wireless" links. If the user is accessing a service through an interactive interface, the interface may be unusable if network interruptions are frequent, even if the latency is low. Infrared local area networks are an example of a "wireless" network with this characteristic. As a user moves between rooms or round corners, the connection can be momentarily lost. An agent that encapsulates as much of the interface as possible can make the performance acceptable. Agents that can function in a stand-alone mode are suitable for nomadic systems that have no "wireless" connections.

Agents may allow servers to use customised communication protocols with clients. To receive an agent initially, the client and server must share some standard protocol. Once the agent is running, though, it can use a specialised protocol for communicating back to its home server or for communicating with other agents that it knows of; e.g. as in the case of the previously mentioned "stock market agent". Mobile agents may even move (teleport) themselves to different locations depending on the need of the application; e.g. querying a large database can be done by moving to the site of the database.

A promising area where mobile agent technology may prove very useful is in updating services in nonstop systems such as telephone switches. If the control program is structured as an agent, one may perceive sending it a replacement agent to take over when new services are to be offered.

The "old agent" and the "new agent" could coordinate their actions in such a way that the new agent gradually takes over from the old as incoming calls are being closed and new incoming calls are being opened. This strategy could be used for software upgrades in general. For example, a spreadsheet program could be structured as a system of agents and, as the vendor upgrades parts of the spreadsheet, the modified agents can be distributed to users via the global information infrastructure for automatic integration into their copies of the program.

Agents can serve a similar function in computer-integrated manufacturing systems. A new component, added to a system, could introduce itself by sending out an agent that provides an interface to the features of the new component and smoothly integrates it into the production control network. Essentially the component reprograms the control network to recognise and use it via its interface agent.

This use of agents for nonstop systems is also applicable to retail outlets where different devices, such as scanners, card readers and printers, are often added to the system. Similarly, new services, such as loyalty card point collection and the like, are added to the software of supermarket cash points. Large superstores now often operate on a 24 hour basis and it is becoming increasingly difficult to power down a busy store to do such additions. Again if the software is structured as agents, both new services and drivers for new hardware components could be added in a nonstop fashion.

Another area of application for mobile agents is groupware. Agents could be sent as attachments to messages to display their interactive content, and they could be used to filter and format electronic news depending on the user's preferences or corporate status. Agents could be used to alert the user of events of interest, such as updates to electronic calendars, or they could be used to negotiate meeting schedules. Mobile agents could also be used in workflow applications, where the agents know where to go in which order and what each human has to do. More ambitiously, mobile agents could be used to structure collections of groupware and distribute updates to software, as mentioned above, or to distribute new clients on client/server systems; e.g. a teleconferencing system could distribute clients to the participants before a session starts.

3.3 Benefits of Mobile Agents

Mobile agents may help people on the move by allowing them to download software useful to them as they are physically mobile. But mobile agents can also help people to stay where they are by enabling them to send software agents remotely on their behalf, thus enabling them to be virtually mobile. In fact, a combination of physical and virtual mobility can eliminate the need for carrying heavy equipment by transporting (teleporting) a user's communication and computing environment to the user's destination. It is much easier to move electrons than molecules and, as discussed above, mobile agents may help a person to use expensive resources more efficiently when physically mobile.

Such applications can be implemented, of course, without the use of mobile agents. There is at present no one convincing "killer" application that stands out and demands mobile agent technology, except perhaps the replacement of software modules in spacecraft. However, it is the enabling factor that mobile agents offer that is important since much tedious implementation work may be eliminated by establishing the infrastructure for mobile agents.

Application developers are already finding the platform independence and automatic memory management offered by Java [Arnold and Gosling, 1996] a decisive factor. However, it is important to realise that Java applets are only scratching the surface of what is about to happen. Developers are finding Java limiting when trying to implement some of the more advanced visions of what agent technology is all about. This is mainly due to the fact that Java, despite being platform and operating system independent, is not a language for distributed computing and definitely not a language for programming "full-grown" mobile agent systems. Although this criticism may seem extreme, since it is always possible to apply an agent oriented programming style in any sufficiently expressive language, the fact is that Java does not include constructs for directly supporting distributed computing, such as communication and persistence, nor does it contain constructs for supporting mobile agents, such as agent transport mechanisms and location change.

4. Mobile Agents—the technical issues

To unleash the full potential of mobile agents it is necessary to address a whole spectrum of issues from many different areas of computing. In fact, mobile agents and mobility seem to touch the foundations of computing, uncovering and breaking many of the tacit assumptions built into the current computing and communication infrastructure. Furthermore, the advent of mobile agents is stimulating solutions to "old" problems; e.g. one of the more spectacular visions for mobile agents is letting an agent or agents roam the internet searching for information. However, the main problem in this vision is not in getting the agents to move from place to place, but rather in solving the problems of schema integration and query optimisation that have been discussed in the distributed database community for decades, plus the fact that most data on the internet is held in unstructured data formats such as ASCII files. Similar arguments apply to questions of security, authentication and "wireless" communication.

Mobile agents are about moving code around in distributed systems, allowing the code to be transported and executed in different locations and carrying out tasks on behalf of somebody. Thus the following questions need to be answered:

- how is the code transmitted
- how is the code executed
- what assumptions can the code make about its execution environment
- what assumptions can the environment make about the received code
- how is the code constructed
- who constructed the code
- on whose behalf is the code executing.

Furthermore, since it is unrealistic to assume that agent based systems will have the luxury of being started from scratch, questions have to be answered about how systems based on mobile agents are to be implemented and executed on legacy systems, such as today's operating systems:

- what assumptions are broken
- how can they be repaired.

Obviously, systems based on mobile agents will need to address sequential programming, but most agent systems will also have aspects of concurrency needing communication and synchronisation. In a globally interconnected world, aspects of distributed computing, such as partial failure, different communication mechanisms and general quality of service, clearly need to be addressed. Furthermore, associated with the issue of physical mobility are the problems of location dependence and independence, disconnectivity and communication mechanisms with different bandwidths and costs to which solutions are required. Many, if not all, of these issues are not particular to mobile agents, but are general issues for reliable distributed and mobile computing.

These issues will be discussed in the following sections dealing with programming languages, architectures and security.

5. Programming Languages

Although much research on mobile agents is carried out in the distributed computing community, there is a strong emphasis on programming languages in the discussion. There are two reasons for this: firstly, agents need to be created and their behaviour described and, secondly, computations have to be able to move between machines.

5.1 Creating agents

Agents can be created in any language, just like the well-known fact that in order to create objects there is no need for an object oriented programming language—in the early eighties a number of books were published on how to perform object oriented programming in COBOL, PASCAL and C. Clearly a programming language with an object model, like C^{++}, Java or CORBA's IDL, makes object oriented programming easier. To quote Peter Nauer:

> *"There are no right or wrong models of computing—there are some models that are more suited for a given purpose than others."*

Hence, for agent programming a new language is not necessarily needed, but a language that directly supports the notion of mobile agents may make the task of constructing such agents easier, since such a language will support the programmer's mental model more directly.

5.2 Moving computations

This being said, however, there is one major difference between conventional systems, including distributed object oriented systems, and mobile agent systems, namely, the question of moving computations between machines. In some distributed object systems, e.g. CORBA (Common Object Request Broker Architecture), it is possible to achieve the effect of moving computations from one location to another by replicating the code of an object, freezing and marshalling its state and then sending it across to a different location, where the computation can continue because the code was already there. This approach only works if the installer of the software has control over the entire system and, at construction time, had enough foresight to know what code might need to be replicated.

In general there is no substitute for moving code, for example, downloading the needed software to an empty laptop before disconnecting and going "on-the-move".

A minimal requirement is that the code is relocatable, but more generally the code should also be machine independent. It is also desirable that the code is able to express constraints that it expects to be satisfied from the underlying platform and to express exceptional behaviour if these constraints are not satisfied. Furthermore, there is a question about the context of the code, especially its state and the communication links that the code carries with it.

Since code transmission, or transmission of computations, is given, some way of describing these computations is needed, and a programming language is just that—a way to describe computations. In principle, one does not need a new programming language for this, any programming language will do, even just sending machine code.

Sending machine code clearly only works for homogeneous architectures, although cross compilation or emulation would go some way towards solving this problem. The internet is a very heterogeneous network, thus sending machine code does not seem a viable option, but, for example, on many corporate PC networks it is safe to assume that almost everybody has a processor compatible with the Intel 386.

However, it is generally considered to be unsafe to receive and execute machine code, unless the code comes from some trusted supplier; e.g. it is generally considered safe to download a program from Microsoft or even a well reputed FTP site on the Internet, but letting arbitrary programs enter your machine is not acceptable. The problem is that machine code is very low level and it is hard, if not impossible, to predict what the code is going to do, and standard operating systems do not provide the right kind of protection for the user's data when such code is running under the user's own account or on the user's PC. This is mainly a problem of standard operating systems, which were not developed to cope with moving code around machines. But changing the operating system is difficult, even for Microsoft! Thus the machine code option for agent code transmitted by the general public should be left out.

5.3 Alternatives for moving computations

One alternative could be to send source code in some conventional language and to compile it on receipt. However, this approach will suffer from the above problem if the language is considered unsafe (e.g. C/C^{++} and even PASCAL). Moreover, this approach suffers from a lack of integration with the run-time environment. Normally, in a conventional language, all the code in a program is assumed to stay in a single address space. To maximise performance, conventional compilers strive to make the code as machine specific as possible, making this a seemingly difficult option as well.

Currently, many people consider that a safer alternative is to define a new scripting language (safe TCL is one such example) and have the code interpreted or have the program compiled to some intermediate representation (e.g. byte code) which can then be interpreted and thus checked at run-time for bad behaviour (Java and Telescript adhere to this approach). It may be quite natural to think that this is the way to go, given the trend towards defensive programming in normal application programming interfaces (APIs), where the programmer will put in checks to ensure that the parameters passed through the API satisfy the conditions for correct functioning of the API. However, this is just because the level of abstraction of the data passed through APIs is very low. If a higher level of abstraction is applied and guaranteed not to be compromised, defensive programming is not necessary. The same argument applies to transmitted code.

A third alternative is to receive intermediate code and to compile it "on-the-fly" (this technique is well developed for functional programming systems—the so called "parse, compile, execute loop"). Here code can be checked statically (or rather before execution) then compiled and executed at the full speed of the native code without fear of error, assuming that the compiler/system support is implemented correctly. There is nothing new here, it is simply that it is possible to take advantage of some considerations on ensuring type safe integration of new computations into a running system that have been developed over many years of research; e.g. Facile [Thomsen et. al., 1996a] uses this approach.

In fact, combinations of interpretation and compilation may be called for depending on the execution platform and/or the size of the agent and its expected run time performance. It is not always worth compiling an agent since it may only run a small program and, perhaps infrequently. Sometimes its execution performance is dominated by slow external communication and sometimes by (slow) user input. In these cases it may be sufficient just to interpret the code. In other situations, where the agent code is bigger or needs to run as fast as possible and be run repeatedly, it may be worthwhile compiling the agent to use native execution. The right choice between interpretation and compilation is difficult to make, and there are only preliminary results to guide developments [Knabe, 1997].

6. Architectures

As already mentioned, an infrastructure to support both clients and servers, sometimes called *agentware*, is needed. In fact, using the client/server metaphor in discussions on *agentware* may be misleading, since agent oriented systems will (have to) move towards a peer-to-peer architecture. As the agent language gets richer, more things need to be supported in its runtime environment and this clearly has implications for the software architecture of systems.

6.1 Traditional approach

Currently, when constructing distributed systems, it is necessary to use multiple programming styles with incoherent programming models, and often it is necessary to resolve conflicts by using low level methods even reverting to the lowest common denominator. Historically, application developers split their system so that application specific subsystems depended directly upon the operating system. However, application implementors often experience difficulties with their applications not being portable because each operating system has its own distinct application programming interface, and different versions of the applications employ different subsystem architectures due to the fact that distinct operating systems provide services that differ in kind and semantics.

6.2 Middleware

To achieve machine and operating system independence layers of software hiding the platform particulars, so called middleware layers, have been developed. Middleware providers distinguish themselves by implementing a particular collection of services that they advertise as useful. For example, the Open Software Foundation's Distributed Computing Environment (OSF/DCE) [Johnson, 1991] offers communication, naming and security services, COMANDOS [Cahill et. al., 1994] defines and implements a virtual machine that hides distribution to the programmer, while ANSAware [APM, 1989] provides support for federated client/server applications and distributed objects in a similar manner to Object Management Group's Common Object Request Broker Architecture (OMG/CORBA) [OMG, 1992a], [OMG, 1992b], [Mowbray and Brando, 1993]. Existing middleware is mainly a collection of useful services, but most often not a very coherent collection. This is due essentially to the fact that existing middleware was developed following a bottom-up approach from low-level services to language support.

Middleware is mainly a set of library functions which can be embedded within a host language or linked with routines in some other language. In addition to this, there is often a collection of macros and possibly a relatively simple language for describing data/simple objects to be transmitted as well as a stub compiler for translating from the common external representation to a representation in the host language (e.g. C/C^{++}, Ada, Fortran).

Attempts to adapt the middleware approach to agent based systems have clearly been pursued. Examples include the Knowledge Interchange Format (KIF) [Genesereth et. al., 1992] and the Knowledge Query and Manipulation Language (KQML) [Finin et. al., 1994], coming mainly out of the distributed AI community. However, as we have mentioned already, mobile agent systems differ from client/server architectures.

6.3 Agentware

Similar to middleware, agentware needs to integrate "services" traditionally found at different layers in current operating systems. Integration of concurrency constructs is needed to allow agents to spawn off sub-agents to carry out tasks locally or remotely, and synchronisation and communication mechanisms are needed for these to be able to synchronise their actions and communicate tasks and results back and forth. A safe interface to the local file system is also needed. But agentware also needs to integrate features of advanced programming language run-time management, e.g. automatic memory management is needed for higher level management of the physical memory, but it is equally important for ensuring that agents cannot peek and poke in other agents' memories. Agentware also needs to integrate concepts found in graphical user interfaces.

Clearly agentware can be implemented to sit on top of existing operating systems and this situation will be predominant in the foreseeable future. However, it is worth revising many of the layers in current systems architectures since agentware will otherwise end up duplicating services from the operating system, communication packages and window management systems. Already the layered systems in the communication subsystems suffer inefficiencies due to the lack of integration, e.g. on most TCP/IP implementations a message is copied three times before it actually "goes on the wire". Similarly, agentware will duplicate operating system features such as memory management, process/thread control and resource management in general. It is this realisation that lies behind the push towards the Java operating system and Java hardware.

7. Security

Mobile agents bring with them the fear of viruses, Trojan horses and other undesirable things. In many respects a mobile agent has similar characteristics to that of a computer virus. It may transport itself from computer to computer, it may use resources (CPU time, disk space, communication bandwidth, etc.), it may spawn copies of itself or create further mobile agents to achieve its goals. The major difference is that a mobile agent is benign and supposedly useful and friendly, but clearly it will not always be sufficient to rely on the creator's good intentions.

The foremost security concern with mobile agents is that they may introduce a very simple way of penetrating the security fire walls that traditional operating systems set up to protect local resources from misuse and tampering. This is due mainly to such systems being conceived at a time when software was difficult to move from machine to machine, and installed software did not communicate with the outside world. The concern in traditional multi-user systems was to protect the users from interference with each other. Such systems assume that the user is in control of the software he or she is executing and will ensure that execution under a user account or on a user's PC could at worst harm that user. However, these system assumptions are not applicable when mobile agents start to move from host to host and are executed in different locations and different environments. Thus it is clearly understandable that IT managers are concerned about the potential risks posed by mobile agents, based on their experiences with viruses in corporate PC networks.

Security aspects can largely be separated into issues concerning authentication, privacy/confidentiality and integrity.

Authentication is not a problem specific to mobile agents, but mobility and mobile agents add a good stress test to existing approaches. Authentication is needed in any multi-user system, in particular in distributed multi-user systems operating in an open network infrastructure. The authentication of users is needed for determining their access rights (for their agents) and other capabilities. This is traditionally done via log-in procedures where user IDs and passwords are checked against account information stored in the computer system. However, this model assumes that the user is going to have a one-to-one session with the computer. This view breaks down when the user views the computer as a communication device to a networked communication and computing infrastructure. In this scenario credentials may have to be checked whenever an agent moves around and/or during execution. The user may need to have a unique ID (e.g. a phone number, credit card, smart card or IP address) or multiple IDs and know when and how to use the relevant one.

Beyond the traditional authentication procedures, most (proposed) solutions are based on encryption and digital signatures. Integrating these with the mobile agent paradigm has the advantage that the problem of letting mobile agents purchase goods or services on behalf of their users can be readily solved as well. However, general solutions in this area are still being sought, partly because there are technical problems to be solved and partly because of political constraints; e.g. US export restrictions on some encryption methods prohibiting their use outside the US. In the same vein, France has restrictions preventing private citizens using any form of encryption.

As discussed above issues regarding privacy and confidentiality are currently being dealt with using techniques based on new programming languages and compiler technology specifically oriented towards safe execution, combined with new secure application interfaces added to existing operating systems or even new operating systems. The main approach to systems based on mobile agents is based on the development of safe languages; i.e. languages that do not allow peek and poke, unsafe pointer manipulations and unrestricted access to file operations. This is often achieved through interpreted languages. Java [Arnold and Gosling, 1996], Safe-TCL [Gallo, 1994], Telescript [White, 1994] and VBScript [Holzner, 1996] are examples of this. Following this approach all (or at least all the perceived) dangerous operations are forbidden or (minimally) monitored. As mentioned earlier the same level (or even a stronger level) of security can also be achieved by advanced compiler technology that allows native code to be generated and executed.

7.1 Particular concerns with Agents

The first concern in agent security is about the client downloading an agent from the net, but there is an almost symmetric issue that can be paraphrased as "agents going to places of ill-repute" in the sense of somebody sending an agent on his or her behalf. Will the person be allowed to do so and, if so, without interference? These are difficult issues since hosts need to execute agents and, for security reasons, need to analyse their code, but agents should not reveal information beyond that strictly necessary for safe execution; e.g. an agent trying to buy the cheapest ticket should not reveal the current lowest price so that a host that sees this could get away with bids just under the current best as opposed to normal cut-throat pricing. The best hope is to apply a style of programming that will be hard to reverse-engineer.

Even when the fear of viruses has been eliminated, mobile agent systems may be a great deal more complex to develop than traditional client/server applications since it is very easy to create agents that will counteract each other or inadvertently "steal" resources from other agents. Just as with client/server systems, constructors of systems based on mobile agents may inadvertently introduce logical bugs such as deadlocks and livelocks. Since an agent can move from place to place, it can be very hard to trace the execution of such systems and, when constructing such systems, special care must be taken. Analyses of such systems requires the most advanced results from concurrency research, e.g. [Borgia et. al., 1996], [Degano et. al., 1997].

Clearly electronic commerce and the global information society are open to fraud, trust misuse, malicious misinformation and even "terrorist-like" attacks, just as in society today. Most agent systems address collaborative agents, or personal agents, in order to simplify user interaction and information gathering. However, offending applets are becoming a problem for some businesses on the internet [Goulde, 1997], and it is envisioned that a whole class of self interested or even hostile agents may emerge [O'Riordan, 1995]. Creating an infrastructure that will allow governments and other authorities to monitor, or at least trace, such antisocial behaviour and produce evidence that can be used for prosecution is a challenge that only few researchers have started to investigate. Clearly a fine balance will have to be struck between a "big brother is watching you" system and a global commercial village where commerce and information interchange can take place freely.

Mobile agents pose many security problems, but companies who decide not to develop or use mobile agent technology because of the security issues will run a high risk of being beaten in the market place by someone who either ignores the security problems, or "hacks" some sort of partial solution to them. Although considerable resources have been spent on computer security, in particular for military systems, the state of the art is still very much in its infancy since many of the issues are not very well known (or perhaps only known by persons with security clearance and therefore not able to share their knowledge). Most security protocols are flawed—the flaws have just not been discovered yet. To get a handle on the security problems posed by mobile agents, good theory and experiments are needed (and spectacular failures, such as the Microsoft Internet Explorer/ActiveX bug, also help to speed up developments). Again, mobile agents are not going to solve these problem but may speed up the search for solutions!

Security needs to embrace the entire system. Partial solutions cannot be trusted to rely on the environment since the environment may be compromised. One security flaw in Java is based on the fact that files loaded from the file system are more trusted than files loaded over the network, but if an attacker manages to place a file in the user's file system by other means, e.g. luring the user to *ftp* the file, the security of the browser may be completely compromised.

In reality, levels of security are a balance between the cost of implementing them, the cost of running them and the cost of security breaches. The security required for buying a house, for example, is very different than that required for buying a T-shirt or a book. Even if you can prove you have been cheated out of five pounds, the cost of a legal procedure could easily make you accept the loss of five pounds. Similarly, the benefits that mobile agent systems promise to bring now outweigh the threat of viruses, hostile hosts and self-interested agents.

8. Next generation Mobile Agent systems

The battle for the first generation mobile agent systems seems to have been won by Java, although Microsoft may launch one more attack based on a revamped ActiveX/VBScript strategy. Therefore, from a research and business strategy point of view, it is worthwhile looking towards the next generation.

Since Java is already showing deficiencies there will be an intermediate generation (generation 1.5) where middleware platform providers, such as OSF/DCE and OMG/CORBA, will try to "get in on the action". Such middleware integrations will clearly serve a need and push the technology in the right direction. The integration of Java with CORBA, for example, will allow legacy applications, already brought into the client/server world by a CORBA wrapping, to be made readily available to the Java world, and Java will add capabilities by allowing the CORBA-enabled client to be downloaded and executed anywhere on the Internet. However, this approach suffers from the complexity of having to master both Java and CORBA. Although they are both object based, they have largely incompatible object models. Therefore, it will be necessary to implement systems that map from one to the other. Furthermore, CORBA has a limited data model and will thus restrict the type of objects that can be exchanged between clients and servers. Moreover, Sun has developed an extension to Java, called the Remote Method Invocation (RMI) interface, which allows developers to construct Java based systems that can invoke Java object methods remotely, in much the same way that Remote Procedure Calls (RPC) allow the invocation of procedures in client/server applications today. The data model for Java's RMI is Java and thus there is no need for cumbersome translations between different data models. Furthermore, since the Java data model is richer than the one supported by CORBA's IDL, a more sophisticated exchange of data will be possible. In addition to this, Java supports easy interfacing to legacy applications through native interfaces and thus threatens to make middleware, such as CORBA and DCE, obsolete. However, the solutions based on middleware integration, and even the Java RMI, are all rather traditional in their approach to distributed computing. This is not difficult to understand since distributed computing is still a bit of a black art with very difficult models and arcane interfaces.

To bring the vision of mobile agent technology successfully to the end-user, it will be necessary to overcome these difficulties. It will be necessary to establish abstract interfaces that are separate from particular communication infrastructures, such as ATM, TCP/IP or X.25, in much the same way that high level languages, such as Java, allow programs to be written without concern for physical memory management. There is no reason why programmers have to manage setting up and terminating network connections, linearising and delinearising objects, splitting and reassembling network packages, etc. This can be handled in much the same way as automatic memory management (in fact, much reuse is possible; e.g. the code for linearisation). First generation agent systems have already demonstrated the usefulness of platform independence. However, platform independent systems have in the past been rejected because they cannot take advantage of the particulars of a given platform. There is clearly a conflict in the desire to be able to write applications capable of moving across various platforms and the desire to take advantage of platform specific features; e.g. graphics or video accelerators, multi-processors or high performance networking. A way to overcome this conflict is to move from a uniform platform to a uniform way of describing platform differences, much in the spirit of the ideas behind intelligent networks and ATM technology. Here quality of service (QoS) requirements may be specified and the transport system can report if these requirements can be honoured, otherwise the system should be capable of graceful degradation. For this to be manageable, it will be necessary to develop intelligent interfaces based on the principles of declarative programming.

8.1 Second generation and future Mobile Agent systems

Second generation agent systems and their successors will have to bridge the gap between intelligent agents and mobile agents. The complexity of mastering this will only be achievable within a coherently integrated framework, based on a well defined programming model (or rather integration of well understood programming models). Indications of this approach may be found in the April system [McCabe and Clarke, 1995] from Fujitsu/

Imperial College, Objective Caml/MMM [Remy, 1994], [Remy and Vouillon, 1997] from INRIA, Erlang from Ericsson [Armstrong et. al., 1996] and Facile [Thomsen et. al., 1996a] from ECRC/ICL. Interestingly these are all based on the integration of polymorphically typed functional programming, some object orientation and the CCS process model [Milner, 1989], together with its higher order and mobile extensions, CHOCS [Thomsen, 1993], [Thomsen, 1995a], LCCS [Leth, 1991] and the pi-calculus [Milner et. al., 1989].

As already mentioned, systems based on mobile agents will exploit sequential programming, but most will also have aspects of concurrency needing communication and synchronisation. Obviously, in an interconnected world, aspects of distributed computing, such as partial failure, location dependence, different communication mechanisms with different bandwidths and costs plus general quality of service need to be considered as well. Mobile agent systems will, in fact, need to bring together many, if not all, aspects of computing—aspects that through decades of separation in layers and subsystems have developed different and incoherent modes of operating. The interactions between the layers and subsystems are very complex, and methods and tools developed for one purpose will have to be used for new purposes. Program analysis, for example, originally developed for code optimisation, is used for security purposes in the Java virtual machine. Many areas in communications, such as point-to-point, asynchronous, broadcast, stream based and RPC based, are still not well understood. There is a continuing debate about data formats and data representation in communication and incorporating code containing data will fuel this discussion further.

Many issues remain in programming languages, whether they be imperative, object oriented, declarative, logic or constraint oriented. Issues of naming, scoping, binding, typing and modularity will have to be resolved. There are even old fashioned questions about syntax (or syntactic confusion), behaviour of constructs (or rather unexpected behaviour such as arrays starting at 0) plus the general question of semantics (which most often is ill defined or loosely defined), which will have to be answered.

Systems aspects, such as memory management, have already been mentioned for security reasons, but other aspects, such as persistence and graphical user interfaces, need to be considered too.

Finally, we have to remember that the concept of mobile agents is about moving programs around; so programs as data objects will be an issue and questions such as their representation, either syntactic, intermediate or native, and whether to compile or interpret the code, need to be answered.

8.2 Programming model

First generation agent languages, such as Java, Safe-TCL, VBScript and Javascript, are rather traditional in that they are based on the imperative and object oriented programming paradigms. These languages, therefore, do not gain much support from formal reasoning, although considerable effort is currently being put into "postmortem" construction of formal foundations for Java. Parts of its type system, for example, have recently been proved to be sound [Drossopoulou and Eisenbach, 1997] and the core language together with some of its concurrency constructs have been given a model based on game theory [Abramsky and McCusker, 1997], but issues such as inheritance, polymorphism and sub-typing, all present in Java, are still not well understood.

The informal treatment of language semantics, in particular concurrency and communication, and security aspects are major problems in the first generation of agent systems. The interactions between the subsystems are very complex and methods and tools developed for one purpose are now being used for new purposes; e.g. program analysis, originally developed for code optimisation, is used for security purposes in the Java virtual machine. However, since the Java byte code verifier uses a "model" of the code, which does not always reflect the operational behaviour of the virtual machine, a security loophole exploiting this fact has recently been uncovered and later demonstrated. Similarly, a security problem with Microsoft's Internet Explorer and embedded ActiveX component occurred because a technology created for a static environment like the desktop was put into the (virtually) mobile environment of the WWW. Technically the problem arises from ActiveX control being based on dynamic binding and thus ActiveX will bind to anything of the same name upon downloading to a client. Malicious web page creators thus only have to guess the name of the program they want to start and, since most DOS/Windows programs will reside in "standard" directories, this is not very difficult.

The security flaws in Java and ActiveX touch upon the reasons why high level languages have been rejected in the past, because sometimes there is a price to be paid for safety. Everything that can be done with a Java applet can also be done using traditional http requests; however, things can happen much faster if the code is being executed locally. ActiveX, which employs almost no security model, can allow things to happen very quickly indeed using native code with unrestricted access to the client machine, where the code may then reside for future use. Java applets employ a much stronger security model and are consequently slower and more clumsy to run. Although there are, as discussed above, flaws in the Java security model, it is a step in the right direction. In general it is not advisable to allow code downloaded from anywhere to perform certain operations, such as direct memory access or reformatting a hard disk, although, in some circumstances, this would be the most efficient way of doing it. Such operations leave openings in the system's security, which will inevitably be discovered and exploited at some point.

8.3 The integration aspect—new features

Integrating all the needed features in a coherent way will require a leap in bringing new techniques, research results and technology together in practice. The most promising solutions follow advanced languages based on static binding, well defined scoping rules and rich interface descriptions, combined with systems based on access lists and capabilities.

The investigation and creation of multi-paradigm programming languages has received much attention in recent years. Most approaches are based on integrating well understood programming models, most of which are formally defined. The state of the theory of computing is being moved forward by such efforts, since points are being highlighted where the existing theories are not applicable, either because they cannot be used to describe systems adequately or because they do not scale to full size systems. Such efforts are needed, since techniques, such as program analysis, traditionally used for optimisation, will now be used as part of the security mechanisms. If there are no adequate theories to describe and prove that these algorithms are doing what they are supposed to be doing, how can they be trusted? As the security flaw in the Java virtual machine has already demonstrated, this is not a trivial question.

However, it has been apparent for a while that flow analysis and modular programming have goals that are in opposite directions. Flow analysis usually needs to take a global view of the code, and optimisations based on flow information assume that this information is complete. Interprocedural flow analysis, or higher order flow analysis, is still in its infancy and difficult to implement efficiently. However, promising results, based on placing flow information in type systems, which can then be used in module interfaces, have recently been developed.

Placing flow information in type systems leads to new type systems with richer properties than purely input/output relations. In a sequential environment these descriptions may be used to propagate information about, for example, memory usage, and in a concurrent environment they may be used for process and communication channel allocation and even deadlock prevention.

Enriching type systems with abstract descriptions of the dynamic behaviour of programs may also be used for security, since it is possible to specify the accepted dynamic behaviour. In fact, unacceptable behaviour will be excluded through the type system and monitoring of the dynamic behaviour will, therefore, not be needed. This is similar to the way that static typing today eliminates the need for checking data in procedure calls, leading to faster execution.

Such type systems can also serve further purposes. In distributed systems, activity will not normally start in one node, but will start in different locations, after which logical links will be progressively established to form the overall system. Currently such systems have to be integrated at very low level, since they do not share a common way of describing and making available interfaces to other systems. However, as interface descriptions get richer, binding based on interfaces will be possible.

Although strong static typing is being recognised as very important in ensuring program correctness and safety, the additional information needed at program construction time may be seen by programmers as an additional burden. Advanced languages, such as ML [Milner et. al., 1990], employ type inferencing, rather than type checking, allowing programmers to write programs almost without type specification. The compiler then infers the type and reports back if there are type errors. Type inferencing can also be applied to the intermediate code representations, used for transporting mobile agents, to compare between the expected type of the agent and the actual type. As the type system gets richer, including abstract descriptions of the dynamic behaviour of the program, a fine balance between decidable and undecidable properties must be found.

Beyond machine inferable properties one can imagine instrumenting agents with machine checkable properties. Such proof carrying code can then be examined by machine assisted verification techniques. Since such techniques might be too difficult for the average user to handle, an industry based on trusted code producers and trusted third party code verifiers could be developed, which, when combined with digital signatures and strong encryption, could lead to a new era of reliable and trusted computing.

9. Conclusions

Agent oriented programming is an emerging and exciting paradigm in computing. It is not hard to predict that, by the end of the century, it will be as important as object oriented programming and client/server based distributed systems were in the eighties and early nineties.

Clearly, mobile agents pose many security problems, mainly due to old security loopholes and the fact that computer security is still in its infancy. However, companies, who decide not to develop or use mobile agent technology because of the security problems, will run a high risk of being beaten to the market place by someone who either ignores the security problems, or "hacks" some sort of partial solution to them.

The first wave of mobile agents has already appeared on the Internet in the form of Java applets and Microsoft has been trying to push its ActiveX/ VBScript technology as a competing technology. Even though Java is a very novel technology it is built on a very conservative evolutionary approach; bluntly, it is just a nicely cleaned up version of C^{++}. However, the language still has some ugly and confusing constructs (e.g. arrays starting at 0) and assignments like (I := ++I—), as well as some direct flaws (e.g.

covariance of arrays which, for type safety, should have been contravariant) and the implementation still has some holes in its integrity (e.g. the code verifier does not always verify the actual code). Most significantly, Java is not a distributed programming language and developers, therefore, are already experiencing difficulties in implementing the more ambitious mobile agent visions. Remedies for this situation are being proposed and are mainly based on middleware integration, such as DCE, CORBA and the Java RMI following a distributed object oriented approach. More "agent like approaches", such as IBM's aglets and General Magic's Odyssey, both based on ideas from General Magic's telescript, have also recently emerged. Since Java has deficiencies in design, architecture and implementation, a certain confusion exists, both because people do not understand all the issues (which is understandable, since they are very complex) and also because of commercial factors, such as Microsoft's attempt to derail the Java steam roller by defining their own APIs.

The second generation mobile agent systems will be able to draw upon substantial experience from the first generation (and generation 1.5). Integration of agent technology from the AI community and the distributed systems community will lead to technologies capable of implementing the most ambitious visions about software agents. New network technologies, such as "wireless" ATM and intelligent telephony, are emerging that will further the need for simple software solutions based on the agent principle. Due to security concerns many of the important issues of computer science, such as structured programming, language design and system engineering, are back on the agenda. Clearly second generation systems will face the usual question of why another new language/system is needed now that the Java standard is here. However, despite commercial pressures, it is important to try not to build tomorrow's legacy systems today. By "thinking mobility" better systems may be created, because many old and well known problems need to be solved properly for security reasons. Moreover, these problems only need to be solved once and future systems will benefit. It is easy to make a mobile system non-mobile, but extremely difficult to make a non-mobile system mobile.

Independently of the underlying technology supporting the implementation of mobile agents, there is the emerging concept of agent oriented programming as a way of thinking about computer systems. Already Java applets are signalling a new way of distributing software. When tools for electronic commerce become more stable and accepted, new ways of licencing software will evolve; e.g. payment could be made on the basis of methods similar to those being considered for video on demand.

A paradigm shift almost always calls for cultural changes and the earlier one starts to prepare for this the smoother such changes will be. The paradigm shift will obviously require technology development, but of equal importance, or perhaps of even more importance, it will also require substantial education and methodology development. Object oriented programming has been around for almost twenty years, but object oriented development methods are only now starting to be mainstream practice. It is therefore important that methods supporting a transition from object oriented programming to agent oriented programming are developed. There will be a need for strong involvement in developing both new standards and collaborative platforms—even perhaps with competitors.

Acknowledgements

Most of the authors' previous work on Facile was carried out at ECRC in Munich, funded in part by several projects in the European Union ESPRIT programme. Some of the current work is also supported by the European Commission and this support is gratefully acknowledged.

A special thanks to Pierre-Yves Chevalier, Xerox Research Lab. in Grenoble, for producing the agent drawing while we were colleagues at ECRC in Munich.

Finally the authors would like to thank their former colleagues at ECRC for contributing to the work on Facile and Mobile Agents.

References

ABRAMSKY, S. and McCUSKER, G., "Linearity, Sharing and State: a Fully Abstract Game Semantics for Idealized Algol with active expressions," In O'Hearn and Tennent (eds.), Algol-like languages, Birkhauser 1997.

ARCHITECTURE PROJECTS MANAGEMENT LIMITED, "ANSA: An Engineer's Introduction to the Architecture," ANSA Technical report, Number TR.03.02., 1989.

ARNOLD, K. and GOSLING, J, "The Java Programming Language," The Java Series, ISBN 0-201-63455-4, 1996.

ARMSTRONG, J., WILLIAMS, M., WIKSTROM, C. and VIRDING, R.: " Concurrent Programming in Erlang" Prentice Hall, ISBN 0-13-285792-8, 1996.

BONNET, PH., BRESSAN, S., LETH, L. and THOMSEN, B., "Towards ECLiPSe Agents on the INTERNET," Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, in conjunction with JICSLP'96, Bonn, Germany, September 2—6, 1996. (http://clement.info.umoncton.ca/~lpnet/final/eclipse-agents.ps.Z)

BORGIA, R., DEGANO, P., PRIAMI, C., LETH, L. and THOMSEN, B., "Understanding Mobile Agents via a non-interleaving semantics for Facile," proceedings of the Third International Static Analysis Symposium (SAS'96), Aachen, Germany, September 24-27, LNCS 1145, Springer Verlag, 1996.

BELL, G., PARISI, A. and PESCE, M., "The Virtual Reality Modeling Language," Specification, Internet, 1995. http://www.sdsc.edu

CAMPIONE, M. and WALRATH, K., "The Java Tutorial: Object-Oriented Programming for the Internet," The Java Series, 1996. ISBN 0-201-63454-6

CAHILL, V., BALTER, R., HARRIS, N. and ROUSSET DE PINA, X. (editors), "The COMANDOS Distributed Application Platform," Springer-Verlag, ESPRIT Research Reports, 312, 1994.

DEGANO, P., PRIAMI, C., LETH, L. and THOMSEN, B., "Analysis of Facile Programs: a Case Study," proceedings of the Fifth LOMAPS Workshop on ANALYSIS AND VERIFICATION OF MULTIPLE-AGENT LANGUAGES, SICS, Stockholm, Sweden, June 24-26, 1996, LNCS 1192, Springer Verlag, 1997.

DROSSOPOULOU, S. and EISENBACH, E., "Java is Type Safe - Probably," to appear in Proceedings of 11th European Conference on Object Oriented Programming, 1997.

FININ, T., FRITZSON, R., MCKAY, D. and McENTIRE, R., "KQML as an Agent Communication Language," Proceedings of the Third International Conference on Information and Knowledge Management, ACM Press, November 1994. (http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps)

GALLO, F., "Agent-Tcl: A white paper," Draft document, posted to the safe-tcl@cs.utk.edu mailing list, December 1994.

GENESERETH, M., FIKES, R. et. al., "Knowledge interchange format," Technical Report, Computer Science Department, Stanford University, 1992.

GOULDE, M., "Digitivity's Applet Management System Secures Applets — Run Java Applets Securely Behind Firewalls," Draft, the Patricia Seybold Group, 1997.

HOLZNER, S., "Web Scripting with VBScript," MIS:Press, ISBN 1-55851-488-0, 1996.

INTERNATIONAL STANDARDISATION ORGANISATION, "Description of MPEG-4," ISO/IEC JTC1 SC29/WG11 N1410, MPEG 96, Chicago, Oct. 1996.

JOHNSON, B.C., "A Distributed Computing Environment Framework: An OSF perspective," Technical report DEV-DCE-TP6-1, Open Software Foundation, Inc., Cambridge, MA, 1991.

KENT, P. and KENT, J., "Official Netscape Javascript Book," Netscape Press, ISBN 1-56604-465-0 , 1996.

KNABE, F., "Performance-oriented implementation strategies for a mobile agent language," Chapter in Mobile Object Systems, Christian Tschudin and Jan Vitek, editors, to appear in the Springer-Verlag Lecture Notes in Computer Science series, 1997.

LETH, L., "Functional Programs as Reconfigurable Networks of Communicating Processes," Ph. D. Thesis, Imperial College, Department of Computing, 1991.

McCABE, F.G. and CLARK, K.L., "April - agent process interaction language," in M. Wolldridge, N. Jennings, editor, Intelligent Agents, LNAI, vol. 890, Springer-Verlag, 1995.

MILNER, R., "Communication and Concurrency," Prentice Hall, 1989.

MILNER, R., PARROW, J. AND WALKER, D., "A calculus of mobile processes," Technical report ECS-LFCS—89—85, Laboratory for Foundations of Computer Science, Edinburgh University, 1989.

MILNER, R., TOFTE, M. and HARPER, R., "The Definition of Standard ML," MIT Press, 1990.

MOWBRAY, T.J. and BRANDO, T., "Interoperability and CORBA-based Open Systems," Object Magazine, pp. 50-54, Sept.-Oct. 1993.

OBJECT MANAGEMENT GROUP, "Object Management Architecture Guide," num. Document 92.11.1, OMG Inc., Framingham, MA, Nov. 1992.

OBJECT MANAGEMENT GROUP, "Common Object Request Borker: Architecture and Specification," num. Document 92.12.1, OMG Inc., Framingham, MA, Dec. 1992.

O'RIORDAN, B., "Self-interest as an opportunity," Draft, personal communication.

REMY, D. and VOUILLON, J., "Objective ML: A simple object-oriented extension to ML," Proceedings of POPL'97, 1997.

REMY, D., "Programming Objects with ML-ART: An extension to ML with Abstract and Record Types," Proceedings of TACS'94, 1994.

THOMSEN, B., "A Second Generation Calculus for Higher Order Processes," Acta Informatica 30, Springer Verlag, 1993.

THOMSEN, B., "A Theory of Higher Order Communicating Systems," Information and Computation, Vol. 116, No 1, 1995.

THOMSEN, B., KNABE, F., LETH, L. and CHEVALIER, P.Y., "Mobile Agents Set to Work," Communications International, July, 1995.

THOMSEN, B., LETH, L. and KUO, T.M., "FACILE – from Toy to Tool," ML with Concurrency: Design, Analysis, Implementation, and Application, Flemming Nielson (Editor), Springer-Verlag, 1996.

THOMSEN, B., LETH, L. and KUO, T.M., "A Facile Tutorial," Invited Tutorial, in Proceedings of CONCUR'96, Concurrency Theory 7th International Conference, Pisa, Italy, August 1996, LNCS 1119, pp 278—298, Springer Verlag 1996.

WHITE, J. E., "Telescript technology: The foundation for the electronic marketplace," General Magic white paper, 2465 Latham Street, Mountain View, CA 94040, 1994.

Biographies

Lone Leth Thomsen

Lone Leth Thomsen joined ICL's Research & Advanced Technology Department as a Principal Researcher on July 1st, 1996. She was formerly at the ECRC in Munich, where she was a senior researcher. At ECRC she worked on the Facile project and she was the ECRC site leader for Esprit BRA working group SEMAGRAPH II 6345 and the Esprit BRA COORDINATION 9102. Jointly with Jean-Jacques Levy and Bent Thomsen she was the Technical coordinator for Esprit BRA CONFER 6564.

She is currently Technical coordinator for Esprit working group CONFER-2 21836 together with Jean-Jacques Levy and Bent Thomsen.

Lone Leth Thomsen received her PhD in computing from Imperial College, London in 1991. She has an MSc in Software Engineering and Computing Systems from Aalborg University Centre, Denmark. She is a member of ACM, IEEE, BCS, EATCS and IDA.

Bent Thomsen

Bent Thomsen joined ICL's Research & Advanced Technology Department as a Principal Researcher on July 1st, 1996. He was formerly at the ECRC in Munich, where he was a senior researcher and teamleader on the Facile project. At ECRC he was the site leader for Esprit BRAs CONCUR2 7166, CONFER 6564 and LOMAPS 8130. Jointly with Jean-Jacques Levy and Lone Leth Thomsen he was the Technical coordinator for Esprit BRA CONFER 6564.

He is currently the ICL site leader for Esprit BRA LOMAPS 8130 and Esprit working group CONFER-2 21836 and, jointly with Jean-Jacques Levy and Lone Leth Thomsen, he is the technical coordinator for Esprit working group CONFER-2 21836.

Bent Thomsen received his Ph.D. in computing from Imperial College, London, in 1991. He has a BSc in Mathematics, a BSc and an MSc in Computer Science from Aalborg University Centre, Denmark. Bent Thomsen is a member of ACM, IEEE, BCS, EATCS and Dansk Magister Forening.

Lone Leth Thomsen and Bent Thomsen are currently establishing a research framework, including a virtual laboratory with Fujitsu Labs, on Mobility and Agent technology, while continuing their research in the Agents area.

For more information and previous work see http://sst.icl.co.uk and http://www.ecrc.de/research or e-mail addresses: lone@sst.icl.co.uk and bt@sst.icl.co.uk

The SY Node Design

G. Allt, P. DeSyllas, M. Duxbury, K. Hughes, K. Lo, J.S.M. Lysons and P.V. Rose

ICL High Performance Systems, Manchester, UK

Abstract

SY is the latest generation of high performance Series 39 processors, and is fully compatible with existing Series 39 systems. The design is built on the highly successful SX design, with the OCP in particular reusing the SX Picode pipeline, but the design does include a number of features that reduce the total cost of ownership. These include: the use of CMOS technology, an SMP (symmetric multiprocessor) architecture, and an optional cheaper multinode interconnection scheme. These new features enable higher performance to be provided at a significant reduction in relative cost throughout the range of scalable product offerings. This paper describes the architecture of the SY node and the techniques used to achieve these objectives.

1. Introduction

SY is the latest model in the Series 39 range of processors. It has been introduced to provide better cost-performance than the existing systems for users of the ICL OpenVME operating system. It provides complete backwards compatibility: all current user programs are capable of running on SY without the need for recompilation.

The design builds on the SX design. SX is built from large ECL gate-arrays manufactured by Fujitsu. The SX node consists of four units: a sophisticated pipelined processor for the execution of 2900 order code and SX picode; an Input Output processor providing up to 16 SMARTfibre or Macrolan channels; an Inter-Node processor allowing up to 8 SX nodes to be joined in a multinode system; and the store. Details of the SX node architecture and design were published in the ICL Technical Journal in 1990 [Eaton et al, 1990], [Abraham et al, 1990].

A major innovation was to abandon the use of high-speed ECL (Emitter Coupled Logic) technology. ECL technology fails to provide the required level of integration required for modern processors and has a high power consumption. SY is built from semi-custom CMOS VLSIs manufactured by Fujitsu to ICL's own designs. Although CMOS gates are inherently slower than ECL gates, the level of integration has enabled us to build an OCP of comparable performance to an SX OCP, but considerably cheaper to develop, manufacture and use. In addition, the greater level of integration provides a much improved level of hardware reliability.

In order to provide large users with an upgrade path from the most powerful SX systems, we had to provide systems that were several times more powerful than a 6-node SX. Although the OCP Picode processor used in SY is the similar to the SX design, the store system has been radically altered to enable 4 OCPs to share a common store as a symmetric multiprocessor. The design includes a coherent caching algorithm that was extensively modelled and a split-transaction protocol Node Bus. This provides a single Series 39 node with at least 3 times the power of an SX node and because of the change in underlying technology a SY node occupies much the same floor space and volume as an SX node.

A significant cost of SX systems was the multinode interconnect. The use of high-speed fibre-optic connections provided the bandwidth required while, at the same time, allowing nodes to be separated physically by up to 750 m (or 2 km in extremis). Although a number of customers required such a physical separation to provide disaster tolerance, a large number were known to site the two nodes adjacently. In SY, we have provided a multinode interconnect that is available in two forms: a 200 MB/second fibre optic link for the high-resilience market and a direct connection for customers who want multinode systems for performance but as cheaply as possible. It is possible to 'mix-and-match' these options for 3- or 4-node systems.

The input/output has been designed around the use of SMARTfibre, although optional fitting of Macrolan couplers is available. Other interconnects such as FDDI, OSLAN or FiberChannel are available via external gateways.

Significant use has been made of existing products and designs for support and maintenance. A large proportion of the diagnostic support firmware has been carried forward from the Node Support Computer for SX (NSX) to that for SY (NSY) and all the generic parts of the support applications VISA and SAM are unchanged.

1.1 Outline Description

The core unit of the SY node is a back plane into which can be plugged the following PCAs (printed circuit-board assemblies):

1 to 4 Order Code Processors (OCPs)

1 Memory Control Unit (MCU)

1 to 4 Main Store Units (MSUs)

1 Coupler Interface Controller (CIC)

1 to 16 Input Output Daughter Boards (IODB)

1 Node Connection Switch (NCS)

These are interconnected as shown in Figure 1.

