A View Mechanism For An Integrated Project Support Environment

by

Alan W. Brown

PhD Thesis

Computing Laboratory
The University of Newcastle upon Tyne

January 1988

ABSTRACT

In the last few years the rapidly expanding application of computer technology has brought the problems of software production increasingly to the fore, and it is widely accepted that current software development techniques are unable to produce high quality software at the rate required to keep pace with this. To try to improve this imbalance, the field of Software Engineering has been advanced as a possible solution, attempting to apply the formal methods of an engineering discipline to software design and implementation. One of the most promising areas to be developed in this field is that of **Integrated Project Support Environments (IPSE's),** which attempt to spread the focus of attention during software development from the coding stage to embrace the whole development cycle, from initial requirements specification through to operational maintenance. These environments hope to improve production efficiency and quality by providing a complete set of support tools to help with each stage of software development, and to supply the necessary tool integration to ensure a smooth transition of use between these tools.

This thesis examines the services provided as part of an IPSE which allow users and tools to interact in a meaningful way throughout the life of a project. In particular, a **view mechanism** is seen as a vital component of these services allowing individual external views of the facilities to be defined which suit different users' needs. A model of a view mechanism for an IPSE is developed, and within a particular implementation of an IPSE, a view mechanism is formally defined and implemented. Finally, the view mechanism is analysed and discussed before concluding with some directions for future research.

ACKNOWLEDGEMENTS

Many people have influenced the work reported in this thesis, and it is my pleasure and privilege to acknowledge their contributions.

Firstly, I am indebted to my supervisor, Peter Hitchcock, who has helped to form many of the ideas reported in this thesis, and encouraged me throughout their development.

I have also been fortunate enough to work with, and receive help from, a number of people in the ASPECT project. In particular, Ray Weedon was a constant source of advice and assistance throughout his time here at Newcastle, Ant Earl encouraged and commented on many of the ideas, and Anthony Hall and Dave Robinson provided guidance to keep me on the straight and narrow. Other ASPECT members also played a part, including Ben Dillistone, Ann Petrie, and Roger Took.

In addition, the contents and presentation of this thesis have been much improved by Brian Randell, who has commented on a number of earlier papers and drafts of this thesis.

Finally, special thanks are due to Moira for her total support throughout the course of this work. Without her encouragement this thesis would never have happened.

Financial support for the ASPECT project has been provided by the Science and Engineering Research Council (SERC) through the Alvey Software Engineering Directorate.

CONTENTS

CHAPTER 4 VIEWS

CHAPTER 5 A MODEL OF AN IPSE VIEW MECHANISM

CHAPTER 1

INTRODUCTION

Software Engineering is concerned with investigating the problems of how to develop large, complex software systems to a strict time-scale, and produce a finished product of high enough quality that it can be used in life-critical situations such as aircraft control and industrial process control. However, efforts at producing such systems have all too frequently been poor, with many systems known to be inefficient, badly maintained, and difficult to use.[Som85] Indeed, it has been suggested[Boe81, DeM82] that

- at least fifteen percent of all software projects never deliver anything; that is, they fail utterly to achieve their established goals.
- overruns of one hundred and two hundred percent are common in software projects.
- the cost of maintaining a software system is typically more than twice the original development cost.

In addition to the existing inadequacies in software development, the trend is for software to be used in more and more diverse applications, driven by awareness of new situations in which computer technology can be applied, together with the continuing dramatic fall in the cost of hardware. This has led to a need for building even larger and more complex software systems. For example, from the initial attempts in the 1960's at automating Air Traffic Control (ATC) systems in the United States, estimates of the present computerised ATC system are that it now totals more than two million lines of code.[Hun87]

In the past the main thrust of attempts to improve efficiency and quality of software has been in the development of individual techniques and automated tools to help with one or more of the development activities,

speeding up that task, and ensuring its correctness. This has resulted in a large number of software development methodologies being defined, with automated tools which help in the application of the methods. However, constructing an environment in which the complete life-cycle of a large software system is supported by grouping together a set of ad hoc tools causes a number of problems. In particular, co-operation and integration between the tools must be actively supported to avoid inconsistencies and duplication of effort between the tools. An **Integrated Project Support Environment (IPSE)** attempts to provide an environment in which large software systems can be developed by integrating a set of tools which support a development methodology within an infrastructure that allows tools to communicate and co-operate in a controlled way. The IPSE infrastructure provides a set of common services appropriate to software development which can be used by the tools. Such services may include facilities for structured data storing and manipulation, for the application of user-defined rules to constrain and control data instances, and for the definition of external views of the services appropriate to different external users.

1.1. Thesis Aims

The main aims of this thesis are

- to investigate past and current trends of software development support, with particularly emphasis on the history and development of IPSE technology.

- to analyse the characteristics of a software engineering database used as the central component of an IPSE.
- to look in detail at the need for, and characteristics of a view mechanism as an integral component of an IPSE infrastructure.
- to formally define and implement an IPSE view mechanism as part of a larger IPSE project, and to analyse its use.

1.2. Thesis Outline

The thesis is organised as follows.

Following the short introduction given in this chapter, chapter two describes the basic architecture and components of IPSE's in general, and looks at their development over the past few years to the point at which standardisation on an IPSE tool interface is currently proposed.

Chapter three then looks in detail at the requirements and characteristics of a database for an IPSE, asking why we use a database at all, and why a commercial database without enhancements is unsuitable for the task. Finally, the database facilities which add to the semantic capture of a database are discussed with particular emphasis on their use in a software engineering database.

Chapter four introduces the notion of views of an IPSE, examining view mechanisms in programming languages and commercial database systems, analysing the requirements of an IPSE view mechanism, and looking at related work following similar aims.

Chapter five describes a model of an IPSE view mechanism building from a simple notion of a view to a mechanism capable of supporting the

requirements defined in chapter four. Using the model some of the problems associated with views, such as updating through a view, are discussed.

Chapter six is a short introduction to the ASPECT project. The ASPECT project was set up to investigate and develop a prototype IPSE, and much of the work reported in this thesis was carried out as part of that project. The chapter provides the wider context of IPSE facilities in which the work was developed, looking only at those parts of the ASPECT project which interact with the view mechanism and influence its design and implementation.

Chapter seven is a summary of a formal definition of a view mechanism for an IPSE using the specification language Z. The complete specification is given elsewhere,[Rob87c] and is summarised in chapter seven to allow the reader to follow the significant details of the specification without getting too enmeshed with the details.

Chapter eight describes the implementation of an IPSE view mechanism as part of the ASPECT project. In particular, the operators which allow a user to create and use view objects are described, together with the underlying structures which support these operations.

Chapter nine provides an analysis of the view mechanism by means of a set of appropriate examples to which the mechanism has been applied. Through these examples we can extrapolate as to its use within a large system development project.

Chapter ten reviews the aims of the thesis, discusses possible developments of the work reported here, and concludes with some final observations.

1.3. Acknowledgements

As the work on which this thesis is based took place as part of a larger project, it is appropriate at this point to acknowledge that chapter six reports work carried out in conjunction with a number of other project members, and that the specification of trees given in Appendix B is a shortened version of a paper by Anthony Hall.[Rob87c]

Apart from the above stated exceptions, all other work reported in this thesis has been the sole responsibility of the author.