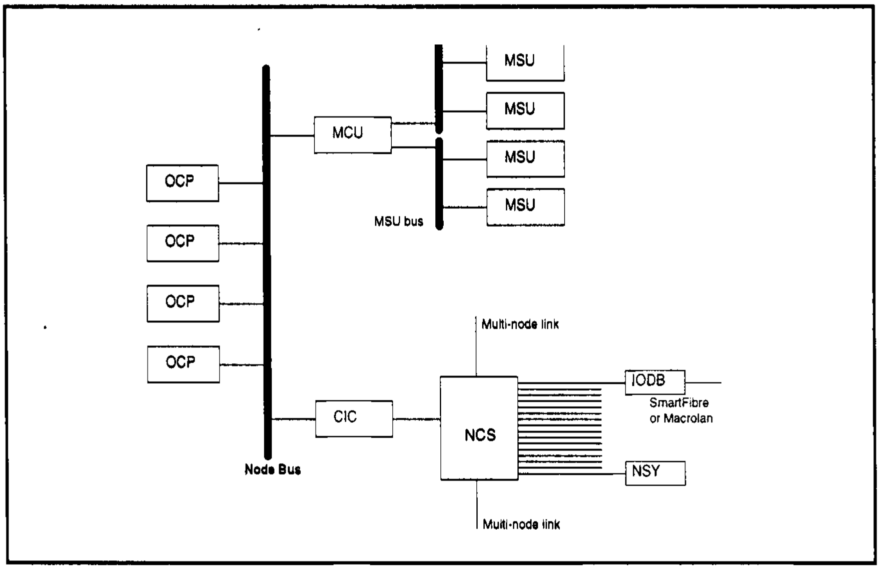


Figure 1: SYNode - principal sub-units

2. Order Code Processing

2.1 Introduction

The OCP plug-in assembly comprises a Series 39 OCP and a 2-level cache structure with an interface to the Node bus, on a single printed circuit board.

The principal task of the SY OCP is to fetch, decode, and execute the Primitive Level Interface (PLI), the target order code of the Series 39 node. A secondary task is to implement the low-level Series 39 architectural features in response to incoming interrupts to PLI processing. As on the previous SX design, these requirements are implemented by converting PLI and interrupts into sequences of a lower level order code, Picode. This section describes the OCP hardware, while subsequent sections describe the Cache structure and Picode. Note that in this document the terms PLI and Picode can refer to the order code or an individual instruction within it, or to a sequence of such instructions.

The SY OCP is based on the design of the SX OCP in order to minimise development costs and design risks. Many improvements over the SX design have been incorporated, to maximise performance and predictability of behaviour. On SX there were two Picode instruction streams (A and B) sharing the same hardware pipeline. The asynchronous interaction of the two streams was difficult to validate, and on SY this feature has been abandoned in favour of a single interrupted stream.

The basic structure of the OCP hardware is shown overleaf in Figure 2.

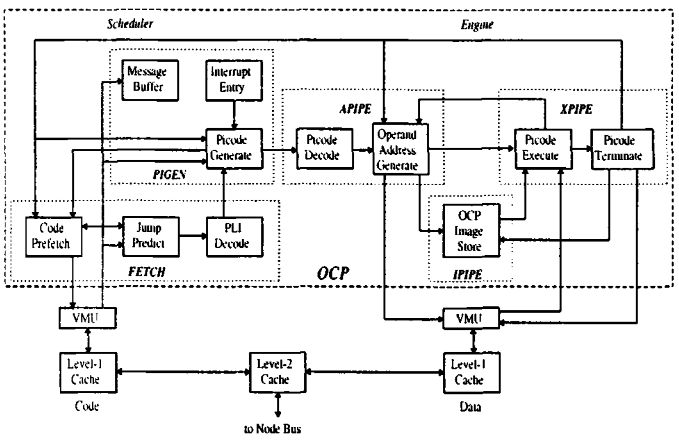


Figure 2: OCP hardware

This structure comprises five separate pipelines, with the following basic functionality:

Fetch	Request of PLI and Picode from the Code Cache, prediction of conditional PLI jumps, and some pre-decoding of PLI
Pigen	Conversion of PLI into Picode sequences, queuing of interrupt sequences, and generation of a stream of Picode instructions for the Engine
Apipe	Picode instruction decoding, and request of operand data from the Data Cache or *Ipipe*
Ipipe	Provides access to OCP-based hardware control registers (Image Store)
Xpipe	Execution of Picode functions, and control of atomic updates to the current process state (registers, the Data Cache, and OCP Image Store).

Hardware interfaces between these pipelines are via various types of multi-line buffer, which allows complete inter-pipeline asynchronism. Inputs to *Fetch*, *Pigen*, and *Ipipe* use FIFOs (first-in-first-out), while *Apipe* and *Xpipe* use Parameter files, a form of random-access register file. Each new Picode instruction is allocated exclusive use of the next parameter file line (slot) by *Apipe*, and keeps that slot through all parameter files until eventual termination. The maximum concurrency of Picode instructions within the Engine is thus 16, the number of slots in these parameter files.

2.2 Normal Picode execution

PLI code is pre-fetched by *Fetch* via the Code Cache, and buffered until required. Incoming code is inspected for predictable jumps, and any found are allowed to modify further code fetches. Each individual PLI is extracted from the buffer, pre-decoded, and passed to Pigen.

In *Pigen*, the PLI is further decoded into a Picode sequence start address and other PLI-related decodes, and checked for any illegal combination of fields. When the previous Picode sequence has ended, this new sequence then starts. Picode is kept in a single segment of virtual store, arranged so that the most frequently accessed code is in the bottom 8k words. Picode addresses in this range get rapid access to their Picode instruction from a fixed store within *Pigen*: Picode outside this range is requested from the Code Cache via Fetch. Picodes subsequent to the first are accessed sequentially or from (Picode) jump destinations until the sequence ends. Picode instructions and related PLI and Picode information are passed to *Apipe*.

Various hardware registers are maintained: some correspond to the PLI Visible Registers, some are private to Picode and others are status flags used internally by the hardware. The definitive copies of these registers are held in *Xpipe*, with the relevant ones updated on successful termination of each Picode instruction. To enhance performance, the registers used in operand address generation have additional copies maintained in the *Apipe*: these can be updated prior to termination where the new data is known.

The *Apipe* accepts each Picode instruction from *Pigen*, allocating it the next cyclic slot number. Each Picode is decoded chronologically, into the actions required by each Engine pipeline. The *Apipe* register copies are used in generating any operand address, which is passed to the Data Cache: read requests will later reply with data and error flags to the *Xpipe*, while for the moment write requests will reply with error flags only. If the operand is within the OCP Hardware Image Store, the address is passed instead to the *Ipipe* which accesses the addressed hardware in a similar manner to the Cache accessing data. If a register to be read by the *Apipe* is not yet valid due to an outstanding write by an elder Picode instruction, the Picode attempting the read will wait until the data is available.

For 'simple' Picode functions involving only registers which are accessible to the *Apipe* and are currently valid, the Picode function is executed here and the result written immediately to the *Apipe* register copy: such functions will later be repeated in the Xpipe. If the necessary input registers are not currently valid or the function is too complex, the register is written later from the *Xpipe* with data from the main execution unit. If the Picode needs no operand fetching, control passes directly from the *Apipe* to the *Xpipe*.

The *Xpipe* is responsible for final execution of the Picode instruction and updating of the process state. The process state is the collection of store data and the PLI register set that defines a breakpoint within a process. This breakpoint can be dumped and restored to allow multiprocessing on a single processor. *Xpipe* processes each instruction chronologically, reading registers and/or operand data then executing the Picode function. Simple Picode functions execute in one beat, whereas more complex functions may take several beats. All functions are implemented directly by hardware for the best possible performance, whereas on SX complex instructions were controlled by a microcode. Meanwhile, error flags are checked for any errors encountered during generation or processing of the Picode. If all is OK, the process state is updated with the results of execution by allowing writes to registers, the Cache or OCP Image Store, and the slot containing the Picode instruction is released for reuse. If not OK, the process state remains unchanged and errors are prioritised into an Exception which is passed to *Pigen*: the (younger) contents of all slots are then nullified, that is they cause no process state update.

More serious errors that indicate illegal Picode, or a corruption of system data, are also collected and inspected at termination time. If a Picode generating such an error attempts to terminate, a "Mayday" is generated, which prevents termination and stops the processor but attempts to ensure that outstanding store writes are completed before initiating a system dump and reload.

2.3 Jumps & Interrupts

Conditional Picode jumps have a condition prediction built in to the Picode function. *Pigen* dynamically detects such jumps and ensures that the next Picode is accessed from the predicted destination. An alternative (non-predicted) destination is calculated and then fetched by *Apipe*. The condition prediction is checked by *Xpipe* and, when wrong, a jump to the alternative destination is issued nullifying all younger activity in the Scheduler and the Engine.

Conditional PLI jumps are predicted by *Fetch*, and subsequent PLI supplied to *Pigen* are from the predicted destination. The condition prediction is dynamic, being derived from the contents of a JPred (Jump Predict) RAM within *Fetch*. This RAM is addressed by a combination of the jump source address and a history of recent jump conditions. The alternative (non-predicted) destination is again calculated/fetched by *Apipe*. The condition prediction is checked by *Xpipe* and, if wrong, a correction jump is immediately issued. This nullifies the OCP, pre-fetches the rest of the current (jumping) Picode sequence and follows this with Picode for the correct next PLI. *Fetch* also updates the JPred RAM to increase future prediction accuracy.

A second major task of *Pigen* is to control Interrupts to PLI Picode. These consist of Exceptions that are errors encountered in generation or execution of specific *Picode* instructions, and Events that represent asynchronous situations requiring Picode support, typically to support low-level architectural features within Series 39. Exceptions are received from *Xpipe*, and Events can be received from anywhere within the OCP. Units external to the OCP requiring Picode support pass a message via the Code Cache to *Pigen*. A non-empty message queue within *Pigen* then generates the necessary event to Picode.

Pending Interrupts are buffered and prioritised, and the chosen Interrupt selects one of several Picode interrupt sequence start addresses. The current Picode sequence is then interrupted by this new sequence, with the new Picode initially preserving the interrupted process state if a later return will be necessary.

A further type of Picode sequence change is a *Warp*. This occurs when a single Picode instruction encounters a difficult descriptor type during execution. *Warps* are signalled to *Pigen* by *Xpipe* and invoke the same hardware as Interrupts (although they are not strictly classed as Interrupts) to cause immediate entry to a Picode sequence to act on the descriptor and complete the PLI Picode sequence that warped.

3. Storage System

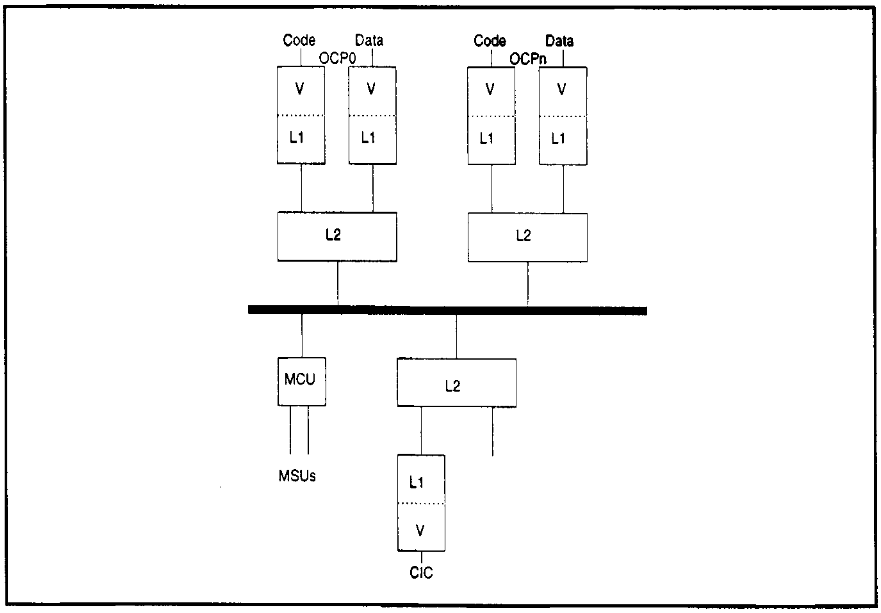


Figure 3: Cache hierarchy

3.1 Cache Hierarchy

Figure 3 shows an outline of the caching hierarchy. For simplicity only

2 OCPs are shown, but more could be fitted. The boxes marked L1 are the first-level cache, and are all identical. When used by an OCP, they form an integral part of the OCP pipeline together with the address translation unit (shown as V on the diagram and known as a VMU). When used by a CIC, the cache and VMU form an autonomous unit. The boxes marked L2 are the second-level cache, and again are all identical. Both level caches reside on the OCP and CIC PCAs. The MCU provides the portal from the Node Bus to the mainstore cards.

Each OCP has two ports into the storage system. The first is through the Code Cache. PLI instructions can be two or four bytes long and the OCP pre-fetches code in 8-byte quanta. After translating a Series 39 virtual address into a real address, the VMU passes the read to the level-1 cache. The second port (Data Cache) is for data and handles fetches and stores of varying lengths up to 8 bytes.

The CIC has only one port: for this reason, one interface of the level-2 cache is left unconnected. The CIC generates stores and fetches on behalf of the IO processors and controls access to multinode data. On this interface data can again be of varying length up to 8 bytes, but also available are 32-byte fetches and stores for IO.

3.2 Cache Coherency

The storage system can be considered to be an array of 32-byte cells. Ownership of a cell moves from cache to cache or mainstore as a client (OCP or CIC) requests access to the data. There are four levels of ownership of a cell, providing distributed coherency control which minimise inter-cache traffic.

Firstly, a cell can be *not valid* in a particular cache. The cache has no knowledge of any data within that cell and must fetch it from somewhere else before it can do anything to that cell. Secondly, a cell may be marked *shared*. In this state the cache's copy of the cell has valid data, but then so may another cache's copy. The cache may return fetch data to a requesting client, but if that client is writing to store, then the cache must ensure that any other copy of the data can no longer be used. The third state is termed *exclusive*, and is used to express that the cache has the only up-to-date copy of the data. This is likely to be the result of a client writing to data within the cell. The final state, which is only used in the level-2 cache, is *inclusive*. This indicates that a formerly *exclusive cell* has been *shared* with another cache, but that mainstore has not been updated.

When a client fetches data from a level-1 cache, the cache must test (or associate) the address. The cache uses two-way set association. In this case a large number of real memory addresses map onto a pair of cache cell addresses, and a test is made whether the real address is currently represented by one of the two cache cells or not. If it is and is valid then the cache can return the data to the client without any further action. If it neither maps nor is valid then a request is made to the level-2 cache to give the level-1 cache shared ownership of the cell.

If the client wants to write to the cell, the level-1 cache must request exclusive ownership of the cell, if it does not already have it. Once a level-1 cache has exclusive ownership it writes to the cell with impunity until another cache requires to access it.

For one cache to acquire or change ownership of a cell, another cache must share or relinquish ownership. To facilitate this, the protocol used is based on a load-give pair of transactions. The request for ownership is called a load, and this will be followed by the give function that grants ownership. A particular cache may not itself be in a position to give a cell, and when it can't it passes the load request on. In the case of a level-2 cache forwarding a level-1 cache's request, the request is broadcast across the node bus to all level-2 caches and mainstore. All level-2 caches then reply with their cell ownership status, and according to such statuses, they and the memory control unit decide from where the data should be given.

Part of the ownership details maintained by a level-2 cache is information on the status within the level-1 caches connected to it. If the level-2 cache decides that the cell is owned by such a level-1 cache then it forwards the load request. Any give response caused by a forwarded load is then returned through the level-2 caches to the originating cache. The subject of cache coherency is reviewed in IEEE Computer [IEEE, 1990].

3.3 Node Bus

The Node Bus is one of the defining features of the SY system. This is a split transaction 128-bit 50 MHz bus connecting all the level-2 caches and the mainstore controller, allowing as many as eight concurrent transactions. Figure 4 above shows a typical split transaction across the node bus. One

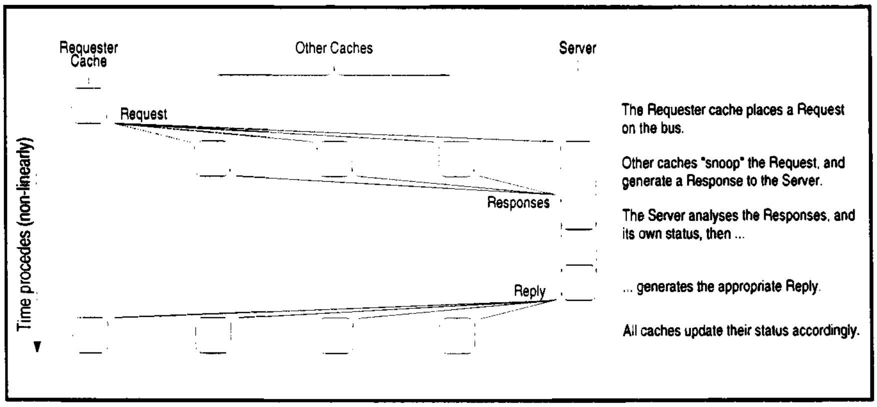

Requester Cache
Other Caches
Server
Request
Responses
Reply
Time procedes (non-linearly)
The Requester cache places a Request on the bus.
Other caches "snoop" the Request, and generate a Response to the Server.
The Server analyses the Responses, and its own status, then ...
... generates the appropriate Reply.
All caches update their status accordingly.

Figure 4: Transaction split across the node bus

level-2 cache, termed the requester, arbitrates a cycle on the Node Bus to request a transaction. All other caches then snoop the request in their association and generate a response. If no cache is able to provide the cell, the MCU then makes a request to the Mainstore and generates a reply with the requested data some unspecified time later. If one cache and only one cache has a valid copy of the data, this server cache generates the reply quickly. If more than one cache has a valid copy of the data, a pre-defined priority is used to identify the cache that provides the data.

A special technology was bought from Fujitsu in which to implement the bus as all the physically separate PCAs are mounted on the motherboard.

In addition to cache coherency traffic, the node-bus supports the passing of messages between clients. A message consists of source and destination address information, function qualifier and up to 16 bytes of data. Messages can be created by client Picode or microcode and are directed at other clients, or to the store system for low-level control. Messages will also be automatically created by a VMU to support multinode activity.

3.4 Protocol validation

The development framework provided us with the capability of defining the protocol used between the caches at the architectural design stage. The language then provided automatic checking that these protocols were continually adhered to. We then had the impetus to capture this checking into the detailed hardware implementation. This allowed us to define checks such as: "Why are you requesting this cell, you already own it exclusively?" or "Why are you asking me to give you this cell, I don't have it?".

These checks provide an immediate indication that the coherency of the caches has been corrupted. This will prevent such a problem, maybe caused by a design or maintenance fault, from resulting in a corruption of system or user data.

3.5 Virtual Memory Management

As indicated above there is a Virtual Memory Management Unit (or VMU) between every level-1 cache and its client processor. Generally, a processor accesses store via its virtual address, but the OpenVME architecture accepts that the same physical store location may be represented by a number of virtual addresses. Also in a multinode system one virtual address may have physical store in more than one node. The main task of the VMU is to handle virtual to real address translation and to detect writes to store in other nodes so that the other nodes get informed.

Any function executed by the VMU is split into at least two passes, a primary (or read) pass and a secondary (or write) pass. On completion of a primary pass, information about the translation is returned to the client together with any read data. The client than decides whether or not to complete the function and how to complete it and instigates the secondary pass.

The core of the VMU is a set of Address Translation (ATU) slaves. These contain the most recently accessed entries in the three sets of translation tables maintained by OpenVME. On a pass through the VMU, the address is associated with the ATU slaves and if all the necessary information is available, allows the pass to complete. If the pass is primary, the translation information is returned to the client and if the function involves reading, a read request is issued to the level-1 cache. If the pass is secondary, a write request may be made to the cache.

If the slaves do not have the required translation data, a read for the table entry is made to the cache, and the function is re-queued. When a function is waiting for table data all subsequent functions are only tentatively executed. Any requests for table data are made, but reads are not actioned as the client expects reads to be executed in order. When the data for a table entry returns from the level-1 cache the queue of instructions is restarted and the function attempted again. If the function was waiting for this particular data, then it will proceed to the next stage of the translation sequence, otherwise it will simply re-queue itself. In a similar manner, a function will be re-queued if for any reason it cannot complete the pass through the VMU. The particular wait condition is remembered and this defines the trigger to restart the queue.

In Series 39 multinode architecture, certain virtual addresses are marked broadcast. This means that a write to the location must also be performed in the other nodes in the system. For such an address, when it is first translated in the primary pass, a flag is set to indicate this. After the secondary pass, a third pass is then scheduled which generates a message that is sent to the CIC. On receipt of this message the CIC passes the write into the multinode network.

The PLI provides a set of store-to-store instructions. These are limited by the OCP dataflow width of 64-bits. However, the VMU has the capability of trapping a sequence of consecutive 8-byte writes and accumulating them into a single 32-byte cell write. This cell write can then be processed by the cache hierarchy more efficiently than the individual writes.

4. Input/output Processing

The SY I/O system, like that for SX, is designed to provide up to 16 fibre-optic LAN connections which may be any combination of SMARTfibre or Macrolan interfaces, subject to a maximum of 12 Macrolans. This allows the majority of existing SX system users to "drop in" the SY node with no changes to their I/O sub-systems. Support for "slow devices" remains unchanged, provided through gateways.

Connection to the LAN is by means of an "I/O Daughter Board" (IODB) which is configured, by means of a "switch", to interface to either SMARTfibre or Macrolan. Each of the IODBs is connected via an asynchronous, full duplex byte interface to the "Node Connection Switch" (NCS) which, for the I/O system, "multiplexes or de-multiplexes" these sixteen interfaces to a single full duplex word interface to the "Coupler Interface Controller" (CIC). The CIC then provides access to the node bus via the level-1 & level-2 caches. (See Figure 1)

Further information for both the CIC & NCS can be found in Section 6: Multinode processing.

As on previous Series 39 machines the OCP supports the OpenVME operating system by providing high level functionality for I/O driving. For SX systems this was provided by special stream B Picode. Since this mechanism no longer exists for SY systems, this functionality has been exported to the IODBs and is now provided by a SPARC based microprocessor optimised for use in embedded systems (Fujitsu SPARClite MB86934) executing a microcode of approximately 64 Kbytes, written in C, and developed specifically for this purpose.
The major hardware components of the IODB are shown in Figure 5 and

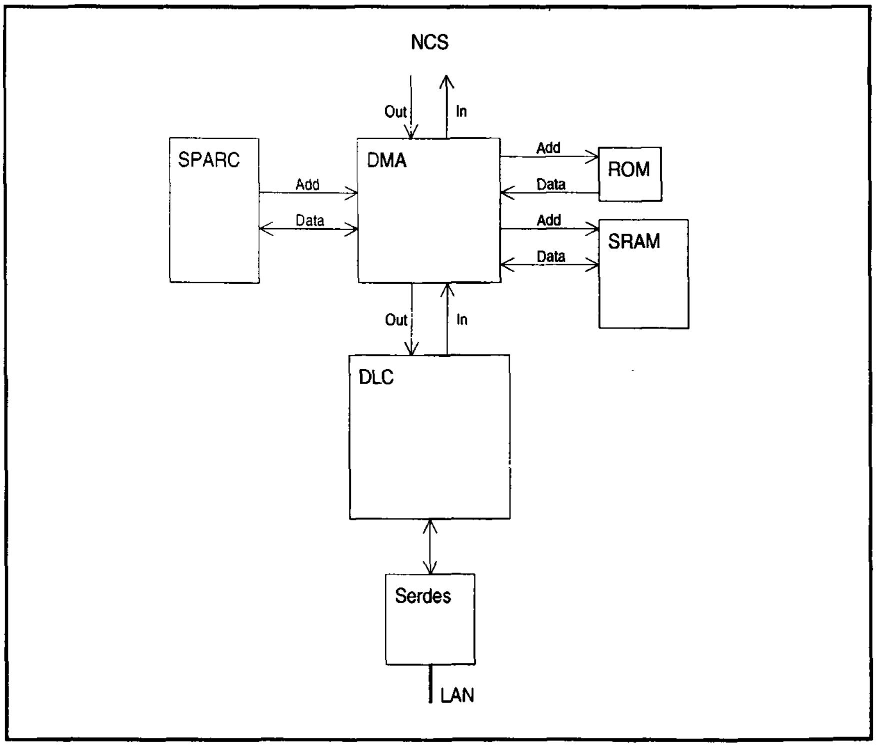


Figure 5: Input/output system

defined below:

SPARClite microprocessor (referred to above)

> Executes microcode to provide the high level functionality required by the OpenVME operating system for I/O driving. Through its interface with the DMA chip (see below) it fully controls the operation of the IODB.

ROM

> Contains the initial SPARClite microcode providing support for initial establishment, loading of the operational microcode and dumping of the contents of the SRAM for diagnostic analysis.

Static RAM

> One Mbyte of Static RAM containing the microcode and buffers for data and control information. The control buffers include the OpenVME architectural Cyclic Output Buffer (COB) for Coupler Information Frames (CIFs), buffers containing internally generated CIFs, buffers for support of Data Phase such as the Assisted Stream Tables and various Page Table buffers. For data, a 256 Kbyte cyclic input buffer, the CIB, and a 16 Kbyte cyclic buffer for output data.

DLC and SERDES Chips

> Used, as on SX systems, to interface, through "updated" Optical transceivers, to the optical fibre LAN.

DMA Chip

> A new CMOS gate array chip designed and developed specifically for SY providing interfaces to the NCS and all other major components of the IODB (SPARClite, SRAM, ROM and DLC). The major functionality of the chip is to provide DMA operations between main store and the buffers in SRAM and to support the transfer of control information between the SPARClite and the OCPs.

The IODB includes comprehensive integrity checking of all data paths, RAMs and control sequences.

A major new feature of the SY I/O system is improved management of "failing" IODBs. The capability to "Hot-Pull" boards allows those with a hard fault to be replaced, reinitialised and configured back into the system without the need to power off or re-IPL.

5. Multinode Processing

Multinode processing on SY is standard Series 39, i.e. a number of nodes are connected together to form a single coherent image of the OpenVME operating system. These nodes share virtual store by replicating that storage on different nodes.

Any writes to shared store are broadcast to other nodes in the system. This ensures the coherence of replicated data throughout the system. Another important requirement is that all nodes must see the public writes applied in the same order—hence it's crucial that there is a mechanism for handling this (see below).

The multinode connections on SX systems require 4 Macrolan and 2 TSN connections per node. In order to scale up to the performance requirements on SY systems—internode traffic increases as node processing power increases—the same connections would have to be increased by up to 16 Macrolans per node. The cost of providing these connections would be disproportionately high compared to the rest of the components in the node. A different approach had to be sought and the resultant design is described below.

5.1 SY Internode Processor INP

The SY INP is the point of communication between nodes in a SY system which can be between one and four nodes. The distance between the nodes can be as short as 20 cm, e.g. between adjacent cabinets, or as long as 2 km. Nodes are connected as two concentric rings (see Figure 6)—unlike SX which requires central boxes and TSN token rings. Communication between the INP and the OCP is done via the Node Bus in the form of 'messages'.

There are 2 main units within the INP, namely the Coupler Interface

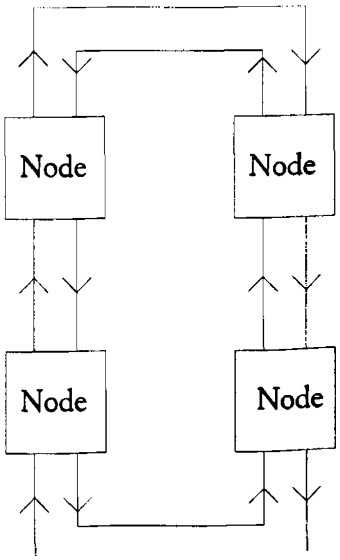


Figure 6: Internode connection

Controller that interfaces to the Node Bus via the level-1 and level-2 caches and the Node Connection Switch that interfaces to the 'ring', IO couplers and the Node Support Computer.

The main advantage of the INP is that it takes the load off the OCPs in dealing with internode and IO traffic so the OCPs can concentrate on their main task of PLI execution.

5.2 Coupler Interface Controller CIC

The CIC unit is used to transmit outgoing internode, IO and Support System messages, via the NCS, to the public write network as well as the IO sub-system.

It also receives incoming internode, IO and Support System messages from the NCS, processes them, performs store accesses on behalf of clients and passes messages to the OCP. These functions are best served by a microcoded engine. It breaks down complex actions into simple functions which can then be transmitted to the storage system.

Another important feature of this unit is the realisation of the node's Real-Time Clock that is held and maintained in the CIC.

5.3 Node Connection Switch NCS

The NCS is essentially a crossbar switch that provides the connectivity to the other nodes in the system and is also the main gateway to the IO couplers (up to 16 IO and an additional interface for the DCM/NSY). This latter interface provides the operational connection of the NSY to the OCP and Mainstore units.

Although the nodes are connected in a 'ring' fashion, the actual path for a broadcast action is taken around a so-called 'hairpin' route. See Figure 7. When a message is to be sent out of the node, it goes up the generate-links and traverses all nodes until it hits the 'top' node which is defined as a synchronisation point for all broadcast actions, i.e. writes to store in all nodes are applied in the order they pass through the 'top' node. When a message reaches the 'top' node, it is 'turned round' and a bit in the message is set indicating that it can now be 'applied' in all nodes. After a message has been made applicable, it is sent down the 'apply-links' and as it passes each node, a copy of the message is sent to the CIC to be applied in each node. The message continues down the 'apply-link' until its reaches the 'bottom' node, passes to the CIC and is then subsequently discarded.

5.4 IO data route

IO traffic is routed through the NCS for a given node as shown in Figure 8. It effectively uses the same crossbar switch for routing the IO data through. A special feature of the IO interfaces is that the NCS is resilient to IO coupler failures and faulty couplers can be configured out without affecting

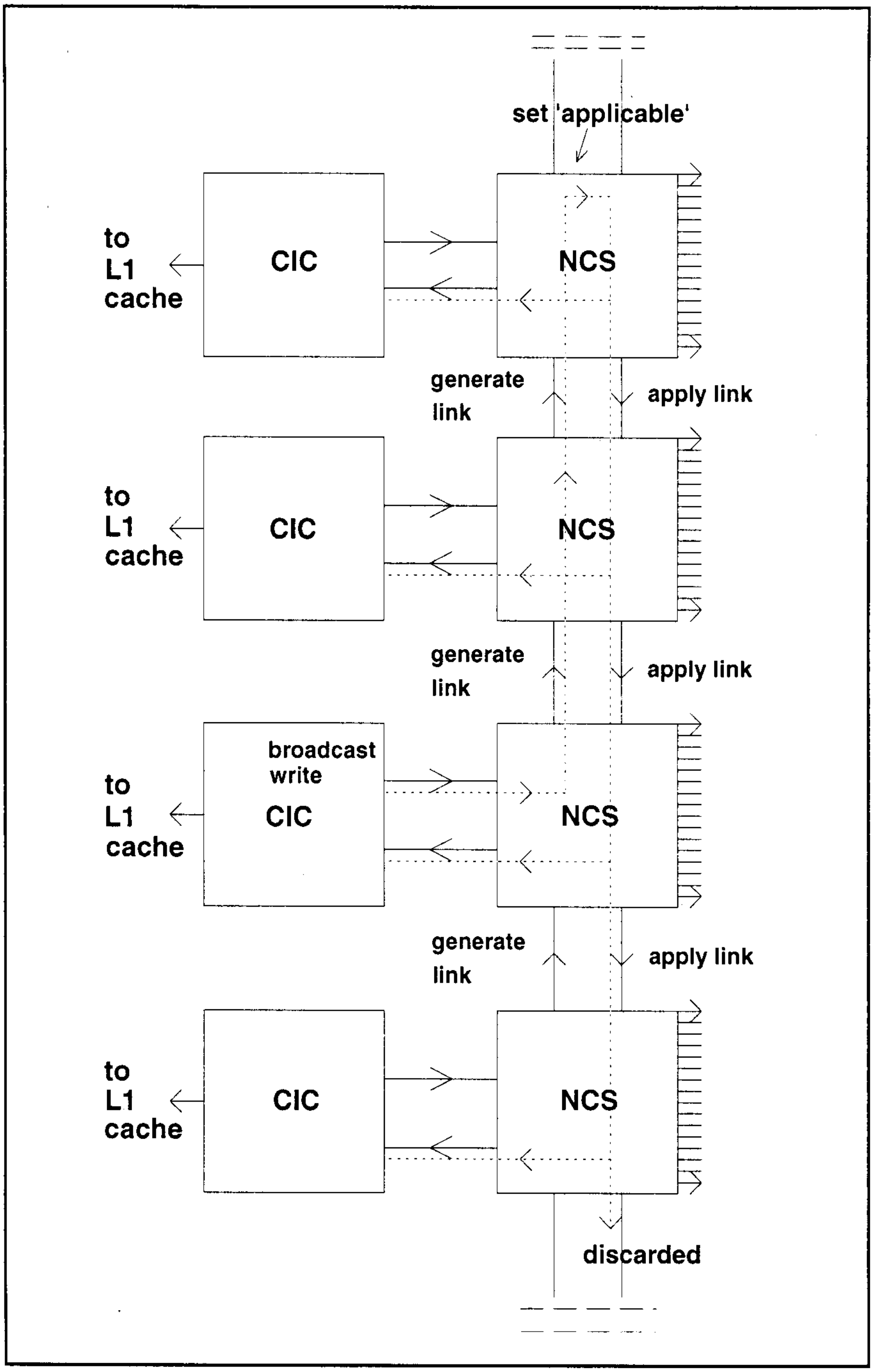


Figure 7: Node connection switch

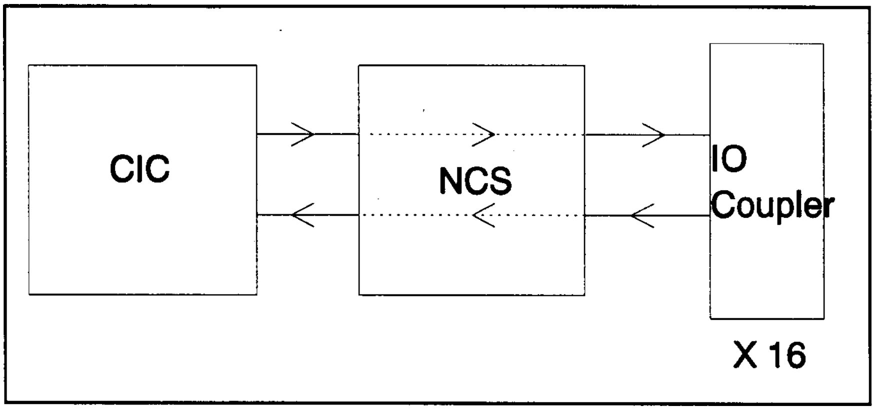


Figure 8: I/O traffic routing

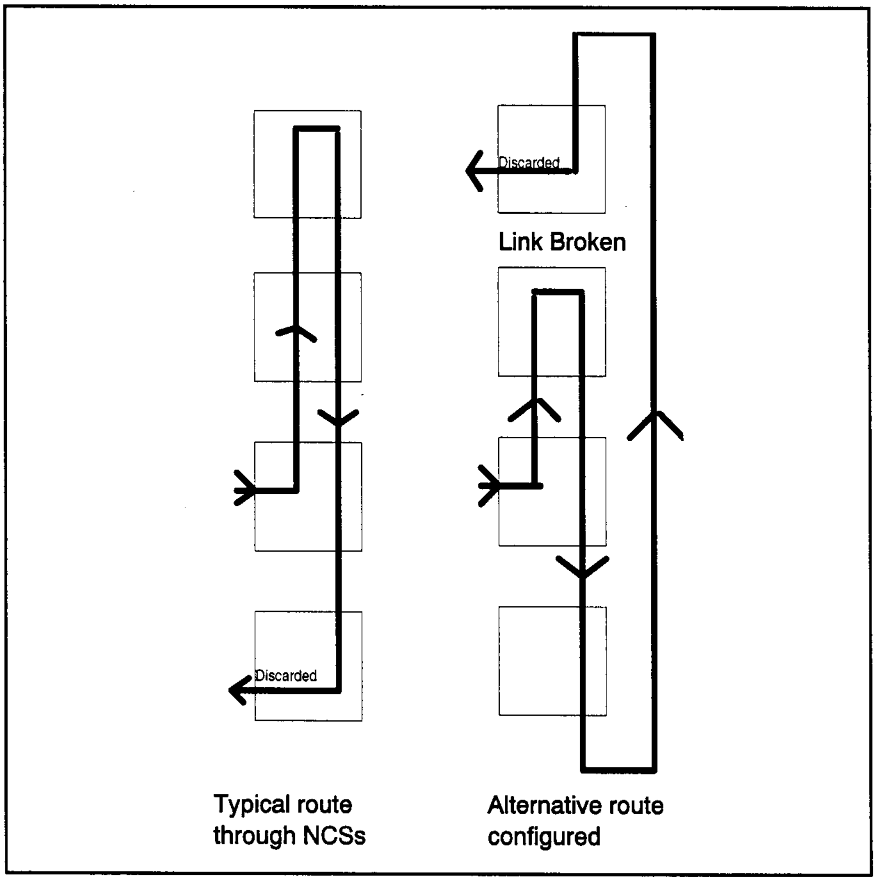


Figure 9: Node connection re-routing

the system. Hot-swap of any IO couplers is supported.

If a link malfunctions, then it is possible to establish a multi-node system avoiding this broken link; see Figure 9. For example, in this diagram, the left hand picture shows a fully functional connection and the public write goes up the stack of nodes to the top node. It then gets marked 'applicable' and sent back down the stack to the bottom node to be discarded. The broken case is shown on the right. The public write cannot traverse the broken link, so it is turned round at the second from top node, made 'applicable' and sent back to the bottom node. From here it is sent down (and back round) to the top node where it is discarded.

5.5 Copper versus Optical connections

If there is no requirement for physical separation of the nodes, they may be connected together with copper interconnect. In this case, the cabinets must be bolted together and the internode cabling is placed within a duct between the two nodes. This provides a single Faraday cage for the nodes that allows us to meet the EMC requirements and reduces ground potential differences between the nodes. Up to 4 nodes may be connected this way with the maximum cable length of 5 m.

If a synchronous-interface approach is adopted on copper connections, then a means of distributing a system clock across up to 4 nodes is required. Whilst this is probably feasible, the method adopted on SY is simpler and removes the need for Phase-locked Loop Oscillators or highly sophisticated and expensive clock tuning engines. The scheme adopted on SY is to send together with the data/control highway a 'travelling' clock that is set roughly two-thirds of the way through the clock cycle time. In this way, transmission line reflections and cross-talk are allowed to die down before valid data is strobed into a FIFO. Reading of the FIFO occurs in a different clock regime, so special circuitry, known as "de-boggling" circuitry, is required to change the clock speed for the data. The interface is thus regarded as asynchronous.

This principle is further applied to the interfaces between NCS and all the IO couplers, and between NCS and the CIC. The advantage is that the OCP/CIC, NCS and the IO couplers (including NSY) can then operate under independent clock-regimes. It allows the basic clock cycle time to be different on these sub-systems, allowing appropriate technology to be chosen for each sub-system.

To support separations where the nodes are not physically adjacent, such as disaster-tolerant systems, an optical interface is used. This solves the problem of widely separated copper interfaces where the logic ground voltage difference would render it unusable.

To meet the bandwidth requirements of a full 4 node SY system, 2 high-bandwidth optical transceivers operating at 1 Gbit/second per link are used.

6. Picode

Picode is the machine code that is obeyed by each OCP on an SY system (see Figure 10). Although the SY Picode is not binary compatible with SX Picode and individual Picode instructions have slightly differing effects on the hardware state, there is considerable compatibility in their effects on the process state. For this reason a large amount of SX Picode source was reused on SY.

Picode is responsible for the following:

- providing the "Primitive Level Interface" to the overlying software
- resolution of program errors
- supporting the Series 39 architecture, scheduling processes and interrupts
- providing fast I/O support
- supporting internode establishment, disestablishment and error reporting
- supporting the interface to the node support computer for SY (NSY)

Of the above, some of the complex arithmetic instruction routines, all of the resolution of program errors, some of the I/O support and some of the interface to the node support computer for SY were reused from the SX Picode.

SX had two Picode instruction streams (A and B) running interleaved on a single OCP. On SY a significant amount of the stream B functionality has been moved to other hardware. Also the SY design provides up to 4 OCPs accessing a single store image. Picode exploits this by assigning each OCP to a software process and running up to four processes in parallel.

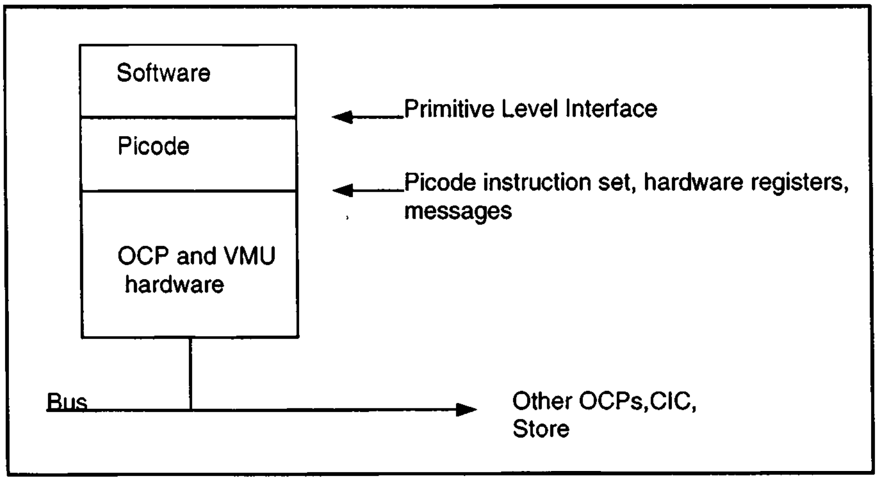

Software
Picode
OCP and VMU hardware
Primitive Level Interface
Picode instruction set, hardware registers, messages
Bus
Other OCPs,CIC, Store

Figure 10: Position of Picode within an SY node

6.1 Multiple OCPs

On earlier Series 39 machines OpenVME's process scheduling algorithm causes one process to be nominated and entered into a software table called the "Node Environment Table" (see Figure 11). On SY the Picode itself can perform process scheduling in order to nominate up to four processes to run concurrently. The Picode, then, has two modes of operation: "Software Process Scheduling Mode", where the software performs process scheduling and enters the nominated process into the node environment table; and "OCP Process Scheduling Mode", where the Picode performs the process scheduling. In software process scheduling mode only one OCP is used, however many are fitted, and this mode is used by the OpenVME loader.

To enable OCP process scheduling mode an extra field has been defined in the node environment table. This is known as the "Process Priority Table Pointer" and holds the address of OpenVME's process priority table.

The SY Picode takes over the management of the process priority table. New primitive procedures are provided to allow software to add a process to the table, remove a process and change a process's priority within the table.

Any time a process priority table is defined or altered, Picode performs a selection algorithm in which a process is selected for each available OCP. The selection algorithm will favour processes of a higher priority and will also try to keep processes on the same OCPs as they have run on before to minimise the switching of local data between OCPs' caches.

A process that needs to perform system-wide tasks may need to run "Preemptively" indicated by causing the "PEP" bit in the program status register to be set. A preemptive thread, once started, must finish before another preemptive thread at the same or lower privilege can start (although a higher privilege preemptive thread may still interrupt it). When calling any of the new primitive procedures to manipulate the process priority table the software process must be pre-emptive.

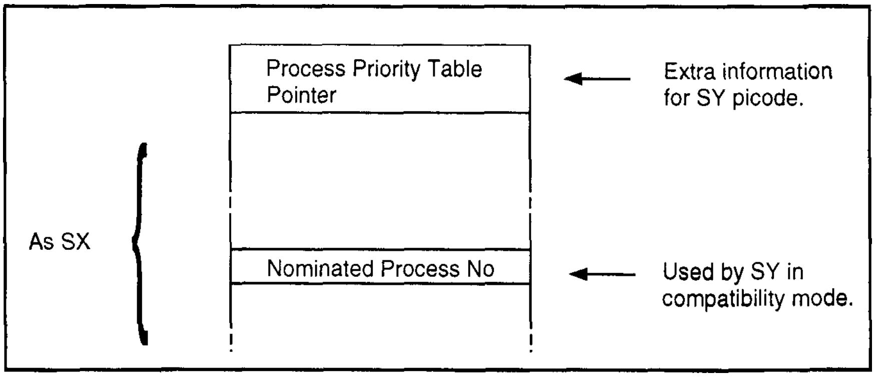

Process Priority Table Pointer
Extra information for SY picode.
As SX
Nominated Process No
Used by SY in compatibility mode.

Figure 11: SY node environment table

After changing the process priority table, when software ends the preemptive thread, a "Nomination Point" occurs. This is where the Picode nominates a process for running on each available OCP. These will be the processes that have been selected by the selection algorithm mentioned above. Note that no change of the node environment table occurs, the Picode keeps its own table of selected and nominated processes.

When software decreases its privilege, when an interrupt occurs or when an interrupt thread finishes the Picode performs a schedule. To schedule, the Picode must select processes to run on each OCP. This Picode is currently running on any OCP. In general, these will be the nominated processes but there may also be interrupts to run, or preemptive processes to complete. If it selects an interrupt to run, this may need to be run on the current OCP or another one. If it needs to run on the current OCP then the interrupt is activated. Otherwise a message is sent to the chosen OCP for the Picode on that OCP to also perform a schedule. If no interrupt is selected, but the scheduling Picode finds that an OCP is no longer running the correct process, that OCP is sent a message telling it to perform a schedule. When the second OCP performs its schedule, it may agree with the first OCP's decision, in which case it goes ahead and changes its process or runs the interrupt. Alternatively it may find either a further discrepancy between the processes being run and those selected or it may find that the list of selected processes and interrupts has changed. In this case it sends another message to ask an OCP to perform a schedule. This continues until no further discrepancy is found. The scheduling algorithm is designed to ensure that at all times each process is running on at most one OCP and that at most only one OCP is preemptive.

6.2 Multinode

On SX the handling of writes to store generated on another node (public writes) was handled by the Stream B Picode. On SY, public writes and semaphores are handled by hardware, with a message sent from the writing processor's VMU to the CIC, then sent to the CIC on the other node and applied to store there. A public real time clock is also provided by the CIC which the Picode can read by sending a message to the CIC. Because the internode hardware is substantially different, the largest part of the internode Picode on SY is to effect internode establishment or disestablishment.

To establish a public write service the Picode will cause all the links between the required nodes to be connected. Note that the nodes are connected together in a chain and a multinode service can only be established between linked nodes. Each link can be either copper, for nodes that are adjacent, or optical for nodes that are separated. Nodes can be connected into a full circle but only a hairpin is required for a multinode service, so under this circumstance Picode will choose not to use one of the links. This is a resilience feature as if a link should fail this will obviously be the link the Picode will choose not to use.

On receiving the establish request from software the Picode chooses a node to be the master node. The Picode in this master node will then grope along the links to find the nodes required for the requested multinode system. As each node is groped it is connected. When the full circle is connected, the Picode then chooses a link to disconnect in order to form the hairpin. This choice is based on two things. Firstly, if the previous multinode service terminated because of a link failure then the link that failed before will be the one chosen to disconnect. Otherwise, the Picode will choose to disconnect an optical link in preference to a copper link. This provides the best performance as an optical link is likely to be joining nodes that are some distance apart. When Picode disconnects an optical link, it will send the link a message to initiate "health check mode" in which the link constantly checks its own state and causes fail information to be sent to OpenVME should any fail be detected.

6.3 I/O Support

On SX I/O support processing was performed by Stream B Picode. Most of this processing has now been exported to processors on the I/O couplers. There is a requirement as on SX, for SY Picode to support fast path I/O sequences. To gain a further performance advantage the multiple OCP aspect of SY is exploited by allowing OCPs to perform I/O fast path sequences in parallel. In order to do this each coupler is set up so that it can individually message one specific OCP, this way two OCPs operating in parallel are guaranteed to be driving different couplers and so no interleaving of messages can take place. See Figure 12.

6.4 Simulation

There were significant innovations in the simulation of SY Picode before the hardware was switched on. Software was written to interface the Picode test bed with the VISA test bed and the Picode unit driver could therefore be tried with the actual Picode before hardware availability.

Because the establishment and disestablishment Picode were written in C, the full establish and disestablish sequences could be tried on a standard UNIX machine, through the use of test harnesses. Such use of C provided a significant improvement in code writing.

Most significantly, OpenVME was run on the Picode test bed up to the point where it required an 'oper' terminal. This is the first time OpenVME has been run in simulation of a new processor design before the hardware was available. This had some major advantages: Picode faults could be removed before hardware availability; faults in the OpenVME changes for SY could also be removed; experience could be gained on how OpenVME behaved in a multi-OCP environment and getting OpenVME to the 'oper' on the real hardware was made easier due to the fact that it had already been done on the Picode test bed. The achievement of this involved enhancing the Picode test bed to allow page discards and the development of software to emulate an IO processor and pass on commands from the Picode test bed to a real disk. This software is now being used on future ICL developments.

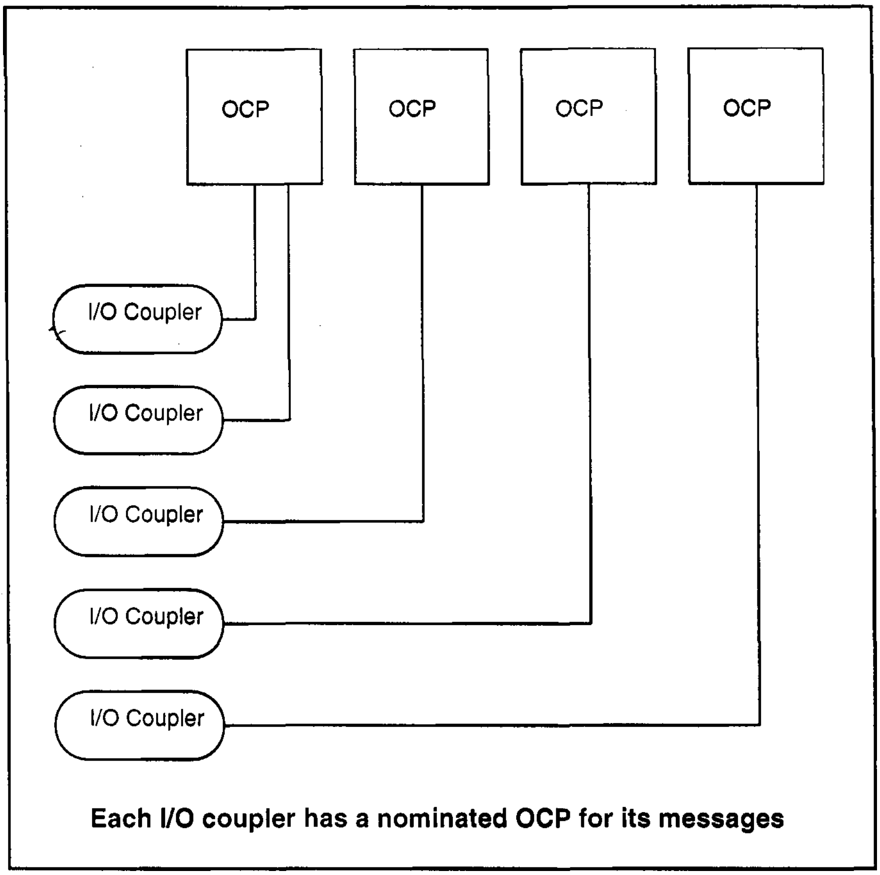


Figure 12: I/O coupler–OCP configuration

7. Performance monitoring & modelling

Performance modelling has played an increasingly important role in all Series 39 system developments as a tool for both performance prediction and design option assessment. For design option assessment the speed of response is more important than the absolute accuracy of result, for performance prediction the reverse is true.

SX project carried performance modelling to new levels of detail by modelling both the processor pipeline and internode connect in significant detail driving the models from detailed traces obtained from both real workloads and ICL specific benchmarks. These workloads may have been running for many hours in order to achieve the desired state before a trace is taken.

The SY project has modelled the same as the SX project but has extended it to include the modelling of the system behaviour with the aim of further improving the accuracy of performance predictions.

7.1 Modelling Approach

Ideally the same behavioural models used to validate the SY design would have been used to predict the performance. Regrettably the time taken for a behavioural model to run is orders of magnitude greater than for a performance model. Further the behavioural models "execute": it would have been impossible to run a behavioural model to the point where it could run the operating system or a real workload.

An alternative adopted by the SX and SY projects is to build Discrete Event simulations models which abstract the behaviour of the system such that the model does not "execute" but simulates with the same timing as the behavioural model. Realistically it is impossible to achieve identical timings in all cases without re-implementing the behavioural models so engineering judgement must be applied to ensure the largest contributors to overall performance are accurate.

The SY performance modelling relies on detailed traces gathered from running real workloads on current machines. These traces detail the exact state for the instruction sequence being executed by a workload. The state includes the value of all registers relevant to the instruction execution but does not include the state of the store. Clearly this information gathering process imposes a significant performance penalty.

The traces can be used directly to drive performance models but they can also be analysed to determine a detailed profile as to what the workload does at a more abstract level—the frequency of public writes, the frequency of access level changes, etc. The information used by a model is dependent on the objectives of the model.

Further information as to the system level behaviour has also been gathered both from the OpenVME system logs and from special traces that minimise the disturbance of the workloads.

7.2 Performance Models

Prior to SY performance modelling tended to be done using S39 systems. All performance modelling for SY has been done using Sun workstations and a performance modelling package written in C^{++}.

The performance modelling package used on SY has been derived from a discrete event modelling package for the Simula language provided by STL (of which ICL was part at the time). The Simula package was used for some preliminary pipeline investigations but was found to be too slow and hence was converted to C^{++} and further extended for our specific needs. This package has been used for all SY performance modelling and has been used by other groups within HPS.

For SY the basic modelling package has been used to support three major models—a pipeline model, a multi-node interconnect model and a system model.

The pipeline model is a highly detailed model of the OCP pipeline and storage hierarchy. It is driven using workload traces taken from previous S39 machines. It has been developed as a highly configurable model which allows many different options to be modelled without the need for new code to be written. This is achieved by constructing a set of stages which interconnect using queues, emulating the behaviour of the real pipeline. These stages can be general or specialised but all follow the same pattern: they take information from their input, perform some function on it, maintain internal state and pass the modified information to their outputs. The model is configured by means of a configuration file which allows queues and stages to be linked together. Various parameters are applied to each stage in order to specialise it. An example of this is cache size. Another file provides parameters that specify the cost functions that govern the time taken for various functions by the stage. This approach allows many options to be considered with minimum need for code to be written at a cost of performance. Where code is needed, such as when new functionality is identified, a new stage can be created as a derivative of an existing stage. The new functionality is added (plus any parameter handling code) and the stage is then available for general use.

The SY node comprises between one and four processors. In its simplest form the pipeline model only models one processor, its caches and the storage system. In order to determine the impact of multiple processors competing for resources within the storage system and to determine the impact of cache invalidation it is possible to configure more than one pipeline BUT this costs significantly more in run time than modelling a single pipeline. In order to determine the performance of multiple processor systems efficiently the events which cause interaction are extracted from a single pipeline model as it runs and fed to a load generator, configurable with up to 4 processors, which presents them to the storage hierarchy after some time displacement. This simulates both the loading of the storage hierarchy by other pipelines and the interaction of specific address access on the caches.

The multi-node model follows a similar pattern to the pipeline mode in that it is configurable and obtains its performance characteristics from input files but rather than being trace driven it is profile driven, the profile being derived from an analysis of the workload traces. This allows the ratio and relationship between events to be changed without the need to generate complex detailed traces. This gives the possibility of creating artificial workloads which will drive the system harder than any real workload and to create profiles for workloads which it is not possible to monitor (see Figure 13).

The system model is a new model for SY which models the detailed system behaviour below the operating system level but above the hardware level—the level implemented by the Picode. It is a simple performance model with the functionality built directly into the model but with timing and workload information taken from a parameter file.

On previous systems this level has been largely ignored as, although its impact on performance was believed significant, the implementation options were severely limited. On SY there has been a radical change to this architectural level brought about by the support of multiple processors within a single node, which has meant a need for greater understanding in this area. In this model the individual processors are modelled running a configurable workload—defined largely in terms of the rates of system call, instruction execution, I/O rates and pre-emptive time. The systems reaction to these events has been captured at an abstract level and the costs of various functions are input as parameters. The model simulates the workload including all messages between processors and the contention for various system resources.

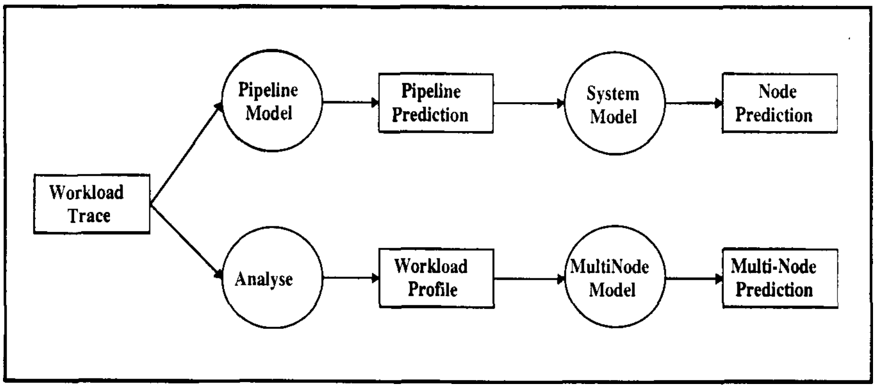


Figure 13: SY performance models

The models are used for two main purposes—design analysis and performance prediction. For design analysis it is the relative difference between two options that is important as it allows the designer to assess the options. Numerous options have been considered during the project life and all models have been used for this purpose. For performance prediction it is the overall performance which is important. For this the three models are combined—the results of one model feeding another as follows

Currently the SY system is undergoing extensive performance trials. Many of the predictions from the performance modelling are being achieved or exceeded

8. Conclusions

SY, like its predecessor SX, has attained the status of being the most complex and powerful machine ever built by ICL. It fully satisfies all the requirements of our customer base in that it provides high performance and high reliability for our top-end customers whilst being capable of being scaled down to satisfy the needs of even our smallest customers at a competitive cost of ownership. The challenge has been immense for both Fujitsu, in providing the state-of-the-art VLSIs to ICL's designs and the ICL designers and implementers who have used to the utmost their intellect, knowledge and skills to make SY a reality.

Glossary

ATU	Address Translation Unit. Hardware which can convert a virtual address to a real address.
Apipe	Picode instruction decoding and operand request part of the OCP.
CIB	Cyclic Input Buffer. Storage space for data arriving from peripheral devices.
CIC	Coupler Interface Controller. Hardware which transfers messages between OCPs and IO and internode couplers.
CIF	Coupler Information Frame. "Commands" sent to the IODB from picode/OpenVME to control the IO operation.
COB	Cyclic Output Buffer. Storage space for data being sent to peripheral devices.
CMOS	Complementary Metal Oxide Semiconductor. High density microchip technology used on SY systems.
Dataflow	Transference of data between registers in the hardware.
DCM	Diagnostic Control Module. Hardware which allows direct access by the node support computer to the SY registers.
Disestablishment	Termination of the internode service causing all nodes in the service to return to being single nodes.

DLC	Data Link Chip.
DMA	Direct Memory Access. Mechanism by which a coupler can write data to the SY system by directly writing into the SY main memory.
ECL	Emitter Coupled Logic. High speed microchip technology which was used for SX.
Establishment	Connection of many nodes together to form one multi-node system.
FDDI	Fibre Distributed Data Interface. An industry standard interface for transferring data.
Fetch	Part of the OCP hardware which fetches and decodes PLI instructions.
FIFO	First in-first out. A buffer which gives back data in the order in which it was stored.
FiberChannel	An upcoming standard high speed optical fibre and protocol for passing data along it.
Gbit	10^9 bits.
HPS	High Performance Systems.
ICL	International Computers Limited plc.
INP	Internode Processor. Hardware to send and receive messages along a fibre optic cable.
Internode	Connection between SY nodes to form a multi-node system.
IO	Input/Output. The action of data being transferred to or from a peripheral device such as a terminal, disc drive or printer.
IODB	IO Daughter Board. Or IO coupler, providing the connection between the node and SMARTfibre or Macrolan.
IPL	Initial Program Load. The act of loading the SY hardware to allow it to start running OpenVME.
Ipipe	Part of the OCP hardware to provide access to OCP based hardware control registers.
JPred	Jump Predict. The jump prediction mechanism.
Kbyte	1,024 bytes.
Mainstore	The computer's memory.
MCU	Memory Control Unit. Hardware allowing access to mainstore.
Microcode	Data which is loaded into a hardware RAM which controls the actions of the hardware.

MSU	A unit of mainstore.
Macrolan	ICL's optical fibre, local area network, connection for peripheral devices.
Mbit	2^{20} bits.
Mbyte	2^{20} bytes.
Multinode	Many nodes connected together to run one SY system. This is done for increased performance and increased resilience over a single node system.
NCS	Node Connection Switch. Hardware to allow information to be transferred between nodes in a multinode system.
NSX	Node Support Computer for SX. A small computer which monitors and provides general support for an SX node.
NSY	Node Support Computer for SY. A small computer which monitors and provides general support for an SY node.
OCP	Order Code Processor. Central processing unit of the SY node which executes PLI instructions and services interrupts.
OSLAN	A 10 Mbit/second coaxial cable communications link and protocol.
OpenVME	ICL's operating system.
Oper	An OpenVME session which is specifically used by a systems operator. This will, in general, be the first session which is made available when an OpenVME service is started.
PCA	Printed Circuit Assembly. Comprising a board and components.
PLI	Primitive Level Interface. The standard interface to the hardware seen by OpenVME applications.
Picode	The machine code which is obeyed by the SY processor.
Pigen	Hardware within the OCP which converts PLI into picode sequences and queues interrupt sequences.
RAM	Random Access Memory. Computer memory in which data can be read and written by supplying an address.
SAM	The OpenVME support and maintenance server.
SERDES	A chip that acts as a serialiser/deserialiser.
SMARTfibre	ICL's high speed optical fibre connection for peripheral

	devices.
SMP	Symmetric Multiprocessor. Comprising many processors using a common store.
SPARC	A specific machine code language.
SPARClite	An off-the-shelf microprocessor which obeys SPARC machine code, optimised for use in embedded systems.
STL	STC Laboratories. Research division of Standard Telephones and Cables.
SX	A generation of ICL series 39 mainframe processors.
SY	The latest generation of ICL series 39 mainframe processors.
Simula	An object oriented programming language used for simulation.
TSN	Token Serial Number. The mechanism by which messages in a multinode system are applied in the same order in each node to give a consistent store image.
VISA	VME Inoperative System Access. This enables diagnostic access to be made to a node.
VLSI	Very Large Scale Integration.
VMU	Virtual Memory Unit. Hardware which translates a virtual address and provides access to store.
Xpipe	Part of the OCP hardware which performs the execution phase of a picode instruction.

Bibliography

EATON, J.R., ALLT, G. AND HUGHES, K., "The SX Node Architecture," ICL Technical Journal, Vol 7, Issue 2, 1990.

ABRAHAM, G.P., FREETH, D.C. AND VOSPER, H., "SX Design Processes," ICL Technical Journal, Vol 7, Issue 2, 1990.

IEEE COMPUTER, "A Survey of Cache Coherence Schemes for Multiprocessors," IEEE Computer, June, 1990.

Biographies

George Allt

George Allt joined ICL in 1968 with an HND in Computer Science from Oldham College of Technology. He initially worked for the Computer Engineering Service Organisation before moving to development in 1971. He has been involved in both 1900 and 2900 machines, primarily in microcode development. George was the design manager for the picode development on SX and was the overall system designer for the SY project.

Pete DeSyllas

Pete DeSyllas joined ICT Stevenage in 1967 having graduated from Exeter University with a BSc in Physics/Maths. He joined the 1901A hardware development team, subsequently was involved in the design of Disk & Communications controllers and then joined the team developing the smaller 2900 Series processors, becoming responsible for the design of the cache/slave storage systems.

He moved to Manchester in 1975, leading teams to design and develop cache systems for the 2966 and Series 39 Level 30 systems. On SY he has led the team responsible for the design and development of the IODB—the unit providing connection between Node and IO Subsystem.

In May 1997 he transferred to Cadence Design Systems Ltd under an outsourcing agreement between the two companies.

Martin Duxbury

Martin Duxbury is a UMIST Physics graduate who joined ICL West Gorton in 1978 from STL Harlow. He has worked on four generations of OCP hardware, from 2966 via Series 39 Level 80 and SX systems to the latest SY. He was the SY OCP system designer and lead the team of engineers implementing the majority of the OCP logic.

In May 1997 he transferred to Cadence Design Systems Ltd under an outsourcing agreement between the two companies.

Kevin Hughes

Kevin Hughes joined ICL Bracknell in 1979 after graduating from Sheffield City Polytechnic with a BSc in Computing Science. He initially worked on 1900 microcoded emulation systems on 2900 systems and moved to West Gorton in 1982 to continue the work on Series 39 where he moved into Series 39 system design specialising in performance. He was system designer on SY and was manager of the performance modelling team.

He was appointed ICL Distinguished Engineer in 1996 and is a fellow of the BCS and is currently a system designer working on the next generation of Series 39.

Kam Lo

Kam Lo joined ICL in 1970 after graduating from the University of Manchester with a BSc in Electronic and Electrical Engineering. He worked initially on developing interconnect technologies for 2900 series products and since then has worked on Series 39 systems with technical design responsibility for various hardware units. On SY he was the manager of the

OCP and IODB design teams.

In May 1997 he transferred to Cadence Design Systems Ltd under an outsourcing agreement between the two companies.

John Lysons

John Lysons joined ICL in 1982 after graduating from Oxford University with MA honours in Mathematics. He has worked in Manchester on three Series 39 mainframe developments.

John has produced microcode for Series 39 level 80, picode for SX as well as software products including an SX microcode compiler, the picode test bed and part of the high level performance models for SX and SY. He became manager of the SY picode project in 1994.

Phil Rose

Phil Rose joined ICL in Manchester in 1976 after graduating with a BA in Engineering (Pt I) and Electrical Sciences from Cambridge University. He worked on the development of successive mainframe products from 2982 to SY where he was a systems designer and a development manager for the hardware design.

In May 1997 he transferred to Cadence Design Systems Ltd under an outsourcing agreement between the two companies.

Discovering associations in retail transactions using neural networks

O.V.D. Evans

Research and Advanced Technology, ICL, Bracknell, UK

Abstract

A pilot application is described in which an unsupervised neural network is used to look for all significant combinations of store items that tend to occur together in supermarket basket transaction files. Such networks can pick up repeating patterns without the need for any preliminary training. Before searching for such associations a number of file pruning and conversion steps are needed. The reason for these is described together with the means by which the results of using the network are viewed and analysed. The results to-date indicate that savings of several orders of magnitude in file searches can be achieved when compared with conventional boolean queries.

1. Introduction

The ability of a neural network to generalise can be exploited in data mining applications to find associations between groups of items. A typical example of such an application is in retail basket analysis. In this case it can be of value to the retailer to find out all the combinations of goods that tend to be purchased together. To discover all likely associations using Boolean key conjunctions in a database search would be impossibly laborious since each likely item would need to tested against all combinations of all other likely items.

A neural network can be used to some effect to find associations in two ways. A supervised neural network can be used as a 'what if' tool in which the network is trained either on a randomly selected sample of actual baskets, or on a set of synthetic baskets the contents of which represent various hypothetical mixes of items to determine the actual prevalence of such mixes. Alternatively, and more usefully, an unsupervised neural network can be trained on an entire basket file and any associations present can be deduced by examining the weightings of the nodes.

Clearly, the application of a competitive neural network towards the discovery of associations between basket items will involve both a considerable measure of data pre-processing and conversion and then subsequent analysis of the results. The bulk of this paper is devoted to describing these data filtering and conversion procedures in order to put the use of the competitive network in a proper perspective. The paper aims to show that the conversion used make the use of neural networks a practical approach to data discovery.

2. Association mining

The purpose of association mining in a retail industry context is to try to discover patterns in the sale of items. Many items are known to have seasonal or regional sales profiles, but of equal interest to the retailer is to know what individual items tend to be bought together—in other words, how many times item X occurs in the same basket of purchases as items Y, Z , etc. Such information can be of value in activities such as restocking, display layout, product promotion and in the exploitation of buying trends and fashions. Some quite unexpected associations have been revealed from such analyses [Derbyshire, 1996], [Baran, 1997].

The actual analysis tries to quantify associations in terms of *confidence* and *support*. These terms, coined by Rakesh Agrawal of the IBM Almaden data mining group [Agrawal and Srikant, 1994], refer respectively to the percentage of records in some database D containing item subsets X which also contain Y, and the percentage of records in the database that contain both X and Y. An association with high confidence means that most records that have X also have Y. An association with high support means that most records in the database have both X and Y.

Although it is quite easy to specify what information is needed, it can be appreciated that, with retail operations stocking 10,000 to 40,000 individually coded items, the combinatorial problem of looking not only for pairwise matches but for more extended associations is computationally intractable using conventional record searching techniques based on Boolean conjunctions. A subsequent section will show that, although the number of items likely to form associations of interest is more likely to be around two orders of magnitude down from the above figures, since files holding thousands of transactions have to be searched, the intractability issue remains formidable. Nevertheless, since the variety, rather than the total number of items sold, determines the size of the neural network, the reduction of this value to a few hundred makes the use of both supervised and unsupervised neural networks acceptable in terms of performance.

Retail transaction records tend to come in the form of large 'flat' files. For the purpose of data discovery, where associations may be sought between different classes of objects, such as products, departments, prices, dates, times of day and checkout positions, it is preferable to work with the 'raw' data rather than with some subsequently 'warehoused' variant. The 'raw' data is easily filtered with simple tools and purpose-built applications and then converted into vectors of active and passive elements needed as input for the neural networks—as explained below. There would be a disadvantage in using 'warehoused' data, or data otherwise assumed into a database, since the original files might need to be reconstituted by means of the relevant join operations before conversion to a form digestible to a neural network. Working with the 'raw' data avoids this overhead and is a not uncommon practice among implementors of data discovery applications. There is also the problem that database access languages may have difficulty with some of the file filtering operations, such as the quorum function, needed to select baskets with a minimum occupancy.

3. Neural networks

Neural networks originally started as a way to simulate the behaviour of networks of biological nerve cells, known as neurons. Each simulated neuron, like its biological counterpart, is designed to carry out a simple threshold calculation by collecting signals at its multiple inputs and summing them. If this sum exceeds a set threshold, the neuron 'fires' by sending out its own signal. The transformation between input sum and output is determined by a non linear function which can vary from a simple 'gap acting' (hard on/hard off) to the more common sigmoid (S-shaped) response.

Although neural networks bear some similarity to their biological precursors, they have evolved into a number of standard types that now bear little resemblance to their biological equivalents other than the ability to learn and recognise patterns, particularly where invariant behaviour is buried in a noisy environment. Such data-rich environments are termed *model-weak,* and model estimation depends on the use of statistical inference techniques, such as non-parametric regression. Neural networks can be regarded as an example of such a technique and have found uses in applications where the only reasonable means of capturing the behaviour of a model is through learning.

In general, two classes of Artificial Neural Network (ANN) can be identified:

- **Supervised ANNs:** commonly the multi-layer perceptron model
- **Unsupervised ANNs:** also known as self-organising maps.

3.1 Supervised networks

These are modelled on the work of McCulloch and Pitts on *association nodes* [McCulloch and Pitts, 1943] and developed into *perceptrons* by Rosenblatt [Rosenblat, 1962]. Perceptrons were conceived as 'black box' genotypic models with a memory mechanism that permits them to learn responses to input stimuli. Although their behaviour approximates to that of a nervous system their components, artificial neurons, bear little relation to their biological counterparts. In a *multi-layer perceptron* (MLP) these neurons are arranged as layers of nodes, in which each node has a weighted input link from all the nodes in the previous layer. The sum of these input signals is multiplied by a weighting factor and the sum then operated on by the node's transfer function. The output is then passed to the next layer, again via a weighted link.

Supervised learning involves propagating an input signal through the MLP structure to its output and comparing the result with some expected value. The error signal generated is then used to update the weightings between the nodes in a process of *back-propagation*. The commonest form of back-propagation in use is the *generalised delta rule* of Rumelhart, Hinton and Williams [Rumelhart et al, 1986]. A series of input vectors known as the *training pattern* is repeatedly presented to the MLP until the root mean square value of the resulting error signals reduces to less than some specified threshold, or a given set of repeat cycles completes. The training process involves the setting of two parameters, the learning rate η, the momentum α, the number of hidden layer nodes and the number of training vectors that make up a *training epoch*. An MLP can be used for data discovery but, due to the need for specific training data (the 'supervision' component), the process is rather indirect and does not compare with the capabilities of an unsupervised network. A fuller treatment of the practical use of supervised neural networks is given in a previous Systems Journal article [Evans, 1997].

3.2 Unsupervised networks

Neural networks can be used for data discovery in cases where there are no data available to train a supervised network. Unlike the case of supervised networks, there are a variety of unsupervised ANNs of which the commonest forms are variants of the competitive network and the Kohonen self-organising map. The distinction between these two types of network is that in the case of the Kohonen *ordered map* there is a process whereby a 'winning' node claims the territory around itself, and in the competitive case [Rumelhart and Zipser, 1986] the second (non-input) layer is organised into *inhibitory clusters*, in each of which only one node can *win* the competition. Figure 1 illustrates a simple competitive network with three input units and three competitive units. The arrowed links between the input units on the left and the competitive units on the right are the weighted inputs to the latter. Figure 1 also includes the frequently used option of inhibitory links between competitive nodes and self-reinforcing links. These, if used, will apply a small decrement to the weights of all nodes but the winning one. In the application described below this option was not used. In such a network a 'winner takes all' strategy means that the winning node reinforces its state by transferring weights from its inactive to its active inputs. A fuller description of the particular competitive algorithm used in the application is given in the Appendix.

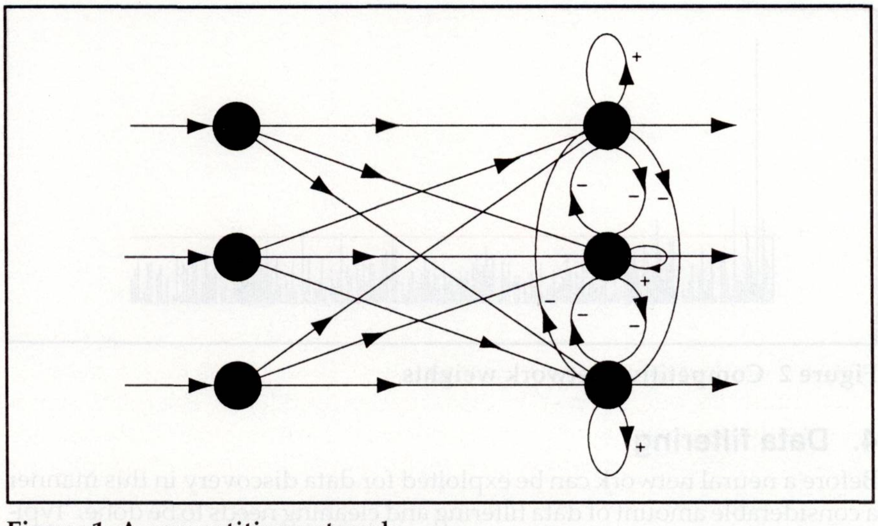


Figure 1 A competitive network

It is important to note that the competitive rules, though simple, reinforce the weights of vector elements that occur together and do not perform simple cumulative totals on each vector element. This means that a competitive network is capable of resolving as many independent associations of vector elements as there are competitive units in the cluster. Such resolution usually becomes apparent after even a single training cycle, so that it is not normally necessary to present the input data more than once to the network. This contrasts with the operation of a back-propagation network, where the input set of training patterns is repeatedly applied to the input until, through back propagation, the difference between the network output values and the expected values reduces to below some pre-set threshold. Another difference between supervised and unsupervised networks is that in the former the output is the set of values of the output nodes of the trained network to subsequently applied input signals. In a competitive network the results of interest are the values of weights between the input nodes and the competitive nodes. The outputs of the competitive nodes merely serve to determine the competition winner for each input vector.

Figure 2 is a typical graphic plot of the weight values of a competitive node after a single training pass. By applying a threshold value (the horizontal line—set by trial and error to some fixed fraction of the maximum weight value) it is possible to separate the highest values as collectively corresponding to clusters of input values that have recurred sufficiently often to have led to reinforcement on every occasion that the node has won the competition. How these values are further analysed to discover associations is described in Section 5 below.

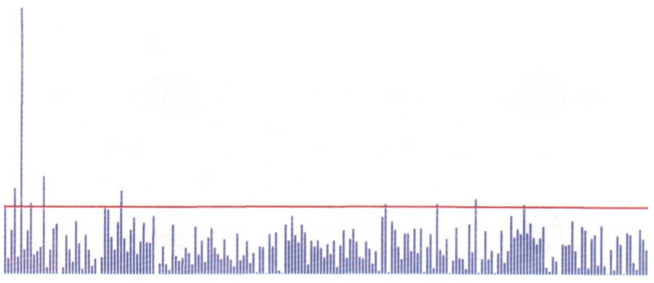


Figure 2 Competitive network weights

4. Data filtering

Before a neural network can be exploited for data discovery in this manner a considerable amount of data filtering and cleaning needs to be done. Typically, out of a product range of 10,000 different items (known by their universal product code, UPC) sold in a day, the most frequently occurring have associations that are already well known while the thousands of items with the lowest ranking are sold so infrequently that they are unlikely to form any significant associations. Somewhat surprisingly, the range of items of interest can be reduced to a few hundred, as can be seen in Figure 3, where 204 items have been selected from the overall rankings. A further stage of data filtering that it should be possible to apply is to select items by department. Figure 4 shows the cumulative count of items ranked by department with three departments selected for further analysis. Such a selection will prune all items belonging to unselected departments from the records of the basket file.

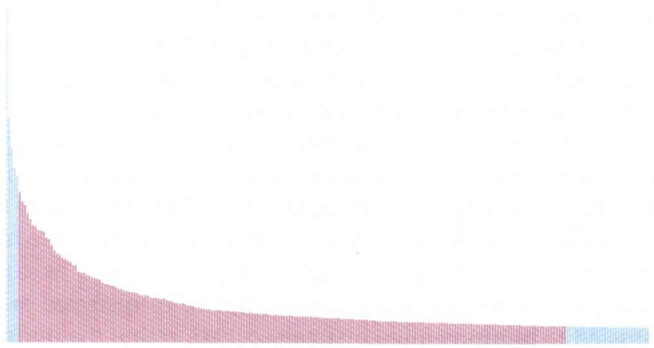


Figure 3 Occurrence of items by rank

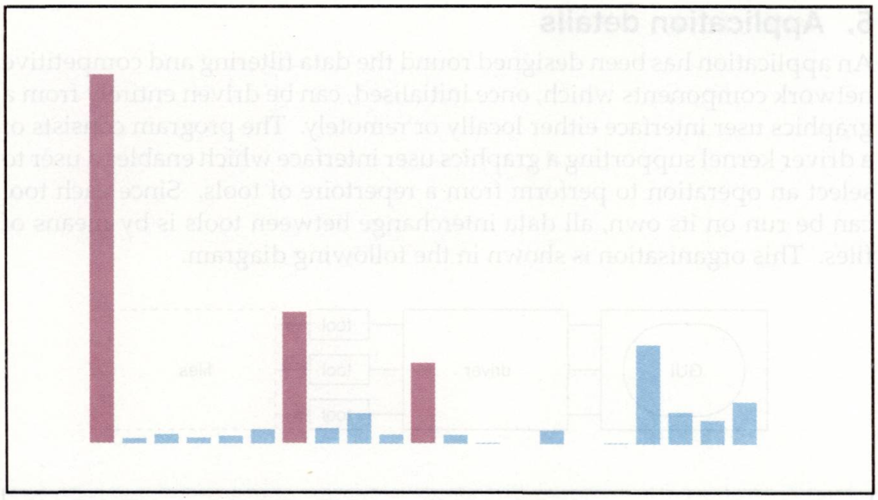


Figure 4 Occurrence of items by department

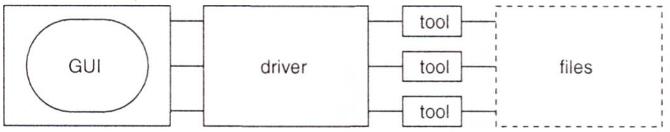
In the same way that the items in each basket can be reduced to just those belonging to departments of interest, it is also possible to weed out all baskets with less than some specified minimum item count, since they are unlikely to produce associations of significant interest. Typically such pruning can reduce the number of records to be analysed by a factor of four. As an example, by setting a minimum quorum of 5 items per basket, the number of baskets to be analysed was reduced from 3,228 records to 767 before eliminating any departments.

With judicious pruning it is possible to reduce the range of items which could form associations from several thousand down to, say, 200. This reduces the possible pairs that can be formed from several tens of millions to 19,900 $({}^{200}C_2)$. Although this still leaves a large combinatorial search space, with 200 inputs it immediately becomes practical to use a neural network whereas 10,000 would present intolerable performance problems. For this purpose each of these 200 items is represented by a presence or absence value in a vector.

For finding associations, multiple occurrences of an item in a basket are of less interest than the variety of items in the basket and are given a 'present' value in the vector. Multiple occurrences, however, can be made to bias the item value logarithmically from say 0.5 for a one-off to some arbitrarily high value of, say, 0.9. Without this bias low occurrences would not generate a detectable signal. A typical neural network for basket association search could therefore have 200 input units and 20 (say) competitive units—this would allow the network to identify up to 20 clusters of input signals, but with the likelihood of partial overlaps.

5. Application details

An application has been designed round the data filtering and competitive network components which, once initialised, can be driven entirely from a graphics user interface either locally or remotely. The program consists of a driver kernel supporting a graphics user interface which enables a user to select an operation to perform from a repertoire of tools. Since each tool can be run on its own, all data interchange between tools is by means of files. This organisation is shown in the following diagram.



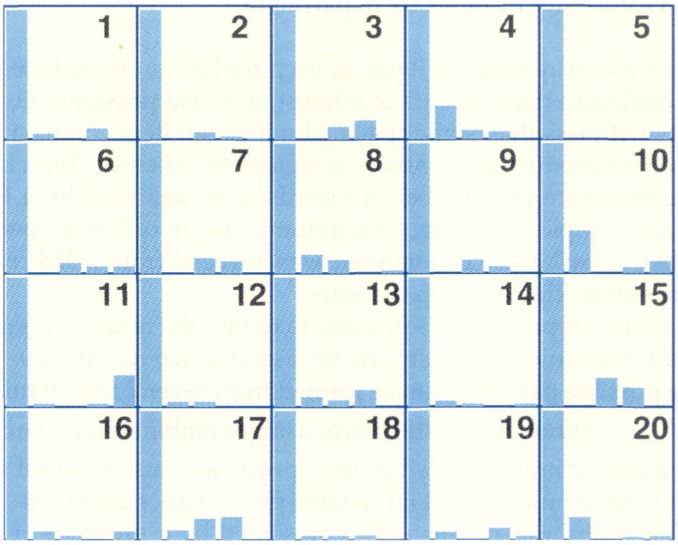


Figure 5 Example of an association map

In a typical data discovery run the following operations would be invoked, by using the corresponding tool and would normally be executed in the sequence listed below.

- **Stats:** This tool is used to extract basic statistics from the raw basket data such as rank ordering by number of items sold, items sold by department and basket occupancy rankings. The statistics, such as item occurrence ranking (see Figure 3) and item counts by department (see Figure 4), are presented to the user, who can then choose a subset for further processing from a continuous range of items in selected departments. The tool takes as its input the raw basket data file and formatting information about it.

- **Prune:** Prune performs several tasks. It will generate a pruned basket file that includes only items from the selected range, from selected departments and of a given user-supplied minimum occupancy. It will also assign a vector index to all the items selected. It will finally configure the competitive network in accordance with the number of items (input units) remaining in the selected range, and the number of competitive units specified by the user.
- **Vectorise:** Vectorise will ask the user to select a portion of the pruned basket file for conversion into pattern vector format to be input to the neural network. Although the entire file can be vectorised, it is often more useful to partition it into parts corresponding to selected times of the day.
- **Competitive neural network (CNN):** The neural network tool will accept a configuration file (produced by the **Prune** tool) specifying the number of input and competitive nodes, the learning rate and the number of training cycles (one, by default) to be executed. The user can control and monitor the operation of the tool by means of its own dialogue box. The output of the network is displayed for every input vector during learning, and the state of the weights to a selected node can be inspected on completion. Figure 2 is an example of such a display. The operator has the option of moving the threshold line from its default position before the state of the weights are written to a file, 'all_nodes.pl', containing for each competitive node, its identity, the threshold value used and a 'hit' list of all the weights, which exceeded the threshold, sorted into descending order of weight values.
- **Analysis:** This tool performs three functions:
  1. For each node it takes a specified number (default 5) of the highest value weights from the 'all_nodes.pl' file, uses their index value (input node number) to pick up the corresponding product code from a table generated by the **Prune** tool and writes the resulting lists to a new file.
  2. Again for each node, the product code with the highest weighting is used as a key to spool off from the main pruned file only those records containing the code. The number of occurrences in the spool file of each of the remaining codes in the list is then found.
  3. The results of the search are then displayed graphically as a matrix

of graphs, each of which displays the associations found for the corresponding node (Figure 5). In each box, the left hand bar is the key code with the remaining bars representing the count of records in the spool file containing both the code for the bar and the key code. Statistics such as numeric occupancy values and percent confidence can the be displayed in an appended dialogue box (not shown in the figure) by 'clicking' on the bar for individual statistics, or in the box space for an overall summary of all the bars in the box.

All the tools described above are presented to the user in a toolbar (not shown in the figure) below the main application workspace and instances to the tool can be called into action by clicking on their icons. Each active tool is capable of a choice of activities selectable from a pull-down menu and can operate independently of each other on filed data. As files are accessed or created during the operation of a tool, these become visible in the workspace along with links to the tools using them. Figure 6 shows what the workspace would look like to the user at the completion of a data discovery run. The box at the upper right of the figure is a message area in which the application can inform the user about its current state. Figures 3, 4 and 5 are all instances of the graphic information that can be selected from the icons in the workspace.

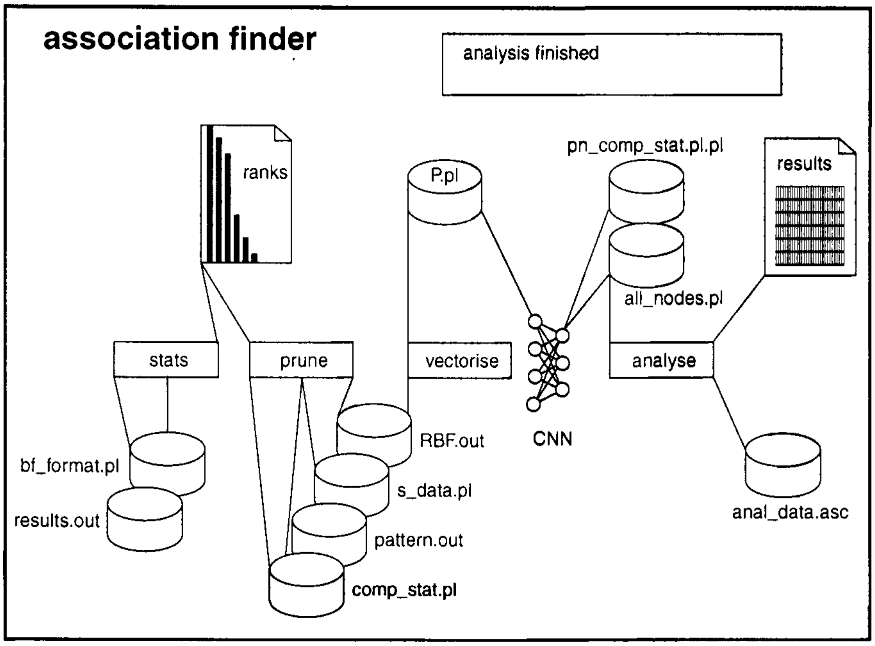


Figure 6 Application user interface

The application was written in ECL^{i}PS^e, a highly efficient constraint logic programming language based on a Prolog developed at the European Computer Industry Research Centre in Munich. ECL^{i}PS^e versions exist for a variety of UNIX platforms. The graphics interface was written in ICL GraphicsPower.

6. Use of supervised networks

Although supervised networks, with their need to be trained on example data, cannot, strictly speaking, be used for data discovery, they can be used in a related activity—hypothesis testing. This involves training an MLP with a set of baskets randomly sampled from the data or artificially made up to represent the 'typical' purchase profiles of various classes of hypothetical customer.

The former approach will yield results—eventually, as the discovery of associations then becomes a hit-or-miss process depending, as it does, on picking up a 'good' set of baskets during random trawls of the file. Early versions of the application did use a proprietary supervised neural net while the competitive net was still under development. The use of this MLP has now been discontinued.

7. Results

The association map shown in Figure 5 was generated from the analysis of the weights of a competitive network having 200 inputs and 20 competitive units after one training cycle using 80 baskets.

As described above, each analysis box, corresponding to a node, is the result of taking the UPC code corresponding to the highest value weight and using it as a key to spool from the original pruned file, a file containing only the records that include the key, and counting the occurrences of the next five highest weighted UPCs in that file.

The merit of the map is that it shows at a glance the strength of the associations discovered by the competitive network. In the actual application, statistics about the associations picked up by a particular node can be displayed by 'clicking' the cursor in the relevant box. A summary of all the associations discovered in the example run is listed in Table 1.

Some associations are quite weak but others, such as those picked up by nodes 8 and 17, show strong associations. In the case of node 17, over 4 items—with 11% of all baskets containing item U000[1] also containing items U003, U004 and U005. In this particular case these associations occurred in a sample of 767 baskets so that although some groupings have high confidence, in general they have low support.

[1] Index values have been used in place of UPC values as basket transaction files and any results derived from them are regarded as commercially confidential by the originators of the data.

Node	Item	No. Occ.	Conf(%)	Node	Item	No. Occ.	Conf(%)
1	U168	24		11	U018	42	
	U088	1	4		U132	1	2
					U103	1	2
	U084	2	8		U002	10	23
2	U011	98		12	U053	25	
	U162	3	3		U118	1	4
	U071	8	8		U116	1	4
	U074	5	5				
	U138	4	4				
3	U108	21		13	U042	40	
	U046	2	9		U021	3	7
	U010	3	14		U135	2	5
					U040	5	12
4	U009	105		14	U041	46	
	U005	28	26		U142	2	4
	U003	9	8		U087	1	2
	U022	8	7		U121	1	2
	U151	2	1				
5	U039	35		15	U097	14	
	U104	2	5		U010	3	21
					U008	2	14
6	U035	38		16	U106	30	
	U075	3	7		U169	2	6
	U146	2	5		U137	1	3
	U052	2	5		U035	2	6
7	U022	44		17	U000	155	
	U016	5	11		U003	18	11
	U031	4	9		U004	18	11
					U005	31	20
					U031	7	4
8	U012	51		18	U106	30	
	U003	8	15		U169	2	6
	U014	6	11		U137	1	3
	U017	1	1		U035	2	6
	U031	1	1				
9	U161	20		19	U033	68	
	U045	2	10		U046	6	8
	U170	1	5		U043	2	2
					U155	8	11
					U124	4	5
10	U123	22		20	U007	128	
	U020	7	21		U000	22	17
	U059	1	4		U063	3	2
	U045	2	9		U162	4	3

Table 1 Summary of associations found

Other runs, involving different contiguous parts of the data file, have been successful in finding that the fourteen occurrences of one particular item always occurred with the only occurrences of one other item. The probability of picking up one of the occurrences of this pairing with random sampling would be about 0.07 and took several dozen runs to discover using the supervised network.

8. Conclusions

It is clear that preliminary pruning can reduce the combinatorial complexity of trying to discover associations between items in a typical transaction file by several orders of magnitude. This is still not sufficient to render the problem tractable by conventional search methods. However, the generalising ability of a suitably configured competitive neural network has made it practicable to highlight potential associations after a single training pass. Then it becomes a simple matter to use the resulting weight statistics to 'drill down' into the original file to test the strength of the clusters discovered by the network. The number of full scans of the file is never more than the number of competitive nodes regardless of the number of item codes over which associations are being sought. With a maximum association membership of five item codes in the example given this means that the total number of additional scans is 80—a scan for each of the non-key items in each node. Some of the latter can be trivial for key items with low support. This compares favourably with the very large number of passes over the database needed for a complete pairwise association search over 200 item codes and the rather greater number in the case of associations of more than two items.

The great merit of the procedure described is that the clusters revealed by the competitive network are merely a means of pruning the number of searches over the transaction file. The results returned are always counts of actual items present. The system is therefore in this sense self-validating for what it finds—but it may of course miss some weak associations if the number of competitive nodes is insufficient to detect them.

Appendix

Competitive Networks

The operation of a simple competitive network with a single layer containing just one inhibitory cluster, as used in the data mining application under discussion, can be described in the following set of rules:

- Every competitive unit in the cluster receives an input from every input unit.
- A competitive unit only learns if it wins the competition against its neighbours; i.e., it has generated the highest output value.
- A stimulus pattern S_j is a vector each element of which can be either *active* or *inactive*. An active element is assigned a value 0.9 and an

inactive one 0.1.

- Each competitive node j generates an output given by the expression

$$y_j = sigmoid(\sum_{i=1}^{N} i_j w_{ij})$$

where N is the number of elements in the input vector and $sigmoid(x)^2$ is the conventional transfer function used in neural nets. The use of a non-linear transfer function is optional and acceptable results can be achieved with a unity function.

- Each unit has weights that average to a constant (normally 0.05). For unit j where the weight on the line from input i is given by w_{ij}, the weights are set to conform to the identity

$$\frac{1}{N}\sum_{i=1}^{N} w_{ij} = 0.05$$

A unit learns by shifting weight from inactive to active input lines such that this identity is maintained. More formally the learning rule that has been adopted is:

$$\Delta w_{ij} = \begin{cases} 0 & for\ a\ non-winning\ unit \\ k\eta y_i & for\ the\ winning\ unit \end{cases}$$

where y_j is the output value of the unit, constant k is +1 for an active input and −1 for an inactive input, and η is the learning constant (set to 0.11 through experiment).

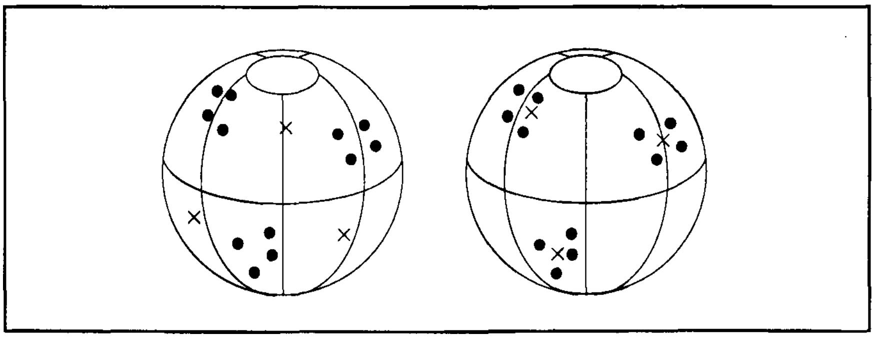


Figure 7 Competitive network before and after learning

The effect of this weight transference is to move the stored pattern in

$^2 sigmoid(x) = \frac{1}{\left(1+\frac{1}{e^x}\right)}$

the winning unit weights a little closer to the input pattern. Figure 7 illustrates this graphically where the input vectors have been normalised to unit length and represented as black dots. The weight vectors, initially set to random values, are marked with Xs in the left hand diagram. After training, their final positions indicate that each of the three nodes has discovered a natural cluster of patterns with the weight vectors now respectively pointing to the centre of gravity of each of the groups, as seen in the right hand diagram. In general, the weight distributions of the individual members of the inhibitory cluster tend to adopt values corresponding to mutually exclusive reinforcing input vector values (but with overlapping elements allowed) so that the multiple occurrence of distinct associations of vector elements can be discovered by examining the weights at each node at the end of a training run. From the relative strengths of these it is possible to derive association rules for the vector elements they represent.

If there is structure in the input vector patterns, the competitive units will break up the patterns into structurally relevant clusters—if any are present. If the stimuli are highly structured then the classifications will remain stable. For less well structured stimuli the classifications will be less stable and may even move about from one node to another in the cluster. The results achieved to-date show little overlap between units. This may be due to the fact that the occurrence of active vector elements in the input patterns corresponds to the overall occurrence of the UPC items they represent which is highly non-uniform as shown in Figure 3.

References

AGRAWAL, R and SRIKANT, R., *Fast Algorithms for Mining Association Rules*. Proceedings of 20th International Conference on Very Large Databases, Santiago, Chile, 1994.

BARAN, U. *Helping Retailers Generate Customer Relationships.* ICL Systems Journal, Vol. 11, No. 2, 1997.

DERBYSHIRE, M.H. *An Architecture for a Business Data Warehouse.* ICL Systems Journal, Vol. 11, No. 1, 1996.

EVANS, O.V.D., *Short-term currency forecasting using Neural networks.* ICL Systems Journal, Vol. 11, No. 2, 1997.

McCULLOCH, W.S and PITTS, W., *A Logical Calculus of the Ideas in Immanent Nervous Activity.* Bull. Math. Biophysics, 5, 115-133, 1943.