CHAPTER 2

INTEGRATED PROJECT SUPPORT ENVIRONMENTS

2.1. Introduction

When a software system is designed and implemented, automated support for the development process is required. Where the software can be written by one person in a few weeks, the amount of support required may be small. For example, individual tools for program editing, compiling, and debugging may be all that are required. However, the current trend is for the use of computer technology in increasingly diverse and complicated applications, leading to the development of larger and more complex software systems. These systems typically require a large group of people to develop them, and may take many months to complete. For example, it is not uncommon for software systems development to involve more than a hundred people over a period of a number of years and produce in excess of a million lines of source code.[Row83, Gla82] With this vast difference in scale, at least the following points may be noted:

- no one person may be fully familiar with the complete system, so it is vital that the separate pools of knowledge can be integrated.
- the finished product may not exist as one single definitive version, but as a set of related versions to suit different user requirements.
- errors will be found in a large system, and hence maintenance of the running system will be an on-going problem.

- it is no longer economically viable to throw away the system and re-implement when operating conditions, or user requirements change. Therefore, the software must evolve over time.
- the management of a software project becomes much more difficult, in particular to control the development and maintenance process and to ensure its continued progress.

Therefore, if we hope to provide automated support for large scale software production, we must provide facilities to deal with these problems, and so support the complete software life-cycle, from requirements definition to operational maintenance, provide recognised paths of communication and co-ordination between the software developers involved in a project, and allow managers to exert control over the development and maintenance process and to accurately monitor its progress. Systems which attempt to provide such support are most often known as **Integrated Project Support Environments (IPSE's).**

2.2. The Problems in a Non-IPSE Environment

Research into different software development techniques and methodologies has been quickly followed by the production of individual software tools which support a method by automating some aspect of the use or application of the method. The tools may provide different levels of support, from simple clerical assistance by recording information in a computer's file store, to extensive checking of this information according to the rules defined by the method.[McD84] Hence, traditional environments in which software is developed consist of a machine operating system and filing system, together with an ad hoc collection of development tools which cover some part of the development

life-cycle, supporting some set of preferred development methods. A software product is progressed by application of the development tools to the data which is generated. This situation is summarised in figure 2.1.

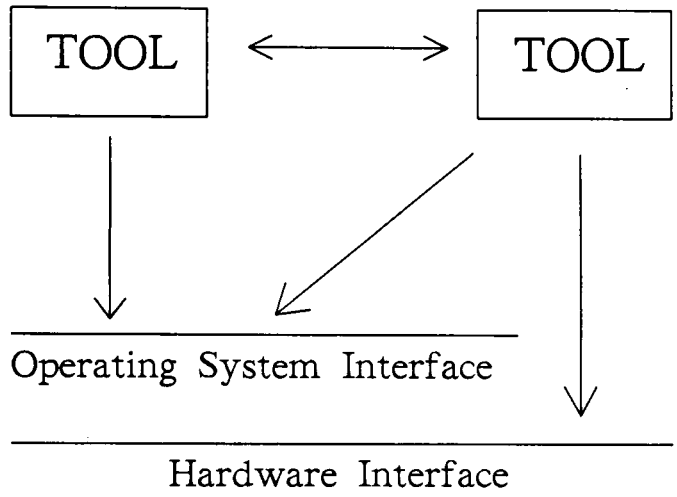

Fig 2.1 Typical Connection of Tools in a Traditional Development Environment.

In figure 2.1 tools access machine facilities through the operating system interface, recording data in the file store it maintains. The principle way in which tools are allowed to interact is through some commonly understood communication protocols embedded within the tools themselves. However, as the tools are often written without knowledge of the other tools with which they may interact, the use of individual tools within a larger development context can result in difficulties. The following problems are often experienced:

- different development tools often overlap in their roles and duplicate effort.
- tools designed to fit different development methods can often interfere with each other, produce inconsistent results, or are totally incompatible.
- the complex relationships and dependencies which exist between data items are often lost, or difficult to determine as these properties are created within the tools themselves and hidden within the data formats they produce as output.
- there is no integrating structure which controls both the ways in which tools are allowed to interact with the data, and the tools which individual users are given access to invoke. As a result, maintaining the integrity of the data is difficult.
- current practice relies heavily on manual co-operation and communication to ensure that project members can work independently and yet as a team. Such problems are compounded in large, distributed projects.

2.2.1. An Example

To illustrate the problems which exist in traditional development environments, and the need for tools to be integrated in their use of data, consider how the UNIX tool MAKE[Fel79] records information about inter-dependencies of program modules.

Figure 2.2 represents diagrammatically how a simple software system may have been constructed, using an example adapted from[Fel79] and developed in.[Hit87b]

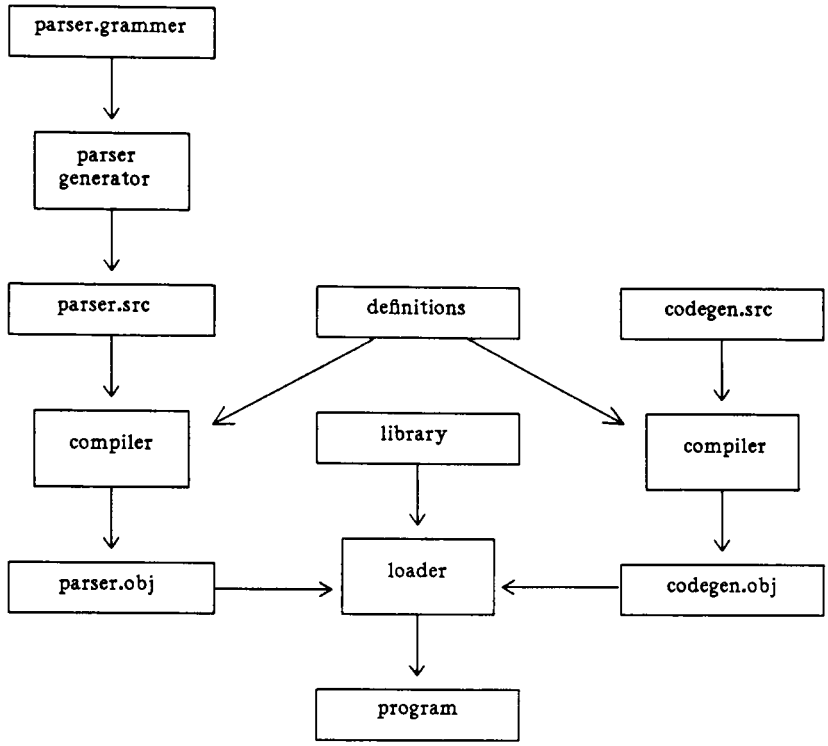

Fig 2.2 Data Flow for a Simple Compiler.

In figure 2.2, a simple compiler is constructed by processing a parser grammer to produce a parser object module, and then compiling this module to produce a relocatable binary version of the parser. This is loaded with the binary produced by compiling a code generation module to produce a running program.