ROSENBLATT, F., *Principles of Neurodynamics.* Spartan Books, 1962.

RUMELHART, D.E., HINTON, G.E. and WILLIAMS, R.J. *Learning Internal Representations by Error Propagation,* in Parallel Distributed Processing, Vol. 1, (edited by D.E.Rumelhart et al), MIT Press, 1986.

RUMELHART, D.E. and ZIPSER, D. *Feature Discovery by Competitive Learning,* in Parallel Distributed Processing, , Vol. 1, (edited by D.E.Rumelhart et al), MIT Press, 1986.

Biography

Owen Evans joined the Advance Research and Development laboratory of ICT Engineering (an ICL predecessor) in 1963 from the oil industry. The research facility has continued in unbroken line to its present guise of the Research and Advanced Technology centre of ICL Group HQ. During his time in the research centre which has spanned virtually the entire evolution of the computer industry, he has worked on computer architecture, memory and microprogram design, system performance evaluation, compiler design and logic languages. In the course of his career at ICL he has managed various projects supported by the Advanced Computer Technology Project, the Alvey programme, and the EC Esprit and Fourth Framework initiatives in the areas of high-level language emulators, text databases, human-computer interaction and constraint logic programming. His current interests are in the areas of constraint logic programming, data mining and neural networks.

Mr. Evans graduated in Engineering from Cambridge in 1959.

Methods for Developing Manufacturing Systems Architectures

S. Murgatroyd[1] and R. Smethurst[2]

[1]MSI Research Inst., Loughborough University, Loughborough, UK
[2]ICL Enterprises, Westfields House, Kidsgrove, Stoke-on-Trent, UK

Abstract

This paper describes the progress of a collaborative research project being carried out at the MSI Research Institute in collaboration with ICL Enterprises. Building on ICLE's (ICL Enterprise's) OPEN*framework* methodology, the project aims to develop generic manufacturing systems architectures, which describe typical manufacturing business processes, typical application architectures and supporting social systems. These generic architectures will form the basis of re-usable models of manufacturing domains for use in subsequent OPEN*framework* assignments. In addition, the project aims to evaluate and specify an integrated set of tools to support the OPEN*framework* methodology and to provide a critique of the methodology in the light of the state-of-the-art in the area (particularly with respect to the development of Enterprise Integration science and the rapidly developing field of domain ontologies) and other comparable approaches. Methodology and tool evaluation is being carried out via a series of case studies, with one such study, involving a major UK electronics manufacturer, being reported in detail in this paper.

1. Project Description

A three-year EPSRC/CDP funded research project has been taking place since May 1995. Entitled "Methods for Developing Manufacturing Reference Architectures", the project is a collaborative venture, between the MSI (Manufacturing Systems Integration) Research Institute of Loughborough University and ICLE which aims to:

i Support the development of a manufacturing systems architecture which lends structure to manufacturing systems used in target enterprises and identifies attributes of necessary underlying information system components.

ii Develop specific manufacturing reference models, providing specialisations of the generic manufacturing systems architecture.

iii Apply, evaluate and promote the use of an integrated set of methods and tools to support the top-down development of the generic architecture and its specialisations.

Four main areas of study are being pursued, as follows:

i Classification and Structure of Manufacturing Architectures and Reference Models. Here existing candidates are being classified and assessed in respect of their capability to populate manufacturing business architectures, technical architectures and social systems.

ii Methods and Tools for Model Capture. This activity has two main thrusts, namely: development of a meta-model based on the OPEN*framework* methodology and an assessment of available methods and tools that can support the methodology.

iii Case Study Application of the Methods and Tools. Prior to being integrated into the meta-model framework, the capabilities of a subset of candidate methods and tools is being evaluated on case study data obtained from collaborating partners. The details of one such case study, carried out in conjunction with a major UK electronics manufacturer, are described later in this paper.

iv Workbench Specification, Application and Enhancement. The design of this workbench is being established in consultation with collaborating partners with a view to them directly exploiting the technology.

2. The OPEN*framework* Methodology

The OPEN*framework* methodology [ICL, 1993] has been developed within ICL and is used by over 400 practitioners worldwide. The methodology is a set of well-documented and method-supported precepts and guidelines to ensure business process re-engineering[1] and system integration projects are managed effectively and that they give rise to solutions that are aligned with business need. It is used to assess customer needs, to define criteria for success and to structure design, development and delivery processes. Whether developing a stand-alone product or a full business process re-engineering solution, the principles remain the same. These are embodied in "The Four Precepts":

i Every system integration project must follow a process defined by OPENframework's "Stages of Solution Delivery"

ii Requirements must be specified with regard to the different Perspectives of people affected by the project

iii Solutions must be expressed in terms of the OPEN*framework* Elements

iv Success criteria must be specified with reference to a defined set of Qualities.

[1] The examination and refinement of organizational processes and activities, manual or automated, that support the functioning of the business. (Re-architecting: a Netron white paper - http://www.netron.com/literat/rearchwp.htm).

The goal of applying the OPEN*framework* methodology is to produce, either in part or in full, architectures made up of the OPEN*framework* Elements, namely Business Architecture, Technical Architecture and Social System.

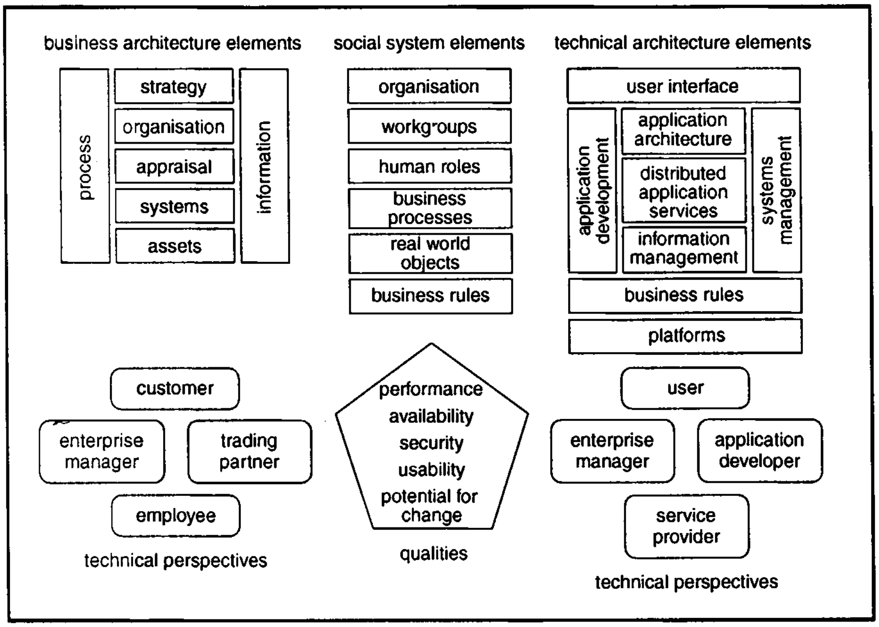


Figure 1: OPEN*framework* Elements, Perspectives & Qualities

3. OPEN*framework* Meta-Model

Early in the project, a meta-model of the OPEN*framework* methodology [Murgatroyd & Gilders, 1995] was produced using entity-relationship-attribute techniques. The aim of this model is to:

i Capture and promote an understanding of the concepts of the methodology and the relationships between the elements of the architectures, social systems, qualities and perspectives.

ii Provide a framework within which to position candidate methods and tools.

iii Provide a mechanism by which the project's relationships with complementary MSI research projects (involved in business strategy development and distributed object systems) may be formalised; thereby characterising key aspects of the nature of the interface between business and manufacturing systems in a formal and consistent manner.

It is envisaged that subsequent case studies to that described in Section 4 will use, in part, ICL's "ProcessWise Workbench" [ICL,1994],[2] a process modelling tool which can be configured according to an internal meta-model; the OPEN*framework* meta-model described above will be used for this purpose. The development of the meta-model is ongoing and is taking into account the recent work in the area of developing enterprise ontologies, more specifically those of the TOVE project [Fox, 1992], Stader [Stader, 1996] and Lenat [Lenat, 1995]. In addition to these ontologies, existing ProcessWise meta-models used in previous consultancy assignments will be incorporated into the meta-model where appropriate. Eventually the meta-model configured ProcessWise Workbench will be used to communicate both the Generic Manufacturing Architecture and specialisations of it in terms of reuseable models of manufacturing enterprises. This is shown in Figure 2.

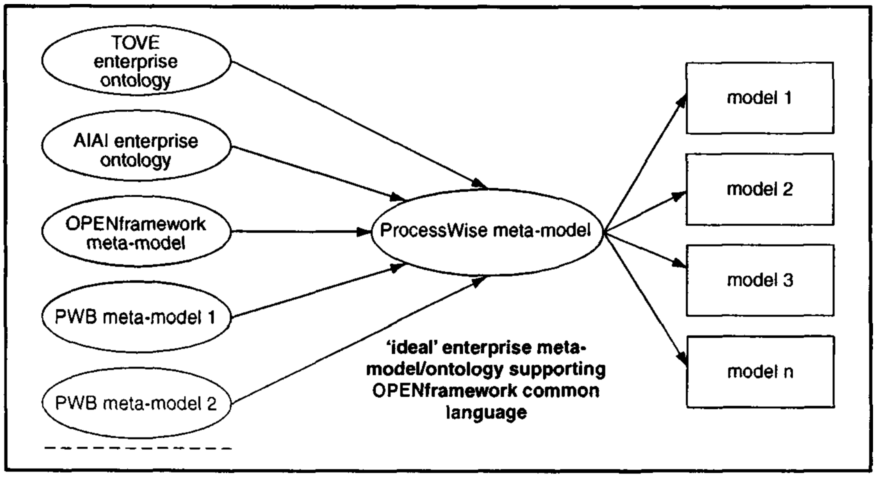


Figure 2: Meta-Model Development & Use

4. Case Study

4.1 Introduction

The case study took place over a three-month period at the premises of a large UK electronics manufacturer. The study had four major aims, namely:

i To provide recommendations to the case study partner regarding improvements to identified manufacturing support processes

ii To provide a means of evaluating the OPEN*framework* methodology

iii To provide a means of evaluating a set of tools for use within the

[2]The ProcessWise Workbench is a PC based tool which allows the modelling, simulation and redesign of business processes. The tool has a powerful meta-modelling facility which allows the workbench to be configured to represent the particular domain(s) of interest.

OPEN*framework* methodology

iv To provide "live" data for the development of the generic manufacturing architecture and its specialisations.

The case study partner had recently undergone a large organisational review and implemented an IT solution to support its manufacturing support processes. For this reason it was not appropriate to consider using the OPEN*framework* approach to define a Technical Architecture but to consider optimising the processes which the IT solution supported. For this reason, the study mainly considered the Business Architecture and Social Systems of OPEN*framework.*

The case study partner's core business is systems design, printed circuit board assembly and systems assembly. This study concentrated on the processes supporting the board and systems assembly activities and deliberately excluded the design processes.

4.2 Scope & Drivers

The study covered all the processes/activities concerned with supporting the manufacture of the case study partner's main product line with a view to providing observations and recommendations to contribute to the overall aims of:

i Reducing order-to-ship lead-time from 4 to 2 weeks

ii Achieving 100% delivery date performance

iii Achieving improved stock levels/inventory control.

After consultation with appropriate staff, the key processes/process areas were deemed to be:

Order forecasting and demand management.

The product is sold and distributed through a network of contracted distributors who produce monthly forecasts of unit sales for a 12-month period. It is the responsibility of the Demand Manager to collate this information and present it to the Sales and Manufacturing Review (SMR) meeting.

Sales and Manufacturing Review (SMR).

This was identified as the key process. The SMR process takes the form of a monthly director-level meeting at which the operations plan for the next period is determined. Sales forecast figures are presented and compared to previous forecast accuracy. The role of this meeting is seen to be to produce the operations plan to support the sales plan.

MRP (Materials Requirements Planning)

The MRP process is carried out automatically by the manufacturing management system and generates procurement requests from the *operations plan and inventory allocations for confirmed orders.*

Procurement

The procurement policy is to procure to the level of the operations plan as decided by the SMR process. Procurement is to forecast, as manufacture is to order.

Order desk.

Orders are received and entered onto the manufacturing management system. The orders are confirmed in consultation with the Master Production Scheduler and a delivery date is generated. Basic information regarding the product is entered at this point to aid the Master Production Scheduler in determining any special product requirements and to enable hardware configuration (full bill-of-materials production).

Hardware and software product configuration.

Prior to manufacture, a full bill-of-materials must be produced (hardware configuration). After assembly, the systems must be software configured prior to delivery.

Master production scheduling.

The Master Production Scheduler loads the manufacturing capacity with manufacturing instructions derived from a monthly-updated operations plan. The Master Production Scheduler has communications channels (formal and otherwise) with the following areas of responsibility: - Manufacturing, Order Desk, Demand Manager, Purchasing and Hardware Configuration.

Sub-system (board level) assembly.

The sub-assembly process is concerned primarily with the production of modules (PCBs) to satisfy demand from both the systems and ship-loose orders (spares etc). A module safety stock buffer is used to regulate the supply to both manufacturing and ship loose. Production orders for modules are raised whenever the level of modules in this buffer falls below a specified safety level.

System Assembly.

The Assembly sub-process represents the manufacturing operations where hardware sub-assemblies and unit software are configured (assembled) to fulfil a particular order. The flow of system orders through this process is essentially constrained by the availability of sufficient system kits and the number of available software configuration engineers.

Warehousing/Delivery.

After system assembly and software configuration, systems are sent to the warehouse for packaging etc. and subsequent delivery to the customer.

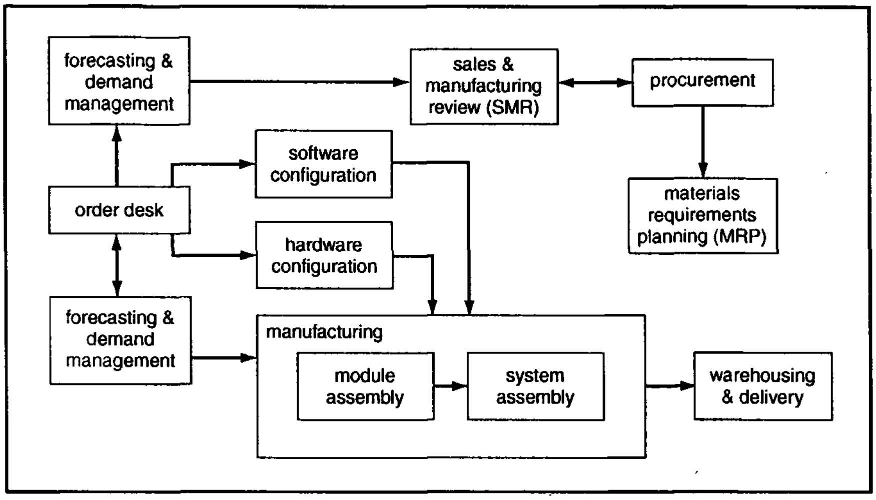

forecasting & demand management
sales & manufacturing review (SMR)
procurement
software configuration
order desk
materials requirements planning (MRP)
hardware configuration
forecasting & demand management
manufacturing
module assembly
system assembly
warehousing & delivery

Figure 3: Process Relationships

At the time of the study, there was excess manufacturing capacity for the volume of product orders that were being received (the product was relatively new and sales volumes were still 'ramping up'). For this reason the manufacturing processes were not considered in detail; i.e. the study did not include the modelling of shop floor layout, individual manufacturing process lead-times, manufacturing shift patterns etc.

4.3 Deliverables

The deliverables from the case study to the industrial partner were to be:

i Documented models of current processes within the scope of the study.

ii Documented what-if? Analyses (simulations) detailing the factors that determine order fulfilment lead-time and inventory control.

iii Documented recommendations for process improvements and possible functional re-alignment to achieve the stated performance goals.

4.4 Tools

The tools used to carry out the case study were:

i Process Modelling – *ithink* [High Performance Systems].

ii Organisational Modelling – VISIO, Visio Corporation.

iii Documentation – Word for Windows, Microsoft.

Since a manufacturing management system had previously been installed, it was not necessary to carry out a detailed information modelling exercise with a view to systems implementation. However, the information flows between processes and the transformation of information from process to process could not be ignored and were documented alongside the production of the process model. If a more formal information representation had been required, IDEF-1, "Integration Definition for Information Modelling (IDEF1X)" [NIST, 1991], or EXPRESS "The NIST EXPRESS Toolkit: Introduction and Overview," [Libes, 1993] would have been used. The majority of modelling was carried out using the *ithink* tool and subsequent model descriptions are presented in *ithink* terms.

Within the scope of this study, the building of a business process model was not the end in itself, rather a means to an end; i.e. to aid the authors in their understanding of the problem domain. The choice of modelling tool and its inherent simulation capabilities enabled the authors to assimilate very quickly large quantities of information and to validate (with the aid of appropriate personnel) their findings through what-if? analyses. The resulting model has been the basis of the recommendations for process improvements, but not exclusively so, since other (unmodelled) relevant information has also helped to form the authors' view of the current situation and hence their recommendations.

4.4.1 Brief *ithink* description

The tool used for modelling the case study partner's order fulfilment activities was *ithink*, developed by High Performance Systems. The authors' selected this package for its flexibility in model construction, graphical processing nature and simple to use simulation capabilities.

A simple model example is now described to give an overview of the *ithink* modelling terminology as a precursor to the description of the process model. It should be noted that this is a very concise introduction to the *ithink* modelling tool and is not intended to provide a complete and thorough description of the software package. The suppliers, High Performance Systems [HPS], should be consulted for more information.

A Simple Model

The *ithink* discrete simulation package allows the creation of reasonably complex models using a number of simple constructs. Figure 4 shows a simple *ithink* model that contains all the basic constructs that can be used to produce a process model, namely flows, stocks, converters and connectors.

The basic idea behind *ithink* models is that processes can be modelled as 'flows' of information between distinct locations, at a rate which maybe either fixed or dependent on some other factor. A complete model can be built by connecting these flows in an order which most accurately reflects the process (or processes) being modelled.

With this in mind, the above model can be broken down into the constructs as follows:

Stocks:	STORES, MANUFACTURING, WAREHOUSE.
Flows:	IN FLOW, OUT FLOW.
Converters:	VALUE, CYCLE TIME.

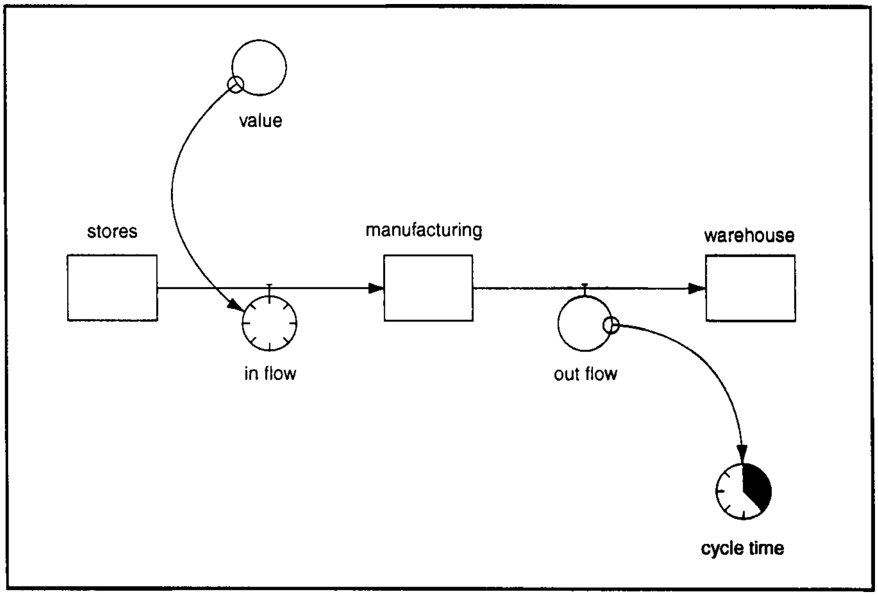


Figure 4: Simple *ithink* Model

Flows represent the transit of 'information units' from one location to another (which may or may not represent an actual physical location).

The *flows* of 'information units' into (and out of) the MANUFACTURING stock are directly controlled by the *flow* values IN FLOW and OUT FLOW. These values may be either fixed or variable; e.g. in the above model IN FLOW may be set according some VALUE dependent on the number of manufacturing orders received and available manufacturing capacity, whereas the OUT FLOW may be dependent on the completion status of manufactured goods and the available warehouse capacity.

Flows are in effect control valves that regulate the rate of *flows* into or out of particular stock items.

Converters can be constant values, definable variables, equations or can be defined graphically.

Stocks can be used to simply hold 'information units' in a model (e.g. a storage bin for nuts and bolts) or can be used to represent the mechanics of a process (e.g. to model the lead-time of a production line). There are a number of *stock* types available in the *ithink* modelling package which have different behavioural characteristics. The types are:

Reservoir:	the simplest ***stock*** type. The number of 'information units' at any given time in a ***reservoir*** depends solely on the number of units added and removed. All units are lumped together; i.e. there is no separation of batches in a ***reservoir.***
Queue:	Information units flow into and out from ***queues*** in a first-in-first-out (FIFO) order. As with ***reservoirs,*** the number of units in a ***queue*** at any given time is dependent on its inward and outward ***flows. Queues*** maintain information about batch sizes; i.e. information enters and exits ***queues*** in discrete chunks.
Conveyer:	Information units traverse a ***conveyor*** with a defined transit time. The outward ***flow*** from a ***conveyor*** will normally be continual but can be halted by using logical decisions determined elsewhere in the process model.
Oven:	The flow of 'information units' into an ***oven*** is controlled. ***Ovens*** process these units in batches and when the processing is complete, dump the units instantly to the outflow. The 'cook time' of units in the ***oven*** is controlled by the outflow logic.

During model construction, the user can specify capacity constraints for all ***stock*** control items used in a process model.

Connectors define the dependencies between specific items in a process model. For example, in Figure 4 the simple *ithink* model shows that the value of the IN FLOW flow is dependent on the value of the ***converter*** called VALUE.

4.4.2 Model Simulation

ithink includes discrete simulation capabilities that allow detailed examination of dynamic behaviour of a process (or collection of processes). The output of this simulation can be presented to the model user in the form of graphs or data tables. The information to be included in these graphs/tables can be defined by the user; e.g. the user may choose to view the flow of units into the stock MANUFACTURING from the simple process model.

The user specifies the 'delta' used for the simulation and the period over which the simulation should run. The delta specifies the unit of time adopted during the simulation (days, months, years etc.) and dictates the frequency at which the model state is recalculated.

The *ithink* package allows a simple measurement of cycle-time information to be calculated. The model, shown in Figure 4, includes a cycle-time measurement that tracks the lead-time of units through the MANUFACTURING process.

4.5 Process

A number of initial interviews were conducted with appropriate staff in order to form an overall view of the key processes with particular emphasis on their inputs, ownership, involved personnel, interfaces with other processes and outputs. This information, together with existing documentation allowed a first-cut model to be developed. This model highlighted the gaps in the knowledge of the problem domain and resulted in a cycle of re-interviewing and re-modelling. Following this re-modelling, typical model simulations were run and shown to the key staff involved in each process to see if the simulations reflected reality in their experience. This enabled some of the major model assumptions to be addressed and corrected and provided valuable model validation. This validation period included attendance at an SMR meeting to gain an appreciation of the quality and detail of information available from which to generate the operations plan.

As a final validation process, the model was presented to the SMR process improvement working group. Again simulations of typical operational scenarios were demonstrated and questioned and the possible uses of the modelling tool within the SMR process itself were discussed.

The total elapsed time for the study was five months through from scoping to documentation and presentation of findings. During this period a number of formal interviews and informal discussions were held with appropriate staff. The key personnel involved were:

i Manufacturing Manager

ii Procurement Executive

iii Procurement Manager

iv Master Production Scheduler

v Demand Manager

vi Order Desk Manager

These people were identified as being key since they possessed intimate knowledge of the current processes and were deemed (informally) to be owners of the processes identified in Section 4.2 (Scope & Drivers). In addition to these people, the work of the study was discussed with the members of the internal SMR process improvement group, bringing the total number of staff involved to approximately twenty. Over the period of the study, approximately thirty interviews and discussions were held and the model went through approximately ten cycles of simulation, validation and re-modelling. The final model was essentially a description of the as-is processes. The simulation capability enabled potential scenarios to be investigated (or more correctly, problems with current operational procedures to be identified) but the model was not sufficiently restructured to represent the to-be situations; in reality the model building exercise provided validated information for the process of deciding on the to-be structure of the processes rather than explicitly defining it. The final model did not represent the entirety of the information gathered and analysed during the study. On-going work is translating this information (i.e. modelling) into suitable forms, using identified methods and tools, particularly ICL's ProcessWise Workbench.

5. Model Description

5.1 Order Desk

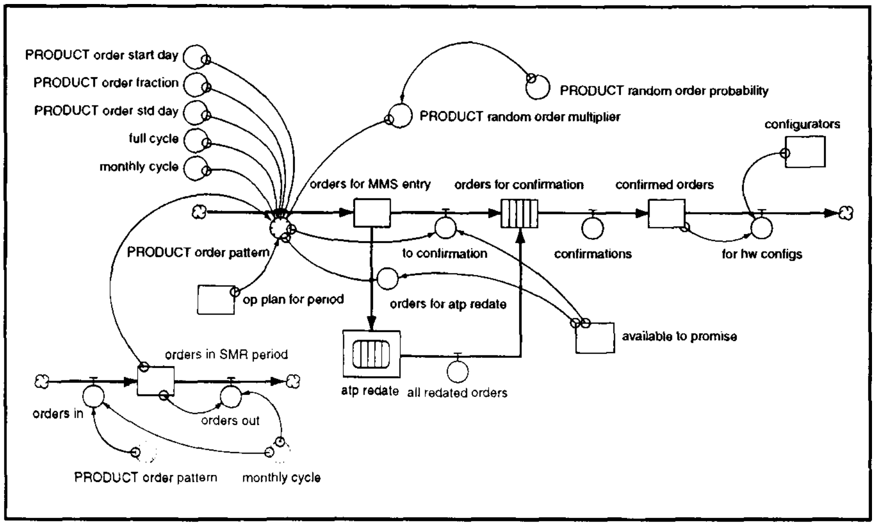


Figure 5: **Order Desk Model–Part 1.**

5.1.1 Description

PRODUCT orders arrive at various rates and in various quantities. The quantities stated in this model are for complete PRODUCT systems. The model allows for the generation of various PRODUCT order patterns using a set of variables depicted in Figure 5 and described below:

PRODUCT_order_fraction

The number of PRODUCT system orders received in an SMR period (31 days) (see Section 5.3, Sales & Manufacturing Review (SMR)) is set to be a fraction of the operations plan for that period; e.g. if the operations plan for that period is set to 40 and the *PRODUCT_order_fraction* is set to 0.75 then, in the absence of any deviation, 30 system orders will be received in that period. The order pattern is normally distributed, the mean of which is *PRODUCT_order_fraction**operations plan for that SMR period.

PRODUCT_order_std_dev

This is the standard deviation for the normally distributed order pattern.

PRODUCT_order_start_day

Model simulations assume that the factory is 'ramped-up' from zero; i.e. there is no inventory and hence no manufacturing capability for a period of time (equal to the longest procurement lead-time for a constituent part of the PRODUCT system). This variable allows the day on which orders first arrive at the factory to be set; i.e. the first day on which it is 'open for business'.

PRODUCT_random_order_probability

Random orders of larger quantity may be introduced into the order pattern using this variable. *PRODUCT_random_order_probability* sets the probability of such large random orders; e.g. if set to 50, on average, there will be a larger order one day in 50. The quantity of this larger order is also random and may be 3, 4 or 5 times that dictated by the normal order distribution. In a similar way, the quantity of ship loose orders may be specified and the process model for this is depicted in Figure 6. Ship loose quantities are specified in terms of modules (boards).

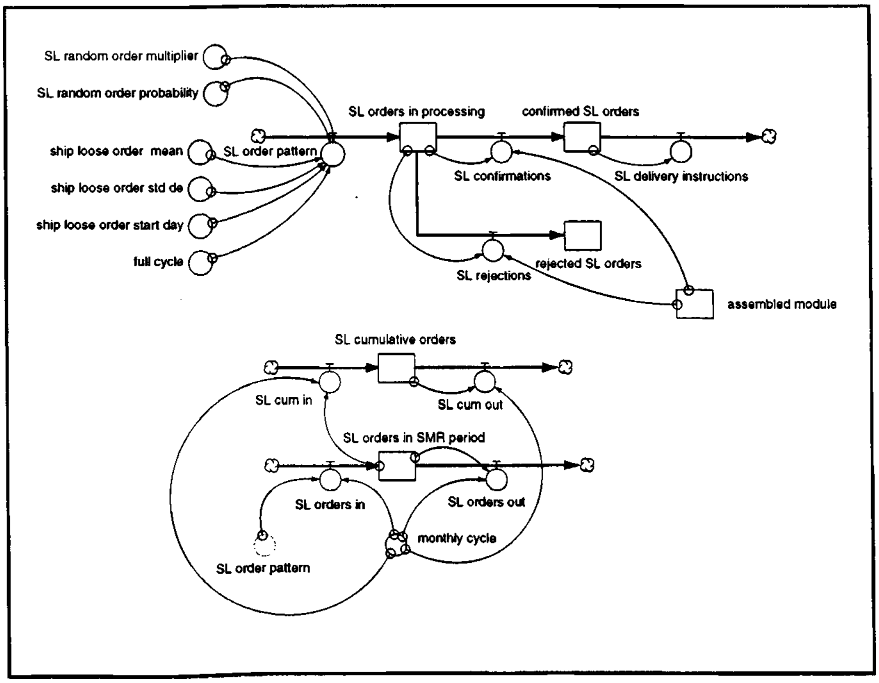


Figure 6: Order Model–Part 2.

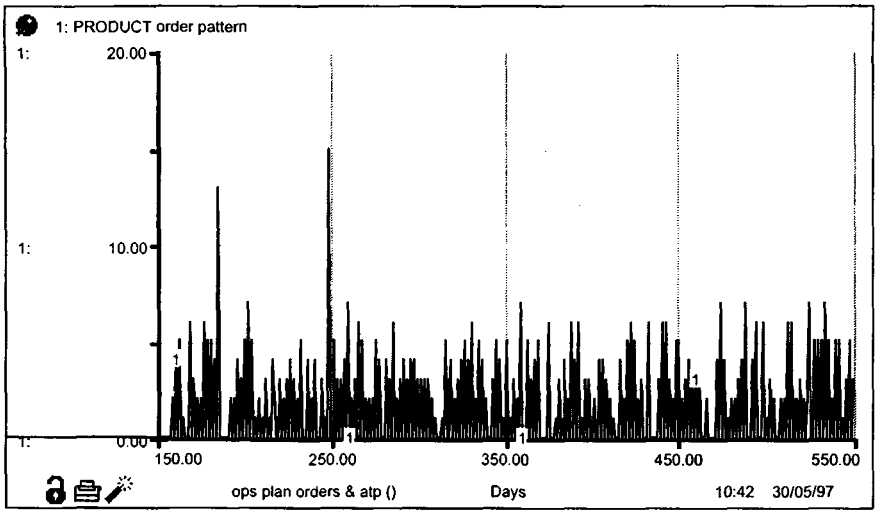


Figure 7: Typical PRODUCT Order Pattern

In Figure 7, a typical PRODUCT order pattern is shown with the following variable values:

PRODUCT_order_fraction = 0.75 (of operations plan of between 60 and 75 systems per SMR period).

PRODUCT_order_std_dev = 2

PRODUCT_order_start_day = 156 (5 SMR periods into factory operation)

PRODUCT_random_order_probability = 50

PRODUCT orders are entered into the Manufacturing Management System (MMS) according to there being sufficient *available_to_promise* (ATP —see Section 5.2, Master Production Scheduling (MPS)). If there is insufficient ATP, the MMS will re-date the order based on future ATP. At this stage the order has not been confirmed to the customer, it resides on the MMS as a "simulated" order. At the point of order entry, a template of basic PRODUCT system information is completed. To confirm the order, the Master Production Scheduler uses this template information. The fact that the order is on the MMS indicates sufficient ATP and if the template does not indicate that the system is a "special" (i.e. a system containing unusual or difficult to procure parts) then the Master Production Scheduler confirms the order and delivery date. At this point the order becomes "active" on the MMS. If the system is a "special", the Master Production Scheduler will specify a delivery date and communicate this to the order desk who will seek acceptance from the customer.

Through this process, orders are confirmed using ATP and template data and confirmation does not depend on there being a complete hardware configuration available.

The model assumes that the time taken from order entry to order confirmation is 3 days.

5.1.2 ATP Redating

The process of ATP redating is used to out-schedule orders that arrive and cannot be immediately satisfied due to existing order book commitments. The order desk and MPS functions jointly own the process.

As mentioned in Section 5.1, if an order quantity cannot be satisfied due to insufficient ATP (due to previously accepted orders and hence committed procured inventory) then the order delivery date is rechecked against the ATP before representation to the customer. The process simply involves using the MMS to look forward in the operations plan to a date where there will be sufficient inventory procured to meet (in full) the anticipated order. If the delivery date attained from this process is acceptable to the customer, then the order becomes simulated and is treated as in Section 5.1.

The modelled ATP redating process works in a similar manner to the real process through using the future operations plan for determining the next available delivery date for a particular order.

The model varies from the actual process in the following ways:

- There is no prioritisation of orders i.e. orders from particularly favoured customers cannot have precedence over other orders.
- Orders for redating are processed in a strictly FIFO manner.

The simple algorithm adopted for the redating only redates an order to a particular operations period if the full order quantity can be manufactured in that period. ATP figures from successive periods are not accumulated in order to redate.

For example, an order arrives for 16 units.

Current ATP (period N) = 3 units.

ATP for period (N+1) = 13 units.

ATP for period (N+2) = 20 units.

In the real process, this order could be fulfilled by the end of period (N+1), since the two successive periods' ATP equals the order quantity. In the model, the order can be redated but cannot be fulfilled until period (N+2) where the ATP figure exceeds the order quantity. If an order cannot be redated in any operations plan period, then it is rejected.

5.2 Master Production Scheduling (MPS)

5.2.1 Description

The main input to the modelled MPS process is the monthly operations plan that is a rolling twelve-month production plan. See Figure 8. This operations plan is agreed at the SMR meeting (Section 5.3, Sales & Manufacturing Review (SMR)).

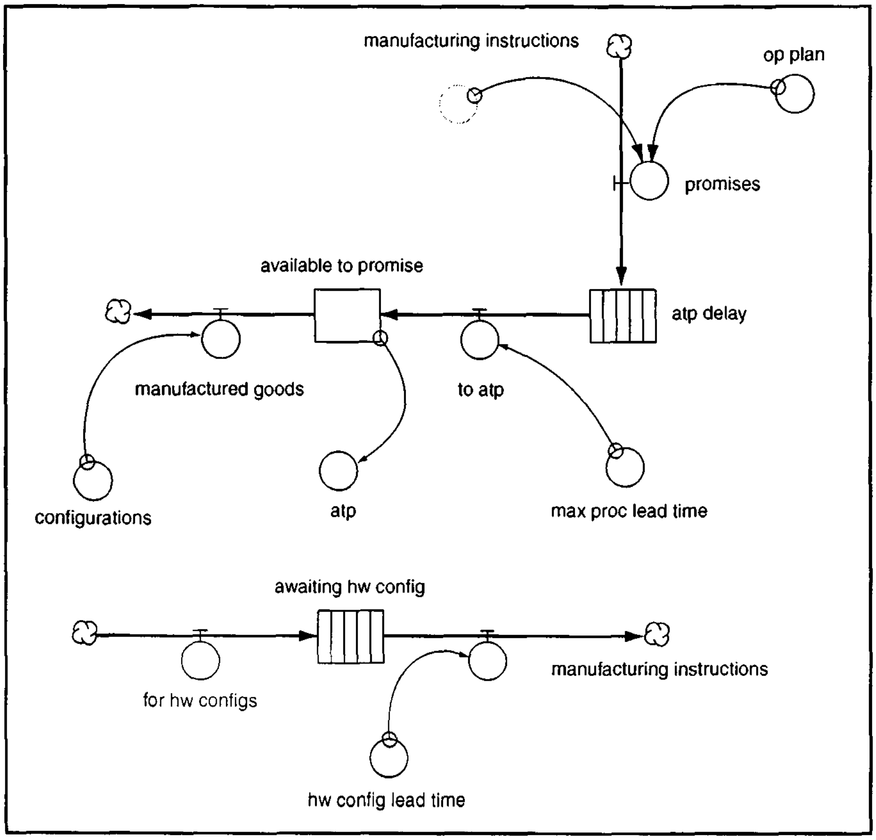


Figure 8: Master Production Scheduling Model

The main output from the MPS is the 'Available to Promise' (ATP) figure for the PRODUCT range. The ATP figure gives a rough indication at any given time of the company's ability to manufacture PRODUCT systems. This figure is calculated from a stock-perspective and does not take into consideration the following factors:

- Manufacturing capacity.
- Configuration capacity (both hardware and software).
- Impact of other product ranges on ability to manufacture PRODUCT systems.

The ATP figure is calculated according to the following equation:

```
ATP = operations plan - confirmed system orders - delivered systems
```

The only factor that increases the ATP is the operations plan figure that occurs monthly. The pull on ATP comes from confirmed orders and delivered systems that can occur on a daily basis. A reservoir stock construct has been used to model ATP since the ATP value is cumulative (i.e. any additional ATP arising from a previous months operation plan which has not been committed to received orders is carried over until consumed). The stock-based perspective of ATP is based purely on the figures in the operations plan, which are based around generic PRODUCT systems. To account for the 'ramping-up' effect of the model (see Section 5.1.1, Description), the model ATP equation does not become active until the first procurement cycle has been completed; i.e. there can be no realistic ATP until there is sufficient initial inventory to satisfy the operations plan figure, so the ATP figure is held at zero until this occurs. Once the initial ramp-up procurement period has expired, the ATP equation shown above is used (where no detailed check on inventory availability is used). The model has the ramp-up procurement period set initially to 100 days (approximately the longest lead-time for any item in a PRODUCT system).

5.3 Sales & Manufacturing Review (SMR)

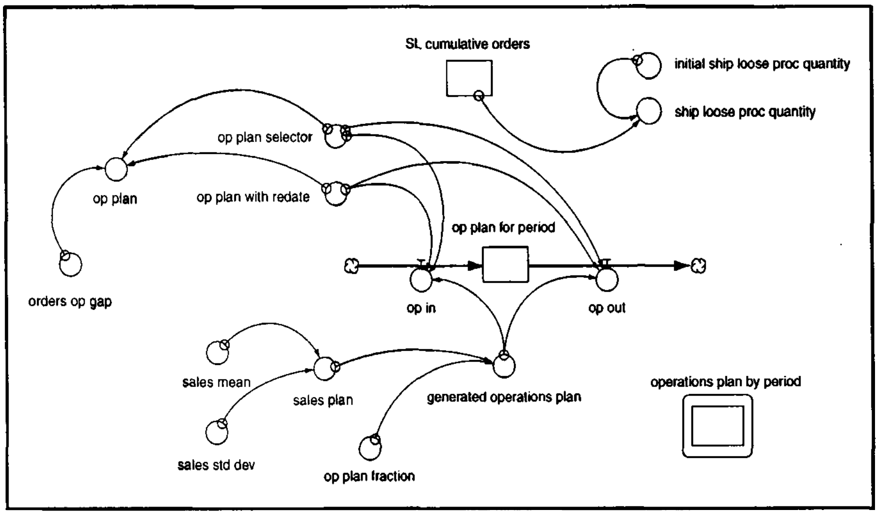


Figure 9: Manufacturing & Sales Review

5.3.1 Description

The SMR (Sales and Manufacturing Resource) process is key to planning procurement and production operations (Figure 9). The SMR process is essentially a monthly meeting where sales forecasts are used to determine a manufacturing operations plan. The procurement policy for the PROD-

UCT is to procure to the operations plan. The operations plan is signed off by the Managing Director as authorisation to initiate procurement.

Representation at the monthly SMR meetings varies, but a typical meeting is attended by:

Sales Director

Operations Director

Master Production Scheduler

Procurement Executive

Demand Manager.

The *op_plan* can be generated within the model in two ways:

i By specifying the *sales_plan* in terms of a *sales_mean* and a *sales_std_dev* (normal distribution) and then multiplying the *sales_plan* by *op_plan_fraction* in order to reflect that often the SMR process does not generate an *op_plan* to support fully the *sales_plan*. (*op_plan_selector* = 0).

ii By taking real *op_plan* figures from the SMR documentation. (*op_plan_selector* = 1).

In Figure 10 is shown a typical operations plan generated from an *op_plan* with the following parameters:

sales_mean = 50

sales_std_dev = 2

op_plan_fraction = 0.7

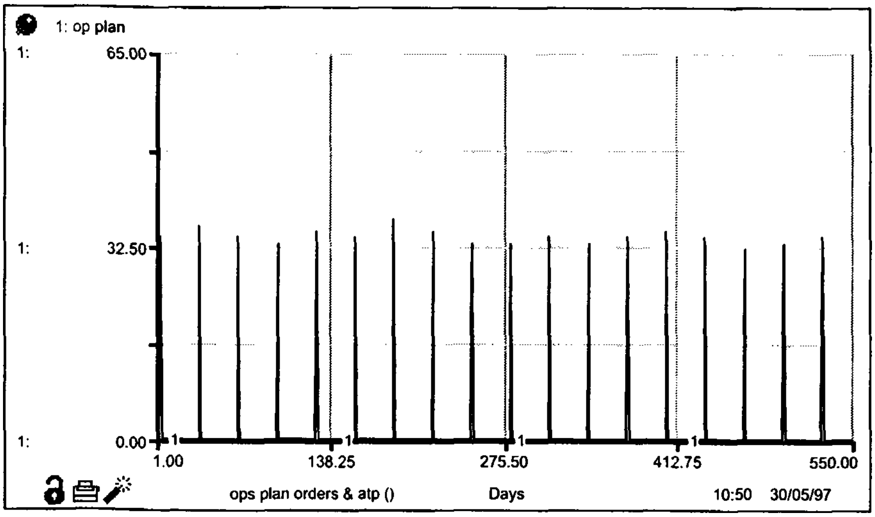


Figure 10: Typical Operations Plan

In reality, an operations plan such as that shown in Figure 10 would not remain fixed over the period of a number of SMR meetings. It is one of the roles of the SMR process to adjust the operations plan in light of the sales forecasts, number of confirmed orders and number of manufactured systems. The model allows for operations plan adjustment via a very simple feedback loop in which order trends are used to either increase or decrease the operations plan. Figure 11 shows the operations plan adjustment in the situation where orders are, on average, lower than sales forecasts (and hence operations plan). It can be seen that the operations plan shows a general downward trend. In the situation where orders are, on average, greater than sales forecasts (and hence operations plan), the corresponding operations plan shows an upward trend.

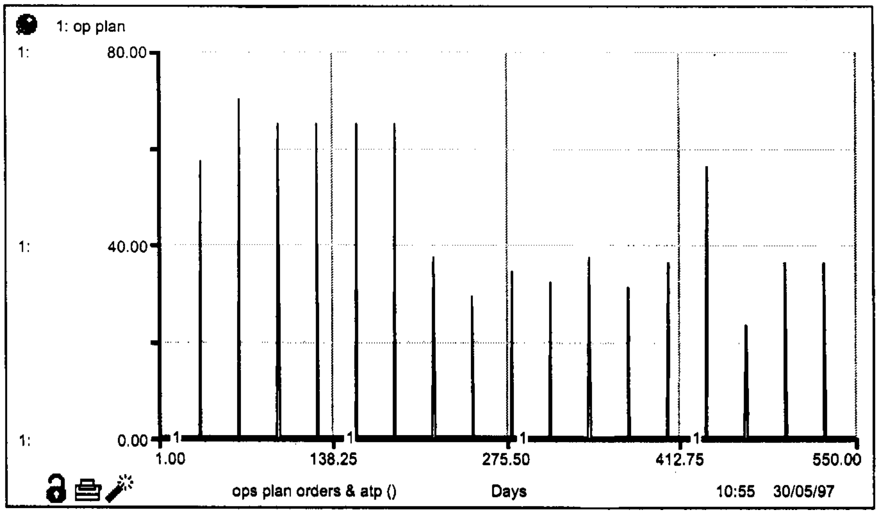


Figure 11: Typical Operations Plan

The feedback loop for the operations plan may be incorporated or not by the use of the *op_correction* switch. Setting *op_correction* = 0 disables the feedback and setting *op_correction* = 1 enables the feedback.

5.4 Materials Requirements Planning (MRP)

5.4.1 Description

The MRP process is used to generate procurement requests from the operations plan and inventory allocations for confirmed orders. The modelled MRP process is extremely simple and generates procurement instructions for complete PRODUCT systems and shipped-loose orders (Figure 12). These procurement instructions are then passed to the procurement model and sub-model (see Section 5.5, Procurement). The two inputs to the MRP process are the operations plan and the shipped-loose procurement quantity. The operations plan figure is used directly to produce a demanded systems procurement figure. A ship-loose procurement order is automatically raised each month and is equal to the ship loose demand from the previous SMR period.

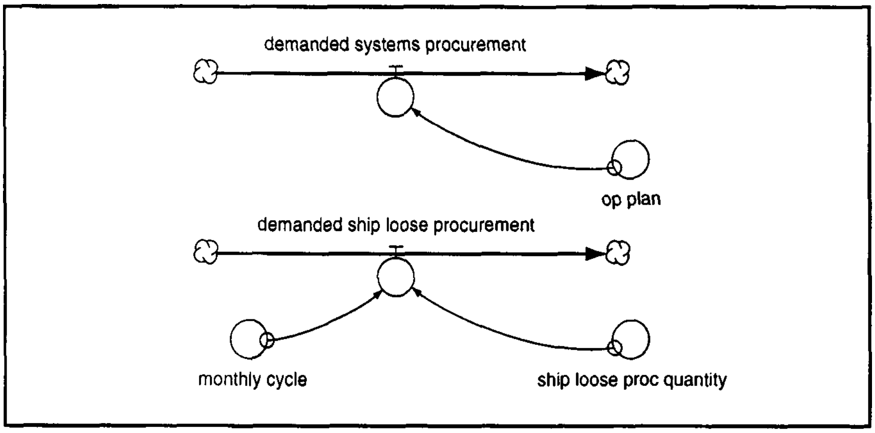


Figure 12: Materials Requirements Planning Model

5.5 Procurement

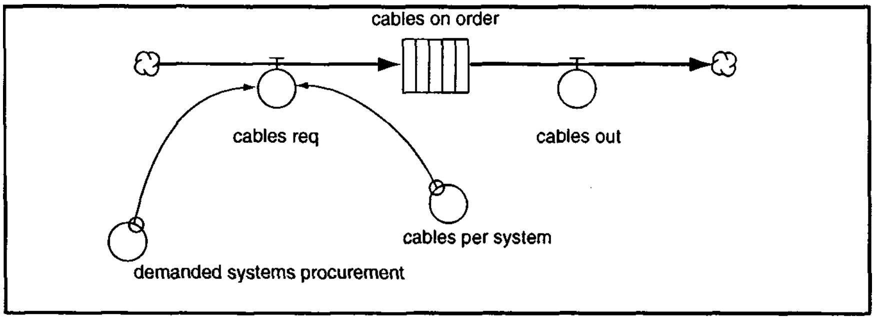


Figure 13: Procurement Model

5.5.1 Description

The procurement process is responsible for supplying the company's manufacturing capability with the components and other raw materials that are needed in support of the PRODUCT range (Figure 13). Orders to procure material can either be dependent or independent-demand. The dependent demand orders typically originate from systems 'orders' included in the operations plan and, to a certain extent, the scheduling is controlled by agreement between the procurement function and the MPS. Certain high-cost items from the PRODUCT bill-of-materials that would typically be included with the dependent demand driven purchase orders are processed manually. Independent demand is normally generated from the MMS. This demand is usually dictated through re-order point control (ROPC). Components ordered according to this method are usually high-volume, low-cost items. There are other mechanisms within the company by which purchase orders may be raised , but these do not significantly contribute to the PRODUCT procurement process.

The modelled procurement process includes a number of simplifications. These include:

- Only modelling the PRODUCT procurement. Any impact of procurement processes for other products upon the effectiveness of the PRODUCT procurement processes was ignored.
- There is no adoption of a 'procurement window' within the current model.[3]
- The PRODUCT was modelled only to the first level of the PRODUCT bill-of-materials. The following items combine to form a complete PRODUCT system:

    Cabling

    General Documentation

    Configuration Documentation

    Connectors

    Modules

Each sub-assembly has a separate procurement operation which permits the use of different operating characteristics; e.g. the procurement lead-time for cabling can be different to that defined for general documentation, connectors etc. The settings used in the process model to simulate the PRODUCT procurement process are shown in Table 1.