To record the dependencies shown in figure 2.2, the MAKE tool relies on a file called a "makefile" which holds these relationships in a special textual form. In figure 2.3, we show how a makefile may look for this example.

```
program:  libl codegen.obj parser.obj
        load libl codegen.obj parser.obj

codegen.obj:  codegen.src defsl
        compile defsl codegen.src

parser.obj:  parser.src defsl
        compile defsl parser.src

parser.src:  parser.gram
        parser-gen parser.gram
```

Fig 2.3 Makefile for Simple Compiler.

It consists of a set of pairs of lines where the first line of a pair defines a dependency, for example, that the module "program" is dependent on "libl", "codegen.obj", and "parser.obj" modules, while the second line is a command which will generate the derived module from its constituent parts. For example, "program" is created by executing the command "load libl codegen.obj parser.obj". Hence, by reference to a makefile the MAKE tool can automatically generate derived modules from their constituent parts without a user having to remember how each module is created.

In isolation, the MAKE tool can be used to great effect for program generation and maintenance. However, as part of a larger development environment in which we would like to provide support for all phases of a software project, there are a number of problems with such a tool. Firstly, the relationships between modules are buried within a makefile in a special form which is specific to the tool. If we now want to use that information within some other tool, for example if we want to analyse the impact of changing a particular module and had such a tool available, then this new tool would need to read in the makefile and decode the relationships which exist, before it can perform its function. No doubt the information it obtains will again be written out in some special form to a file, and if we now wanted to run a report generator with that information, it too would have to

interpret the data and convert it to a form it recognised. This not only results in duplication of effort within different tools, having to re-interpret the structured information each time, but is also a source of problems when changes to the data format take place. It is not now possible to make any changes to the way data is stored without changing all the tools which access the data.

A second consequence of this approach is that maintenance of makefiles for large systems is a particularly difficult task. This is because the complexity of the system is reflected in the complexity of the makefile, and as a result it is difficult to assess how the makefile should be amended to reflect changes in the way a system is built. If the structures and relationships hidden within a makefile were to be held explicitly in some appropriate form, it would then be easier to alter this information when changes are needed.

Finally, if we now wanted to apply some form of constraints on how systems are built, for example to apply project standards for system building, then with the makefile approach we would find this particularly difficult to achieve as it is not clear at what point such rules could be applied. This is because integrity checking of the data is carried out in an ad hoc manner within individual tools themselves. If we could remove the task of integrity maintenance of data from each of the separate tools, record the rules in an accessible form, and apply the checks to the data at a central point, then we would have a mechanism which gave us much more confidence in the correctness of the data.

2.2.2. Tool Families

A partial solution employed by many existing development systems is to tightly couple sets of tools into families which share a common pool of knowledge, as shown in figure 2.4.

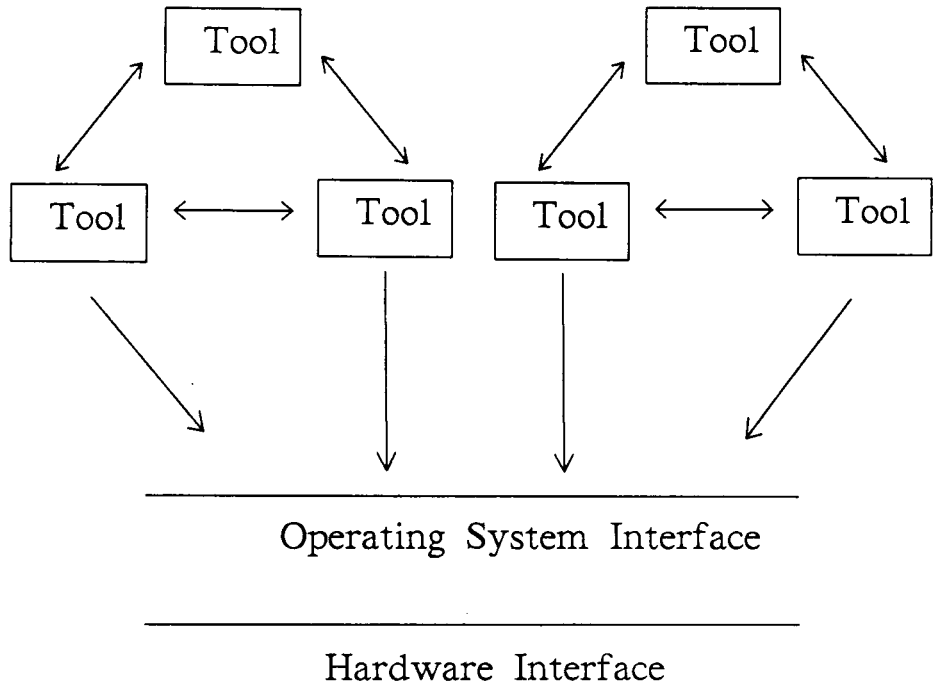

Fig 2.4 Grouping of Tools into Families.

Recognising the need for co-operation and integration between tools, one approach has been to build families of tools which can work in concert to support some part of the development or maintenance process.[Row83, Do176] These tool families share a common data format and operating conventions which allows them to interact in a meaningful way. A typical example is the set of UNIX tools which make up a documenter's workbench.[AT&84] This

allows, for example, a single document to contain tables, equations, and diagrams, and to be processed by passing the document through a sequence of appropriate tools which share a formatting style and a set of formatting conventions.

However, the conventions on which a set of tools are based are still embedded within the tools themselves, and, for example, to add a new tool to a development environment would involve understanding the communication and data conventions of the tool sets with which you would expect the new tool to interact. This would involve re-implementing all the checks and structuring operations which already exist within the other tools. Also, if you tried to use the new tool with tools built on different conventions, then the results would not be as expected.

2.3. The IPSE Approach

Having recognised the problems of integrating a set of individual tools to support the complete life-cycle of a large software project, researchers then began to focus their attention on the means of integration within a support environment, rather than on the individual tools themselves. By providing integrated support for software development, it was hoped that much more control could be maintained over the development process, while at the same time easing the transition of a software product from one development phase to the next.

The basis of providing integrated support has been through attempting to remove from the individual tools many of the data structuring and control facilities which typically are duplicated in each tool, and to maintain them at a central point which allows them to be more easily amended and

consistently applied.[Bro87] This has been coupled with an increasing recognition that effective support for software development can only be achieved if the tools work together within an effective development process.

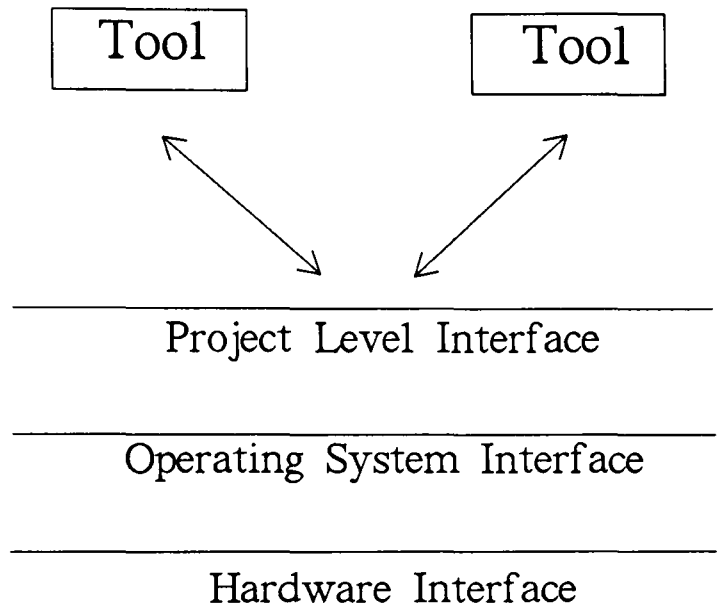

Fig 2.5 Tool Interaction in an IPSE.

In figure 2.5 we can see that the tools communicate by interacting at a higher abstract level than the operating system interface. The interface at which tools and users interact has been raised so that it is closer to the actual problem domain, being concerned with project level objects such as users, their roles within the software development process, and the tasks they carry out, rather that the operating system concepts of files and program processes. We can think of this more abstract level as a *project level*

interface, as through this interface facilities and services are available which help control software development at the project level. Typically, the facilities offered are for recording structured data using a database, and for the controlled sharing of data between users at the level of documents and programs.

Examining environments which provide integrated support for software development, we can divide systems into two approaches based on the exposure given to the project level interface, and consequently the ease with which new tools can be added to the environment to support new methods and techniques.

2.3.1. Closed IPSE's

In the first approach, which provides what we can call **closed** environments, steps towards fully integrated environments have been taken by providing a set of tools for supporting the life-cycle of a project within a preferred model of the software development process. Typical of this approach is the *Perspective* IPSE.[Sys84] In a *Perspective* environment, tools communicate mainly via a database, which records all relevant information about a software project throughout its life cycle. The database is structured in a way which allows meaningful relationships between data items to be maintained, for example, the relationship between a specification document and its implementation in a programming language. The tools provided form a fixed set supporting a single method of project development. In this case, a set of tools support the MASCOT design notation,[Jac83] with automatic translation of validated MASCOT designs into an extended form of the Pascal programming language. Additional tools then allow compilation and debugging of programs written in this extended form of Pascal. Further facilities exist to allow

communication between project members while controlling the sharing of data between them through the use of versioned data items.

A large number of closed IPSE's have been built in the last decade, each supporting some preferred set of software development techniques. An early examples was TOPD,[Hen76] which supported a top-down approach to program design which translated into COBOL code. Later examples include TOOLPACK,[Ost83] which is specifically designed to support small-scale development of mathematical software implemented in FORTRAN, and ARCTURUS,[Sta84] which supports the use of Ada as a command, design, and programming language. Many more such systems are described in.[Hau82]

The important features of a closed IPSE are that although tightly coupled support is provided by the IPSE for a designated development methodology, no mechanisms are available for the IPSE user to add new tools to support a different development approach, and very limited facilities to tailor the IPSE to suit a particular organisation's needs. Consequently, a closed IPSE may provide excellent automated support if the development techniques supported mirror closely an organisation's existing development strategy, and are well suited to the application system under development, but otherwise may involve the organisation in a large amount of re-learning, or may not be directly applicable to some systems. Similarly, as ideas and opinions concerning software development continue to evolve, support for new methods and techniques will also be required. With many closed IPSE's it is not easy to add support for such techniques without a considerable amount of re-implementation of the IPSE itself by the IPSE vendor. The additional project level services which are available in an IPSE, such as the improved data structuring facilities of a project database, are not made accessible to the end users who may want to customise the IPSE to suit individual project needs.

2.3.2. Open IPSE's

Recognising the problems of inflexibility which are evident with closed IPSE's, some of the more recent work in IPSE development has led to the creation of what can be called open environments.[Ste87] In this approach the role of the IPSE is seen as the provision of an infrastructure into which tools can be embedded. The IPSE provides control of all data developed during the life-time of a project by providing a set of facilities accessible through a more appropriately structured interface than is conventionally available in a simple tool and operating systems environment. So, for example, facilities available at this interface may include support for the structuring and storing of information such as documents and program modules, for configuration and version control of data items, and for the sharing of information between groups of users. As this provides the lowest level at which any of the IPSE services may be accessed, and is therefore the interface at which tools are written, this interface is known as the Public Tool Interface (PTI) for the IPSE.[Lyo86] The required openness of the IPSE is achieved by making the PTI extensible in order to support new methods and tools, and configurable, so that a project can impose particular methods of working if desired.

One of the first IPSE's to begin to recognise the advantages of providing an open environment was CADES[McG80] which was specifically built to support the development of ICL's VME/B operating system. Although originally configured with a fixed set of tools, a form of PTI was made available to allow new tools to be integrated within a CADES system.[Rob87b]

2.3.2.1. Basic Architecture

The architecture of open IPSE systems has been initially guided by work carried out to define requirements for an Ada Programming Support Environment (APSE), reported in the STONEMAN document.[Bux80] Here, to support the development of large programs written in Ada, the need for automated support of the program development process led to the definition of a set of requirements which outline the basic architectural components of an APSE. To summarise the report, the basis of any APSE must be a database which records data items and their relationships in a structured and accessible form. The database provides the integrating factor for tools which communicate through this structured repository for data. A Kernel Ada Programming Support Environment (KAPSE) is then defined as the database together with communication and run-time support to allow a set of Ada programs to be executed. With the addition of a minimal set of tools to support the creation and maintenance of Ada programs, a Minimal Ada Programming Support Environment (MAPSE) is envisaged. Finally, an Ada Programming Support Environment (APSE) is constructed by extending the MAPSE to provide support for different programming methodologies and techniques. This architecture is summarised in figure 2.6.

An important feature of the architecture shown in figure 2.6 is the interface between the KAPSE and MAPSE facilities. Essentially, this interface provides access to the set of services provided by the KAPSE to the tools implemented as part of the MAPSE and APSE. In this way it is equivalent to what we have called a PTI.

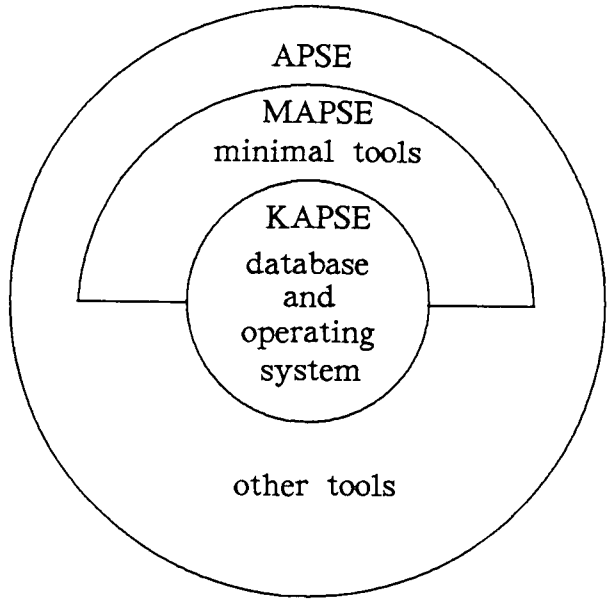

Fig 2.6 Basic Architecture of an APSE.