5.5.2 The 'Generic' PRODUCT System

Although each PRODUCT order is usually unique (i.e. the production processes are predominantly configure/assemble to order), the commonality of sub-assemblies required for specific orders allows an average system bill-of-materials to be defined—the 'Generic System'. This generic system, illustrated in Table 1, is used to drive the procurement orders raised as a consequence of the operations plan. The template used for the generic system is updated normally on a weekly basis from historical data (in the real-

[3] In reality, the procurement process does not automatically track the decisions made in the determination of the operations plan. Although the operations plan is generated for a rolling twelve-month production period, it is accepted that the procurement orders are only released to a point in time where confidence in the predicted order level is high, and such that held stock levels can be maintained at an economic level. The setting of the 'procurement horizon' is also made with consideration to the order response rate and product lead-time.

world process) but is fixed within this model. The factors that control the generic system within this model can be set within the flight simulator.

Sub-Assembly	Number per Generic System	Procurement Lead-time	Procurement Lead-time standard deviation
Cabling	1	50	2
General Documentation	1	10	1
Config Documentation	1	20	1
Connectors	1	50	2
Modules	1	max_proc_lead_time	3

Table 1: Procurement Details Used in Process Model Simulations

5.6 Inventory

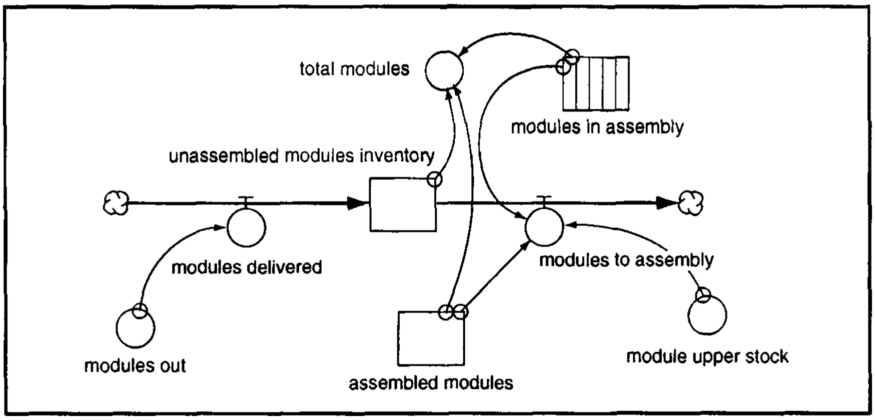


Figure 14: Inventory Model

5.6.1 Description

The receipt and storage of procured materials and components has been modelled collectively as 'Inventory' (Figure 14). The Inventory area serves only to hold material until requested by the manufacturing processes or independent ship-loose demand. The inventory process is contained in an *ithink* sub-model in order to mask the detail from the main model. The sub-model contains five similar process branches to represent the inventory levels of the five PRODUCT sub-assemblies: Cabling, General Documentation, Configuration Documentation, Connectors and Modules. Each sub-model process branch operates in the same manner: sub-assemblies arrive from suppliers (goods receipt modelled simply as procurement orders delayed by a procurement lead-time) and arrive in a unique inventory bin for each sub-assembly. Each inventory bin is depleted by manufacturing requests, whereby a number of sub-assemblies is withdrawn from the inventory stocks to satisfy an order for a specified quantity of generic PRODUCT systems. The only variance in the inventory sub-process model occurs in the issue of Module inventory. The flow of Module inventory is controlled purely by the level of the Modules safety stock buffer. The Inventory sub-model is also used to generate a number of statistics that are used elsewhere in the model. These values are:

total_modules: the number of modules (and potential modules) in the process. This figure includes finished modules and work-in-progress.

assembled_available_kits: the number of complete generic systems, which can be manufactured with existing sub-assembly inventory. This excludes Module inventory that exists as work-in-progress.

total_available_kits: (used solely for monitoring purposes in simulation). As above, but includes all Module inventory including WIP.

5.7 Configuration

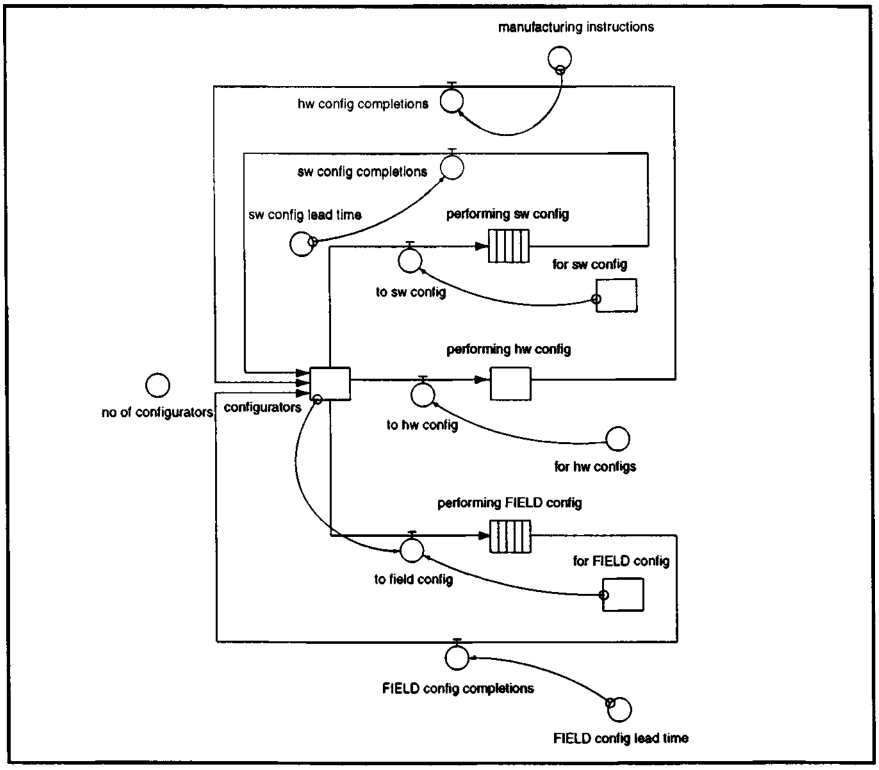


Figure 15: Configuration Model–Part 1.

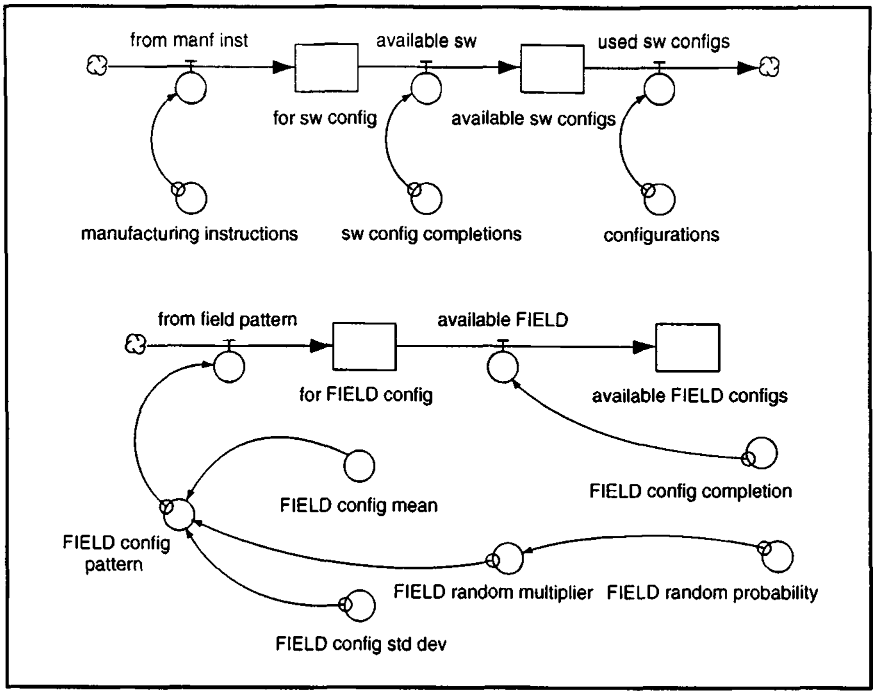


Figure 16: Configuration Model–Part 2

5.7.1 Description

Hardware and software configuration are important processes for fulfilment of PRODUCT orders. A PRODUCT system requires a hardware configuration (a detailed bill-of-materials on the MMS) prior to system manufacture (assembly) and a software configuration prior to delivery to the customer. There is a limited resource (configuration engineers) to carry out this work and in addition to the PRODUCT configuration activities there is demand for FIELD – ISDX software reconfiguration. For this reason the FIELD – ISDX reconfiguration load is modelled.

The configuration model is based around the available configuration resource, *configurators* (Figures 15, 16). It is assumed that one configurator (i.e. a configuration engineer) is able to produce a PRODUCT hardware configuration, a PRODUCT software configuration and an FIELD – ISDX reconfiguration; i.e. the configurator is multi-skilled and can perform one of these tasks at any one time. The *configurators* resource is allocated based on confirmed PRODUCT orders (which initiates the generation of a hardware configuration). Once this hardware configuration is available and there is sufficient inventory, a set of *manufacturing_instructions* is generated which initiates the generation of a software configuration. In addition, it is possible to specify a loading for FIELD reconfiguration (PRODUCT already deployed which requires reconfiguration as part of an upgrade) in the same manner as the *PRODUCT_order_pattern*, normally distributed with *FIELD_config_mean*, *FIELD_config_std_dev* and *FIELD_random_probability*.

configurators = 14

hw_config_lead_time = 2 days

sw_config_lead_time = 1 day

FIELD_config_lead_time = 5 days

It is assumed that it takes one configurator these times to produce one configuration per system and that all systems require individual configurations; i.e. if on one particular day an order is received for 5 PRODUCT systems it, 5 configurators will be required to produce the 5 (individual) hardware configurations. Also, 5 configurators will be required to produce 5 (individual) software configurations.

Figure 17 shows the number of available *configurators* over time, as they are required to perform hardware and software configurations in response to a *PRODUCT_order_pattern* in the absence of any FIELD reconfiguration loading.

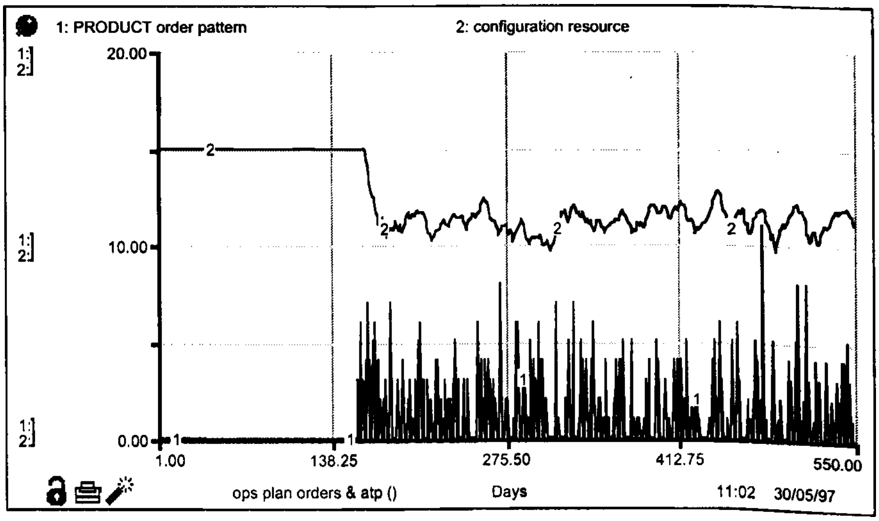


Figure 17: Typical Configuration Engineer Resource Usage

In Figure 18 is shown the available configurator resource with a FIELD reconfiguration demand loaded over the *PRODUCT_order_pattern* depicted in Figure 17. The FIELD reconfiguration demand is defined by the following parameters:

FIELD_config_mean = 10
FIELD_config_std_dev = 2
FIELD_random_probability = 50

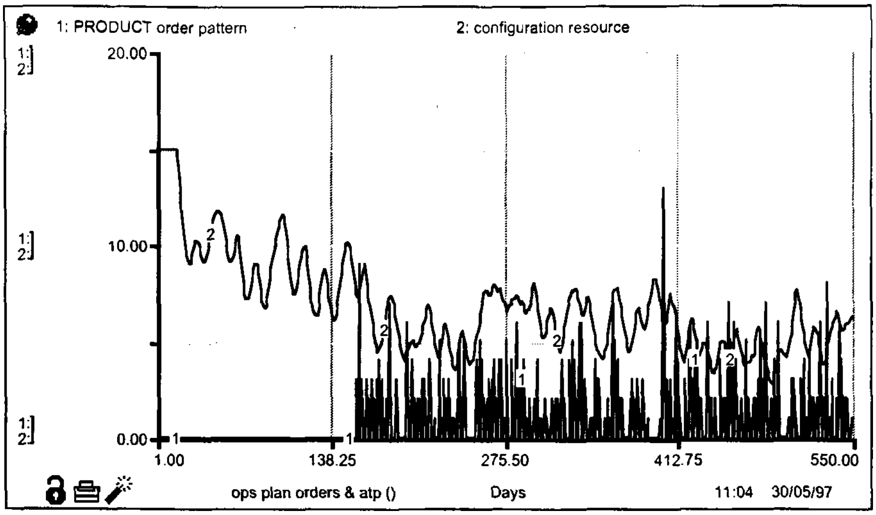


Figure 18: Typical Configuration Engineer Resource Usage

Within the configuration model there is no concept of prioritisation of configuration effort. *Configurators* are allocated to tasks as demanded without regard to when their results are required; i.e. 'as it lands on the desk'. The configuration model impacts other areas of the model since *manufacturing_instructions* cannot be issued without a hardware configuration being available and assembled systems cannot be delivered until a software configuration is available and loaded. Unavailable hardware and software configurations give rise to increased *delivery_lead_time.*

5.8 Sub Assembly

5.8.1 Description

The Sub-Assembly process is concerned primarily with the production of modules to satisfy demand from both the systems and ship-loose orders. A module safety stock buffer is used to regulate the supply to both manufacturing and ship-loose. Production orders for modules are raised whenever the level of modules in this buffer falls below the specified safety level. The sub-assembly process model contains a representation of the module safety stock buffer. The safety buffer is replenished from the Inventory area whenever the safety level has been breached. At present only a minimum safety level is specified for this buffer; this level can be set from the flight simulator control panel. The pull on the module stock buffer can be from either ship-loose demand or from system demand. System demand (in the form of manufacturing instructions) pulls off the required modules to satisfy the defined number of generic systems. Ship-loose demand can be of any size. The Sub-Assembly Model is shown in Figure 19.

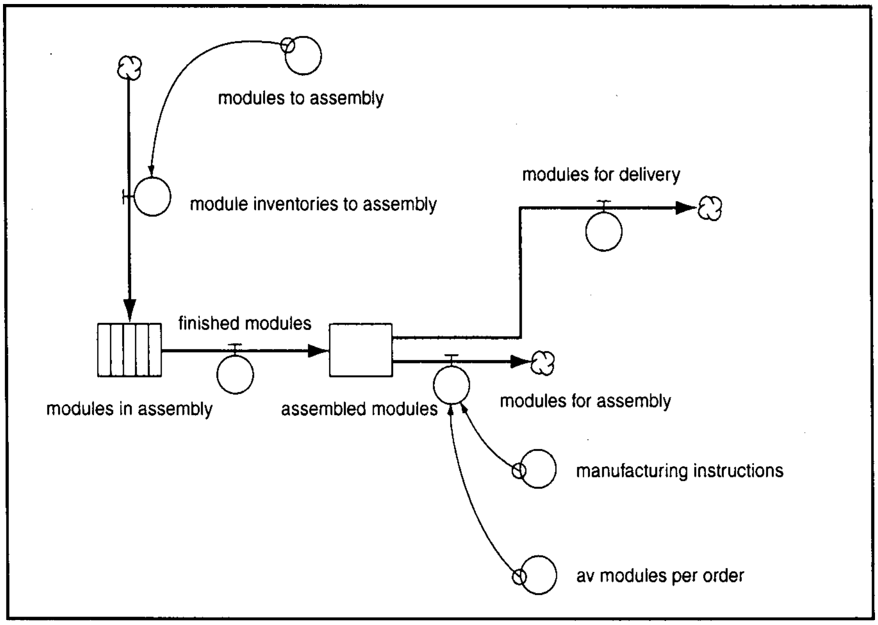


Figure 19: Sub Assembly Model

5.9 System Assembly

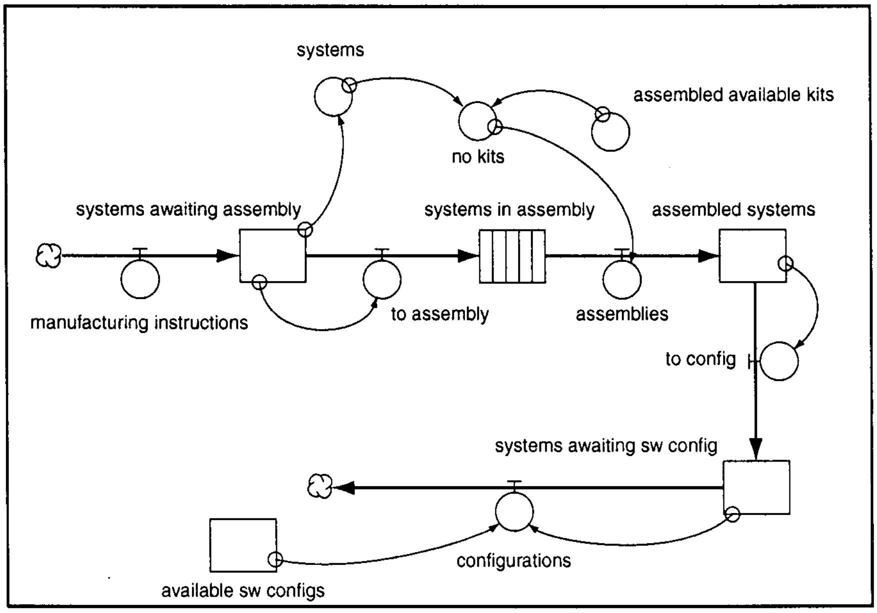


Figure 20: System Assembly

5.9.1 Description

The system assembly process (Figure 20) represents the manufacturing operations where hardware sub-assemblies and unit software are configured (assembled) to fulfil a particular order. The flow of system orders through this process is essentially constrained by the availability of sufficient system kits and the number of available software configurations. The systems orders are processed in the model in a strictly first-in-first-out (FIFO) order. No orders can proceed unless the previous order has been satisfied. There is no prioritisation of orders allowed in the model, an order already in the process cannot be removed or cancelled. The time required to fully assemble the hardware required for a PRODUCT system is set to six days in the process model. After hardware assembly, the systems are sent for software configuration. The model contains the stock *assembled_systems* merely for simulation monitoring purposes; i.e. to specifically monitor the "number of assembled systems from hardware configuration" without reference to software configuration.

5.10 Warehouse

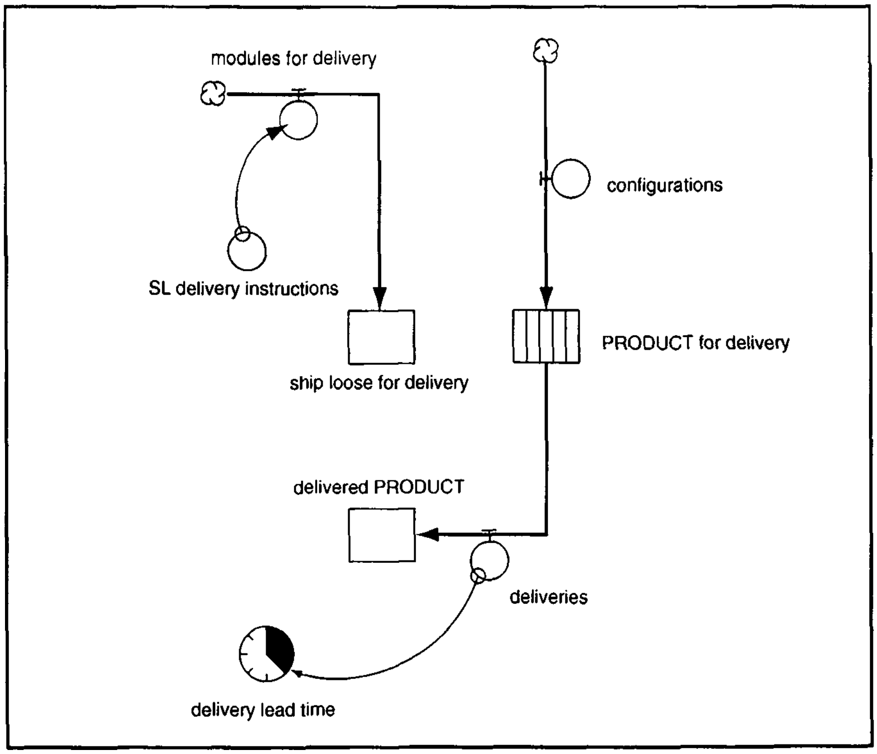


Figure 21: Warehousing Model

5.10.1 Description

After system assembly and software configuration, systems are sent to the warehouse for packaging etc. and subsequent delivery to the customer. The warehouse processes (Figure 21) are not modelled in detail. This part of the model simply includes a time associated with the warehouse processes. The model assumes that the warehouse processes take two days. The *delivery_lead_time* variable measures the time taken for a particular order to reach delivery from initial order inquiry.

6. Results of Study

The study gave rise to many recommendations for improvement of the order fulfilment processes, particularly the SMR process which was seen to be key to the company's manufacturing operations. Many of the recommendations were for organisational change or re-alignment and as such were not a direct result of model simulations. The model building activity, in conjunction with knowledge of the OPEN*framework* methodology, forced the authors to seek information and solutions outside of what was possible to model and simulate. The model lent credibility to the recommendations and proved to be an invaluable catalyst for discussions within the company. Particularly important was the ability to simulate realistic operating scenarios and to provide a range of what-if? analyses far beyond that possible with manual methods alone. The essence of the recommendations for process improvement was as follows:

- The order fulfilment processes were individually well understood but their interaction and contribution to the business as a whole were not. A division of process ownership between function areas within the company exacerbated this problem.
- The SMR process should be widened in scope to perform company-wide requirements planning to ensure that the business goals are met. This would broaden the scope of the current SMR process that sees its function as only producing operations plans.
- Business goals should be translated into metrics by which the performance of the SMR process can be constantly measured.
- The company and not a functional area of it should own the SMR process. This should be reflected in its compulsory attendance list.
- Sales forecast accuracy is key to the performance of the manufacturing operations. Sales forecast figures should be visible at the SMR meetings and included in a formal feedback loop to adjust operations plan levels (as opposed to current ad-hoc methods).
- Hardware and software configuration activities and their resource levels play a key part in determining manufacturing lead-time and should be included in the SMR planning activity. To allow this, the

ownership of these processes should change to allow the SMR process to determine configuration resource levels.

- Order confirmation based on available-to-promise will lead to inventory level problems in the event that sales actually meet forecast demand and ship loose demand remains uncoupled from the determination of the operations plan.
- A tool such as *ithink* should be used as part of the SMR process to perform continuous what-if? analyses.

7. Conclusions

The recommendations of the study were well accepted and are being incorporated into the company's on-going process improvement initiatives. Knowledge of the OPEN*framework* methodology enabled the study to be carried out in a structured manner and although the company did not have day-to-day visibility of it, their knowledge that the study was being carried out within a well established framework gave them confidence that the recommendations would be well founded. Using the principles of OPEN*framework* enabled the modelling study to be kept in perspective with the overall aims of the company and was particularly useful in producing a realistic and acceptable scope for the study. The *ithink* modelling tool only allowed coverage of some of the aspects of the study; being able to visualise its position within the OPEN*framework* meta-model made it possible to better relate the modelled information to some of the more subjective aspects of the study such as personal motivations, organisational structures and the use of the supporting IT. Without knowledge of the OPEN*framework* methodology, the study would not have been as successful as it was since it would have been easy to consider the modelling exercise and its resultant model as the sole reason for carrying out the study. The OPEN*framework* methodology ensured that the goals of the study were kept firmly in view and that the modelling exercise provided appropriate support in achieving them. Whilst much of the data gathered during the study was specific to the particular company, the study provided much valuable generic process information to further the development of the Generic Manufacturing Architecture.

Acknowledgements

The Author would like to thank the CDP directorate of the EPSRC and the members of the MSI Research Institute for their continued support of this research work.

Bibliography

FOX, M.S., "The TOVE Project: A Common-sense Model of the Enterprise," Proceedings of the International Conference on Object Oriented Manufacturing Systems, Calgary, Alberta, Canada, 1992.

HPS, High Performance Systems, Inc., 45 Lyme Road, Suite 300, Hanover, NH 03755 Phone: (800) 332-1202 or (603) 643-9636 Fax: (603) 643-9502, URL http://www.hps-inc.com/products/ithink/ithink.html.

ICL, "OPEN*framework* – The Systems Architecture: An Introduction," Prentice Hall, 1993.

ICL, "ProcessWise Workbench User Guide" – ref. PWB/usrguide/P5.4, issue 1.0, Process Management Centre, Forest Road, Feltham, Middx, TW13 7EJ.

LENAT, D.B., "CYC: A Large-Scale Investment in Knowledge Infrastructure," Communications of the ACM, 38, no. 11, November, 1995. (See also other articles in this special issue).

LIBES, D., "The NIST EXPRESS Toolkit: Introduction and Overview," Don NISTIR 5242, National Institute of Standards and Technology, Gaithersburg, MD, 1993.

MURGATROYD, I.S. AND GILDERS, P.J., "OPEN*framework* Meta-Model," MSI Research Institute, Loughborough University, Loughborough, UK.

NIST, "Integration Definition for Information Modelling (IDEF1X)", Federal Information Processing Standards Publication, 184, National Institute of Standards & Technology, Gaithersburg, USA.

RICHMOND, B., "System Dynamics/Systems Thinking: Let's Just Get On With It," International Systems Dynamics Conference, Sterling, Scotland, 1994.

STADER, J., "Results of the Enterprise Project," Proceedings of Expert Systems '96, the 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge, UK, December, 1996.

Biographies

Shaun Murgatroyd

Shaun Murgatroyd graduated in Mechanical Engineering from Leeds University in 1985. After positions with the UK Atomic Energy Authority and American Can UK Ltd, he joined the Department of Manufacturing Engineering at Loughborough University in 1987 as as Research Assistant. He has worked on a number of manufacturing systems integration research projects in collaboration with industry involving significant secondment periods at collaborator sites. Now a Senior Research Associate in the MSI Research Institute, his expertise lies in the application of modelling methodologies, tools and techniques within manufacturing industry.

Roy Smethurst

Roy Smethurst joined ICL (then English Electric Leo Marconi) Kidsgrove in 1964 after graduating from Nottingham University with a BSc in Mathematics. He worked initially on Compilers (KDF6, KDF7, KDF9, System 4 COBOL and PL/I). Subsequently he moved to a system design role on Series 39 VME, where he was the design authority for VME's SCL System, Work Management, Spooling and File Transfer. In 1991 Roy joined the OPEN*framework* architects and produced architectures for the Availability quality, CALS (Continuous Acquisition and Life-cycle Support), Manufacturing Systems and Multimedia. For the last three years Roy has been an OPENframework Consultant with major assignments with ICL Pathway, BZW, Avery Berkel UK and RAF Strike Command. He is a member of the BCS and a Chartered Engineer.

Demystifying Constraint Logic Programming

O.V.D.Evans

Research and Advanced Technology, ICL, Bracknell, UK

Abstract

This short paper is intended to refute the view that logic programming languages such as Prolog are esoteric and therefore inaccessible to anyone with a grounding in conventional sequential programming. By first demystifying logic languages through a series of worked examples the groundwork is then laid for explaining, again with examples, the data-driven processes that form the basis for the power and conciseness of logic programming languages, such as the ECL^{i}PS^e platform described elsewhere in this issue.

1. Why logic programming

The ECL^{i}PS^e platform supports a logic programming language paradigm. Although a number of constraint programming systems exist which are independent of logic languages, in the particular case of ECL^{i}PS^e some grounding in logic languages is essential for a good understanding of how constraints work and what their benefits are. The basic principles of logic programming are first introduced with a number of programmed illustrations. A second subsection includes a systematic description of a complete worked example. Since this description aims to demystify the art of constraint logic programming its theoretical aspects have been excluded, as far as possible, in order to concentrate on the practical task of producing useful programs. The use of examples for describing the operation of the various features of ECL^{i}PS^e conforms to the style adopted in the relevant reference manuals. Throughout this work the assumption is made that the reader has encountered conventional programming languages at some stage.

2. Logic programming basics

Logic programming is the process of constructing a logical expression and evaluating it for consistency for a given set of value assignments to its terms. The evaluation of a logic program is normally called a query because what is being done is to test whether a particular assertion, the query, is consistent with the facts and rules that make up the program. The answer to the query is therefore either logically *true* or *false*. If the query succeeds it will return the answer *true*, but in the course of succeeding it will have given values to various variables of interest and possibly completed other actions—all as *side effects*. What the query will have evaluated is in essence a single large logical expression consisting of instantiated variables and the logical operators *and*, *or* and *not*. The significance of regarding a logic program as a single logical function is that such an expression is true or otherwise regardless of the order of execution of its terms—in other words it is completely declarative. Since, in practice, the program interpreter needs to execute the expression in some systematic order, much of the development of logic programming languages has centred around eliminating any deficiencies that could arise due to order-dependent execution. The most notable features contributing to order-independence being the concept of backtracking so that disjunctive conditions can be explored, and the delayed execution of expressions that still include variables that are instantiated at some later point in the predicate firing order. It will be seen later that constraint logic programming is a logical extension of the process.

Any practical realization of logic programming as exemplified in a language such as Prolog will have a syntax that enables such logical expressions to be structured into more digestible sub-expressions. The so-called *Horn-clauses* in Prolog, and used in ECL^{i}PS^e, are such sub-expressions, each representing one of the rules that has to be true for the query to succeed in a way that helps to distinguish the scope of variables local to a particular rule from those shared with other rules. These rules can be nested and recursive. Typically a logic program will consist of a top-level conjunction of rules (called *predicates* in Prolog) each of which consists of one or more sets of *goals* which can in turn invoke other predicates.

Since logic programming languages are declarative this means that the control structures normally used in sequential languages to handle repeated operations are replaced by predicates containing goals defining the non-end and end conditions of any such activity. In working through a list, say, the non-end goal will repeatedly call itself recursively until it fails—but at this point the end condition goal will hold and the predicate will be satisfied. Another related property of a declarative programming language is that it is a single-assignment language. This means that no variable can be bound to a value more than once within a solution path—but quite clearly if alternative solutions are available through backtracking, the overall query can be re-satisfied several times with different values.

In logic programming languages a distinction is made between assignment, which is treated, if supported at all, as an extra-logical primitive, and *unification*. Unification is a completely symmetric primitive which for say, two variables **X** and **Y**,[1] will test that their respective structures are equivalent and that any individual values within the structure are identical or

[1] By convention the leading character of a variable identifier is an upper case letter to distinguish it from facts, predicates or atoms which always start with lower case letters.

become identical after local unification. The program fragment (with its result):

```
1 ?- X=[1,_,2,_,3,_,[A-5,7+C]],Y=[_,a,_,b,_,c,[6-
B,D+8]],X=Y.

Y = [1, a, 2, b, 3, c, [6 - 5, 7 + 8]]
X = [1, a, 2, b, 3, c, [6 - 5, 7 + 8]]
D = 7
B = 5
C = 8
A = 6
yes
```

is an example, first of the unification of `X` and `Y` respectively to the list structures shown which contain a mixture of integers, atoms (`a,b,...`), uninstantiated variables (`A,B,_,  ...`), and a sublist—and then to each other. The result illustrates that variables can be bound to values wherever they occur. It can be noted that operators such as + and - have no meaning in logic programming and are merely treated as symbolic literals. In this relatively simple example the unification algorithm has worked out that the conjunction of the three unifications is consistent with the value bindings listed and therefore returns the answer 'yes'. Any contradictions will cause failure.

A more useful example is given in the following complete predicate. This shows how rules are invoked by a query, in this case by applying unification to evaluate the logical outcome of a series of comparisons to bind the correct value to the query variable. In this case the program consists of a small database of five facts **`height/3`** and one rule **`contour/2`**, where the number after the name is the customary form of specifying the number of terms governed by the predicate, also known as its "arity".

```
height(0,500,green).
height(501,1000,yellow).
height(1001,1500,brown).
height(1501,2000,purple).
height(2001,3000,white).

contour(Height,Colour):-
     Height>=LV, Height<HV,height(LV, HV, Colour).
contour(Height,off_map):- Height<0.
contour(Height,off_map):- Height>3000.
```

In this case the query asks what colour of contour band (on a map) corresponds to the query height. When the query **`contour(2444,C)`**. is entered the following steps are executed by the logic language interpreter:

1. The first goal of the **`contour/2`** predicate is invoked.

2. The first term **`Height`** compared with **`LV`**. **`LV`** is undefined so the goal is delayed.
3. The second term **`Height`** compared with **`HV`**. **`HV`** is undefined so the goal is delayed.
4. The first instance of the fact **`height/3`** is invoked, which will cause its two first terms (0 and 500) to be unified with **`LV`** and **`HV`** respectively.
5. The two delayed goals **`>=`** and **`<`** are now woken up and will both fail.
6. The first goal of **`contour/2`** will now backtrack and invoke the next instance of **`height/3`** and reunify **`LV`** and **`HV`**. The second set of comparisons will also fail as will those involving the third and fourth invocations of **`height/3`**.
7. The fifth invocation of **`height/3`** will cause the first rule to succeed because the two comparisons will now hold, and the third term of the rule head can now be unified and the query satisfied, giving **`C = white`**.

If the query is entered with a **`Height`** value for which none of the facts hold then the first goal of the **`contour/2`** predicate will fail and the remaining two are tested in turn. This is because all three instances of the predicate are regarded as alternatives any of which can satisfy the query conditions in the same way that the facts **`height/3`** are interrogated in turn. This caters for the *or* operator in logic programming in the same way that the comma separators are equivalent to the *and* operator. It is quite feasible to write the entire program as a single logical expression—this is left as an exercise for the reader.

The example is intended to illustrate how the basic logical operators *and* and *or* are applied to resolving a query that requires both unification, goal delay and backtracking for its solution. More importantly, it also demonstrates that many operations imply unification without the need to apply the equality operator explicitly. The structure of the individual goals of the predicate is in Horn-clause form—consisting of a head, a neck and a body. The head consists of the name of the predicate and the list of terms that enable it to communicate with the rest of the program. The neck separates the head from the body and can be dispensed with if there is no body. The body of a predicate goal is any sequence of clauses separated by commas—each clause normally being either a call to a predicate or a fact.[2] An exam-

[2] Strictly speaking Prolog does not distinguish between facts, predicates and queries as all three share the same syntax. For example in foo(X,Y,Z) the variables can be free, integers, atoms, compound terms or other predicates.

ple of a bodiless goal is the end condition goal in the following predicate, where **index/3** holds if the elements of one list correspond to the elements of another list processed in some manner.

```
index([],[],_).
index([X|T1],[I-X|T2],I):- +(I,1,NI),index(T1,T2,NI).
```

The predicate will hold when the second list is a list of the elements of the first list prefixed by an ascending index and the separator '–'. There is a non-end goal and an end goal, the latter satisfying the end conditions without the need for a neck and a body. This is an example of how a repetitive task, such as the processing of elements in a list can be defined with just the two conditions—one relating some operation on the head elements of the lists with a recursive call to do exactly the same on the tails of the lists, and the end condition (both lists empty). The + function is an example of a built-in predicate that is *extra-logical*. This means that in the equivalent representation `NI is I+1` the variable `NI` is *assigned* the arithmetic sum of `I` and `1` and not the value `I+1`. Extra-logical predicates, of which all arithmetic functions are a class, are so-named because the result variable is not instantiated through unification and equally importantly because the assignment is not reversible. It will be seen later that the latter restriction does not apply when arithmetic operations are represented as constraints.

The above short description of the basics of constraint logic programming is necessarily incomplete but it should prove sufficient to guide the reader through the details of the short ECL^{i}PS^e application described in the next section as an exercise in the use of a logic programming language before embarking on any use of constraints.

3. ECL^{i}PS^e as a logic program

The notation used in the program examples of the previous section is in fact the one shared by the constraint logic programming language ECL^{i}PS^e with the majority of other Prolog-based languages. This section will use the ECL^{i}PS^e syntax to describe a working example of a program representing the behaviour of a vending machine—vending machines appear to be a popular vehicle for describing the operation of logic programs. The selection of goods, prices and stock levels have been taken from an actual vending machine. The application behaves purely as a logic program and does not depend in any way on constraints.

The complete vending machine paradigm can be described by the following logical expression:

> *A sale is achieved if a selected item exists* **and** *it is in stock* **and** *the sum tendered is greater than or equal to the item price* **and** *if the sum is greater than the price there is enough cash float to make change* **or** *the tender is returned if the item is out of stock* **or** *the tender is*

less than the price or *if the float is not enough to make change.*

If the sale condition holds then there is some additional housekeeping to perform such as updating the float and the stock of items—since the latter cause irreversible effects they can be regarded as extra-logical actions. The program, listed below, can be divided into the following parts:

1. a database of facts
2. a predicate with a number of alternative goal conditions
3. some database update predicates.

Facts are named relationships between atomic values. For a vending machine there are three sets of facts:

- a description of the items on sale
- the stock level of the items on sale
- the float, being the amount of money in the machine.

Facts can be either static or dynamic. Dynamic facts need to be specified explicitly—in this example by means of the compiler directive `:- dynamic ...` at the head of the program. The fact `goods/3` is static since it describes the permanent properties of the sale items such as price, item code and description. The remaining facts are dynamic because as sales are made the stock levels will go down and the float will increase. The facts `stock/2` relate each item code to its stock level and the single fact `c_float/1` holds the current cash balance in the machine.

The predicate `sale_achieved/4` caters for the six input conditions that the vending machine will recognise. These are respectively for the query

```
sale_achieved(Tender,Item_no,Item,Change).
```

with the variables `Tender` and `Item_no` instantiated to integer values and the returning item description, `Item` and change, `Change`:

> **`Valid sale, correct tender:`**
> The `stock/2` database is searched for a match on its first term to the value of `Item_no`. If the second term, unified to `N` is non zero, the database `goods/3` is searched for a match on its first two terms, `Tender` and `Item_no` respectively. If a match is found the third and fourth terms of the query are unified to the third term (description) of the matching `goods/3` fact and zero (no change) respectively. During the search process the individual database facts are searched for unification to the search keys. Failed matches will cause local backtracking until the matching fact is found. If the goal succeeds to this point the amount of the tender is added to the float and the stock

holding for the item concerned reduced by one. Since arithmetic addition is an extra-logical operation the built-in predicate **is** is called to evaluate the expression **c_float+Tender** in which the addition operator functions as such and not as literal and where **c_float** causes the corresponding fact to be dereferenced to its value automatically. Failure of any of the searches will cause another goal to be selected.

Valid sale, change required:
As before the stock is checked but in the case of the **goods/3** database only a match of **Item_no** to the second term is sought. If a match is found the third term of the query is unified with the third term of the matching fact. The amount of change is then calculated and checked to be less than the current float. If it is this value is unified with the **Change** term of the query and the **stock/2** and **c_float/1** facts are updated as before.

Valid item, cannot make change:
This goal caters for the condition that although the item exists and is in stock, too much tender has been input for change to be made. This goal succeeds by unifying the first term of the query (tender) with the last (change) and returning the message "No change" as the third term. No facts are updated.

Valid item, out of stock:
This choice succeeds if the query variable **Item_no** unifies with the first term of a **stock/2** fact for which the second term is zero. Again the first and fourth query terms are unified (tender returned) and an "Out of stock message" returned. No facts are updated.

Valid item, short tender:
This choice succeeds if the second term of a fact **goods/3** unifies with the query variable **Item_no** and the amount of the tender is less than the item price. The tender is returned with the message "Check price". No facts are updated.

Invalid item code:
This goal succeeds if the item code entered is outside the range covered by the set of facts **goods/3**. It is also an example of an alternative syntax for disjunctions. Instead of specifying two separate goals for checking the higher and lower limits, the built-in *or* predicate ';' can be used in the general **form (Clause_1 ; Clause_2 ; ... Clause_n)**. Again, the tender is returned with the message "Check selection" and no updates are made. In vending machines where item choice is

made by pressing a unique button this goal would be redundant.

Since the program needs to return the tender if it makes no sale, the predicate must not be allowed to fail so that all eventualities must be catered for. However since only one goal needs to be satisfied once for any particular query it is normal to add to each goal a condition that disables the query processor from returning all the possible solutions to the query. This is an efficiency measure as it prevents the query processor from "following fruitless computation paths that the programmer knows could not produce solutions" [Sterling and Shapiro, 1986]. This condition is known as the *cut* and its purpose is to act as a barrier to backtracking so that a conjunctive goal followed by a cut will produce at most one solution because it commits the solver to all choices made up to the point that the cut occurred. In the example below the cut would be added just ahead of the stop symbol in each of the goals as `, !.` where `'!'` is the symbol for the built-in cut predicate. Use of the cut predicate remains controversial and while the above description is appropriate in the current context, it should not be taken as comprehensive and the cited reference or similar should be consulted for a fuller description of its use.

The remaining predicates in the example program are used to update the fact database. Facts can be added to the database using the built-in predicate **assert/1** and likewise retracted with **retract/1**. Compiled facts have to be declared as dynamic before these predicates can be used. Here the two built-in predicates are used to retract out-of-date stock and float values and to replace them with current values. If a dynamic fact is asserted it is merely added to the database of that fact, so that in order to update a fact it is necessary specifically to retract the old copy. Since these procedures carry out program modification they should be used with caution since they can give rise to unexpected behaviour.

```
:- dynamic c_float/1,stock/2.

% Goods description database - stocked goods
goods(26,2,"Fruit pastille").
goods(28,4,"Opal fruits").
goods(24,5,"Ready salted crisps").
goods(29,7,"Wine gums").
goods(24,10,"Salt & vinegar crisps").
goods(30,13,"Crunchy bar").
goods(19,14,"Polo").
goods(24,15,"Cheese & onion crisps").
goods(20,20,"TUC sandwich").

% Unstocked goods
goods(0,1,_). goods(0,3,_). goods(0,6,_).  goods(0,8,_).
goods(0,9,_). goods(0,11,_). goods(0,12,_).goods(0,16,_).
goods(0,17,_). goods(0,18,_). goods(0,19,_).

% Stock database
stock(1,0).    stock(2,20).   stock(3,0).    stock(4,20).
stock(5,20).   stock(6,0).    stock(7,20).   stock(8,0).
stock(9,0).    stock(10,20).  stock(11,0).   stock(12,0).
stock(13,20).  stock(14,20).  stock(15,20).  stock(16,0).
stock(17,0).   stock(18,0).   stock(19,0).   stock(20,20).

% Cash float
c_float(100).

% Six vending conditions
sale_achieved(Tender,Item_no,Item,0):-
     stock(Item_no,N),N>0,goods(Tender,Item_no,Item),
     New_c_float is c_float+Tender,
     float_ud(New_c_float),stock_ud(Item_no).
sale_achieved(Tender,Item_no,Item,Change):-
     stock(Item_no,N),N>0,goods(Price,Item_no,Item),
     Tender>Price,Change is Tender-Price,c_float>=Change,
     New_c_float is c_float+Price,
     float_ud(New_c_float),stock_ud(Item_no).
sale_achieved(Tender,Item_no,"No change",Tender):-
     goods(Price,Item_no,_),Tender<Price,
     Change is Tender-Price,c_float<Change.
sale_achieved(Tender,Item_no,"Out of stock",Tender):-
stock(Item_no,0).
sale_achieved(Tender,Item_no,"Check price",Tender):-
     goods(Price,Item_no,_),Tender<Price.
sale_achieved(Tender,Item_no,"Check selection",Tender):-
     (Item_no>20;Item_no<1).

% Vending machine housekeeping
float_ud(New_c_float):-
    retract_all(c_float(_)),assert(c_float(New_c_float)).

stock_ud(Item_no):-
     stock(Item_no,S),NS is S-1,
     retract_all(stock(Item_no,_)),
     assert(stock(Item_no,NS)).
```

The example program illustrates some of the basic features of the logic programming language incorporated into ECL^{i}PS^e but clearly there are many aspects that the program does not include. Principal among these is the extensive library of built-in predicates which range from input-output control, internal database handling (of which `assert/1` and `retract/1` are only a part), term, list and string manipulation as well as a comprehensive list of arithmetic functions. These are not described in the worked example since once the logic programming aspect ECL^{i}PS^e has been demystified, access to these features becomes easy and straightforward.

4. Why constraints

The purpose of using constraint technology is as an efficiency aid in applications where in order to find a solution a search has to be made through a large number of alternative value assignments to variables. By defining in advance all the constraints governing the variables involved in such searches it is possible to eliminate all impossible alternatives before the search is started. In many computationally hard problems, this has been known to tip the balance between solvability and unsolvability. For problems known to be intractable constraints can be added progressively until a solution is obtained. Another desirable feature of constraint programming is that as the constraints are progressively set up—as a set of bounded domain variables and the relationships between them—propagation will occur every time a new constraint is created and subsequently whenever any of the variables involved undergoes a change in domain. Because this happens autonomously, constraint solvers can be said to be one of the few large-scale applications of data-flow programming where control of computation is predominantly data driven.

4.1 Constraint basics

The description by example of the operation of the constraint solver in ECL^{i}PS^e is based on some acquaintance with Prolog-like logic programming languages as described in the preceding sections. Also since the application program used as the vehicle for describing the behaviour of the ECL^{i}PS^e constraint solver is a variant of the one already described, the reasonable assumption is made that the foregoing sections have already been read and digested.

A constraint logic program written in ECL^{i}PS^e will appear superficially to be no different from one without constraints. For example, the program below can be described as holding if all its goals are true:

```
foo(X,Y,Z,...):-
        g1(...),
        c1(X,Y),
        g2(...),
        c2(Y,Z),
        c3(X,Z),
        g3(...),
        g4(...).
```

In this case however the `g...` goals are conventional goals and the `c...` goals are constraints with `X,Y,Z` being variables shared between the various goals. The difference becomes apparent when the program is executed. Once the first constraint `c1(X,Y)` has been set up and the order of execution (firing order) reaches `c2(Y,Z)`, since `Y` is common to `c1(X,Y)` and `c2(Y,Z)`, `c1(X,Y)` will be re-activated. The same process will apply when `c3(X,Z)` is reached since `Z` is shared with one previous constraint and `X` with the other. If some other constraint declared in a predicate other than `foo(X,Y,Z,...)` also shares any of the three domains through the terms at the head of the predicate then changes in that constraint will also propagate into the predicate. The constraints will execute in an order governed solely by the sequence in which the domains are referenced inside the constraints and not in the order of declaration in the program.

In using constraints to solve a combinatorial problem involving a search for a set of values that satisfies the constraint requirements the following three steps must be executed—here used in the context of a trivial scheduling example:

1. The scope of the problem is set by defining the bounds of all the domain variables involved. For example the variables `S1, S2, S3` representing say, task start times of the three tasks `1, 2` and `3` of some job, can all be given finite integer domains ranging from `0` to `100` with the command `[F, S1, S2, S3]::0..100`, where `F` is the job finishing time. The :: operator is used by ECL^{i}PS^e to define a domain and the .. to define its lower and upper limits.

2. The constraints of the problem are defined. Supposing the three tasks with durations of, say, `30`, `25`, and `28` need to be fitted into a span of 100 time units so that they are executed in sequence and their durations do not overlap, then the following precedence constraints would ensure this ...

```
S2#>=S1+30,S3#>=S2+25,F#>=S3+28.

S1 = S1{[0..17]}
S2 = S2{[30..47]}
F = F{[83..100]}
S3 = S3{[55..72]}
```

... where the `#` operator stipulates that the equality/disequality identities in which it appears are to be treated as constraints by the program interpreter and not as in-line goals. The result of applying the constraints to the domain variables defined earlier is shown under the constraints that produced them. It is important to note that propagation applies to both sides of any identity equally—any domain variable can be affected.