2.3.2.2. Further Developments

The approach advocated in STONEMAN has gained a wide measure of acceptance over the last few years, and a number of efforts have been made following similar principles. Here we briefly summarise the direction of this work.

In the United States, mainly funded by the Department of Defense (DOD), work has been concentrated on implementations of APSE's following the guidelines of STONEMAN. In Europe, and in particular the UK, with funding from both the Alvey and ESPRIT programmes, IPSE work has been much more diverse in nature. While the ESPRIT work has been mainly concentrating on the tools which will be needed to populate IPSES's and a single framework through which they can interact, the Alvey programme has funded

three concurrent projects which are investigating IPSE design and construction. The first of these is the ASPECT project.[Hal85] Following many of the principles established in STONEMAN, ASPECT has been working over the past three years on research into, and prototype development of, an open, extensible IPSE. Much of this work is reported in detail later in the thesis. The second project, ECLIPSE, is much more concerned with intercepting current proven technology and constructing a production-standard IPSE to support both Alvey and ESPRIT funded tool producers.[Ald85] This is primarily a set of tools written to the interface provided by an implementation of the Portable Common Tool Environment (PCTE), which is a PTI defined within the ESPRIT programme as the interface to which all of its tool writers can work. The third, and much more ambitious project, called IPSE 2.5, is attempting to extrapolate from current technology to define the direction of future IPSE research in the next decade.[War86] In particular, it is concentrating on support for formal methods of software development. Its name comes from the Alvey definition of a set of existing tools linked by a file system as a first generation IPSE, those built on a database as second generation, and those built on knowledge-based techniques as third generation.[Mor87, Mai86] According to these definitions, while ASPECT and ECLIPSE are second generation IPSE's, this project is attempting to establish a path between second and third generation systems. However, as the project was begun much more recently than the other two it is still very much in an exploratory phase.

Work in industry is similarly moving in the direction of open IPSE's, most notably perhaps with recent work at IBM aimed at defining the architecture of a software engineering support facility.[Hum85, Hof85] The architecture proposed emphasises the need for a set of Common Tool Services (CTS) accessed by tools through a Common Tool Interface (CTI), analogous to a PTI. It will be interesting to see how this work develops and how it influences

other commercial IPSE builders.

2.3.2.3. Standardisation on a PTI

Early work in this area was motivated by the United States Department of Defense (DOD), whose analysis of their massive spending on computer software determined that a great deal of their resources were being spent on re-writing software in different languages to suit different operating conditions. This led to the design and development of the Ada language, and the recognition that the environment in which Ada programs were developed would itself have to be defined. Initial observations and requirements for Ada programming Support Environments (APSE's) were outlined in the STONEMAN document.[Bux80] When a number of concurrent projects were underway to develop APSE systems along the lines suggested in the STONEMAN report, the US DOD soon realised that the problems experienced by implementing software in many different languages would soon exist for tool writers who would not know which APSE interface to use when implementing tools. Potentially, tool writers would spend much of their time porting tools to work in different APSE systems. Hence, at the beginning of 1982, work began on defining a set of APSE standards for use within the US DOD which would permit the sharing of tools and other software between US DOD supported APSE's. In 1985 a proposed Military Standard Common APSE Interface Set (CAIS) was released for evaluation.[US.85a] Leading on from this work a set of requirements for a more advanced APSE interface set has been developed,[US.85b] and work is currently underway to enhance the first version of the CAIS to meet those requirements.

Motivated by a similar need for a common interface for tools, though for programs written in a variety of languages, not just Ada, the European

ESPRIT programme has provided funds for the definition of a Portable Common Tool Environment (PCTE) which can be used throughout the ESPRIT community.[Bul85, Sys87] All tool writers funded by the ESPRIT programme will write tools to this public interface, which will facilitate the integration of tools within a common framework. Implementations of the PCTE are now available, and many more are expected in the near future.

Hence, at present, support for a standard PTI is divided between those who advocate the CAIS, and in particular its enhancements to provide extended support, and those who support use of the PCTE. Both technical and political arguments abound in this debate, with many people desperate for standardisation as soon as possible, including tool developers who are trying to avoid unnecessary re-writing of their tools to suit different IPSE interfaces. However, many IPSE researchers favour a more cautious approach to standardisation, recognising that experience of using IPSE systems is remarkably thin on the ground. It remains to be seen how these arguments will develop.

2.4. Support Environments for Non-Software Design Activities

The process of software development shares a number of characteristics with other development activities, most notably Very Large Scale Integration (VLSI) circuit design, architectural design, and aerospace design. These activities can be grouped collectively under the title of "Engineering Design". It is interesting, then, to examine the approaches taken to provide computerised support in these related areas and compare them with the IPSE technology described above.

Computer-Aided Design (CAD) systems have been available for a number of years, and it has been recognised that in the past there have been

more attempts to provide automated support in design activities such as VLSI circuit design than for software design.[Stu83] What is interesting, however, is to note that the development history of support environments for non-software design activities has very closely followed that described above for software. In particular, the abundance of individual tools supporting some part of a particular design method are now giving way to integrated tool sets supporting more of the design process.[Kat83] Indeed, in parallel with the work in IPSE's for software development support, attention in non-software design activities is now turning from individual tools towards controlled tool interaction through the use of a design database.[Enc82, Ket87, Buc84]

Analysis of the characteristics of design data have revealed a number of problems with applying conventional database technology in the design environment.[Kat83] In the next chapter we perform a similar analysis for software engineering data, and it is interesting to note how closely the results equate to similar analyses of VLSI circuit design data.

Cross-fertilisation of ideas between different design activities is currently underway in some areas, most notably in the support for long-lived design transactions and complex data objects.[Kim85, Kat84, Lor81] It is clear, however, that continued collaboration will be of benefit to all concerned, with the result that the design process itself may be better understood, and the designer's needs more adequately supported.

2.5. Summary

The need to provide automated support to aid the process of complex software systems design, implementation, and maintenance, led to the production of individual tools to support individual development methods and techniques. Integration of the various facilities provided has been necessary to ensure that the complete project life-cycle is supported in a convenient, flexible manner within a model of the development process. This chapter has briefly examined the motivation and objectives of this work, with particular emphasis on the key aims of openness and integration upon which an IPSE is founded.

In the next chapter we look in detail at the support required at the heart of an IPSE: the mechanisms for data capture, analysis, and control.

CHAPTER 3

SOFTWARE ENGINEERING DATABASES

3.1. Introduction

From the earliest attempts to provide integrated support for the software process, the use of a database as the central repository for all information associated with a project was seen as a key element.[Bux80, Ost81] This was driven by the fact that centralised control of data had been achieved in other application areas as a direct consequence of the use of the rapidly advancing database technology. In particular, through explicitly recording the data in a structured form it can be controlled, queried, and manipulated giving a higher degree of integrity than is usually the case using a simple filing system as the basis for a persistent data store.

In this chapter we examine the importance of using a database as a structured repository for all project development data, and look in detail at the special requirements of a database designed to control software engineering data.

We conclude this chapter by examining some of the facilities which could be used as part of the superstructure of a software engineering database to control and maintain the integrity of the data. In particular, rules, transaction, and view mechanisms are needed to further control user interaction with a database.

3.2. Why use a Database for Software Engineering ?

Having recognised the need for providing automated support for the software development process, it is important to realise that underlying the active support provided by a set of tools must be the basic mechanisms for capture, control, and analysis of the data generated during the lifetime of a software project. This basic principle is further examined below.

3.2.1. A View of the Software Process

There are many ways to view the role of an IPSE in the task of software development, but perhaps one of the most fundamental is to see an IPSE as providing the infrastructure for supporting a complex decision-making process. From this perspective, the development of a software project is seen as the manipulation of a set of complex objects governed by past experience, and directed by a plan for the future. In this way, at any point in a project's development we are attempting to verify that the present state of the system is consistent with past experience, while ensuring that we have a complete, integrated plan for the product's future development. This must be achieved within the context of changing project requirements, the existence of errors and inconsistencies, and a myriad of organisational constraints, all of which are an inherent part of any large project.

Seen in this way, software development is a complicated exercise in information control and manipulation. Any environment designed to support this process must be fundamentally based on mechanisms for data capture, analysis, manipulation, and control.

3.2.2. Advantages of using a Database

The main attraction of using a database for any application is that it provides centralised control of its operational data[Dat86] (where operational data is the term used to describe the information of interest to an organisation which is recorded within the database). For software engineering applications, the operational data will include all the information generated during software development. In particular, requirements analysis, systems specifications, design documents, source and object code units, and testing and error reports must all be recorded. In addition to data concerning the software product, details of the project development process itself must be recorded such as which developers are responsible for each program unit and how each of the project activities are related.

We now re-evaluate the advantages generally associated with the use of a database, emphasising the particular needs of software engineering applications.

3.2.2.1. Reducing Redundancy

One of the consequences of central control of data is that the redundancy of stored data can be considerably reduced and controlled. In most cases a single centrally controlled copy of a data item can be maintained, which is sharable both by a number of different applications, and by many different users. In the development of large software projects, control of redundant data is a vital function for the following reasons.

3.2.2.2. Avoiding Inconsistencies

Reducing redundancy is a great help in removing inconsistencies in the data. For example, if an error is found in a program unit, then when the unit is amended the same change must be made to all copies of that unit, and all larger components that have been built using the unit must be notified and rebuilt. This property of change propagation is much easier in a centrally controlled environment, especially using a database that not only stores the data items themselves, but also explicitly records the relationships between those items, such as where each item is used.

3.2.2.3. Enforcing Standards

One of the problems in software engineering is that there are so many different ideas about how software should be designed and implemented. In an uncontrolled environment, this leads to incompatibility and inconsistencies between data developed using different methods. Maintaining central control of all development data is an important method of enforcing a set of development standards, providing a central point at which all design data entering the database can be validated before it is stored.

3.2.2.4. Maintaining Integrity

There are two aspects to integrity. The first is that access to data can be monitored, so that confidential information can be strictly controlled. Using a database this notion can be expanded so that each user is presented with an individual subset of the complete data, by using a view mechanism. This allows the user to get on with his/her work without distraction from irrelevant data, and provides a mechanism for restricting access to privileged

information.[Kel85]

The second type of integrity is that of ensuring that the data recorded is accurate and conforms to constraints we may wish to impose on it. This is partly covered by reducing redundancy, but additional data validation is possible by defining a central set of validation procedures, or integrity constraints, which can be applied to data before entry to the database.[Laf79] In this way, we remove the need for validation and consistency checks to be embedded redundantly in each individual application program accessing the data. Now we are able to control these constraints centrally, apply them consistently, and allow them to be updated without affecting the applications themselves.

3.2.2.5. Enhancing Data Independence

In most simple filing systems, inherent in each data application is a knowledge of how the data is recorded in secondary storage. For example, when data is stored as files, many tools which access those files must know how the data is structured within a file. This means that if there was some change made to the data format, then all of those tools would need to be re-written. With a database, however, tools work with a virtual interface, so that if the underlying storage structures were to change, the applications using the data would be insulated from this and could usually continue to be used. However, the database itself would need to be amended to change the mapping between the virtual interface and the physical storage structures. (This is a major topic of database design, and is dealt with in greater detail in[Dat86, Ull80).

3.3. Why not use an Existing Database ?

In general, database development has been geared towards commercial applications, to hold such information as personnel details and bank records. This has led to the development of techniques which make databases highly efficient in a commercial environment, but not necessarily so suitable for other application areas whose characteristics differ greatly from the usual environment. Table 3.1 is a summary of the main differences between the requirements for a commercial database, and a database to record software engineering data.

We now look in more detail at the requirements for a software engineering database by comparing the mechanisms used in commercial databases with those required to support software engineering applications.

3.3.1. The Nature of Design Data

In commercial databases the data held is usually fixed length and simple, consisting of values such as salaries, bank account numbers, and employee names and addresses. In contrast, data in a software engineering database will be complex, variable length, and could even be graphical. For example, the design of a software component may be recorded as an interconnection of smaller component designs, and these dependencies are an important part of the design data. They could be recorded in some graphical form, in a number of related versions, or be only partially complete. All this information must be recorded along with the design itself.

Commercial Database	Software Engineering Database
Most information is static and can be described *a priori*, so the schema is also static and compiled.	Continuous evolution of information - data about the environment itself (tools, methods, etc.), and the products being developed.
Update to the schema is infrequent, and controlled by a group of Database Administrators.	Change to the schema is expected and frequent. Many users will need to change the schema.
Data stored is atomic and fixed-length (eg. strings and numbers).	Data stored is atomic, but also structured. Could be graphical data, design documents, programs, and so on. Also, data items may be large and complex, and of variable length.
Small number of entity types, with large number of instances of each type. Often only simple, fixed relationships between entity types exist.	Many entity types, with fewer instances of each type. Complex relationships may exist between entity types, and new relationships may be created.
Initially loaded with large amount of data. Slow, constant rate of data growth.	Less data initially loaded. Rapid growth of database (both of structure and contents), which slows down after completion of design phase and increases sharply during implementation and testing.
Single-valued data items, which are updated in place.	Versions of data items are vital. Dependence on versions, and relationships between versions must be explicitly recorded.
Transactions are short, atomic, and can be used as the basis for concurrent data access.	Long-lived transactions (>hours) which may leave the database inconsistent for long periods. Cannot be conveniently used as locking units.

Table 3.1 Comparison of Commercial and Software Engineering Databases.

3.3.1.1. Complex Objects

In an application such as VLSI design, a typical object is represented by a large collection of records that belong to different record types.[Kim85] Unlike commercial databases, accessing and manipulation of this data is often based on high level object-based notions, so that the granularity of a database object need no longer be just a few records of similar type. There must be some method to control database operations at much higher levels of abstraction.