3. The residual domains of the variables are searched for a solution by instantiating each domain in turn to one of the elements in its range after pruning—this process is commonly called *labelling*. A built-in predicate `indomain/1` exists for this purpose. It starts with the smallest element in the domain and on backtracking successive elements are taken. Once an instantiation of the variable is found which is

consistent with the constraints the goal is satisfied. Normally the built-in is used as part of a predicate that works through the list of variables to be labelled.

5. ECL^{i}PS^e as a constraint logic program

As a worked example of a constraint program in ECL^{i}PS^e it is possible to continue to use the vending machine paradigm. The problem to be solved in this case is to make change in coin denominations that exactly make up the amount returned. As a subsidiary problem it is possible to specify that this amount should be made up of the minimum number coins that will do the job.

The problem of selecting from a set of assorted integers a subset that exactly sums up to a given value is one of a class of 'knapsack' problems[3] which, when scaled up sufficiently, becomes computationally infeasible to solve—such problems are generally known, with commendable understatement, as computationally 'hard'. In the case of selecting coins from a set of only six denominations with repeats allowed the problem is trivial but serves to illustrate the operation of a constraint solver.

To illustrate the problem the fact `c_float/1` introduced in the vending machine example of the previous section is changed from a simple integer value to a list of the current holdings of coins in descending denomination value from 50p to 1p. The sub-problem of making the correct change for the vending machine can be defined as follows:

Given an integer value C find what sum of multiples of the integer values 50, 20, 10, 5, 2 and 1 exactly equals C where these multiples individually do not exceed the limits listed in the fact `c_float/1`.

The program is specified as a query which for a required value of change will return a list of the number of coins present in the change in their various denominations, and the total number of coins returned. As will be seen later this last term will be used as an optimizing parameter. With the term `C` instantiated to the value 13, the program will execute the following goals:

1. For each element listed in the fact `c_float/1` the goal `get_denom_ds/2` will create in a second list a corresponding domain with bounds between zero and the element value. This specifies that the number of coins of any denomination in the change must not exceed what is in the float.

2. The goal `d_count/3` is called to count the total number of coins returned in the change—used later for optimization. The operator `\==` in its third clause holds when the two terms are not identical.

[3] The integers represent rod lengths and the problem is to find what subset of them would exactly fit a notional one-dimensional knapsack.

3. A constraint is defined so that the sum of the denomination amounts, each multiplied by its denomination value, must equal the required change amount. The moment this constraint is defined some *a priori* propagation will take place since the only values for the 50p and 20p domains that can satisfy the instantiation of `C` to 13 are the ground state zero. Likewise the domains for 10p and 5p will be reduced to `0..1` and `0..2` respectively. This pruning will take place autonomously.
4. The goal `labelling/1` is called with a list of the domains for which it will attempt to find a set of ground values that satisfies the sum constraint. The goal applies the built-in predicate **`indomain/1`** to each domain variable of the list in turn starting with the smallest element of each domain and taking successively increasing values wherever backtracking occurs. Each time a new value is generated, all the constraints referencing the domain (in this case just the sum constraint) are woken up and re-evaluated for consistency. A first solution is achieved with the result list instantiated to `[0, 0, 0, 0, 5, 3]` and the number of coins returned is 8—which is incidentally also the number of backtracks executed to arrive at the solution.

The example program is listed below—in practice some mechanism for updating the float would also be included. In addition to the result given above, eight further solutions will be found to be available.

```
% Cash float in denominations of 50p, 20p, 10p, 5p, 2p,
1p respectively
c_float([2,4,3,7,6,4]).

mk_ch(C,[D50,D20,D10,D5,D2,D1],S):-
    c_float(FL),get_denom_ds(FL,[D50,D20,D10,D5,D2,D1]),
    d_count([D50,D20,D10,D5,D2,D1],0,S),
    C#=50*D50+20*D20+10*D10+5*D5+2*D2+D1,
    labeling([D50,D20,D10,D5,D2,D1]).

get_denom_ds([],[]).
get_denom_ds([X|Xt],[D|Dt]):-
    D::0..X,get_denom_ds(Xt,Dt).

labeling([]).
labeling([X|Xt]):-indomain(X),labeling(Xt).

d_count([],C,C).
d_count([0|Dt],C,FC):-d_count(Dt,C,FC).
d_count([D|Dt],C,FC):-D\==0,NC is C+D,d_count(Dt,NC,FC).
```

The salient points to be noted from this worked example are that as constraints become defined, *a priori* constraint propagation will take place automatically. By reducing the domains of the variables in each constraint—in some cases down to ground values, the residual domain space that needs to be searched is considerably reduced. The search process also propagates constraints—every ground state that preserves the consistency of the set of constraints is retained and the labelling process can proceed to the next variable. Although the reduction of search space in the above example will have little impact on performance, the mechanism can have a critical effect in rendering a large range of problems solvable by reducing the order of their complexity.

The example described above will produce one solution each time that the query is executed until no further solutions are possible. This means that it is possible to apply further cost criteria to select an optimum solution from this solution space. For a vending machine various options are available. In this example the one adopted minimizes the total number of coins returned in the change. To achieve this the coin total produced in the previous example can be used as the objective function by turning it into a domain that can be progressively reduced as the optimization finds better solutions. The following changes to the example program will achieve this:

```
top(C,L,M):-
      c_float(FL),sum_c(FL,0,N),M::1..N,
      min_max(mk_ch(C,L,M),M),!.
top(_,"Can't make change",_).

mk_ch(C,[D50,D20,D10,D5,D2,D1],M):-
      c_float(FL),get_denom_ds(FL,[D50,D20,D10,D5,D2,D1]),
      d_count([D50,D20,D10,D5,D2,D1],0,S),M#=S,
      C#=50*D50+20*D20+10*D10+5*D5+2*D2+D1,
      labeling([D50,D20,D10,D5,D2,D1]).
```

The main difference is that the change making predicate now becomes a goal term of the built-in optimizer predicate **min_max/2**[4] with the second term being the objective function. This latter is the value that the optimizer will seek to find a lowest value for and has therefore been converted from the integer in the first example to a domain variable here. The optimizer is now called from a top-level predicate **top/3** which also includes the definition of the new domain variable **M** with an upper bound representing the maximum number of coins in the float. The only change to the body of the original program is the inclusion of the additional constraint **M#=S**, which will prune the 'number of coins' domain every time there is a reduction. Executing the optimized program under ECL^iPS^e will produce the results...

[4] Built-in predicates are available from a library of pre-defined procedures. **min_max/2** uses the *branch and bound* method in which partial solutions found to be worse than a previous solution force failure and a recomputation. Better solutions update the objective function by lowering its upper bound and the search restarts from the beginning.

```
[eclipse 29]: top(13,L,N).
Found a solution with cost 8
Found a solution with cost 7
Found a solution with cost 5
Found a solution with cost 4
Found a solution with cost 3

L = [0, 0, 1, 0, 1, 1]
N = 3
yes.
```

... where it can be seen that, again for a change requirement of 13 and the same float make-up as before, the number of coins returned has been reduced progressively from 8 to 3 to achieve the optimum solution of one each of 10p, 2p and 1p. It will also be noted that the top-level predicate also caters for the case where no combination of denominations in the float will meet the change requirement—this would cause the main goal to fail through over-constraint and call the alternative.

6. Ease of use and other ECL^{i}PS^e features

The above examples of both logic programming and constraint logic programming are intended merely as an introduction to the art and to serve as the basis of a hands-on practical course. ECL^{i}PS^e, in common with other well established logic programming languages, has an on-line language interpreter that executes any valid sequence of procedures entered into the command line. It therefore becomes very easy not only to explore the logic language and finite domain library features described above, but also other extensions such as the interval domain (applying constraints to real values) and generalized propagation libraries (turning any predicate into a constraint), and the so-called 'glass box' features of ECL^{i}PS^e with which it is possible to examine in detail the nature and behaviour of the individual domain attributes of variables.

A feature of many rule based so-called 'expert systems' applications is their ability to provide explanations of the reasoning path that led to a solution. This is usually just a record of the firing order of the internal rules of the expert system. The equivalent in a constraint based system would be to make a record of the sequence of constraints visited in the course of *ab initio* pruning and subsequent labelling in addition to the non-constraint goals visited. The granularity of this process is so fine that even the designers of an application would have difficulty in interpreting the results. This is where the 'glass box' tools supported by ECL^{i}PS^e come into their own as by using them it is possible to focus on the behaviour of particular domains and if necessary display the progress of pruning graphically. These features have proved to be particularly valuable in resolving on the one hand why programs fail through over-constraint, and on the other pinpointing the causes of insufficient pruning which renders the subsequent search phase inefficient.

Bibliography

STERLING, L. and SHAPIRO, E., "The Art of Prolog", MIT Press 1986.

Biography

Owen Evans joined the Advance Research and Development laboratory of ICT Engineering (an ICL predecessor) in 1963 from the oil industry. The research facility has continued in unbroken line to its present guise of the Research and Advanced Technology centre of ICL Group HQ. During his time in the research centre which has spanned virtually the entire evolution of the computer industry, he has worked on computer architecture, memory and microprogram design, system performance evaluation, compiler design and logic languages. In the course of his career at ICL he has managed various projects supported by the Advanced Computer Technology Project, the Alvey programme, and the EC Esprit and Fourth Framework initiatives in the areas of high-level language emulators, text databases, human-computer interaction and constraint logic programming. His current interests are in the areas of constraint logic programming, data mining and neural networks.

Mr. Evans graduated in Engineering from Cambridge in 1959.

Constraint Programming

Mark Wallace

IC-Parc, William Penney Laboratory, Imperial College, London, UK

Abstract

Constraint Programming is a paradigm that is tailored to solving hard search problems. Its success is due to combining clean problem modelling with efficient problem solving. This is achieved by extending declarative modelling techniques with efficient specialised constraint handlers and high-level control. The paper starts by reviewing the historical roots of constraint programming and introducing the concept of a constraint store. The latter, which holds a set of primitive constraints, of which the global consistency is enforced, is illustrated by a particular constraint programming language called CLP($\Re$). The use of constraint propagation to improve the efficiency of a program by reducing the size of the search space is discussed and a simple application is used to illustrate the dramatic improvements in program runtime that may be achieved. The implementation of constraint programming is examined and, in particular, the embedding of constraint programming code in general-purpose programming languages is discussed. Constraint programming methods are now being used in many applications, some of which are briefly surveyed. The paper concludes by summarising current developments and indicating areas of likely importance in the near future.

1. Introduction

Constraint programming is a paradigm that is tailored to hard search problems. To date the main application areas are those of planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming incorporates techniques from mathematics, artificial intelligence and operations research, and it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance. The direct representation of the problem, in terms of constraints, results in short, simple programs that can be easily adapted to changing requirements. The integration of these techniques into a coherent high-level language enables the programmer to concentrate on choosing the best combination for the problem at hand. Because programs are quick to develop and to modify, it is possible to experiment with ways of solving a problem until the best and fastest program has been found. Moreover more complex problems can be tackled without the programming task becoming unmanageable. A tutorial introduction to constraint logic programming can be found in [Fruhwirth et.al, 1992].

Constraint logic programming (CLP) combines logic, which is used to specify a set of possibilities explored via a very simple inbuilt search method, with constraints, which are used to minimise the search by eliminating impossible alternatives in advance. The programmer can state the factors which must be taken into account in any solution, – the constraints, state the possibilities – the logic program, and use the system to combine reasoning and search. The constraints are used to restrict and guide search.

The whole field of software research and development has one aim, namely, to optimise the task of specifying and writing and maintaining correct, functioning programs. Three important factors to be optimised are:

- correctness of programs
- clarity and brevity of programs
- efficiency of programs

Constraint programming is, perhaps, unique in making a direct contribution in all three areas. This is why it is such an exciting paradigm.

2. History

In 1963 Sutherland introduced the Sketchpad system, a constraint language for graphical interaction. Other early constraint programming languages were Fikes' Ref-Arf, Lauriere's Alice, Sussmann's CONSTRAINTS and Borning's ThingLab. These languages already offered the most important features of constraint programming: declarative problem modelling and efficient constraint enforcement; propagation of the effects of decisions; flexible and intelligent search for feasible solutions. Each of these three features has been the study of extensive research over a long period.

The current flowering of constraint programming owes itself to a generation of languages in which declarative modelling, constraint propagation and explicit search control are supported in a coherent architecture that makes them easy to understand, combine and apply.

2.1 Declarative Modelling and Efficient Enforcement

2.1.1 Algorithm = Logic + Control

Declarative programming has a long history yielding languages such as LISP, Prolog and other purer functional and logic programming languages, and of course it underpinned the introduction of relational databases and produced SQL which, for all its faults, is today's most commercially successful declarative programming language.

There has been a recognition that declarative programming has problems with performance and scaleability. One consequence has been a swing back to traditional procedural programming. However, constraint programming, whilst recognising that efficiency is an important issue, retains the underlying declarative approach. The idea is not to abandon declarative programming (that would amount to throwing away the baby with the bathwater), but to augment it with explicit facilities to control evaluation. Hence Kowalski's maxim that Algorithm = Logic + Control.

When constraints are used in an application, both the issues of modelling and performance are considered. An early use of constraints was in the modelling of electrical circuits. Such circuits involve a variety of constraints from simple equations (the current at any two points in a sequential circuit is the same), to linear equations (when a circuit divides, the current flowing in is the sum of the currents flowing out), to quadratic equations (voltage equals current multiplied by resistance) and so on. A constraint solver that can handle all the constraints on a circuit would be prohibitively inefficient. Consequently Sussmann sought to model circuits using only a simple class of constraints. He showed that the lack of expressive power of simple constraints can be compensated for by using multiple orthogonal models of the circuit. The different constraints of the different models interact to produce more information than could be extracted from the models independently.

2.1.2 Constraints for Multi-Directional Programming

In many early constraint systems, constraints were little more than functions which were evaluated in a data-driven way. The logic programming paradigm, however, suggested that programs should be runnable "in both directions". In addition to evaluating a function, f(X), yielding the result Y, it must be possible to solve the equation f(X) = Y for a given value Y but unknown arguments X.

Naturally when a function is evaluated "backwards" i.e. from its result producing its input—it is no longer a function. Attempts to integrate functional and logic programming motivated much research on equation solving systems, and in the end spawned constraint logic programming.

It was recognised that constraint solving lies at the heart of logic programming, in its built-in unification. Researchers began to replace (syntactic) unification with other equation solvers. An important example of this was Boolean unification: this is a solver for equations between Boolean expressions, whose possible values are only true or false. This development has now found a commercially successful application for design and verification of digital circuits. Moreover Boolean unification is also being applied to the design and verification of real-time control software.

2.1.3 Constraint Logic programming

Soon an even more radical step was taken when it was recognised that unification could be replaced by any constraint system and solver, provided certain conditions were satisfied. There was no need for a unification algorithm (which reduces an equation between expressions to an equivalent set of variable assignments). Indeed the constraints need not be equations at all.

The resulting scheme [Jaffar and Lassez, 1987] called the Constraint Logic Programming Scheme, and written CLP(X), was illustrated by choosing mathematical equations and inequations as the constraint system, and the Simplex algorithm as the solver. This instance of CLP(X) is called CLP($\Re$), and is described in a later section.

It has inspired a whole research area, exploring the interface between logic and mathematical programming. One resulting constraint programming language is 2LP (Linear Programming and Logic Programming), which embeds mixed integer programming in a constraint programming system. Another is Newton, which uses interval constraints to solve hard mathematical problems involving polynomials.

Whilst constraint logic programming offers a powerful modelling language, new constraint propagation algorithms and a clean execution model, mathematical programming offers some sophisticated algorithms, highly optimised implementations and a wealth of industrial application know-how.

The next step beyond the standard constraint logic programming scheme was to include more than one constraint system and solver in a single system. Even CLP($\Re$) was, in fact, such a combination including syntactic unification, Gaussian elimination and the Simplex. CLP systems nowadays include a variety of solvers which exchange information through shared variables. For numeric variables, in addition to the above solvers, there may also be a Groebner base rewriting system for handling polynomial equations, a very powerful CAD solver and a weaker but very useful constraint handler for reasoning on numeric intervals. The latter three system are typically useful for non-linear constraints, containing expressions in which variables are multiplied together.

2.2 Propagation

2.2.1 Early Applications

Constraint propagation was used in 1972 for scene labelling applications, and has produced a long line of local consistency algorithms, recently surveyed in [Tsang, 1993].

Constraint propagation offers a natural way for a system to spontaneously produce the consequences of a decision. (Propagation is defined in the dictionary as "dissemination, or diffusion of statements, beliefs, practices"). Propagation is the most important form of immediate feedback for a decision-maker.

Propagation works very effectively in interactive decision support tools. In many applications constraint programming is used in conjunction with other software tools, taking their results as input, performing propagation, and outputting the consequences. Typically feedback from the propagation tool is given in the form of a spreadsheet interface.

Many early applications of constraint programming were related to graphics: geometric layout, user interface toolkits, graphical simulations, and graphical editors. Constraint propagation has played a key role in all these applications, with the result that control over the propagation has been thoroughly investigated: leading to a current generation of very high-performance constraint-based graphics applications.

2.2.2 Constraint Satisfaction Problems

On the other hand constraint propagation has been the core algorithm used in solving a large class of problems termed constraint satisfaction problems (CSP's). Standard CSP's have a fixed finite number of problem variables, and each variable has a finite set of possible values it can take (called its domain). The constraints between the variables can always be expressed as a set of admissible combinations of values. These constraints can be directly represented as relations in a relational database, in which case each admissible combination is a tuple of the relation. CSP's have inspired a fascinating variety of research because despite their simplicity they can be used to express real, difficult, problems in a very natural way.[1]

One line of research has focussed on constraint propagation, showing how to propagate more and more information (forward checking, arc-consistency, path-consistency, k-consistency, relational consistency and so on). For example arc-consistency is achieved by reducing the domains of the problem variables until the remaining values are all supported; a value is supported if every constraint on the variable includes a tuple in which the variable takes this value and the other variables all take supported values.

Even if none of the arc-consistent domains are empty, this does not imply the CSP has a solution. To find a solution it is still necessary to try out specific values for the problem variables. Only if all the variables can be assigned a specific value, such that they all support each other, has a solution been found. One algorithm for solving CSP's, which has proved useful in practice, is to select a value for each variable in turn, but, after making each selection, to re-establish arc-consistency. Thus search is interleaved with constraint propagation. In this way the domains of the remaining values are reduced further and further until either one becomes empty, in which case some previous choice must be changed, or else the remaining domains contain only one value, in which case the problem is solved.

Another line of research has investigated the global shape of the prob-

[1] The class of CSP problems is NP complete.

lem. This shape can be viewed as a graph, where each variable is a node and each constraint an edge (or hyper-edge) in the graph. Tree-structured problems are relatively easy to solve, but research has also revealed a variety of ways of dealing with more awkward structures, by breaking down a problem into easier subproblems, whose results can be combined into a solution of the original problem. Picturesque names have been invented for these techniques such as "perfect relaxation" and "hinge decomposition".

More recently researchers have begun to explore the structure of the individual constraints. If the constraints belong to certain classes, propagation can be much more efficient or it can even be used to find globally consistent solutions in polynomial time. Indeed there are nice sufficient conditions to distinguish between NP-complete problem classes and problem classes solvable with known polynomial algorithms.

2.2.3 Constraint Propagation in Logic programming

The practical benefits of constraint propagation really began to emerge when it was embedded in a programming language [Van Hentenryck, 1989]. Again it was logic programming that was first used as a host language, producing impressive results first on some difficult puzzles and then on industrial problems such as scheduling, warehouse location, cutting stock and so on [Dincbas et.al., 1988]. The embedding suggested new kinds of propagation, new ways of interleaving propagation and search and new ways of varying the propagation according to the particular features of the problem at hand.

These advantages were very clearly illustrated when, using lessons learned from the Operations Research community, constraint logic programming began to outperform specialised algorithms on a variety of benchmark problems. However the main advantage of constraint programming is not the good performance that can be obtained on benchmarks, but its flexibility in modelling and efficiently solving complex problems.

A constraint program for an application such as Vehicle Scheduling, or Bin Packing, not only admits the standard constraints typically associated with that class of problems, but it also admits other side-constraints which cause severe headaches for Operations Research approaches.

Currently several companies are offering constraint programming tools and constraint programming solutions for complex industrial applications. As host programming language, not only logic programming but also LISP and C^{++} are offered.

2.3 Search

The topic of search has been at the heart of AI since GPS. Some fundamental search algorithms were generate and test, branch and bound, the A* algorithm, iterative deepening, and tree search guided by the global problem structure, or by information elicited during search, or by intelligent backtracking.

The contribution of constraint programming is to allow the end user to control the search, combining generic techniques and problem-specific heuristics. A nice illustration of this is the *n*-queens problem: how to place *n* queens on an *nXn* chess board, so that no queens can take each other. For *n* = 8, depth-first generate-and-test with backtracking is quite sufficient, finding a solution in a few seconds. However, when *n* = 16, it is necessary to interleave constraint propagation with the search so as to obtain a solution quickly. When *n* = 32, however, it is necessary to add more intelligence to the search algorithm. A very general technique is to choose as the next variable to label the one with the smallest domain; i.e. the smallest number of possible values. This is called first-fail. It is particularly effective in conjunction with constraint propagation, which reduces the size of the domains of the unlabelled variables (as described for arc-consistency above). For *n* queens, the first-fail technique works very well, yielding a solution within a second. Unfortunately even first-fail doesn't scale up beyond *n* = 70. However there is a problem-specific heuristic which starts by placing queens in the middle of the board and then moving outwards. With the combination of depth-first search, interleaved with constraint propagation, using the first-fail ordering for choosing which queen to place next, and placing queens in the centre of the board first, the 70-queens problem is solved within a second, and the algorithm scales up easily to 200 queens.[2]

3. Programming with a Constraint Store

3.1 Primitive Constraints

The traditional model of a computer store admits only two possible states for a variable: assigned or unassigned. Constraint programming uses a generalisation of this model. A so-called constraint store can hold partial information about a variable, expressed as constraints on the variable. In this model, an unassigned variable is an unconstrained variable. An assigned variable is maximally constrained: no further non-redundant constraints can be imposed on the variable, without introducing an inconsistency.

Primitive constraints are the constraints that can be held in the constraint store. The simplest constraint store is the ordinary single-assignment store used in functional programming. In our terms it is a constraint store in which all constraints have the form Variable = Value.

The first generalisation is the introduction of the logical variable. This

[2] Solutions for n queens can be generated by a deterministic algorithm, however this problem is useful for providing a simple and illuminating example of techniques which also pay off where there are no such alternative algorithms.

allows information of the form Variable = Term to be stored, where Term is any term in first-order logic. For example it is possible to store X = f(Y). The same representation can be used to store partial information about X. Thus, if nothing is known about the argument of *f*, we can store X = f(_). This is the model used in logic programming and, in particular, by Prolog.

The storage model used by logic programming has a weakness, however. This is best illustrated by a simple example. The equation $X - 3 = Y + 5$ is rejected because logic programming does not associate any meaning with – or + in such an equation.

The extension of logic programming to store equations involving mathematical functions was an important breakthrough. Equations involving mathematical functions are passed to the constraint store, and checked by a specialised solver. In fact not only (linear) equations but also inequations can be checked for consistency by standard mathematical techniques. It is necessary, each time a new equation or inequation is encountered, to check it against the complete set of equations encoutered so far.

Linear equations and inequations are examples of primitive constraints. Thus we have an example of a constraint store. Further constraint stores can be built for different classes of primitive constraints, by designing constraint solvers specifically for those classes of constraints.

We use the term storage model, rather than data model, because the facility to store constraints is independent of the choice of data model—object-oriented, temporal etc. On the other hand, the term storage model as used here does not refer to any physical representation of the stored information.

Definition: *A constraint store is a storage model which admits primitive constraints of a specific class. Each new primitive constraint that is added to the store is automatically checked for consistency with the currently stored constraints.*

This definition of a constraint store specifies an equivalent operation to writing to a traditional store. However no equivalent to the read statement is specified. There are two important facilites useful for extracting information from a constraint store.

Firstly it is useful to retrieve all those constraints that involve a given variable, or set of variables. For example, if the constraint store held three constraints, $X \geq Z$, $Y \geq X$, $W \geq Z$ the constraints involving X would be $Y \geq X$ and $X \geq Z$. However retrieving only constraints explicitly involving a variable may not give a full picture of the entailed constraints on the variable. For example, the store $X \geq Y$, $Y \geq Z$ entails $X \geq Z$. The mechanism necessary to return all the constraints and entailed constraints on a given variable or set of variables is termed projection. A very useful property of a class of primitive constraints is the property that the projection of a set of primitive constraints on a given variable, or set of variables, is also expressible as a set of primitive constraints. If the primitive constraints have this property it is possible, for example, to drop a variable from the constraint store when it is no longer relevant. This is the equivalent to reclaiming the store associated with a variable in a traditional programming language when the variable passes out of scope.

Secondly it is useful to retrieve particular kinds of entailed constraints from a constraint store. For example it is very useful to know when a constraint store entails that a particular variable has a fixed value. For example the constraint store $X \geq Y, Y \geq 3, 3 \geq X$ entails that both X and Y have the value 3.

3.2 CLP($\Re$)

The constraint logic programming scheme, written *CLP(X)*, is a generic extension of logic programming to compute over any given constraint store. Logic programming over a constraint store has all the advantages of traditional logic programming, plus many further advantages for high-level modelling and efficient evaluation. If the constraint store holds primitive constraints from the class X, logic programming over this constraint store is termed *CLP (X)*. In this section we shall use a particular class of primitive constraints, linear equations and ineqqualities, termed $\Re$, to illustrate the scheme. We shall use an example from [Colmerauer, 1990] to illustrate how it works.

> Given the definition of a meal as consisting of an appetiser, a main meal and a dessert, and given a database of foods and their calorific values, we wish to construct light meals i.e. meals whose total calorific value does not exceed 10.

A *CLP($\Re$)* program for solving this problem is shown in Figure 1.

```
lightmeal(A,M,D) :-
    I>=0, J>=0, K>=0,
    I+J+K <= 10,
    appetiser(A,I),
    main(M,J),
    dessert(D,K),


main(M,I) :-
    meat(M,I).
main(M,I) :-
    fish(M,I).
```

```
appetiser(radishes,1).
appetiser(pasta,6).



meat(beef,5).
meat(pork,7).



dessert(fruit,2).
dessert(icecream,6).
```

Figure 1: The Lightmeal Program in *CLP($\Re$)*

A *CLP ($\Re$)* program is syntactically a collection of clauses which are either rules or facts. Rules are as in logic programming with the addition that they can include constraints, such as $I + J + K \leq 10$, in their bodies.

The intermediate results of the execution of this program will be descibed as computation states. Each such state comprises two components, the constraint store, and the remaining goals to be solved. We shall represent such a state as **`Store @ Goals`**. *CLP ($\Re$)* programs are executed by reducing the goals using the program clauses. Consider the query **`lightmeal(X,Y,Z).`** which asks for any way of putting together a light meal. The initial state has an empty constraint store and one goal:

```
@ lightmeal(X,Y,Z).
```

This goal is reduced using the clause whose head matches the goal. The goal is then replaced by the body of the clause, adding any constraints to the constraint store:[3]

```
X=A,Y=M,Z=D, I+J+K =< 10, I>=0, J>=0, K>=0 @
appetiser(A,I), main(M,J), dessert(D,K)
```

The execution continues, choosing a matching clause for each goal and using it to reduce the goal. Variables which neither appear in the original goal, nor any of the currently remaining goals are projected out, as described above. A successful derivation is a sequence of such steps that reduces all the goals without ever meeting an inconsistency on adding constraints to the store. An example is:

```
X=radishes, Y=M, Z=D, 1+J+K=<10, J>=0, K>=0 @
main(M,J), dessert(D,K)

X=radishes, Y=M, Z=D, 1+J+K=<10, J>=0, K>=0 @
meat(M,J), dessert(D,K)

X=radishes, Y=beef, Z=D, 1+5+K=<10, K>=0 @
dessert(D,K)

X=radishes, Y=beef, Z=fruit  @
```

Note that at the last step the constraint $1 + 5 + 2 =< 10$ is added to the store, but it is immediately projected out.

Next we give an example of a failed derivation. The initial goal is the same as before, but this time **`pasta`** is chosen as an appetiser instead of **`radishes`**:

```
X=A,Y=M,Z=D, I+J+K =< 10, I>=0, J>=0, K>=0 @
appetiser(A,I), main(M,J), dessert(D,K)

X=pasta, Y=M, Z=D, 6+J+K=<10, J>=0, K>=0 @
main(M,J), dessert(D,K)
```

[3] Strictly all variables in the clause are renamed, but we omit this detail for simplicity.

```
X=pasta, Y=M, Z=D, 6+J+K=<10, J>=0, K>=0 @
meat(M,J), dessert(D,K)
```

At the next step whichever clause is used to reduce the goal **meat(M,J)**, an inconsistent constraint is encountered. For example, choosing beef requires the constraint $J = 5$ to be added to the store, but this is inconsistent with the two constraints $6 + J + K \leq 10$ *and* $K \geq 0$.

When the attempt to add a constraint to the store fails, due to inconsistency, a *CLP (X)* program abandons the current branch of the search tree and tries another choice. In sequential implementations this is usually achieved by backtracking to some previous choice of clause to match with a goal. However there are or-parallel implementations, where several branches are explored at the same time, and a failing branch is simply given up, allowing the newly available processor to be assigned to another branch.

4. Constraint Propagation

4.1 Propagating Changes

A key innovation behind constraint programming is constraint propagation. Propagation is a generalisation of data-driven computation. Consider the constraint $x = y + 1$, where x and y are variables. In a constraint program, any assignment to the variable y (e.g. $y = 5$) causes an assignment to x ($x = 6$). Moreover the very same constraint also works in the other direction: any assignment to x (e.g. $x = 3$) causes an assignment to y ($y = 2$).

In a graphical application, constraint propagation can be used to maintain constraints between graphical objects when they are moved. For example if one object is constrained to appear on top of another object, and the end-user then moves one of the objects sideways, the other object will move with it as a result of constraint propagation.

In general each object may be involved in many constraints. Consequently the assignment of a new position to a given object as a result of propagation, may propagate further new assignments to other objects, which may cause further propagation in their turn. If each constraint between two objects is represented as an edge in a graph, the propagation spreads through the connected components of the graph.

When a particular object is assigned a new position, and the change is propagated from the object to other objects, there is a causal direction. In this case we can assign a direction with each edge of the (connected component of) the graph. As long as the graph is free of cycles, the propagation behaviour is guaranteed to terminate, and produce the same final state irrespective of the order in which constraints are propagated. Efficient algorithms, such as the DeltaBlue algorithm, have been developed for propagation of graphical constraints. They work by firstly generating the directed graph whenever an object is moved, and then compiling this directed graph into highly efficient event-driven code.

However if the graph contains cycles both these issues arise. Consider, as a simple example, the three constraints C1, C2 and C3 specified thus: C1: $x = y + 1$, C2: $y = z + 1$ and C3: $z = x + 1$.

Assigning $y = 3$ may start a non-terminating sequence of propagations cycling through the constraints C1 (which yields $x = 4$), then C3 (which yields $z = 5$), then C2 (which yields $y = 6$) and then C1 again and so on. Alternatively the same assignment $y = 3$ could propagate through C2 yielding $z = 2$, thence $x = 1$ and $y = 0$ via C3 and C1. In this case the propagation also goes on for ever, but this time the values of the variables decrease on each cycle. Thirdly, the same assignment $y = 3$ could yield $z = 5$ via C1 and C3, and $z = 2$ via C2.

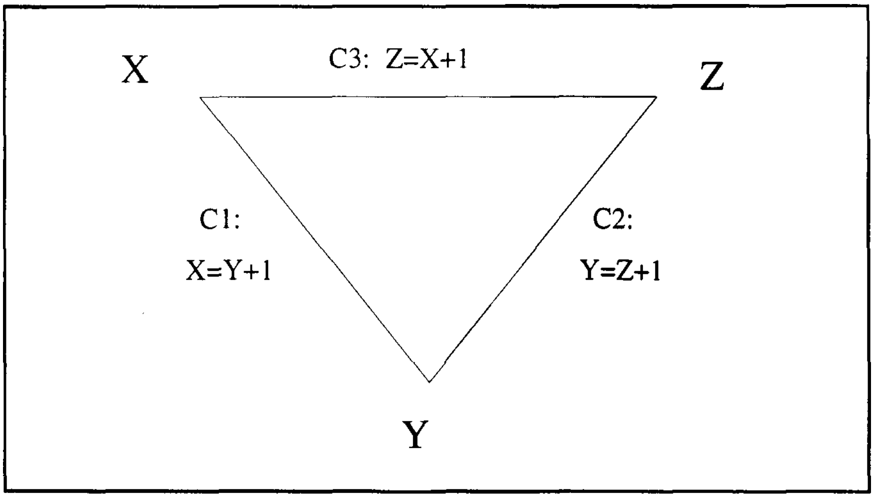


Figure 2: A Constraint Graph with a Cycle

In this example the original constraints are, logically, inconsistent. However similar behaviour can occur when the constraints are consistent. In the following example there are constraints on three variables. C4: $x + y = z$, C5: $x - y = z$. Suppose the initial (consistent) assignments are $x = 2, y = 0, z = 2$. Now a new assignment is made to z: $z = 3$. If constraint propagation on C4 yields $y = 1$, then propagation on C5 yields $x = 4$. Now propagation on C4 and C5 can continue for ever, alternately updating y and x.

Propagation algorithms have been developed which can deal with cycles, but if the class of admissible constraints is too general, it is not possible to guarantee that propagation is confluent and terminating.

4.2 Active Constraints

4.2.1 Propagating New Information

The changes propagated by label propagation, as discussed in the previous section, are variable assignments as held in a traditional program store. However in this section we explore the application of constraint propagation to constraint stores, which maintain partial information about the program variables expressed as primitive constraints. Using a constraint store, it is possible to develop a quite different model of computation in which the store is never destructively changed by propagation, but only augmented. One great advantage of this combination is that confluence properties are easy to establish, and consequently there is little need for the programmer or applications designer to know in what order the propagation steps take place.

4.2.2 Constraint Agents

We have encountered two very different kinds of constraints. Primitive constraints are held in a constraint store, and tested for consistency by a constraint solver. On the other hand propagation constraints actively propagate new information, and they operate independently of each other. Propagation constraints are more commonly called constraint agents. The behaviour of a constraint agent is to propagate information to the underlying store. In case the underlying store is a constraint store, the information propagated is expressed as primitive constraints.

Constraint agents are processes that involve a fixed set of variables. During their lifetime they alternate between suspended and waking states. They are woken from suspension when an extra primitive constraint on one or more of their variables is recorded. Sometimes, after checking certain conditions, the woken agent simply suspends again. Otherwise the agent exhibits some active behaviour which may result in new agents being spawned, new primitive constraints being added to the store, or an inconsistency being detected (which is equivalent to an inconsistent constraint being added to the store). Subsequently the agent either suspends again, or exits, according to its specification.

4.2.3 Some Built-in Constraint Agents

The simplest constraint agent is one which adds a primitive constraint to the constraint store and then exits. The most fundamental example is assigning a value to a variable, eg. `X=3`. This agent adds $X = 3$ to the constraint store and exits.

The next two examples are disequality constraints, which will be illustrated in the next section. The first disequality constraint is invoked by the syntax `X~=Y`. This agent does not do anything until both X and Y have a fixed value. Only when the primitive constraints in the store entail $X = val_x$ and $Y = val_y$ for some unique values val_x and val_y, does the agent wake up. Then its behaviour is to check that val_x is different from val_y. In case they are the same, an inconsistency has been detected.

If the constraint store holds finite domain constraints, then the more powerful constraint agent invoked by the syntax `X ## Y` can be used. This agent wakes up as soon as either X or Y has a fixed value. It then removes this value from the finite domain of the other variable and exits.

4.3 Map Colouring

4.3.1 The Map Colouring Program

As a toy example let us write a program to colour a map so that no two neighbouring countries have the same colour. In constraint logic programs, variables start with a capital letter (e.g. `A`), and constants with a small letter (e.g. `red`).

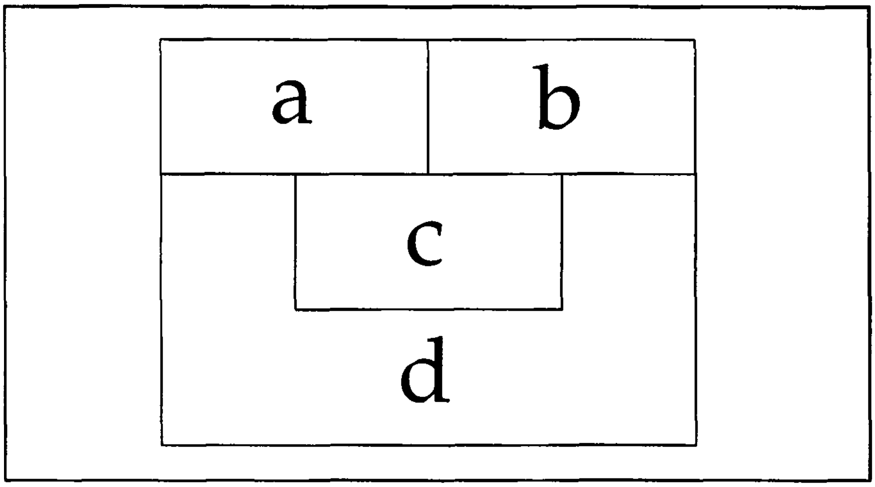


Figure 3: A simple map to colour

A generic logic program, in Prolog syntax, that tries to find possible ways of colouring this map with only three colours (red, green and blue) is as follows:

```
coloured(A,B,C,D) :-
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
    chosen(A), chosen(B), chosen(C), chosen(D).

chosen(red).
chosen(green).
chosen(blue).
```

In this program the (as yet undefined) predicate *ne* constrains its arguments to take different values. We will show how different definitions of *ne* cause the program to behave in different ways.

The predicate *chosen* can be satisfied in three ways. At runtime the system tries each alternative in turn. If a failure occurs later in the computation, then the alternatives are tried on a last-in first-out basis.

The first definition of *ne* uses the original disequality of Prolog:

```
ne(X,Y) :- X\=Y.
```

If invoked, when either of its arguments are uninstantiated variables, `X\=Y` simply fails. To avoid this incorrect behaviour it is possible to place the constraints after all the choices. In this case the program correctly detects that there is no possible colouring after checking all 81 alternatives.

A more efficient Prolog program can be written by interleaving choices and constraints, but this requires the programmer to think in terms of the operational behaviour of the program on this particular map. The same effect can be achieved much more cleanly by using the above program with a new definition: `ne(X,Y) :- X~=Y`. This disequality predicate delays until both arguments have a fixed value. It then immediately wakes up and fails if both values are the same. If the values are different it succeeds. This program detects that our map cannot be coloured with three colours after trying only 33 alternatives.

Another disequality constraint is available in CLP, which assumes its arguments have a finite set of possible values. We can use it by defining:

```
ne(X,Y) :- X##Y.
```

This disequality delays until one of its arguments has a fixed value. This value is then removed from the set of possible values for the other. To obtain the advantage of `X##Y` it is necessary to declare the set of possible values for each variable, by writing `[A,B,C,D]::[red,green,blue].`, as follows:

```
coloured(A,B,C,D) :-
    [A,B,C,D]::[red,green,blue],
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
    chosen(A), chosen(B), chosen(C), chosen(D).

ne(X,Y) :- X##Y.

chosen(red).   chosen(green).   chosen(blue).
```

This program detects that the map cannot be coloured after trying only six alternatives.

Although this example is so trivial that it is quite simple to solve it in Prolog, the CLP program scales up to complex maps and graphs in a way that is impossible using a programming language without constraints.

4.4 Building Constraint Agents

4.4.1 Guards

Constraint agents can be built by directly defining their waking behaviour using the notion of a "guard". As an example we take a resource constraint on two tasks, *t1* with duration *d1* and *t2* with duration *d2* forcing them not to overlap. The variable ST_1 denotes the start time of *t1* and ST_2 denotes the start time of *t2*. Suppose we wish to define the agent constraint $agent(ST_1, ST_2)$ thus: if the domain constraints on the start time of *t1* and *t2* prevent *t1* from starting after *t2* has finished, constrain it to finish before *t2* has started.

This behaviour can be expressed as follows:

```
agent(ST_1,ST_2) <==>           % agent name and parameters
       ST_1 #< ST_2 + d2   |    % guard
           ST_1 + d1 #<= ST_2 % body
```

The guard will keep the agent suspended until the domains of ST_1 and ST_2 are reduced to the point that the inequality $ST_1 \leq ST_2 + d2$ holds for every possible value of ST_1 and ST_2. When this is true, it wakes up and executes the body, invoking a new agent `ST_1+d1 #<= ST_2`. Of course this guard may never become true, in case task *t2* runs before task *t1*. To cope with this alternative we add another guard and body, yielding the final definition:

```
agent(ST_1,ST_2) <==>           % agent name and parameters
     ST_1 #< ST_2 + d2   |      % guard1
         ST_1 + d1 #<= ST_2 ;  % body1
     ST_2 #< ST_1 + d2   |      % guard2
         ST_2 + d2 #<= ST_1     % body2
```

This agent wakes up as soon as either of the guards is satisfied and executes the relevant body. As soon as one guard is satisfied, the other guard and body are discarded.

4.4.2 Agents Defined by Specific Codes

Constraint programming systems implement their built-in agents using specific codes which wake up, for example, whenever the upper or lower bound of the domain constraint on a given variable is altered.

Using specific codes it is also possible to build complex constraints and constraint behaviours to obtain good performance on large complex problems. Indeed this is the approach that has been very successfully applied on job-shop scheduling benchmarks and incorporated into commercial constraint programming tools such as CHIP and ILOG SCHEDULE.

5. Implementation and Applications

5.1 Constraints Embedded in A Host Programming Language

5.1.1 Control

A constraint programming language is the result of embedding constraints into a host programming language. The host program sends new constraints to the constraint handler, under program control. The information that can be returned to the host program depends on how tightly the constraints are embedded in the host language. Most basically the constraint handler can report consistency or inconsistency. Given a closer embedding it can also return variable bindings. Most closely it might allow the host programming language to be extended with guards and other annotations, so as to allow host program statements to be suspended and woken up like other constraint agents.

We can diagram the behaviour of a constraint program as follows:

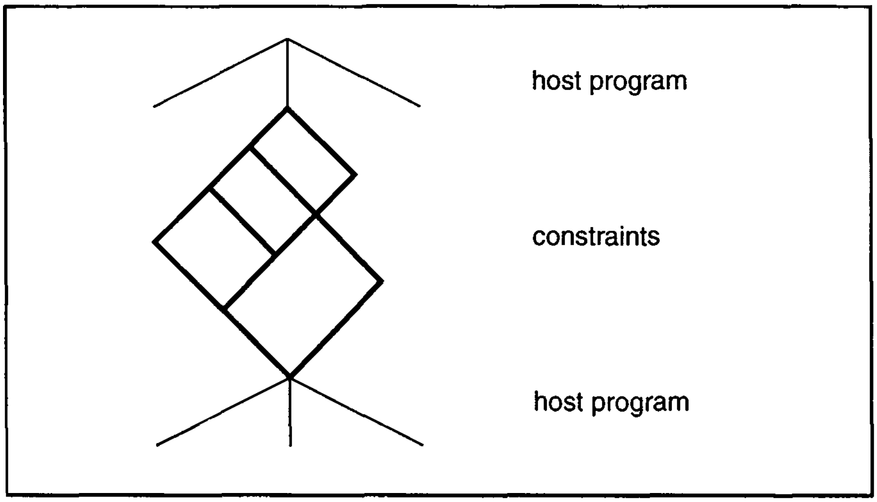


Figure 4: Control in Constraint Programming

The diagram shows three successive phases occurring during program execution.

In the first phase the host program is executing under explicit program control. The host program performs such tasks as input/output, event handling, and search. It may execute for some time before finally sending a constraint to the handler. The diagram illustrates the host program performing search over three branches. For the purposes of the diagram, it does not matter how these branches are explored (sequentially, or in parallel) and how they are expressed (by recursion over a set of alternatives, or by non-deterministic choice and backtracking). The succeeding phases of the execution are only shown for the second branch.

In the second phase the constraint handler is executing, and its control is constraint-driven. The constraint handler only becomes active when it receives a new constraint from the host program. The behaviour of the constraint handler (represented in the diagram as a network of thick lines) is defined by a set of atomic behaviours (each of which is represented by a single arc in the network). An atomic behaviour is the posting of a new constraint to the store, or a single propagation step performed by a constraint agent, or the invocation of the body in a guarded constraint. When no more constraint propagation is possible, the constraint handler returns to the host program with success, and the host program resumes control, as illustrated in the third phase of the above diagram.

5.1.2 Concurrency

The challenge for embedding constraint handling in a host programming language is to deal with constraint agents acting concurrently. Assuming the programmer has little or no control over the waking and resuspending of constraint agents, the constraint programming framework must ensure that the final result of constraint propagation, before the host program resumes control, is independent of the order in which the agents wake up and post new basic constraints to the constraint store. The concurrent constraints framework [Saraswat, 1993] enables this condition to be met for large classes of practically useful constraint agents.

5.1.3 Logic Programming as a Host Language

Constraints fit hand in glove with declarative host programming languages. Three of the most influential constraint programming languages were embedded in Prolog: PrologIII, CLP($\Re$) and CHIP. Whilst all three system are still developing further, there are many new constraint programming systems emerging including ECL^{i}PS^e, Oz, 2LP, and Newton.

From a theoretical point of view the extension of logic programming *to Constraint Logic Programming* (CLP) has been very fruitful. A good survey is [Jaffar and Maher, 1994]. For example ALPS, a form of logic programming with guards, was an extremely influential language, becoming the forerunner of the *Concurrent Constraints* paradigm [Saraswat, 1993]. Concurrent constraint programming has in turn provided a very clean model of concurrent and multi-agent computing. Constraints can also be modelled in terms of information systems, which allows us to reason about the behaviour of constraint programs at an abstract level.

5.1.4 Libraries for Embedded Constraint Programming

The constraint programming technology has matured to the point where it is possible to isolate some essential features and offer them as libraries or embedded cleanly in general purpose host programming languages.

For example isolating constraints as libraries has made possible the development of sophisticated constraint-based scheduling systems, see [Zweben and Fox, 1994]. More generally there are commercially available libraries supporting constraint handling such as the CHIP and ILOG C^{++} constraint libraries.

5.2 Applications of Constraint Programming

Based on a few constraint programming languages which support the storage of basic constraints and the waking and resuspension of constraint agents, the technology has achieved a number of remarkable successes on benchmarks and, more importantly, real industrial applications. A recent survey of practical applications of constraint programming [Wallace, 1996] estimated the annual revenue from constraint technology at around $100 million per annum.

One early application, developed in 1990, was to container port planning in Hong Kong [Perrett, 1990]. The application was built by ICL, using finite domain constraints. Another early user was Siemens, who have applied Boolean constraints to problems of circuit design and integration. Both Siemens and Xerox are now applying constraints to real time control problems.

Constraints are used for graphical interface design and implementation at Object Technology International. Constraint-based scheduling has made a big impact in the USA, with applications in heavy industry, NASA and the Army. The application developers are typically small companies such as Recom, Red Pepper and the Kestrel Institute.

One company, ILOG, has sold constraint technology both in the USA and Europe. ILOG also have applications in south east Asia. Its French rival, Cosytec, is perhaps the only company to make all its business from constraint technology and applications. [Cras, 1993] gives a survey of industrial constraint solving tools.

Areas where constraint programming has proved very successful include scheduling, rostering and transportation. Constraints are used for production scheduling in the chemical industry, oil refinery scheduling, factory scheduling in the aviation industry, mine planning and scheduling, steel plant scheduling, log cutting and transportation, vehicle packaging and loading, food transportation scheduling, nuclear fuel transportation planning and scheduling, platform scheduling, airport gate allocation and stand planning, aircraft rescheduling, crew rostering and scheduling, nurse scheduling, personnel rostering, shift planning, maintenance planning, timetabling, and even financial planning and investment management.

There is a regular conference on the Practical Applications of Constraint Technology, presented on `http://www.demon.co.uk/ar/PACT97/index.html`.

6. Current Developments

6.1 Constraints in the Computing Environment

Naturally there is a great deal of useful research exploring ways of using constraints in an object oriented programming environment, in databases, and on the internet. The field of constraint databases, in particular, has thrown up a growing community of researchers who are exploring the theoretical and practical possibilities of storing constraints in databases, imposing constraints on databases, and retrieving constraints from databases. This work is starting to be noticed in the field of geographical information systems. There is a growing need for databases to handle space, in two and three dimensions, and time. Examples are environmental monitoring and protection, air traffic control, and reasoning about motion in three dimensions. The constraint database technology appears to address these requirements.