The above is also true in software design; a typical software object is constructed from many smaller components, and such an object hierarchy is important in maintaining control of the development process.[Bro85a, Bro85b]

3.3.1.2. Version Control and Configuration Management

The design process is both tentative and iterative. This has a profound effect on the growth of a design database, as it is necessary to record amendments to a data object as new versions of that object. Hence, in a design database it is important to control the dependencies between objects, so that a request for a data item obtains the correct version. IPSE's now typically contain version control and configuration management facilities as an integral part of their architecture.[Ost83, Leb84]

3.3.2. Transactions

If a transaction is considered to be the unit of database consistency and concurrency, then a typical transaction in a commercial application, such as removing money from one bank account and placing it into another, is of short duration, requires only a few database records, and occurs atomically in the sense that the complete transaction takes place, or it fails and the data is unchanged. For a design activity the notion of a transaction is very different, and has the following characteristics:

- conversational, requiring frequent interaction with the system before completion.
- may last many hours, especially as the complete design may be considered as one long database transaction.
- may use many records, as the objects accessed may be complex and highly inter-related.
- the concept of atomicity is not very applicable, as undoing a transaction in this environment may remove many hours of work.

These characteristics have a number of important implications, most notably in the areas of concurrency control and maintenance of integrity.

3.3.2.1. Concurrency Control

Whenever concurrent multiple access to shared data is possible, some form of concurrency control must be implemented to prevent data corruption by interference. In a commercial database the control technique most commonly used is known as locking, whereby a request for a data object sets a locking flag to say that the object is in use. Conflicting requests for that object are then denied until the lock holder releases the object by resetting

the locking flag.

However, this simple technique is of little use in a design application as the object to be locked could be large and the length of time the lock is required may not make it sensible for other designers to wait for the object to be released before proceeding. Other methods must therefore be used, taking advantage of the fact that most design objects are developed in isolation, and are shared at controlled times though a version mechanism.[Kat84] For example, in *Perspective* interaction with IPSE data is restricted to the creation, amendment, and controlled sharing of document versions.[Sys84]

3.3.2.2. Integrity

The tentative and iterative nature of the design process leads to a conversational style of database interaction. Integrity rules have a much more important role in this environment, and can be used to verify a design throughout this interaction, and hence influence a design's development. Thus, in addition to the static form of integrity checking provided through data typing, dynamic integrity checking must be applied at project-related points in the development of a software product.

For example, consider the design of some software component in a large software system. An initial design attempt could be an abstract description of the component using a high-level design language, or some graphical notation. At this point it is useful to perform some basic form of consistency checking to validate the overall design, as early detection of design faults is a vital service. The feedback from these checks will help decide how to continue the design and where to focus attention, promoting a structured approach to the design process by encouraging incremental development. In this way the design continues, slowly being developed and validated until

a final design is produced which can be more rigorously checked before allowing others to make use of it.

The importance of integrity rules is clear, but it is not obvious what to do when any of the rules is violated. The choice lies between:

- rejecting the offending data and only allow it to be recorded in the database when it is amended to pass the checks.
- allowing the data to stand, but having some form of warning to let other users know that the data has not been fully validated.

As there will be occasions when the integrity rules will need to be violated, usually when a design is incomplete, it is useful to be able to apply each of these methods when necessary.

3.3.3. Summary

The use of a database at the heart of an IPSE is based on the sound principle of the need for control of software development data. However, as has been shown above, using a database in a software engineering context places a whole new set of requirements on the database and its associated management system, and conventional database technology may not be easily applicable or appropriate in some areas.

In the next section we continue the examination of the use of a database for recording software development data by looking at the ways in which a database allows us not only to record data in a useful form, but also to provide more meaningful user interaction with the data. This is seen as essential in an application such as software engineering where the data recorded is highly inter-related, and accessed by tools outside the direct control of the database.

3.4. The Importance of Semantics Capture

Although it is true to say that the function of a database is to store information in structures that permit easy retrieval and manipulation, it is also possible to give a higher-level interpretation to the role of a database. At this more abstract level, a database may be seen as providing a model of some portion of a real world system. The database schema is now a formal description of that model, while the data values recorded in the database represent the state of the real world system at some point in time.

Seen in this way, it is important to ensure that the data values which can be stored in the database are restricted to the subset which reflect possible actual values in the real world. We can call this set the "meaningful" set of data values. Clearly, it should be the goal of any database system to be able to specify (in some formal way) the set of meaningful data values, and to provide mechanisms for their capture and subsequent manipulation. This work often comes under the title of "Semantic Data Modelling", as we are attempting to capture more of the semantics of the data rather than treating data as a set of static values. In particular, it is important to record the relationships and dependencies between data items, as well as the value a data item may have. By holding such information, we are able to constrain the set of values a data item may hold, and even to control when update operations (insert, amend, and delete) are permitted on a data item.

A more pragmatic reason for ensuring the accuracy of data concerns the propagation of erroneous data. In any system using a database, one of the consequences of applications sharing data is that many other applications may potentially retrieve, amend, or delete that data. As a result, any errors made by one application, such as changing a data item to an inappropriate value, can quickly affect other applications. In this way, it is very easy for a small amount of incorrect data to cause great problems in a shared

database. By capturing more of the meaning of the data structures which make up the database, and constraining the allowable values that such structures may hold, the possibility of such errors occurring can be reduced.

For Software Engineering applications, the need for enhanced semantics capture is apparent, as we are dealing with highly structured data items which exist in a complex network of inter-relationships. Not only that, but the tools and applications which access the data are often written by external agencies, and therefore not subject to direct control by the database administrators. This implies that we have no control over the integrity checks performed by such tools, and cannot rely on them leaving the database in a consistent state. In the following sub-sections we describe a few of the mechanisms which have been used to help add more meaning to the data stored in a database, and hence improve support for maintaining a consistent database.

3.4.1. Mechanisms for Capturing Semantics

A number of mechanisms are available within a database system to capture and control the semantics of the data recorded.

3.4.1.1. Data Modelling

A data model provides the formal framework for a database within which data can be represented and manipulated.[Dat83] There should be a clear and direct mapping between the representation of the real world as given by the model, and the real world elements themselves.

We can reduce a data model to three basic components:

- a set of object types
- a set of operators on instances of those types
- a set of general integrity rules

To model some part of the real world that is of interest, we attempt to identify common object types and relationships which exist. These are then represented in the model. By examining the behaviour of these objects in the real world, we can construct a set of operators to manipulate instances of these object types, and can also draw up a set of general rules which govern the possible states of the objects.

The aim in recent years has been the development of data models which reflect more of the structure of the real world entities that are being modelled. This is a way of capturing more of the meaning of the data being recorded, and hence more adequately constraining the model to resemble the real world system. For example, the relational model uses the single data structure of a relation to represent the existence of an entity, properties associated with an entity, and to record relationships between entities. However, later data models,[Ham81, Myl80, Cod79] have a much richer set of object types, which allow them to distinguish between different classes of object, supporting each one with appropriate structures, operators and integrity rules.