6.2 Mixed Initiative Programming

One of the great bugbears of constraint programming is how to deal with overconstrained problems. As Jean-Francois Puget put it, "What solution do you return when there are no solutions?" The traditional approach in mathematical programming is to associate a penalty with each violated constraint and seek the solution which minimises the total penalties.

A related approach is to decide between the different constraints which ones are more important than which others. The constraint program then only enforces a constraint if this does not cause a more important constraint to be violated.

The drawback is that it is not easy for the user to estimate the importance of a constraint, and the solution produced by the software may well not be the best solution in the opinion of the end users of the system. Moreover this approach is a black box, and the end users receive no feedback about "nearby" solutions, which might prove better on the ground.

Accordingly one current area of research is how to help the end user solve overconstrained problems, and multi-criteria optimisation problems which have different, and possibly conflicting, optimisation criteria. The challenge is to allow the end-user to explore the solution space interactively, eliciting information about solutions, and potential solutions, which enables the user to choose the very best solution for his or her purposes. This is called mixed initiative programming.

6.3 Interval Reasoning

Intelligent software systems have often been highly specialised for symbolic computation, but weaker on numeric computation. This is one reason why the combination of symbolic constraint solving and mathematical programming are proving to be so interesting both in theory and practice.

One recalcitrant problem for numeric computation is the problem of numeric instability. Under certain, unlikely, circumstances, tiny rounding errors in the basic mathematical routines can unexpectedly cause serious errors in the final result. The difficulty is that these errors are hard to predict, and no practical way has been found to predict them.

A way to contain this instability is to reason on numeric intervals, instead of numbers, ensuring that at each calculation the interval is rounded out so as to ensure the actual solution lies inside it. Unfortunately these intervals tend to grow too wide to be useful. Recently, however, using intervals as primitive constraints in a constraint programming framework some excellent results have been obtained for well-known mathematical benchmarks. These results compete with the best mathematical programming approaches, in particular when the input intervals are quite wide.

Intervals appear in a multitude of different contexts as a way of approximating values. In particular they are used in database indexing, in constraint propagation, and for specifying uncertainty.

The author predicts that interval constraints will play a crucial role in spatial and temporal databases and in the handling of uncertainty.

6.4 Stochastic Techniques

Organisations are increasingly able to capture an up-to-date picture of their global resources, and they are seeking to optimise their use of these resources. However for large organisations this optimisation problem is unmanageable: no algorithm could ever find the guaranteed best solution for the whole organisation.

Stochastic techniques are a way of exploring very large solution spaces and finding good solutions even when it is only possible to visit a (vanishingly) small proportion of the solutions. Well-known techniques include simulated annealing, genetic algorithms and tabu search. A drawback is that for structured problems, where constraints impose complex dependencies between different parts of the solution, stochastic techniques are not able to enforce these constraints.

Recently researchers have begun to explore ways of embedding constraint propagation in stochastic algorithms, thus ensuring that the solutions visited by the algorithm satisfy the problem constraints. To date such hybrid algorithms have been rather loosely coupled. For example, the stochastic technique only works on a small subset of the problem variables, producing skeleton solutions. These are then fleshed out using constraint handling techniques, and the cost of the resulting full solution is calculated, and fed back to the stochastic algorithm which generates another skeleton solution.

Tightly integrated algorithms combining techniques from mathematical programming, constraint programming and stochastic algorithms are now the vision of a growing research community. These algorithms may still not be the "golden bullet" that cuts through all forms of complexity, but they would certainly represent an important step in the right direction.

Bibliography

COLMERAUER, A., "An Introduction to Prolog III," Communications of the ACM, 33(7): 69-90, July, 1990.

CRAS, J.Y., "A Review of Industrial Constraint Solving Tools," ISBN 1-898804-001, AI Intelligence, 1993.

DINCBAS, M., SIMONIS, H. and VAN HENTENRYCK, P. "Solving large scheduling problems in logic programming," EURO-TIMS Joint International Conference on Operations Research and Management Science, Paris, July, 1988.

FRUHWIRTH, T., A. HEROLD, A., V. KUCHENHOFF, V., LE PROVOST, T., LIM, P., MONFROY, E. and WALLACE, M., "Constraint logic programming: An informal introduction," Logic Programming in Action (edited by Comyn, G. and Ratcliffe, M.), Springer, 1992.

JAFFAR, J. and LASSEZ, J.L., "Constraint logic programming," POPL'87: Proceedings of 14th ACM Symposium on Principles of Programming Languages, pp 111-119, Munich, January, 1987.

JAFFAR, J. and MAHER, M., "Constraint Logic Programming: A Survey," Journal of Logic Programming, 19/20:503-581, 1994.

PERRETT, M., "Using Constraint Logic Programming Techniques in Container Port Planning," ICL Technical Journal, Vol 7, 3, 1991.

SARASWAT, V.A., "Concurrent Constraint Programming," MIT Press, 1993.

TSANG, E., "Foundations of Constraint Satisfaction," Academic Press, 1993.

VAN HENTENRYCK, P., "Constraint Satisfaction in Logic Programming," Logic Programming Series, MIT Press, Cambridge, MA, 1989.

WALLACE, M.G., "Practical applications of constraint programming," Constraints, 1(1), 1996.

ZWEBEN, M. and FOX, M.S., (Editors), "Intelligent Scheduling," Morgan Kaufmann, 1994.

Biography

Mark Wallace is currently seconded from ICL to IC-Parc, Imperial College, where he is Deputy Director. Dr. Wallace has been with ICL for some 15 years, during which he completed a PhD in natural language processing at Southampton University, and then spent a decade at the European Computer Industry Research Centre (ECRC), first working on knowledge bases and, for the last three years, leading ECRC's constraint reasoning project. At IC-Parc he manages several research and application development projects, as well as participating in the development of ECL^{i}PS^e II.

ECL^{i}PS^e – A Platform for Constraint Logic Programming

Mark Wallace, Stefano Novello and Joachim Schimpf

IC-Parc, William Penney Laboratory, Imperial College, London, UK

Abstract

This paper introduces the Constraint Logic Programming (CLP) platform ECL^{i}PS^e, which is designed to support not only an implementation of CLP but also mathematical and stochastic programming techniques. ECL^{i}PS^e is designed for solving difficult "combinatorial" industrial problems in the areas of planning, scheduling and resource allocation. The platform supports experimentation with different hybrid algorithms. These algorithms have two aspects: constraint handling, and search. Various different constraint handling facilities are available as ECL^{i}PS^e libraries, including finite domain propagation, interval propagation and linear constraint solving. In ECL^{i}PS^e the same constraint can be treated concurrently by several different handlers. With regard to search behaviour, ECL^{i}PS^e supports both constructive and repair based search and allows these to be combined into hybrid search techniques.

1. Introduction: The ECL^{i}PS^e Philosophy

The first generation of constraint programming languages focused on a single technique: constraint propagation, as described in Section 4 of [Wallace, 1997]. Whilst constraint propagation has proven itself on a variety of applications, it cannot alone suffice to produce solutions efficiently for typical practical industrial problems.

Over the years Operations Researchers have designed highly efficient algorithms for several classes of problems, such as set partitioning, matching, knapsack, and network flow problems, using techniques based on Mixed Integer Programming (MIP). More recently stochastic techniques, such as Simulated Annealing, have achieved striking results on optimization problems such as the travelling salesman problem.[1]

ECL^{i}PS^e is designed to take advantage of all these results, by supporting industrial scale MIP functionality, and stochastic techniques, as well as constraint propagation and solving.

More importantly, real industrial problems seldom fit into a specific class: the pure travelling salesman problem rarely comes up in real life because there are typically many salesmen available to cover the different customers, certain customers can only be visited at certain times of day, also roads are busier at certain times of day so the journey time may vary with the time of day, and anyway the poor salesmen need some time to rest—they can't usually complete their circuits before lunchtime! These "side constraints" may belong to another problem class—such as the class of set covering problems, or scheduling problems.

Industrial problems typically have constraints that belong to different problem classes—they are in a sense "hybrid". Accordingly it is not enough to offer a wide choice of algorithms for solving such problems: the main requirement is to be able to mix and match the algorithms, i.e. to build hybrid algorithms.

ECL^{i}PS^e is designed to support the fast development of specific hybrid algorithms tuned to the problem at hand. It is not assumed that the first algorithm implemented by the application developer is guaranteed to be the best one: rather ECL^{i}PS^e provides a platform supporting experimentation with different hybrid algorithms until an appropriate one is found which suits the particular features of the application.

In the next section we shall explore ECL^{i}PS^e as a problem modelling language. We distinguish two kinds of model: the *conceptual* model, which captures the problem specification, and the *design* model, which is tuned for efficient solving on a computer. ECL^{i}PS^e is designed to support both kinds of models and the mapping between them.

In the following sections we shall examine the ECL^{i}PS^e facilities for handling constraints. In [Wallace, 1997] we encountered different kinds of constraints—primitive constraints, *propagation* constraints and *constraint agents*. ECL^{i}PS^e supports various classes of built-in constraints, both *primitive* constraints and *propagation* constraints. ECL^{i}PS^e also allows complex constraints and constraint behaviours to be constructed from the built-in classes, thus supporting *constraint agents.*

After constraint handling we return to the second major aspect of problem solving: the search for solutions.

We will separate this discussion into two subsections. The first is concerned with *constructive* search, and the second with *repair-based* search. Constructive search explores the consequences of making choices for decision variables one-at-a-time. Each choice reduces the set of viable choices for the remaining decisions. By contrast repair-based search explores the consequences, not of making decisions, but of *changing* them. In this case the new choice is typically compared with the previous one, in the context of other suggested choices for the other decision variables. Initially it is not expected that the suggested choices are necessarily consistent with the constraints. The idea of changing the choices is to reduce the number of constraint violations until all the constraints are finally satisfied.

[1] The travelling salesman problem is to find the shortest route which starts at a certain point, visits a given set of destinations (customers) and returns to the starting point at the end.

Finally there is a brief section on the ECL^{i}PS^e system, its external communication facilities, how to embed it, documentation and how to obtain the system.

2. ECL^{i}PS^e as a Modelling Language

2.1 Overview of ECL^{i}PS^e as a Modelling Language

ECL^{i}PS^e is tailored for solving combinatorial problems. Such problems are characterised by a set of decisions which have to be made (where each decision has a set of alternative choices) and a set of constraints between the decisions (so a certain choice for one decision may preclude, or entail, certain choices for other decisions).

In ECL^{i}PS^e each decision is modelled by a variable, and each choice by a possible value for that variable. The constraints are modelled by relations between the variables. As an example consider the map colouring program, with four countries to colour, as shown in the following code:

```
:- lib(apply_macros).

coloured(Countries) :-
  Countries = [A,B,C,D],
  applist(value,Countries),
  ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D).

value(red).
value(green).
value(blue).
```

This problem was also used to illustrate constraint logic programming in [Wallace, 1997]. ECL^{i}PS^e is a constraint logic programming language, using the same syntax as Prolog. Hopefully this syntax will already be familiar to many readers. At the same time, we also hope that any readers, who have suffered from the limitations of Prolog, will not automatically conclude that ECL^{i}PS^e suffers from the same limitations.

The problem involves four decisions, one for each country. These are modelled by the variables *A, B, C* and *D*. *Countries* is just a name for the list of four variables. Each decision variable, in this problem, has the same set of choices, modelled as possible values for the variables (*red, green* and *blue*). There are six constraints, each of which is modelled by the same relation (*ne* meaning *not equal to*).

The first command :- `lib(apply_macros).` loads an ECL^{i}PS^e library. Much of the functionality of ECL^{i}PS^e is held in different libraries, some of which will be introduced in the next section. The library *apply_macros* holds the definition of the *applist* predicate, which applies a predicate to each element of a list. *applist(value,Countries)* is equivalent to *value(A), value(B), value(C), value(D).*

2.2 Why Logic Programming

The requirements on ECL^{i}PS^e are of two kinds: to enable such problems to be modelled simply and naturally; and to enable the resulting problem model to be solved efficiently. The separation of modelling and solving is supported in ECL^{i}PS^e by distinguishing the conceptual model, expressed as a "pure" logical ECL^{i}PS^e program, from the design model, which is constructed from the conceptual model by adding control to the ECL^{i}PS^e program.

This combination of requirements is difficult to satisfy—perhaps impossible if a completely general modelling language is required, suitable for every kind of application. However the applications for which ECL^{i}PS^e is designed are decision support applications involving combinatorial problems.

Logic programming is peculiarly apt for modelling problems of this kind for two reasons:

- It is based on relations
- It supports logical variables.

Since every combinatorial problem is naturally modelled as a set of variables and a set of constraints (i.e. relations) on those variables, the facilities of logic programming precisely match the requirements for modelling combinatorial problems.

Every predicate in a logic program defines a relation, either explicitly as a set of facts, or implicitly in terms of rules. We can recall the example from [Wallace, 1997]. The predicate *meat* was defined by two facts:

```
meat(beef,5).
meat(pork,7).
```

whilst the predicate *main* (meaning "main course") was defined by two rules:

```
main(M,I) :- meat(M,I).
main(M,I) :- fish(M,I).
```

Variables in logic programming are logical variables. Thus it is entirely natural to initialise the problem variables (for example by writing *Countries* = [*A*, *B*, *C*, *D*]) and then to constrain them (for example by writing *ne(A,B)* and so on).

We briefly compare ECL^{i}PS^e as a modelling language with formal specification languages, mathematical modelling languages, mainstream programming languages and object oriented languages.

2.2.1 Formal Specification Languages

Formal specification language are designed for formality, but not for execution. Consequently they include constructs, such as universal quantification, which are precisely defined but are not constructive. In other words there are constructs which cannot be mapped onto any (practical) algorithm.

Luckily the class of problems for which ECL^{i}PS^e is designed have a finite set of decision variables each of which admits only finitely many alternatives. Consequently it is only necessary to support a restricted form of logic[2] which is easier to understand and easier to implement. The nearest thing ECL^{i}PS^e offers to universal quantification is iteration over finite sets, as for example the goal *applist(value, Countries)* in Section 2.1.

The restricted logic of ECL^{i}PS^e has a benefit that the mapping from the conceptual model of the problem to the design model is an extension of the conceptual model rather than a rewriting. This means that when problem requirements change it is natural to capture these changes in the conceptual model, and then carry them through to the design model. The result is that during application development the conceptual model and the design model remain in step. This avoids many of the pitfalls which await developers working on applications whose specifications are changing even during application development.

2.2.2 Mathematical Modelling Languages

There already exists a class of modelling languages designed for combinatorial problems. These are the mathematical modelling languages typically used as input to mixed integer programming (MIP) packages. We further discuss MIP, and how to use it through ECL^{i}PS^e, in Section 3.4 below.

Although the syntax is different, mathematical modelling languages share many of the features of logic programming. They support logical variables, and constraints. They support numerical constraints which, though not supported in traditional logic programs, are supported by constraint logic programs as we shall see in the following section. They support named constraints, which is achieved in constraint logic programming by introducing a predicate name, e.g. `precede(T1,T2) :- T1 >= T2.`

There are two facilities in constraint logic programming which are not available in mathematical modelling languages.

The main one is quite simple: in constraint logic programs it is possible to define a constraint which involves a disjunction. Mathematical programming cannot handle disjunction directly. The second difference is that logic programming allows new constraints to be defined in terms of existing ones, even recursively. In mathematical programming the model is essentially flat, which not only complicates the model but also reduces reusability within an application and across applications.

To illustrate the advantage of handling disjunction in the modelling language, we take a toy example and present two models: a mathematical

[2] technically called Horn clauses.

programming model and a constraint logic programming model.

Consider the constraint that two tasks sharing a single resource cannot be done at the same time. The constraint involves six variables: the start times S_1, S_2, the end times E_1, E_2, and resources R_1, R_2 of the two tasks. The specification of this constraint is as follows:

> *either* the two tasks use distinct resource (R_1 *ne* R_2) *or* $task_1$ ends before $task_2$ starts ($E_1 \leq S_2$) *or else* $task_2$ ends before $task_1$ starts ($E_2 \leq S_1$).

First we shall show how it can expressed as a mathematical model without disjunctions. For this purpose it must be encoded using numerical equations and inequalities, together with integer constraints.

The disjunctions can be captured by introducing three 0/1 variables, B_{r1}, B_{r2}, and B_t, and using some large constant, say 100000, larger than any possible values for any of the six variables. Now we can express the constraint in terms of numerical inequalities as follows:

$$R_1 + 100000 * B_{r1} + 100000 * B_{r2} \geq R_2 + 1$$
$$R_2 + 100000 * B_{r1} + 100000 * (1 - B_{r2}) \geq R_1 + 1$$
$$S_1 + 100000 * (1 - B_{r1}) + 100000 * B_t \geq E_2$$
$$S_2 + 100000 * (1 - B_{r1}) + 100000 * (1 - B_t) \geq E_1$$

If $B_{r1} = 0$ then the two tasks use different resources. In this case, if also $B_{r2} = 0$ then $R_1 \geq R_2 + 1$, otherwise $B_{r2} = 1$ and $R_2 \geq R_1 + 1$. It is an exercise for the reader to prove that if $B_{r1} = 0$ then the tasks can overlap. Otherwise, if $B_{r1} = 1$, then $B_t = 0$ entails $S_t \geq E_2$ and $B_t =1$ entails $S_2 \geq E_1$.

In ECL^{i}PS^e this constraint can be expressed directly in logic, as illustrated below:

```
taskResource(S1,E1,R1,S2,E2,R2) :-
   ne(R1,R2).
taskResource(S1,E1,R1,S2,E2,R2) :-
   R1=R2, S1 >= E2.
taskResource(S1,E1,R1,S2,E2,R2) :-
   R1=R2, S2 >= E1.
```

We note that the ECL^{i}PS^e model is a conceptual model, whilst the mathematical model is a design model. The point here is that in ECL^{i}PS^e both models can be expressed, whilst mathematical modelling can only express a design model. Indeed we shall show in Section 4.1 a design model written in ECL^{i}PS^e that is very close to the conceptual model.

Another ECL^{i}PS^e design model, which is also close to the conceptual model, is handled in ECL^{i}PS^e by an automatic translator which builds the MIP model and passes it to the MIP solver of ECL^{i}PS^e . This translator is described in [Rodosek et al., 1997].

Whilst the above example shows that such complex constraints can be expressed in terms of numerical inequalities, as required for MIP, the encoding is awkward and difficult to debug. It becomes increasingly difficult as the constraints become more complex (e.g. the current example immediately becomes even harder if the resources have a finite capacity greater than one).

Notice, finally, that the mathematical model requires resources to be identified by numbers, whilst the constraint logic programming model imposes no such restriction as we shall show in Section 4 below.

2.2.3 Mainstream Programming Languages

Naturally the implemented solution to an industrial problem must be delivered into the industrial computing environment. It is sometimes argued that this is only possible if the solution is implemented in a mainstream programming language such as **C**, **C++** or even **Java**. There are two arguments supporting this view, firstly that of how to embed it (it is easier and more efficient to pass data and control between modules written in the same programming language), and secondly that of system support (mainstream language programmers are much easier to find and replace than specialist programmers).

Whilst this argument only supports a mainstream programming language being used for implementation, and not conceptual modelling, it has consequences for the modelling language as well on the assumption, which we discussed above, that the conceptual model should be close to the design model. Thus if the design model is encoded in a mainstream programming language, then either the conceptual model is compromised, becoming more like a design model, or the gap between the conceptual model and design model grows very wide.

Sadly the attempt to tackle combinatorial problems with mainstream programming languages has too often foundered because the implemented solution has proven not to solve the actual industrial requirement (often because requirements change during application development). The solution cannot then be modified to meet the actual, or new, requirements within a reasonable cost and time-scale.

Given that the core combinatorial optimization problem is best solved by a specialised programming platform (either mathematical or constraint-based), the problem of embedding has to be solved.

One approach is to embed constraint solving in a mainstream programming language. However, as we shall see in Section 5 below, search and constraint handling are closely interdependent. Even if the search is encoded in a mainstream programming language, the programmer is required to understand in detail not only the data structures used by the constraint handlers, but their operational behaviour.

In practice, packages providing an embedding of constraints in mainstream programming languages also encapsulate search within the package. The application developer is required to control the search. To avoid any mismatch between the host programming language and search control within the package, a popular approach is to implement the package as a library of the host programming language.

The result is that the separation of conceptual modelling and design modelling is given up in favour of staying within the confines of the expressive capabilities of the host programming language. This approach not only requires specialist programmers to develop and support the application, but it also sacrifices the modelling advantages of mathematical and constraint logic programming.

In fact the problem of embedding has been overcome, although first generation constraint logic programming languages were deficient in this area. ECL^{i}PS^e can be fully embedded in C and C^{++}, and indeed uses an external solver, written in C to handle linear constraints, since the runtime cost of such an interface is perfectly acceptable even for a tightly integrated component such as a constraint handler.

2.2.4 Object Oriented Languages

ECL^{i}PS^e supports object-orientation through two distinct features, *modules* and *structures*. Modules support *behavioural* object orientation, and structures support *structural* object orientation.

Because of the nature of combinatorial problems, the only requirement for behavioural object orientation is in the constraint handlers. The implementation of each constraints library is hidden inside a module, and access to the internal data structures is only through predicates exported from the module.

The remaining objects that can occur in an ECL^{i}PS^e model have attributes but no behaviour, and so they require only structural object orientation.

In our first example we modelled a map colouring problem using only variables and constraints. It can be argued, however, that for more complex applications, the conceptual model can benefit from a notion of object, into which variables can be built. For example in modelling a resource scheduling problem the notion of a *task* with certain attributes is useful. A task might have an *identifier*, a *start time*, an *end time* and a *duration*. as shown in the example below.

After declaring structures for *tasks* and *times*, the programmer can access any of their attributes independently.

Each ECL^{i}PS^e prompt (e.g. `[eclipse 1]:`) is followed by a user query (e.g. lib(structures).). In the rest of this paper, "query *N*" always refers to the query which is preceded by the prompt `[eclipse N]:`.

```
[eclipse 1]: lib(structures).
*  structures loaded
```

```
[eclipse 2]: define_struct(task(id, start, end, dura-
tion)).
             define_struct(time(hour, minute)).
*  yes.

[eclipse 3]: T=task with [id:a, duration:10].
*  T = task(a, _, _, 10)
*  yes.

[eclipse 4]: T1=task  with [id:a3,start:S3,end:(time with
hour:H3)],
             T2=task  with [id:a4,start:S3,end:(time with
hour:H4)],
             H3>H4.
*  T1 = task(a3, S3, time(H3, _), _)
*  T2 = task(a4, S3, time(H4, _), _)
*  yes.
```

The programmer enters `lib(structures).` to which the system responds `structures loaded`. I have added a star to the beginning of each line showing a system response.

Query 2 defines the attributes for objects in the classes *task* and *time*. Query 3 shows how the user can equate a variable with a structured object (i.e. the variable is instantiated to the structure). ECL^{i}PS^e automatically constructs unknown values (written _) for the unspecified attributes.

Query 4 illustrates something of the expressive power needed in a constraint programming language which supports objects. Not only do the objects T1 and T2 share an attribute value—this is a shared sub-object—but they also have non-shared sub-objects of which the attributes are connected by a constraint. Such a constraint, between distinct objects, is not expressible within the traditional object-oriented framework.

2.3 The Conceptual Model and the Design Model

The main benefit of constraint logic programming over other platforms for solving combinatorial problems is in the closeness between the conceptual model and the design model. ECL^{i}PS^e takes full advantage of this by offering facilities to choose different annotations of the same conceptual model to achieve design models which, whilst syntactically similar, can have radically different behaviour.

2.3.1 Map Colouring

Let us start by mapping the conceptual model for the map colouring example, discussed in Section 2.1, into a design model which uses the finite domain constraint handler of ECL^{i}PS^e.

The design model is encoded as shown below:

```
:- lib(fd).
```

```
coloured(Countries) :-
   Countries=[A,B,C,D],
   Countries :: [red,green,blue],
   ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
   labeling(Countries).

ne(X,Y) :- X##Y.
```

The design model extends the conceptual model in four ways:

1. The ECL^{i}PS^e finite domain library is loaded (using :- *lib(fd)*)
2. An explicit finite domain is associated with each decision variable (using *Countries :: [red, green, blue]*)
3. The finite domain built-in disequality constraint is used to implement the *ne* constraint (*using ne(X,Y) :- X##Y*). ## is a special syntax for disequality used by the finite domain constraint solver
4. This program includes a search algorithm, invoked by the goal *labeling(Countries).* As we shall see later, this predicate tries choosing, for each of the variables *A, B, C* and *D* in turn, a value from its domain. It succeeds when a combination of values has been found that satisfies the constraints.

Naturally this is a toy example, and it is not always so easy to turn a conceptual model, such as the ECL^{i}PS^e program in Section 2.1, into a design model, such as the program listed above. Nevertheless constraint logic programming, and in particular ECL^{i}PS^e, have made much progress in achieving a close relationship between the conceptual model and the design model. The different components of the ECL^{i}PS^e system all support the separate development of a clear, correct conceptual model, and an efficient design model, and they also support the mapping between the two.

2.3.2 Having Enough Change in Your Pocket

Let us now take a more interesting problem, which has been set as a recent challenge within the MIP community. The problem is apparently rather simple: what is the minimum number of coins a purchaser needs in their pocket in order to be able to buy any one item costing up to one pound, and guarantee to be able to pay the exact amount?

The problem involves only six decision variables, one for the number of coins of each denomination held in the pocket (the denominations are 1, 2, 5, 10, 20, 50).

The conceptual model for this problem is as follows:

```
:- lib(apply_macros).

solve(PocketCoins,Min) :-
   PocketCoins=[P,Tw,Fv,Te,Twe,Ff],                    %1
   applist(range(0,99),[Min|PocketCoins]),               %2
   Min = P+Tw+Fv+Te+Twe+Ff,                              %3
   fromto(1,99,genc(PocketCoins)),                       %4
   minimize(Min).                                        %5

genc(PocketCoins,Total) :-
   Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],                      %6
   applist(range(0,99),Coins),                           %7
   Total = P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,         %8
   maplist( '<=',Coins,PocketCoins).                     %9
```

The lines are numbered, using the syntax %N, as % is a comment symbol in ECL^{i}PS^e. This program is now described line-by-line.

1. The variable *PocketCoins* is just a shorthand for the list of six variables, *[P, Tw, Fv, Te, Twe, Ff]* which denote the number of coins of each denomination held in the pocket.
2. *[A,B,C]* is a list, but ECL^{i}PS^e allows lists to be written in an alternative syntax *[Head | Tail]*. Thus *[Min | PocketCoins]* is simply another way of writing the list of seven variables, *[Min, P, Tw, Fv, Te, Twe, Ff]*. The command *applist(range(0,99), [Min \ PocketCoins])* associates a range (between 0 and 99) with each of the variables.
3. *Min* is the total number of coins in the pocket, as enforced by the equation *Min = P+Tw+Fv+Te+Twe+Ff.*
4. To ensure that these coins are enough to make up any total between 1 and 99, we now impose 99 further constraints, one for each total. *genc(PocketCoins,Total)* is called for each value of *Total* between 1 and 99.
5. *minimize(Min)* simply specifies that the best feasible solution to the problem is one which minimises the value of the variable *Min.*
6. *genc(PocketCoins,Total)* initialises another set of coins *[P1, Tw1, Fv1, Te1, Twe1, Ff1]* needed to make up the total *Total.*
7. This set of coins is also initialised to range between 0 and 99.
8. Their total value is constrained to be equal to *Total.* This constraint is enforced by the equation *Total = P1+ 2*Tw1 + 5*Fv1 + 10*Te1 + 20*Twe1 + 50*Ff1.*
9. Finally the constraint that the required coins of each denomination must be less than, or equal to, the number of coins of that denomination in the pocket, is enforced by the constraints: *P1 <= P, Tw1 <= Tw, Fv1 <= Fv, Te1 <= Te, Twe1 <= Twe, Ff1 <= Ff.* These constraints are generated by the single command *maplist(<=, Coins, PocketCoins).*

Let's start by trying mixed integer programming on this problem. To do this we add *integer* declarations for each of the integer variables, and change the constraints to use the syntax recognised by the (external) MIP solver accessed via the ECLⁱPS^e library eplex. For equality we use the syntax `$=`, and for inequality we use `$>=`. The design model is shown:

```
:- lib(apply_macros).
:- lib(eplex).

solve(PocketCoins,Cost) :-
   PocketCoins=[P,Tw,Fv,Te,Twe,Ff],
   applist(range(0,99),[Min|PocketCoins]),
   Min $= P+Tw+Fv+Te+Twe+Ff,
   fromto(1,99,genc(PocketCoins)),
   optimize(min(Min),Cost).

genc(PocketCoins,Total) :-
   Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],
   applist(range(0,99),Coins),
   Total $= P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,
   maplist('$=<',Coins,PocketCoins).

range(Min,Max,Var) :-
   integers(Var),
   Var $>= Min,
   Var $=< Max.
```

This program passes all the `$=` and `$>=` constraints to the *CPLEX* mixed integer programming package (CPL93), and invokes the CPLEX branch and bound solver, to minimise the value of the variable *Min*. This minimum is placed in the variable *Cost*.

As such this model can only solve the problem of producing the exact change up to 59 pence (replacing 99 with 59 in the above program). For the full problem the system runs out of memory. There are standard MIP solutions to this problem: these can run for many hours and it is a tough challenge to reduce this time to minutes.

An ECLⁱPS^e program for solving the "Coins" problem using the facilities of the ECLⁱPS^e finite domain constraint solver implemented in the ECLⁱPS^e the library *fd* is given below:

```
:- lib(apply_macros).
:- lib(fd).

solve(PocketCoins,Min) :-
   PocketCoins=[P,Tw,Fv,Te,Twe,Ff],
   applist(range(0,99),[Min|PocketCoins]),
   Min #= P+Tw+Fv+Te+Twe+Ff,
   fromto(1,99,genc(PocketCoins)),
   minimize(labeling(PocketCoins),Min).
```

```
genc(PocketCoins,Total) :-
   Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],
   applist(range(0,99),Coins),
   Total #= P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,
   maplist('#<=',Coins,PocketCoins).

range(Min,Max,Var) :-
   Var::Min..Max
```

In this case the **#=** and **#>=** constraints and the optimization predicate *minimize* are implemented in the ECL^{i}PS^e finite domain library. This program proves within a few seconds that the minimum number of coins a purchaser needs in his pocket to make up any total between 1 and 99 is eight coins. One solution is: *P*=1, *Tw*=2, *Fv*=1, *Te*=1, *Twe*=2, *Ff*=1.

We have shown how the same underlying model for the "Coins" problem can be passed to different solvers so as to use the best one. However in ECL^{i}PS^e it is not a choice of either/or: the same constraints can easily be passed to several solvers at the same time. For instance we can define **X $#= Y** to be both **X $= Y** and **X #= Y** and replace = in the above model with **$#=**. We can treat **>=** similarly.

Whilst for this problem the finite domain solver alone solves the problem most efficiently, we have encountered practical examples where the combination of both solvers outperforms each on its own.

3. Solvers and Syntax

ECL^{i}PS^e offers several different libraries for handling symbolic and numeric constraints. They are the *fd* (finite domain) library, the *range* library, the *ria* (real interval arithmetic) library, and finally the *eplex* (MIP) library.

3.1 The *fd* (Finite Domain) Library

The finite domain library has been used and refined over a 10 year period. As a result it has a great many constraint handling facilities. It is best seen as three libraries.

The first is a library for handling symbolic finite domains, with values like *red, machine_1* etc. The built-in constraints on symbolic finite domain variables are equality and disequality: these constraints can only hold between expressions which are either constants or variables. These constraints can also be used when the domains are numeric.

The second is a library for handling integer variables, and numerical constraints on those variables. The library propagates equations and inequalities between linear expressions. A linear numeric expression is one that can be written in the form $Term_1 + Term_2 + \dots + Term_n$, where each term can, in turn, be written *Number* or *Number * Variable*. The number can be positive or negative.

An example is the expression 3 * X + (-4) * Y + 3 (which we would simply write 3 * X - 4 * Y + 3).

The third is a library supporting some built-in complex constraints. Two examples of such constraints are the **alldistinct** constraint, which constraints a set of variables to take values which are pairwise distinct, and the **atmost** constraint, which constrains at most *N* variables from a given set to take a certain value.

3.1.1 The fd Symbolic Finite Domain Facilities

In Sections 2.1 and 2.3.1, above, we showed a map colouring problem and its solution. The domains associated with the countries were *red, green* and *blue*. These were declared as finite domains, with the usual syntax: *X :: [red, green, blue]*.

The problem could have been modelled using numbers to represent colours, so there is no extra power in allowing symbolic finite domains as well as numeric ones. However when developing ECL^{i}PS^e programs for real problems, it is a very great help to use meaningful names so as to distinguish different types of finite domain variables. In particular it is crucial during debugging.

The basic constraints on finite domain variables, together with predicates for accessing and searching these domains are illustrated below:

```
[eclipse 1]: lib(fd).
*   fd loaded

[eclipse 2]:  X::[a,b,c].
*   X = X{[a, b, c]}
*   yes.

[eclipse 3]: X::[a, 3.1, 7].
*   X = X{[3.0999999, 7, a]}
*   yes.

[eclipse 4]: X::[a,b,c], dom(X,List).
*   X = X{[a, b, c]}
*   List = [a, b, c]
*   yes.

[eclipse 5]: X::[a,b,c], Y::[b,c,d], X#=Y.
*   X = X{[b, c]}
*   Y = X{[b, c]}
*   yes.

[eclipse 6]: X::[a,b,c], X##b.
*   X = X{[a, c]}
*   yes.

[eclipse 7]: X::[a,b,c], indomain(X).
*   X = a     More? (;)
*   X = b      More? (;)
*   X = c
*   yes.

[eclipse 8]:  [X,Y,Z]::[a,b,c],  X##Y,  Y##Z,  X##Z,
labeling([X,Y,Z]).
*   X = a
*   Y = b
*   Z = c      More? (;)

*   X = a
*   Y = c
*   Z = b      More? (;)
*   yes.

[eclipse 9]: [X,Z]::[a,b,c], Y::[a,c],
   deleteff(Var,[X,Y,Z],Rest), indomain(Var).

*   X = X{[a, b, c]}
*   Y = a
*   Z = Z{[a, b, c]}
*   Rest = [X{[a, b, c]}, Z{[a, b, c]}]
*   Var = a     More? (;)
*   yes.
```

The programmer enters `lib(fd).`to which the system responds **`fd loaded`**.

The second query associates a symbolic finite domain with the variable *X*. In response ECL^{i}PS^e prints out the variable name and its newly assigned domain. The fact that the variable has an associated domain does not require any changes in other parts of the program, where X may be treated as an ordinary variable.

Query 3 shows that symbolic domains can include values of different types.

Query 4 shows the use of the *dom* predicate to retrieve the domain associated with a variable.

Queries 5 and 6 illustrate the equality and disequality constraints, and their effects on the domains of the variables involved. Finite domain constraints use a special syntax to make explicit which constraint library is to handle the constraint, for example it uses **#=** instead of **=**.

Queries 7, 8 and 9 illustrate search. Strictly one would not expect search predicates to belong to a constraint library, but in fact search and constraint propagation are closely connected.

Query 7 shows the *indomain* predicate instantiating a domain variable *X* to a value in its domain. ECL^{i}PS^e asks if more answers are required, and when the user does indeed ask for more, another value from the domain of *X* is chosen, and *X* is instantiated to that value instead. When the user asks for more again, *X* is instantiated to the third and last value in its domain, and this time ECLⁱPS^e doesn't offer the user any further choices, but simply outputs *yes*.

Query 8 illustrates the built-in finite domain *labelling* predicate. This predicate simply invokes *indomain* on each variable in turn in its argument. In this case it calls *indomain* first on X, then Y and then Z. However the variables are constrained to take different values by three disequality constraints, and only those labellings that satisfy the constraints are admitted. Consequently this query has six different answers, though the user stops asking for more after the second answer.

Query 9 illustrates a heuristic based on the *fail first* principle. In choosing the next decision to make, when solving a problem, it is often best to make the choice with the fewest alternatives first. The predicate *deleteff* selects a variable from a set of variables which has the fewest alternatives: i.e. the smallest finite domain. In the example there are three variables, X, Y and Z representing three decisions, *deleteff* picks out Y because it has the smallest domain, and then *indomain* selects a value for Y. The third argument of *deleteff* is an output argument: *Rest* returns the remaining variables after the selected one has been removed. These are the decisions yet to be made.

3.1.2 The *fd* Integer Arithmetic Facilities

For numeric finite domains the *fd* library admits equations, inequalities and disequalities over numeric expressions.

Additionally the *fd* library includes some built-in optimization predicates. These are all illustrated below:

```
[eclipse 1]: lib(fd).
*  fd loaded

[eclipse 2]: X::1..10.
*  X = X{[1..10]}
*  yes.

[eclipse 3]: X::1..10, mindomain(X,Min).
*  X = X{[1..10]}
*  Min = 1
*  yes.

[eclipse 4]: [X,Y]::1..10, X#>Y+1.
*  X = X{[3..10]}
*  Y = Y{[1..8]}
*  yes.

[eclipse 5]: [X,Y]::1..10, X#>Y+1, Y#=6.
*  X = X{[8..10]}
*  Y = 6
*  yes.
```

```
[eclipse 6]: [X,Y,Z]::1..10, X #= 2*(Y+Z).
*   X = X{[4..10]}
*   Y = Y{[1..4]}
*   Z = Z{[1..4]}
*   yes.

[eclipse 7]: X::1..10, mindomain(X,Min).
*   X = X{[1..10]}
*   Min = 1
*   yes.

[eclipse 8]: [X,Y,Z]::1..10, X #= 2*(Y+Z), Y##Z,
    minimize(labeling([X,Y,Z]),X).
*   Found a solution with cost 6
*   Y = 2
*   Z = 1
*   X = 6
*   yes.
```

Query 2 illustrates how a numeric finite domain can be initialised just by giving lower and upper bounds, instead of the whole list of members. In fact, internally, finite domains are stored as lists of intervals (for example *[1..5, 8..10, 15]*).

Query 3 shows how the user can find out the lower bound of a variable's numeric finite domain. There is a similar predicate for retrieving the upper bound.

Queries 4, 5 and 6 illustrate some features of finite domain constraint propagation.

Query 4 shows the pruning achieved by a simple numerical finite domain constraint. Notice that both the domains of *X* and *Y* are pruned—constraints work in all directions.

Query 5 illustrates that a finite domain constraint remains active even after it has achieved some pruning. This query is the same as query 3, with an extra constraint imposed subsequently. The *X # > Y + 1* constraint is still active, and prunes the domain of *X* still further from *[3..10]* to *[8..10]*.

Query 6 shows that, in the interest of computational efficiency, the mathematical constraints only narrow the bounds of the finite domains. In this example the domain of *X* could theoretically be reduced to *[4,6,8,10]*, but this would require much more computation - especially if the finite domains were quite large!

Query 7 is an example of the use of the built-in *minimize* predicate. This predicate returns an admissible labelling of the variables *X*, *Y* and *Z* which yields the smallest value for *X*. In general any search procedure can be substituted for *labelling([X,Y,Z])* as the first argument to *minimize*. For example we could have used *minimize((indomain(X), indomain(Y), indomain(Z)), X)*.

3.1.3 The *fd* Complex Constraints

There are two motivations for supporting complex constraints. One is to simplify problem modelling. It is shorter, and more natural, to use a single *alldistinct* constraint on *N* variables than to use $n*(n-1)/2$ (pairwise) disequalities.

The second motivation is to achieve specialised constraint propagation behaviour. The **`alldistinct`** constraint on *N* variables, has the same semantics as $n*(n-1)/2$ (pairwise) disequalities, but it can also achieve better propagation than would be possible with the disequalities. For example if any *M* of the variables have the same domain, and its size is less than *M*, then the **`alldistinct`** constraint can immediately fail. However if two variables X and Y have the same domain, with $M>1$ elements, the constraint *X ## Y* can achieve no propagation at all. Thus the pairwise disequalities are unable to achieve the same propagation as the *alldistinct* constraint.

The constraint *atmost(Number, List, Val)* constrains the number of the variables, *Number*, in the list, *List*, to be not greater than the the value, *Val*. This is a difficult constraint to express using logic. One way is to constrain each sublist of length *Number+1* to contain a variable with value different from *Val*, but the resulting number of constraints can be very large.

A more natural way is to constrain all the variables to take a value different from *Val*, and to allow the constraint to be violated up to *N* times. The *fd* library supports such a facility with the constraint *#=0(T1, T2, B)*. This constraint makes *B=1* if *T1=T2* and *B=0* otherwise. It is possible to express *atmost* by imposing the constraint $\# = (Var_i, Val, B_i)$ or each variable Var_i in the list and then adding the constraint $B_1 + \ldots + B_m \# <= N$. The built-in *atmost* constraint is essentially implemented like this.

The other *fd* constraints (**#<**, **#>**, etc.) can be extended with an extra *0/1* variable in the same way.

The *fd* library includes a great variety of facilities, which are best explored by obtaining the ECL^{i}PS^e extensions manual [Brisset et al., 1997] and looking at the programming examples in the section on the *fd* library there.

3.2 The *range* Library

The range library does very little itself, but it provides a common basis for the interval and the MIP libraries. By contrast with the *finite domain* library, the *range* library admits ranges of which the upper and lower bound are either real numbers or integers. The library enables the programmer to associate a range with one or more variables, as illustrated below:

```
[eclipse 1]: lib(range).
*  range loaded

[eclipse 2]: X::0.0..9.5, lwb(X,4.5).
*  X = X{4.5 .. 9.5}
*  yes.
```

```
[eclipse 3]: X::4.5..9.5, X=6.0.
*  X = 6.0
*  yes.

[eclipse 4]: X::4.5..9.5, X=1.0.
*  no (more) solution.

[eclipse 5]: X::0.0..9.5, lwb(X,4.5), integers([X]).
*  X = X{5 .. 9}
*  yes.
```

In query 2, the programmer enters `X::0.0..9.5, lwb(X,4.5).`, and the system responds by printing out the resulting range. When the variable is instantiated, the range is checked for compatibility, as shown by queries 3 and 4.

Finally, what might be treated as type information in other programming paradigms, can be treated as a constraint in the constraint programming paradigm. Thus we can add a constraint that something is an integer in the middle of a program, as shown by query 5.

3.3 The *ria* (Real Interval Arithmetic) Library

The *ria* library supports numeric constraints which may involve several variables. Throughout program execution, *ria* continually narrows the ranges associated with the variables as far as possible based on these constraints. In other words *ria* supports propagation of intervals, using the range library to record the current ranges, and to detect inconsistencies.

The constraints handled by *ria* are equations and inequalities between numerical expressions. The expressions can be quite complex, they can include polynomials and trigonometrical functions. This functionality is quite similar to that offered by *fd*, except that *fd* can only propagate linear constraints. On the other hand, the finite domain library uses integer arithmetic instead of real number arithmetic, so it is in general more efficient than *ria*.

Since interval propagation will be described in the next issue of the Journal [Yakhno et al., to be published], we shall confine ourselves here to a single example showing *ria* at work.

Suppose we wish to build a garden house, whose corners must lie on a given circle. The house should be a regular polygon, but may have any number of sides. It should be as large as possible within these limitations. (Note that the more sides the larger the area covered, until it covers practically the whole of the circle.) However each extra side incurs a fixed cost. The problem is to decide how many sides the garden house should have?

If it had six sides, the house would look as illustrated overleaf in Figure 1.

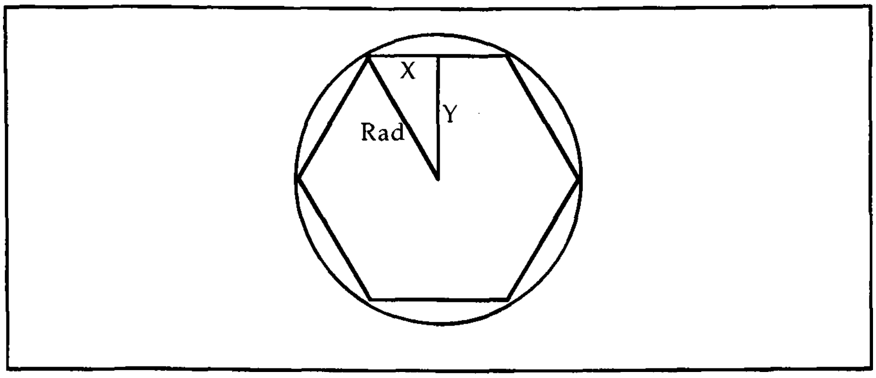


Figure 1: The Garden House

The area of the house is *2*6*A* where *A* is the area of the triangle in the illustration. The area of an *N*-sided house can be modelled in ECLⁱPS^e as shown below:

```
:- lib(ria).

area(N,Rad,Area) :-
   X*>=0, Y*>=0, N*>=3, integers(N),
   Rad *>=0, Area*>=0,
   Area *=< pi*sqr(Rad),
   cos(pi/N) *= Y/Rad,
   sqr(Y)+sqr(X) *= sqr(Rad),
   Area *= N*X*Y.

cost(N,Rad,W1,W2,Cost) :-
  W1*>=1, W2*>=1, Cost *>=0,
   area(N,Rad,Area),
   Cost *= W1*Area-W2*N.

tcost(N,Cost) :-
   cost(N,10,1,10,Cost).
```

N is the number of sides, and *Area* is the area of the house. The variable *Rad* denotes the radius of the circle, and *X* and *Y* are the lengths of the sides of the triangle, as illustrated in Figure 1.

ria requires its constraints to be written with a specific syntax (eg `X *>= Y` instead of just `X >= Y`). This distinguishes *ria* constraints from linear and finite domain constraints, which each have their own special syntax.

To work out the payoff between the area and the number of sides, we define the cost-benefit of the house to be *W1*Area-W2*N*, where *W1* and *W2* are weighting factors that we can chose to reflect the relative costs and benefits of the area against the number of sides. In the model shown above, *tcost* returns the cost-benefit of an N-sided house in case the radius of the circle is 10, and the weights are *W1=1* and *W2=10.*

We can place this program in a file called *house.pl*, and then use ECLⁱPS^e to find out some costs by first "consulting" the file, as illustrated by query 1 below:

```
[eclipse 1]: [house].
*  range  loaded
*  house.pl   compiled
*  yes.

[eclipse 2]: tcost(3,C).
*  C = C{99.90 .. 99.9}
*  yes.

[eclipse 3]: tcost(4,C).
*  C = C{160 .. 160}
*  yes

[eclipse 4]: tcost(6,C).
*  C = C{200 .. 200}
*  yes.

[eclipse 5]: tcost(7,C).
*  C = C{204 .. 204}
*  yes.

[eclipse 6]: tcost(8, C).
*  C = C{203 .. 203}
*  yes.

[eclipse 7]: tcost(N,C).
*  N = N{3 .. 31}
*  C = C{0.0 .. 284}
*  yes.

[eclipse 8]: tcost(N,C), squash([C],1e-2,lin).
*  N = N{3 .. 31}
*  C = C{0.0 .. 224}
```

Queries 2-6 would seem to indicate that the seven sided house is best for the given cost weightings.[3]

However it is also interesting to see whether the interval reasoning system itself can achieve useful propagation without even knowing the number of sides of the house. We show this in query 7.

An upper bound on the number of sides is extracted due to the constraint that the cost-benefit must be positive, but the propagation on the cost-benefit is rather weak. In cases like this, propagation can be augmented by a technique known as squashing, as illustrated in query 8. This tech-

[3] The intervals returned from ria are much narrower than this, but for this paper I have reduced the output to three significant figures.

nique will be described in detail in [Yakno et al., to be published].

We now give two short examples showing limitations of interval reasoning in general. This will motivate the introduction of a linear constraint solver in ECL^{i}PS^e, to be described in Section 3.4.

The two limitations are that interval reasoning cannot, even in some quite simple examples, detect inconsistency among the constraints; and in cases where the constraints have only one solution, interval reasoning often fails to reflect this in the results of propagation.

This is illustrated by the two simple examples below.

```
[eclipse 1]: lib(ria).
*   ria loaded

[eclipse 2]: X+Y *=<1, Z+X*=<1, Y+Z*=<1, X+Y+Z*>=2.
*   X = X{-Inf.0 .. Inf.0}
*   Y = Y{-Inf.0 .. Inf.0}
*   Z = Z{-Inf.0 .. Inf.0}
*   yes

[eclipse 3]: X+Y *= 2, X-Y *= 0.
*   X = X{-Inf.0 .. Inf.0}
*   Y = Y{-Inf.0 .. Inf.0}
*   yes
```

In this case the system failed to detect the inconsistency in query 2, and did not deduce that only one value was possible for *X* and *Y* in query 3. The answer is not incorrect, as *ria* only guarantees that any possible answers must lie in the intervals returned - it does not guarantee the existence of an answer in that interval. Nevertheless it would be useful to have available a more powerful solver to recognise cases such as these.

3.4 The *eplex* (External CPLEX Solver Interface) Library

Equations and inequalities between linear numeric expressions, as defined in Section 3.1 above, are a subset of the constraints which can be handled by *ria*. However this class can be handled very powerfully, so much so that any inconsistency between the constraints is guaranteed to be detected. Techniques for solving linear constraints have been at the heart of operations research for half a century, and highly efficient linear solvers have been developed.

One of the most widely distributed, scalable and efficient packages incorporating a linear constraint solver is the CPLEX MIP package (CPL93). CPLEX offers several algorithms for solving linear constraints including the Simplex and Dual Simplex algorithms. These algorithms are supported by sophisticated data structures, and the package can handle problems involving ten of thousands of linear constraints over ten of thousands of variables.

In the discussion so far, we have not yet mentioned an important aspect of most industrial combinatorial problems. Not only is it required to make decisions that satisfy the constraints, but they must also be chosen to optimize some measure. In the travelling salesman problem for example, the decisions of what order to visit the cities are based on optimizing the total distance travelled by the salesman.

One feature of available packages for solving linear and mixed integer problems, is support for optimization. Indeed they not only offer optimization as a facility, but require it. Therefore in illustrating the two example where *ria* performed badly, using instead the *eplex* library, we shall insert a dummy optimization function. The use of *eplex* is shown below:

```
[eclipse 1]: lib(eplex).
*   eplex loaded

[eclipse 2]: X+Y $=< 1, Z+X $=< 1, Y+Z $=< 1, X+Y+Z $>= 2,
    Opt $= 0, optimize(min(Opt),Cost).
*   no (more) solution.

[eclipse 3]: X+Y $= 2, X-Y $= 0, optimize(min(X),Cost).
*   X = 1.0
*   Y = 1.0
*   Cost = 0.0
*   yes.

[eclipse 4]: X+Y $= 2, X-Y $= 0, optimize(max(X),Cost).
*   X = 1.0
*   Y = 1.0
*   Cost = 0.0
*   yes.
```

Where *fd* uses a **#**, and *ria* uses a *****, *eplex* uses a **$**.

Query 2 is the same set of constraints whose inconsistency is not detected by *ria*. *eplex*, however, recognises their inconsistency.

In order to establish that there is only one possible value for *X* we have had to use two queries, 3 and 4, first finding the minimum value for *X* and then the maximum. Although the same value for *Y* was returned in both solutions, the *eplex* library has still not proven that *1* is the only possible value for *Y*.

For problems involving only real number (or *continuous*) variables, linear constraint solving alone suffices to solve the problem. However industrial problems typically include a mixture of real number and integer variables. For example in problems involving discrete resources the decision as to which resource to use for a task cannot be modelled as a continuous variable.. Traditionally operational researchers will use a binary (or *0/1*) variable or an integer variable. Most resources are discrete, for example machines for jobs, vehicles for deliveries, rooms for meetings, berths for ships, people for projects, and so on. Another fundamental use of discrete variables is in modelling the decision as to the order of doing things—for example, visiting cities in the travelling salesman problem, or performing tasks on the same machine. From the point of view of the programmer adding the constraint that a variable is integer-valued is straightforward. However the effect of such a constraint on the performance of the solver can be disastrous, because mixed integer problems are much harder to solve than linear problems.

```
[eclipse 1]: lib(eplex).
*   eplex loaded

[eclipse 2]: X+Y $>= 3, X-Y $= 0, optimize(min(X), C).
*   Y = 1.5
*   X = 1.5
*   C = 1.5
*   yes.

[eclipse 3]: integers([X]), X+Y $>= 3, X-Y $= 0,
    optimize(min(X), C).
*   Y = 2.0
*   X = 2
*   C = 2.0
*   yes.
```

The *eplex* library uses standard range-variables provided by the range-library, which facilitates interfacing to other solvers. The interface to CPLEX enables state information to be retrieved, such as constraint slacks, basis information, and reduced costs. Also many parameters can be queried and modified. A quite generic solver demon is provided which makes it easy to use CPLEX within a data-driven CLP setting.

The notion of solver-handlers encourages experiments with multiple solvers. A pair of predicates make it possible to read and write problem files in MPS or LP format.

MIP packages such as CPLEX and XPRESS , which is also currently being integrated into the *eplex* package, are surprisingly effective even for some problems involving many discrete variables. Their underlying linear solvers reflect a carefully chosen balance between flexibility and scalability. They offer less flexibility than the linear solvers which are usually built into constraint programming systems, such as *CLP($\Re$)*, but much better scalability.

It has proved possible, within ECL^{i}PS^e, to achieve much of the flexibility of *CLP($\Re$)* within the restrictions imposed by MIP solvers [Rodosek et al., 1997].

4. Complex Constraints

Whilst constraint programming languages offer a broad selection of built-in constraints, each new industrial application typically requires a number of application-specific constraints which aren't among the built-in constraints.

Let us take, as an ongoing example, the constraint that two tasks sharing a single resource cannot be done at the same time. This constraint was introduced in Section 2.2.2 above.

The constraint involves six variables: the start times S_1, S_2, end times E_1, E_2 and resources R_1, R_2 of the two tasks. The specification of this constraint is as follows:

> *either* the two tasks use distinct resource (R_1 *ne* R_2) *or* task$_1$ ends before task$_2$ starts ($E_1 \leq S_2$) *or else* task$_2$ ends before task$_1$ starts ($E_2 \leq S_1$).

We shall compare three different ways of handling this constraint. First we recall how it can be encoded using numerical equations and inequalities, together with integer constraints. This is the encoding necessary to allow it to be solved using MIP algorithms, as available through the *eplex* library. However the MIP package is not necessarily the best algorithm for handling such a constraint.

Indeed experience with practical applications suggests that the more *0/1* variables it is necessary to introduce to handle each constraint, the less efficient MIP becomes. The inefficiency arises partly because the MIP constraints are handled globally and the cost of handling extra constraints and boolean variables increases very fast with their number.[4] An additional factor is that until the boolean variables take a value very close to *0* they have very little effect on the search.[5]

By contrast we shall show how it can be handled using two further libraries of ECL^{i}PS^e - the *propia* library and the *chr* library. These libraries allow the constraint to be modelled much more simply and handled more efficiently.

4.1 The *propia* (Generalised Propagation) Library

A major issue in defining complex constraints is how to handle disjunction. The resource constraint of our running example can be quite easily expressed using a disjunction of finite domain constraints. Indeed ECL^{i}PS^e allows us to express disjunction as alternative clauses defining a predicate, so the constraint can be expressed as a single ECL^{i}PS^e predicate thus:

[4] Using the Simplex or Dual Simplex algorithms this cost goes up, in the worst case, exponentially with the number of constraints and variables.

[5] Technically the constraints in which they appear are rarely facet-inducing cuts.

```
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1 ## R2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1#=R2, S1 #>= E2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1#=R2, S2 #>= E1.
```

The purpose of the *propia* library is to take exactly such disjunctive definitions and turn them into constraints.

This is illustrated in below:

```
:- lib(propia).

propiaTR(S1,R1,S2,R2) :-
   [S1,S2]::0..100, [R1,R2]::[r1,r2,r3],
   E1 = S1+50,  E2 = S2+70,
   fdTaskResource(S1,E1,R1,S2,E2,R2) infers most.

fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1 ## R2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1#=R2, S1 #>= E2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
   R1#=R2, S2 #>= E1.
```

The syntax *Goal infers most* turns any ECL^{i}PS^e goal into a constraint. It is supported by the *propia* library.

The behaviour of this constraint is to find which values for each variable are consistent with the constraint. The constraint has the propagation behaviour described in [Wallace, 1997]: it repeatedly attempts to reduce the domains of its variables further every time any other constraints reduce any of these domains. Some examples of this behaviour will now be discussed with reference to the following:

```
 [eclipse 1]: [fdTaskResource].
*    propia     loaded
*    fdTaskResource.pl compiled.
*    yes.

[eclipse 2]: propiaTR(S1, R1, S2, R2), R1#=r1, R2#=r1.
*    S1 = S1{[0..50, 70..100]}
*    R1 = r1,
*    S2 = S2{[0..30, 50..100]},
*    R2 = r1
*    yes.

[eclipse 3]: propiaTR(S1,R1,S2,R2), R1=r1, S2#>=35, S2#<=45.
*    S1 = S1{[0..100]}
*    R1 = r1,
*    S2 = S2{[35..45]},
*    R2 = R2{[r2, r3]}
*    yes.
```

In query 2, the constraint deduces that, since the tasks cannot overlap, $task_1$ cannot start between *51* and *69*, and $task_2$ cannot start between *31* and *49*. In query 3, since the tasks are bound to overlap, the constraint deduces that $task_2$ must use either resource r_2 or r_3.

Other behaviour can be achieved by writing *Goal infers consistent* or *Goal infers ground* instead. These behaviours, together with other facilities of the *propia* library are described in the ECL^{i}PS^e extensions manual [Brisset et al., 1997].

4.2 The *chr* (Constraint Handling Rules) Library

The ECL^{i}PS^e programmer has little control over the behaviour of complex predicates using the *propia* library.

For example in the *fdTaskResource* query 2, illustrated above, the constraint detects "holes" inside the domains of the variables *S1* and *S2*. However experience in solving scheduling problems suggests that the computational effort expended in detecting such holes is rarely compensated by any reduction the amount of search necessary to find solutions. Whilst this propagation is too powerful, the other alternatives available in the *propia* library are too weak.

The most useful behaviour for the constraint is to do nothing until one of the following conditions hold:

- If the tasks are guaranteed to overlap, constrain them to use distinct resources
- If the tasks must use the same resource, and one of the tasks cannot precede the other, constrain that task not to start until the other task has ended.

Notice that this is, unfortunately, not the behaviour achieved by the MIP encoding, either.

This behaviour can be expressed in ECL^{i}PS^e using the Constraint Handling Rules *chr* library. The required ECL^{i}PS^e encoding remains quite logical, but it needs a new concept, that of a *guard*. A rule with a guard is not executed until its guard is entailed, until then it does nothing. The data-driven implementation of guarded rules uses the same mechanisms as the data-driven implementation of constraints discussed in the following section.

The syntax for guarded rules is rather different from the syntax for ECL^{i}PS^e clauses encountered so far. This syntax is illustrated by the accompanying encoding of the *chrtaskResource* constraint. In this example the constraint handling rules use finite domain constraints in their definitions.

```
chrTR(S1,R1,S2,R2) :-
```

```
    [S1,S2]::0..100, [R1,R2]::[r1,r2,r3],
    E1 = S1+50,  E2 = S2+70,
    chrTaskResource(S1,E1,R1,S2,E2,R2).

constraints chrTaskResource/6.

chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
    R1 #= R2, E1 #> S2   |   E2 #<= S1.
chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
    R1 #= R2, E2 #> S1   |   E1 #<= S2.
chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
    E1 #> S2, E2 #> S1   |   R1 ## R2.
```

Logically each of these three rules states the same constraint: either *R1* ≠ *R2* or *S2* ≥ *E1* or *S1* ≥ *E2*. However each rule uses a different "if...then" statement. For example the first rule says that if *R1=R2* and *E1>S2* then S1 ≥ E2.

In order to use constraint handling rules, it is necessary to translate them into the underlying ECLⁱPS^e language using an automatic translator. The constraints must be written to a file called *file.chr* - in our example we shall use *chrTaskResource.chr*. To illustrate the loading and use of constraint handling rules, some examples are given:

```
[eclipse 1]: lib(chr), lib(fd).
*   chr loaded
*   fd loaded

[eclipse 2]: chr(chrTaskResource).
*   chrTaskResource.chr compiled.
*   yes.

[eclipse 3]: chrTR(S1, R1, S2, R2), R1#=r1, R2#=r1.
*   S1 = S1{[0..100]}
*   S2 = S2{[0..100]}
*   R1 = r1
*   R2 = r1
*   yes.

[eclipse 4]: chrTR(S1, R1, S2, R2), R1=r1, R2=r1, S1#<=65.
*   S2 = S2{[50..100]}
*   R1 = r1
*   R2 = r1
*   S1 = S1{[0..50]}
*   yes.

[eclipse 5]: chrTR(S1,R1,S2,R2), R1=r1, S2#>=35, S2#<=45.
*   S1 = S1{[0..100]}
*   R2 = R2{[r2, r3]}
*   R1 = r1
*   S2 = S2{[35..45]}
*   yes.
```

Query 3 yields less propagation than *propiaTR* because this implementation does not punch holes in the variables' domains.

Query 4 does, however, produce new information, because not only do both tasks use the same resource, but also the constraint $S1 \leq 65$ means that $task_1$ must precede $task_2$. The constraint deduces that the latest start time for *S1* is actually 50, and the earliest start time for *S2* is also (by coincidence) 50.

Query 5 uses the fact that the tasks must overlap to remove r_1 from the domain of *R2*.

The *chr* library offers many more facilities, including multi-headed rules, and augmentation rules. These facilities can be explored in detail by studying the relevant chapter in [Brisset et al., 1997], and trying out the example constraint handling rule programs which are distributed with ECL^{i}PS^e.

4.3 Explicit Data Driven Control

The *propia* and *chr* libraries are implemented using a set of underlying facilities in ECL^{i}PS^e which support data-driven computation. The main feature supporting data-driven computation is the *suspension*. This is illustrated below:

```
[eclipse 1]:    lib(suspend), lib(fd).
*  suspend loaded
*  fd loaded

[eclipse 2]: suspend(writeln("Wake up!"),1,X->inst),
             writeln("Do this first"),
             X=1.
*  Do this first
*  Wake up!
*  X = 1
*  yes.

[eclipse 3]: suspend(writeln("Wake up!"),1,X->inst),
             current_suspension(S),
             suspension_to_goal(S,Goal,M),
             call(Goal,M).
*  Wake up!
*  *      ...
*  yes.

[eclipse 4]: suspend(writeln("Wake up!"),1,X->inst),
             current_suspension(S),
             kill_suspension(S),
             X=1.
*  S = 'WOKEN GOAL'
*  X = 1
*  yes.
```

```
[eclipse 5]:X::1..10,
            suspend(writeln("Wake up!"),1,X->min),
            X#>3,
*   Wake up!
*   X = X{[4..10]}
*   yes.
```

A suspension is a goal that waits to be executed until a certain event occurs. Each suspension is associated with a set of variables, and as soon as a relevant event occurs to any one of the variables in the set, the suspension "wakes up" and the goal is activated. One such event is instantiation: all the suspensions on a variable wake up when the variable is instantiated.

Query 1 loads the *suspend* library, and it also loads the *fd* library, which will be used in the last example. It is preferable to load all the libraries that may be needed at the start of the session.

Query 2 suspends a goal *writeln("Wake up!")* on one variable X. The goal will be executed as soon as X becomes instantiated *(X → inst)*. When woken the goal will be scheduled with a certain priority. The priority is given as the second argument of *suspend*. In this case the priority is *1*, which is the highest priority. The remainder of query 2 performs another write statement and then instantiates X. The output from ECL^{i}PS^e shows that the suspended goal is not executed, until X is instantiated, after the system has already output *Do this first*.

Query 3 shows various facilities for explicitly handling a suspension. The current suspensions can be accessed. (It is also possible to access just the suspensions on a particular variable.) A suspension can be converted to a goal.[6] A suspension can be "killed", so that it is no longer accessible or wakeable. The suspension has no connection to the goal, however, which can still be executed.

Query 5 illustrates another kind of event that can wake up a suspended goal. In this case the goal is suspended until the lower bound of the finite domain associated with X is tightened (*X →min*).

There are other events which can wake suspended goals associated with other constraint handlers, but the most general event is that the variable becomes more constrained in *any* way (expressed as *X → constrained*). Goals suspended in this way will wake when any new constraint on X is added (an *fd* constraint, a *ria* constraint, or an *eplex* constraint).

Finally it is also possible to retrieve goals suspended on a given variable, or those associated with a given event on a given variable.

Based on this simple idea it is possible to define a constraint behaviour explicitly. As a simple example let us make a constraint that two variable differ by more than some input number *N*. We will call the constraint *ndiff(N,X,Y)*, where *N* is the difference, and *X* and *Y* the two variables. Its

[6] The variable *M* denotes the module in which *writeln* is defined.

behaviour will be to tighten the finite domains of the variables. In the following code, a behaviour for an *ndiff* constraint is implemented and, since underlying *fd* constraints are used, the *fd* library is loaded:[7]

```
:- lib(fd).
:- suspend.

ndiff(N,X,Y) :-
   mindomain(X,XMin),
   maxdomain(Y,YMax),
   YMax<XMin+N, !,
   X#>=Y+N.

ndiff(N,X,Y) :-
   mindomain(Y,YMin),
   maxdomain(X,XMax),
   XMax<YMin+N, !,
   Y#>=X+N.

ndiff(N,X,Y) :-
   suspend(ndiff(N,X,Y),3,[X,Y] -> any).
```

The first clause for *ndiff* checks if the lower bound for X is so close to the upper bound for Y that X cannot be less than Y (if it was, then to satisfy the *ndiff* constraint we would need to have $Y >= X + N$). In this case impose the constraint that $X \# >=0\ Y + N$.

The second clause does the symmetrical test on the lower bound of Y and the upper bound of X.

If neither of these conditions is satisfied, then *ndiff* doesn't do anything. It just suspends itself until the finite domains of X or Y are tightened (*[X,Y] -> any*).

This very same mechanism of suspended goals is used to implement all the built-in constraints of ECL^{i}PS^e. For example the constraint $X \# > Y$ is implemented using a goal which is suspended on two events: a change in the maximum of the domain of X, and a change in the minimum of the domain of Y. Typically all the finite domain built-in constraints are suspended on events which occur to the finite domains of their variables.

Before concluding this subsection, we should observe that the different constraint libraries of ECL^{i}PS^e are supported by a very flexible facility. The information about each kind of constraint on a variable is held in a data structure which is attached to the variable called an *attribute*. When the *fd* library is loaded, each variable in ECL^{i}PS^e has a finite domain attribute. If the variable has no finite domain, this attribute contains nothing, and the behaviour of the variable is just as if it had no attribute. On the other hand if the variable does have a finite domain, then the attribute stores the finite

[7] The *fd* library automatically loads the *suspend* library, so it is not actually necessary to load *suspend* explicitly.

domain, as well as pointers to all the suspended goals which are waiting for an event to occur to the finite domain.

Naturally *ria* constraints and *eplex* constraints are stored in other attributes, and they have their own suspended goals attached to them.

Any ECL^{i}PS^e user can define and implement a completely new constraint handling library in three steps:

1. A new attribute, storing information about the new class of constraints, must be defined
2. Events specific to this class of constraints must be specified
3. New constraint behaviours must be implemented in terms of goals which suspend themselves on these events.

The ECL^{i}PS^e extensions manual [Brisset et al., 1997] gives an example of defining such a new constraint library.

5. Search

5.1 Constructive Search

5.1.1 Branch and Bound

In the preceding sections we have encountered two optimization procedures, the finite domain procedure *minimize* and the MIP procedure *optimize*. Both optimization procedures implement an algorithm called *branch and bound*, which posts a new constraint, each time it finds a solution, that the cost of future solutions must be better than the cost of the current best solution. Eventually the new constraint will be unsatisfiable, and the algorithm will have proved that it has found the optimum.

5.1.2 Depth-First Search and Backtracking

We have also encountered the finite domain search procedure *labeling*, which successively instantiates a list of finite domain variables to values in their domains. In ECL^{i}PS^e the default search method is depth-first search and backtracking on failure. Of the complete search methods available, this is in practice the best because search algorithms with a breadth-first component quickly grow to occupy too much memory. We will discuss some incomplete search methods below.

5.1.3 *Guesses* - Constraints Imposed During Search

Search is, of course, much more general than just labelling. Certainly, for combinatorial problems, it involves making guesses that may later turn out to have been bad. However a guess need not involve guessing a value for a variable, as is done in labelling. For example if a variable X has range *[0..100]*, instead of guessing a precise value for X, it may be useful to perform a binary chop, first guessing that $X \geq 50$, and then, if the guess turns out to be bad, guessing that $X < 50$. A guess in the most general sense is the posting of a new (non-redundant) constraint which narrows the search space. However there is no guarantee that such a guess does not rule out solutions to the problem, therefore the system must also explore the remainder of the search space on backtracking. Typically this is done by imposing the negation of the constraint. However the negation of an inequality $\geq$ is a strict inequality $<$, which can't be handled by linear programming. However in case X is an integer variable, and N an integer, the negation of $X \geq N$ is $X \leq N-1$ which can be handled.

5.1.4 MIP Search

Finite domain propagation only narrows domains, and does not guarantee to detect all inconsistencies. Thus there is no guarantee that a partial labelling (which assigns consistent values to some of the variables) can always be extended to a complete consistent labelling. However the linear constraint solver available through *eplex* does indeed guarantee to detect all inconsistencies between the linear constraints. On the other hand a linear solver does not take into account the constraint that certain variables can only take integer values, thus it can return proposed solutions in which non-integer values are proposed for integer variables. The linear solver can efficiently find an optimal solution to the problem in which integrality constraints on the variables are ignored. Such an optimum is termed an optimum of the "continuous relaxation" of the problem, or just the "relaxed optimum" for short.

This suggests a different search mechanism, in which a new constraint is added to exclude the non-integer value in the relaxed optimum returned by the linear constraint solver. If the value for integer variable X was 0.5 in the relaxed optimum, for example, a new constraint $X \geq 1$ might be added. Since this excludes other feasible solutions such as $X=0$, this new constraint is only a guess, and if it turns out to be a bad guess then the alternative constraint $X \leq 0$ is posted instead.

This is the search method used in MIP when *optimize* is called in the *eplex* library.

5.1.5 Search Heuristics based on Hybrid Solvers

MIP search can be duplicated in ECL^{i}PS^e by passing the linear constraints to CPLEX and using the proposed solutions to decide which new constraint to impose (i.e. guess) next. Whilst there is little point in precisely duplicating the MIP search control with ECL^{i}PS^e, it allows the ECL^{i}PS^e programmer to define new search techniques using information from both the *fd* library and from *eplex*. For example the choice of which variable to guess a constraint on next can be guided by the size of the finite domain, as recorded in the finite domain library, and then the choice of what value to label it to can be guided by its value in the relaxed optimum returned from *eplex*.

This search technique is supported by the ECL^{i}PS^e library *fdplex*, and is illustrated below. The *fdplex* version of *indomain* selects the value closest to the value at the relaxed optimum returned by *eplex*.

```
:- lib(fdplex).

mylabeling([]).
mylabeling(Vars) :-
   deleteff(Var,Vars,Rest),
   indomain(Var),
   mylabeling(Rest).

solve(X,Y,Z,W) :-
  [X,Y]::1..5,
   [Z,W]::1..100,
   10*Z+7*W+4*X+Y #= 49,
   Cost #= Z-2*W+X-2*Y,
   minimize(mylabeling([X,Y,Z,W]),Cost).
```

Indeed it is instructive to watch the search taking place using the ECL^{i}PS^e tracing facilities, so we shall load the above program into a file called *fdplexsearch.pl*. Now we shall run it as shown as follows:

```
[eclipse 1]: [fdplexsearch].
*  fd loaded
*  range loaded
*  eplex loaded
*  fdplex loaded
*  yes.

[eclipse 2]: spy(mylabeling), spy(indomain).
*  spypoint added to mylabeling / 1.
*  spypoint added to indomain / 1.
*  yes.

[eclipse 3]: solve(X, Y, Z, W).
*  CALL mylabeling([X{eplex:1.0, range : 1..5, fd:[1..5]},
*                   Y{eplex:5.0, range : 1..5, fd:[1..5]},
*                   Z{eplex:1.2, range : 1..3, fd:[1..3]},
*                   W{eplex:4.0, range : 1..4, fd:[1..4]}])
                            (dbg)?- leap
*  CALL   indomain(Z{eplex:1.2, range : 1..3, fd:[1..3]})
                            (dbg)?- leap
*  EXIT   indomain(1)   (dbg)?- leap
*  CALL  mylabeling([Y{eplex:5.0, range : 1..5, fd:[1..5]},
*                   X{eplex:2.0, range : 1..5, fd:[2..5]},
*                   W{eplex:3.7, range : 1..4, fd:[2..4]}])
                            (dbg)?- leap
*  CALL   indomain(W{eplex:3.7, range : 1..4, fd:[2..4]})
                            (dbg)?- leap
```

```
*  EXIT   indomain(4)  (dbg)?- leap
*  CALL   mylabeling([3, 2]) (dbg)?- no debug
*  Found a solution with cost -11
*  X = 2
*  Y = 3
*  Z = 1
*  W = 4
*  yes.
```

In query 1 the *fdplex* is loaded, and it automatically loads the other libraries which are needed. Query 2 sets spypoints on two predicates. Now each time either of these predicates are called, and when they exit, the debugger stops and allows the programmer to study the state of the program execution. Query 3 calls the program defined in the search code above. Before labelling starts the domains of the variables have already been reduced by finite domain propagation. The reduced domains are automatically communicated to the *range* library, and passed into the linear solver. The linear solver (CPLEX) has already been invoked by *eplex* and has returned the values of the variables *X,Y,Z,W* at the relaxed optimum.

Now *deleteff* selects the variable with the smallest domain, which is Z. The *fdplex indomain* predicate labels Z to the integer value nearest to its value at the relaxed optimum. This wakes the *fd* constraint handler which tightens the domain of X, and it wakes the linear solver which returns a new relaxed optimum with new suggested values for the other variables.

This time the variable with the smallest domain is W, and this is the one selected for instantiation. Once this has been instantiated to the integer value closest to its suggested value, *fd* propagation immediately instantiates the remaining values.

At the next spy point the user enters *n* (for *no debug*) and tracing is switched off. The optimal solution is indeed the one found first, which testifies to the usefulness of the combined heuristic used in the search.

5.1.6 Incomplete Constructive Search

For real industrial applications, the search space is usually too large for complete search to be possible. The branch and bound search yields better solutions with longer and longer delays until, in many cases, it fails to yield any new solutions but continues searching fruitlessly.

In cases where complete search is impractical, the heuristics guiding the search become very important. If bad heuristics are chosen the search may methodically explore some unpromising corner of the search space yielding very poor solutions which fail to drive the branch and bound search into more fruitful areas. Good heuristics depend on good constraint handling: the information returned from the constraint handlers is crucial in enabling the heuristics to focus search on promising regions. Moreover once some good choices have been made, propagation can achieve even better results supporting even better heuristics for future choices. This positive feedback produces a virtuous spiral.

Received wisdom suggests that local search techniques, based on solution repair, achieve faster convergence on good solutions than constructive search. However on several industrial applications our experience has shown the contrary. Good heuristics, tailored to the application at hand, have proved more effective in yielding high quality solutions than techniques based on solution repair.

5.1.7 Intelligent Backtracking and *nogood* Learning

ECL^{i}PS^e offers facilities for programmers to define specific constructive search algorithms. Intelligent backtracking has been implemented in ECL^{i}PS^e. It is not offered as a library, however, because in practice any reduction in the amount of search due to intelligent backtracking is vitiated by the cost of accessing and updating the necessary data structures.

The information about which constraints are involved, when a failure occurs during search, is useful for recording combinations of variable values which are mutually inconsistent. Such conflict sets can be used to impose extra constraints called *nogoods* which are learned during search.

nogood learning has also been implemented in ECL^{i}PS^e and is proving useful on some benchmark examples, but as yet no library supporting *nogoods* is available. A paper describing this work [Richards & Richards, 1996] is available from the IC-Parc home page (whose URL is given in Section 6 below).

5.2 Solution Repair

At the end of the previous section we suggested that even for incomplete search, constructive search with good heuristics can outperform solution repair. However there are many important examples, such as job-shop scheduling and travelling salesman problems, where repair performs better than constructive search. Moreover repair is very important in handling *dynamic* problems, which change after an initial solution has been found. The problem may be changed because the user is unsatisfied with the solution for reasons which are not captured in the implementation, and adds new constraints to exclude this solution. Otherwise the change may be due to external circumstances such as unplanned delays, rush orders, cancellations, and so on.

ECL^{i}PS^e uses the concept of the *tentative* value to support solution repair. This is the same concept that is used to return proposed values for variables from the linear solver, as discussed in the preceding section. In the case of repair, however, the tentative value comes not from a constraint handler, but from the original solution to the original problem.

When the problem changes, some of the tentative failures may no longer satisfy some of the new constraints. Indeed the simplest change is to change the value of a variable, which is to impose a new constraint constraining that variable to take a new value different from its tentative value. In this case the tentative value violates the new constraint. In case there are no violations, of course, the tentative values comprise a feasible solution to the new problem and there is no need to repair the solution at all.

The purpose of the ECL^{i}PS^e repair library is to support the process of detecting a variable whose tentative value is in conflict with a constraint, and in detecting further violations that result from choosing a value for a variable that differs from its tentative value.

5.2.1 "Constructive" Repair

There are several very different repair algorithms that arise from different choices of how to change the value of a variable from its tentative value. The algorithm most similar to constructive search simply instantiates the variable to the chosen new value. In this case the tentative values do no more than support a specific heuristic within a constructive search algorithm. Notice that the heuristic can do more than simply choose the tentative value as the first guess for each variable during labelling. It can also take into account for each value for a variable the number of other tentative values with which it conflicts according to the constraints. Thus when a variable is labelled to a new value, the value is chosen so as to minimise disruption to the original solution.

The ECL^{i}PS^e *repair* library defines primitives for setting a tentative value for a variable (*tent_set*) and for looking it up (*tent_get*). It also supports a special annotation which changes the behaviour of a constraint from propagation to simply checking against the tentative values of their uninstantiated variables. The annotation is written *Constraint r*, where *Constraint* can be any built-in or user-defined constraint. Whenever the check fails, the constraint is recorded as a *conflict constraint*, and full propagation on the constraint is switched on. The set of conflict constraints can be accessed via the predicate *conflict_constraints*. This can be used in the search procedure to decide which variable to label next.

A built-in search predicate called *repair* is provided which selects a variable whose tentative value violates a repair constraint, labels it and succeeds when all the remaining variables have consistent tentative values.

We illustrate this repair algorithm (with an example from the IC-Parc ECL^{i}PS^e library manual [Schimpf et al., 1997]) below:

```
solve(X,Y,Z) :-
    [X,Y,Z]::1..3,            % the problem variables
    Y ## X r,                 % state the constraints
    Y ## Z r,
    Y #= 3 r,
[X,Y,Z] tent_set [1,2,3],  % set existing solution
repair,                    % invoke repair labeling
[X,Y,Z] tent_get [NewX,NewY,NewZ]. % get repaired solu-
tion
```

The solutions found are *[1,3,1]* and *[1,3,2]*, which means that only *Z* has been repaired. Initially only the constraint *Y* #= *3* is inconsistent with the solution so variable *Y* is repaired to take the value *3*. This now affects the constraint *Y* ## *Z* , and *Z* must be repaired to either *1* or *2*.

The constraint *Y* ## *X* is not affected by the update. In particular, *X* keeps the value of the existing solution, and is not even being labelled by *repair/0*.

Constructive repair is also known as *informed backtracking* and has been used successfully on a variety of benchmarks [Minton et al., 1992].

5.2.2 Weak Commitment

Instead of instantiating a variable in order to repair it, an alternative method is simply to change its tentative value. This approach requires no backtracking, since every conflict can be fixed by just changing tentative values. The disadvantage is that cycles can easily occur in which two variables repeatedly switch their tentative values.

A very successful algorithm based on repairing tentative values is called *Weak Commitment* [Yokoo, 1994]. On starting all the variables have tentative values. Variables in conflict are repaired - by instantiating them - until either there are no more conflicts and the algorithm terminates, or the remaining conflicts cannot be repaired. The latter situation occurs when the next variable in conflict cannot be instantiated to any value that is consistent with the variables instantiated so far.

When such a dead-end is encountered, the weak commitment algorithm simply uninstantiates all the variables, setting their tentative values to the values they had when they were instantiated. Then the algorithm restarts, fixing conflicts as before.

5.2.3 Local Improvement

Constructive repair and weak commitment are two algorithms designed to find feasible solutions to a problem. In case the problem additionally requires some cost to be minimised, the repair must be adapted to return better and better solutions.

For unconstrained problems, local improvement can be achieved by just changing the value of some variable, having chosen the variable and value such that the cost of the new solution is better than the cost of the previous solution. This idea underlies the various hill-climbing algorithms as well as stochastic techniques such as Simulated Annealing and Tabu search.

For problems with constraints, changing the value of a variable will not necessarily yield a feasible solution. The ECL^{i}PS^e repair library can be used, however, to find a feasible solution which incorporates the change.

A simulated annealing program has been written in ECL^{i}PS^e which ensures that moves respect the problem constraints. The program has been compared with a pure simulated annealing approach which simply associates a cost with violated constraints and otherwise treats the problem as unconstrained. Experiments showed that the "constrained simulated annealing" program outperformed the pure one.

For an industrial application the repair library has been used together with the *eplex* linear constraint library. In the algorithm used for this application, the relaxed optimum is checked against the repair constraints, and at each step a violated constraint is strengthened in such a way that the next solution returned from *eplex* must satisfy it. The algorithm outperforms standard MIP search because the problem is a dynamic constraint problem: there is an original solution and the requirement is to modify that solution to satisfy some new constraints.

Details of these algorithms are beyond the scope of this article, but hopefully this brief survey has offered a glimpse of the power of repair-based search in combination with the different solvers of ECL^{i}PS^e.

6. The ECL^{i}PS^e System

ECL^{i}PS^e runs under the UNIX operating system (specifically SunOS 4 on Sun-4 hardware, Solaris on Sparc machines and Linux on PC's), and will be available under Windows-NT (version 4.0) by the end of 1997.

ECL^{i}PS^e is can be embedded in **c** and **c**$^{++}$ programs. It is available in the form of a linkable library, and a number of facilities are available to pass data between the different environments, to make the integration as close as possible. Naturally facilities are also provided to allow ECL^{i}PS^e to invoke **c** and **c**$^{++}$.

A tightly integrated graphical system is very useful for program development, and ECL^{i}PS^e offers such an integration to the Tcl/Tk toolkit, which is public domain software available under Unix and Windows. Typically ECL^{i}PS^e is invoked from Tcl which is driven directly by user interactions. An example graphical environment for ECL^{i}PS^e developers is the graphical constraint environment *Grace*, available as an ECL^{i}PS^e library. Grace is implemented using ECL^{i}PS^e and Tcl.

The ECL^{i}PS^e system includes a great deal of documentation which can be printed, but is best to use on-line as a set of web pages. The home page for ECL^{i}PS^e at IC-PARC is

```
http://www-icparc.doc.ic.ac.uk/eclipse
```

This page offers a dozen links to other pages giving the following information:

- If you are interested in ECL^{i}PS^e:
  - Overview of ECL^{i}PS^e features
  - How to obtain the system
  - What is ECL^{i}PS^e's current status?
  - Application descriptions
  - Related papers and technical reports
  - Other CLP-related sites
- If you are already using ECL^{i}PS^e:
  - Latest release notes
  - Manuals and other documentation
  - Programs, libraries and tools
  - The User Group Mailing List
  - How to report a bug
  - Download the latest version

The manuals cover the non-constraint facilities of ECL^{i}PS^e [AGGOUN et al., 1997] as well as the facilities supporting constraints [Brisset et al., 1997], [Schimpf et al., 1997]. There is additional information covering the graphical user interface library, and how to embed code in **C** and **C^{++}**.

Background references can be found in the list of publications reachable from the IC-Parc home page at:

`http://www-icparc.doc.ic.ac.uk/`

7. Conclusions

The ECL^{i}PS^e platform has been under development for over ten years. During that time constraint programming has established itself not only as an important research area, but also in live industrial applications. The market for constraint technology is growing dramatically, to the point that the major vendor of MIP technology (CPLEX) has been recently taken over by a constraint technology vendor (ILOG).

Over the last five years ECL^{i}PS^e has moved on from its early roots in logic programming and constraint propagation, to a focus on hybrid algorithms. A tight integration between MIP and CLP has been developed and hybrid algorithms based on this combination have proved their efficiency in industrial applications. However hybrid search algorithms, in particular utilising solution repair, have also been a focus of research and development.

Based on growing experience with hybrid algorithms, we have been able to separate the features of the different algorithms both from each other, and from the underlying problem model. Consequently we have reached the point where ECL^{i}PS^e can be used to express a clear, precise and neutral conceptual model of an application, and this model can then be extended and annotated at the implementation stage. The result of implementation is a design model which implements fine-grained hybrid algorithms tailored to the application at hand.

This work has been based on experience gained from a variety of industrial applications. IC-Parc has developed applications for several of its industrial partners, and each application has contributed to the final architecture of the ECL^{i}PS^e platform. Ongoing applications, with partners such as British Airways, Wincanton Transport and Bouygues, continually give rise to new hybrid techniques, and these results will feed back into ECL^{i}PS^e, as the algorithms are encapsulated and added as new libraries.

Nevertheless the real benefit of ECL^{i}PS^e comes not from the algorithms that are already encapsulated as libraries, but from the ease with which new hybrid algorithms can be developed and validated, and delivered into the industrial computing environment.

Bibliography

AGGOUN, A. et.al., "ECL^{i}PS^e user manual," IC-Parc, 1997.

BRISSET, P. et.al., "ECL^{i}PS^e Extensions manual," IC-Parc, 1997.

CPLEX. Using the cplex callable library and cplex mixed integer library. Technical Report Version 2.1, CPLEX Optimisation Inc., 1993.

MINTON, S., JOHNSTON, M. D., PHILIPS, A. B. and LAIRD, P., "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems," Artificial Intelligence, 58, 1992.

RICHARDS, T. and RICHARDS, B., "Nogood learning for constraint satisfaction," (Technical Report, IC-Parc, 1996), Proceedings of CP 96 Workshop on Constraint Programming Applications, 1996

RODOSEK, R., WALLACE, M. and HAJIAN, M., "A new approach to integrating mixed integer programming with constraint logic programming," (Technical Report, IC-Parc, 1997), Annals of Operations Research (to be published).

SCHIMPF, J., NOVELLO, S. and EL SAKKOUT, H., "IC-Parc ECL^{i}PS^e Library Manual," IC-Parc, 1997.

WALLACE, M.. "Constraint programming," ICL Systems Journal, Vol 12, 1, 1997.

YOKOO, M., "Weak-commitment search for solving constraint satisfaction problems," Proceedings of 12th National Conference on Artificial Intelligence, pp 313-318, 1994.

ECL^{i}PS^e is jointly owned by ICL and IC-Parc and can be obtained by ftp from IC-Parc by emailing **`eclipse-request@doc.ic.ac.uk.`** Full documentation is also obtainable from the web site at: **`http://www-icparc.doc.ic.ac.uk/eclipse`**

Biographies

Mark Wallace

Mark Wallace is currently seconded from ICL to IC-Parc, Imperial College, where he is Deputy Director. Dr. Wallace has been with ICL for some 15 years, during which he completed a PhD in natural language processing at Southampton University, and then spent a decade at the European Computer Industry Research Centre (ECRC), first working on knowledge bases and, for the last three years, leading ECRC's constraint reasoning project. At IC-Parc he manages several research and application development projects, as well as participating in the development of ECL^{i}PS^e II.

Stefano Novello

Stefano Novello has been a Research Fellow at IC-Parc, working on ECL^{i}PS^e II, since 1996. He completed an MSc at Imperial college in 1988 and then went to ECRC, eventually taking on the management of the KCM project with responsibility for developing and supporting Europe's first and fastest Prolog machine. From 1992 to 1996 he worked at Encore computers, before joining IC-Parc. At IC-Parc, Stefano has made ECL^{i}PS^e II into a 'C' library and is developing a Windows version of the system.

Joachim Schimpf

Joachim Schimpf is one of the original designers of ECL^{i}PS^e. He obtained his Dipl. Inform. from the Technical University of Munich in 1986 and then joined Siemens. He moved to ECRC in 1988, where he became co-designer of Sepia and ECL^{i}PS^e, and developed the parallel implementation of ECL^{i}PS^e. He moved to IC-Parc in 1995, as a Principal Research Fellow. Since then Joachim has designed ECL^{i}PS^e II and developed the mathematical programming functionality of the system.

Previous Issues

Vol.8 Iss.2 - November 1992

Open Networks - The Key to Global Success
Infrastructure of Corporate Networks in the Nineties
Broadband Networking
FDDI - The High Speed Network of the Nineties
The Evolution of Wireless Networks
Communications Technology for the Retail Environment
RIBA - A Support Environment for Distributed Processing
Information Technology: Support for Law Enforcement Investigations and Intelligence
Standard for Keyboard Layouts - The Origins and Scope of ISO/TEC 9995
ESS - A Solid State Disc System for ICL System for ICL Series 39 Mainframes

Vol.8 Iss.1 - May 1992

Defining CASE Requirements
ICL's ICASE Products
The Engineering Database
CASE Data Integration: The Emerging International Standards
Building Maintainable Knowledge Based Systems
The Architecture of an Open Dictionary
The Use of a Persistent Language in the Implementation of a Process Support System
ALF: A Third Generation Environment for Systems Engineering
MASP/DL: The ALF Language for Process Modelling
The ALF User Interface Management System
A New Notation for Dataflow Specifications

Vol.7 Iss.4 - November 1991

Systems Management: A Challenge for the Nineties - Why now?
The Evolution within ICL of an Architecture for Systems Management
Manageability of a Distributed System
Distribution Management - ICL's Open Approach
Experience of Managing Data Flows in Distributed Computing in Retail Businesses
Generation of Configurations - a Collaborative Venture
Operations Management
OSMC: The Operations Control Manager
The Network Management Domain
An Overview of the Raleigh Object-Oriented Database System
Making a Secure Office System
Architectures of Knowledge Base Machines
The Origins of PERICLES - A common on-line Interface

Vol.7 Iss.3 - May 1991

Introduction to the technical characteristics of ISDN
ISDN in France: Numéris and its market
The Telecoms Scene in Spain
Future Applications of ISDN to Information Technology
A Geographical Information System for Managing the Assets of a Water Company
Using Constraint Logic Programming Techniques in Container Port Planning
Locator - An Application of Knowledge Engineering to ICL's Customer Service

Development philosophy and fundamental processing concepts of the ICL Rapid Application Development System RADS
A moving-mesh plasma equilibrium problem on the ICL Distributed Array Processor

Vol.2 Iss.3 - May 1981

A dynamic database for econometric modelling
Personnel on CAFS: a case study
Giving the computer a voice
Data integrity and the implications for back-up
Applications of the ICL Distributed Array Processor to econometric computations
A high-level logic design system
Measures of programming complexity

Vol.2 Iss.2 - November 1980

The ICL Information Processing Architecture, IPA
VME/B: a model for the realisation of a total system concept
Birds, Bs and CRTs
Solution of elliptic partial differential equations on the ICL Distributed Array Processor
Data routing and transpositions in processor arrays
A Bayesian approach to test modelling

Vol.2 Iss.1 - May 1980

Security and privacy of data held in computers
CADES - software engineering in practice
ME29 Initial Program Load: an exercise in defensive programming
Project Little - an experimental ultra-reliable system
Flow of instructions through a pipelined processor
Towards an 'expert' diagnostic system
Using Open System Interconnection standards

Vol.1 Iss.3 - November 1979

Meteosat 1: Europe's first meteorological satellite
An analysis of checkpointing
Statistical and related systems
Structured programming techniques in interrupt-driven routines
The content addressable file store - CAFS
Computing in the humanities
The data dictionary system in analysis and design

Vol.1 Iss.2 - May 1979

Computers in support of agriculture in developing countries
Software and algorithms for the Distributed Array Processor
Hardware monitoring on the 2900 range
Network models of system performance
Advanced technology in printing: the laser printer
The new frontier: three essays on job control

To order back issues

Contact

Sheila Cox
Research and Advanced Technology,
ICL, Lovelace Road, Bracknell, Berks., RG12 8SN
Telephone +44 (0)1344 472900
Fax +44 (0)1344 472700
Email: S.D.Cox@bra0102.wins.icl.co.uk

or

The Editor, V.A.J. Maller
Telephone +44 (0)1438 833514
Email: V.A.J.Maller@ste0418.wins.icl.co.uk

ICL Systems Journal

Guidance for Authors

1. Content

The ICL Systems Journal has an international circulation. It publishes papers of a high standard that are related to ICL's business and is aimed at the general technical community and in particular at ICL's users, customers and staff. The Journal is intended for readers who have an interest in computing and its applications in general but who may not be informed on the topic covered by a particular paper. To be acceptable, papers on more specialised aspects of design or application must include some suitable introductory material or reference.

The Journal will usually not reprint papers already published but this does not necessarily exclude papers presented at conferences. It is not necessary for the material to be entirely new or original. Papers will not reveal matter relating to unannounced products of any of the ICL Group companies.

Letters to the Editor and book reviews may also be published.

2. Authors

Within the framework defined in paragraph 1, the Editor will be happy to consider a paper by any author or group of authors, whether or not an employee of a company in the ICL Group. All papers are judged on their merit, irrespective of origin.

3. Length

There is no fixed upper or lower limit, but a useful working range is 4,000-6,000 words; it may be difficult to accommodate a long paper in a particular issue. Authors should always keep brevity in mind but should not sacrifice necessary fullness of explanation.

4. Abstract

All papers should have an Abstract of approximately 200 words, suitable for the various abstracting journals to use without alteration.

5. Presentation

5.1 Printed (typed) copy

A typed copy of the manuscript, single sided on A4 paper with the pages numbered in sequence, should be sent to the Editor. Particular care should be taken to ensure that mathematical symbols and expressions, and any special characters such as Greek letters, are clear. Any detailed mathematical treatment should be put in an Appendix so that only essential results need be referred to in the text.

5.2 Electronic version

Authors are requested to submit either a magnetic disk version of their copy in addition to the manuscript or an e-mail attached file or both. The format of the file should conform to the standards of any of the widely used word processing packages or be a simple text file.

5.3 Diagrams

Line diagrams will, if necessary, be redrawn and professionally lettered for publication, so it is essential that they are clear. Axes of graphs should be labelled with the relevant variables and, where this is desirable, marked off with their values. All diagrams should be numbered for reference in the text and the text marked with the reference and an appropriate caption to show where each should be placed. Authors should check that all diagrams are actually referred to in the text and that copies of all diagrams referred to are supplied. If authors wish to submit drawings in an electronic form, then they should be separated from the main text and be in the form of EPS files. If an author wishes to use colour, then it is very helpful that a professional drawing package be used, such as Adobe Illustrator.

5.4 Tables

As with diagrams, these should all have captions and reference numbers. If they are to be provided in electronic form, then either a standard spreadsheet (Excel) should be used or the data supplied as a file of comma/tab separatedvariables. A printed version should also be supplied, showing all row and column headings, as well as the relevant units for all the quantities tabulated.

5.5 References

Authors are asked to use the Author/Date system, in which the author(s) and the date of the publication are given in the text, and all the references are listed in alphabetical order of author at the end. e.g. in the text: "...further details are given in [Henderson, 1986]" with the corresponding entry in the reference list:

HENDERSON, P. Functional Programming Formal Specification and Rapid Prototyping. IEEE Trans. on Software Engineering SE*12, 2, 241-250, 1986.

Where there are more than two authors it is usual to give the text reference as "[X et al ...]". Authors should check that all text references are listed; references to works not quoted in the text should be listed under a heading such as Bibliography or Further reading.

5.6 Style

A note is available from the Editor summarising the main points of style - punctuation, spelling, use of initials and acronyms etc. preferred for Journal papers.

6. Referees

The Editor may refer papers to independent referees for comment. If the referee recommends revisions to the draft, the author will be asked to make those revisions. Referees are anonymous. Minor editorial corrections, as for example to conform to the Journal's general style for spelling or notation, will be made by the Editor.

7. Proofs, Offprints

Printed proofs are sent to authors for correction before publication. The Editor will, however, always be prepared to send electronic versions to authors who have access to software compatible with the production system used for the Journal—Microsoft Word, Microsoft Excel, Adobe PageMaker, Adobe Illustrator and Adobe Photoshop.

8. Copyright

Copyright of papers published in the ICL Systems Journal rests with ICL unless specifically agreed otherwise before publication. Publications may be reproduced with the Editor's permission, which will normally be granted, and with due acknowledgement.

ICL Systems Journal May 1997

ICL Research & Advanced Technology
Lovelace Road
Bracknell
Berkshire RG12 8SN