3.4.1.2. Rules

Although it has been stated that integrity constraints can be supported implicitly through the structure of the database schema, and indeed it has been said[Sch84] that the quality of a data model is directly related to its ability to do so, it is clear that this can apply only to generally applicable constraints on the data objects. For example, a general integrity constraint supported by a data model could be that when two entities are related via an association, then the existence of the association implies the existence of the two entities themselves (this is known as a referential integrity rule). We can claim this to be a general rule of the model, independent of the particular application data that is being modelled. On the other hand, for a particular application of a database, there will be certain instances of the data object types that conform to the general integrity constraints, but are invalid for this application. For example, in a personnel database, we may wish to impose the constraint that the "marriage" relationship that associates two entities of type "person" can never exist between two people of the same sex. Such an association, however, would not violate the referential integrity constraint given above.

A separate mechanism to deal with the definition and enforcement of application-specific rules is required to support these integrity constraints. Although there have been a number of papers discussing the need for such systems,[Dat81] and even proposed designs for systems,[Wil80, Lil84] most DBMS include only very primitive rule mechanisms, or no rule systems at all. Here we note a number of issues which need to be addressed in implementing a more complicated rule system.

- Defining a Rule. There must be a language in which the rules can be conveniently expressed, together with some means of checking that the rules so defined are valid.
- Checking a Rule. The process of checking a rule against a database state can be very complicated to implement. Typically, a triggering mechanism is used, whereby rules are enforced whenever some action takes place. An example might be creating a instance of an object type, at which point rules constraining valid instances of the object type could be applied.
- Violation of Rules. If a rule is violated by some operation, then it is not obvious what action should be taken. Three possibilities are: rejecting the operation by returning to a state immediately before its execution, accepting the operation but issuing a warning message, and attempting to amend the operation to conform with the rule.

3.4.1.3. Views

One of the consequences of maintaining data in a shared database is that all applications which access the data must work with a single, predefined set of data structures and operators. However, in some database applications, and particularly Software Engineering, a large and diverse group of users and applications will wish to make use of the data. For example, it is expected that a complete IPSE would be used by project managers, quality assurance groups, clerical and administrative staff, as well as software engineers and programmers. If the database consists of a single, globally available set of data structures and operators, then the following problems may occur:

- data structures and operators suited to one application, or class of user, may be inefficient, unnatural, and time-consuming to other users.
- the level at which database interaction occurs may be too abstract for some users (eg. programmers), while too low-level for others (eg. project managers). The working requirements of these users are very different.
- there will be some data which is of relevance to only a subset of users, and we may want to restrict the access to such data by other users so that they are not confused by this unnecessary information.
- often databases contain sensitive information which we may not wish to be generally available. Many different levels of security may exist, and we need a mechanism to control access to data at different levels.[Den86] Typical examples are personnel data such as salary details, and strategic information which is embedded in software applications such as radar and defence systems.
- for similar reasons, we may want to restrict the operations different classes of user are allowed to perform on data. For example, at the lowest level, a user may be given read-only access to certain information.

Early in the development of database systems the need for such functions was apparent, and a mechanism which was often used to tackle some or all of these problems was the notion of a view.[Tsi78]

A view mechanism allows the creation of abstract interfaces to a database. Each interface can be tailored to a particular class of users needs, at an abstract level suited to those user's style of interaction. The view mapping, which defines the interface in terms of the underlying database, filters out unnecessary or sensitive data, and may derive new abstract data and operators.

In a Software Engineering application, the necessity and importance of such a views mechanism is investigated in this thesis. Clearly tools which access the data will interact with the database at different abstract levels which can be supported by a view mechanism. More importantly, it is expected that existing tools will be ported to new IPSE's, potentially requiring a great deal of re-writing. A view mechanism, however, could be used to define abstract interfaces to the database which require a minimum amount of change to the tool to allow it to run in the new environment. A method of providing such a mechanism is a major aim of this thesis.

3.4.1.4. Transactions

Interaction with the database is usually through the execution of a sequence of database operations which we wish to be considered as a single atomic unit. For example, the operation of reserving a seat on an airplane in an Airline Reservation System may consist of two smaller operations of checking a seat is available, and then making a booking for the seat. Obviously, these two operations must be bound together to be treated as a single operation. This unit of interaction is known as a database transaction, and is supported by a transaction mechanism. Typical functions of a transaction mechanism are to prevent other users from accessing data which is currently being amended, and to undo partially completed transactions in the face of failure or user activated transaction abort. Traditional approaches to these problems rely on locking of all information which is to be used in a transaction until the transaction either terminates successfully, or the information is released without change.

In more recent database applications, most notably in design applications such as Computer-Aided Design, VLSI design, and indeed Software Engineering, the traditional notion of a transaction must be extended to deal with the more conversational style of interaction which exists. Transaction mechanisms for database systems have been adapted to cope with this, mainly founded on the principle that most design data exists as a set of related versions, and that the traditional "update-in-place" found in commercial database systems rarely occurs. It is more common that a document is developed in isolation, and made available to other users at controlled points using a version mechanism. Hence, access to design data will generally be for reading purposes only, with the possibility that further versions of the item are developed. This is an active area of research which is currently receiving a great deal of attention.[Kat84, Kat83, Lor81]

3.5. Summary

Having examined in detail the advantages of using a database for software engineering data, an analysis of the requirements for a software engineering database has revealed a number of problems with the application of traditional database technology. In summary we can say that traditional database technology is inappropriate for this application because:

- the highly structured nature of design data and its complex interrelationships are not easily represented with traditional data models. In particular, support for user defined rules may be needed to maintain the integrity of the data.

- user interaction with the data cannot be adequately modelled using the traditional notion of a database transaction. A particular problem is the use of a transaction as the unit of database concurrency.
- allowing data interaction at different abstract levels is vital within a software engineering database. The traditional database notion of a view falls well short of the support that is required for this.

Providing database support for software engineering data is an active area of current research, as shown by a recent bibliography.[Ber87] However, the remainder of this thesis is concerned with examining one of the mechanisms seen as important for a database system, and which has received little attention in the context of software engineering databases - the use of a view mechanism. We begin by looking at some of the principles behind such a mechanism, before defining and implementing a view mechanism for use in a particular IPSE system.

CHAPTER 4

VIEWS

4.1. Introduction

Common to a number of areas in computing science is the notion of providing different abstract interfaces, or views, of data and operations at a fixed level. In investigating the advantages of providing a mechanism to support such a process as part of an IPSE, we can examine these other areas to give a guide as to how an IPSE view mechanism should be designed, implemented, and used. In this chapter we discuss the use of views in programming languages and databases, and from these areas establish some of the principles which govern the later work on the design and implementation of a view mechanism for an IPSE.

4.2. Abstraction, Views, and Abstract Data Types

Human beings understand the world and solve the problems it poses by means of a process called **abstraction.** For example, when trying to understand how a car works in order to be able to fix it, we need to know something about engines, gears and the things we can do to them. However, when learning to drive a car, we abstract from our knowledge of how the car works and concentrate on manipulating the steering wheel, depressing the accelerator, and so on. These are alternative ways to view the same objects, with emphasis being placed on their different functional roles.

Similarly, it has been recognised that since computer systems are written to model real-world problems, they are more likely to be correct, adaptable, and maintainable, the more closely the data structures and operations they use mirror those embodied in the problem being tackled.

To facilitate this, it has been necessary to build into programming languages various mechanisms to define and manipulate the data structures being modelled at different levels of abstraction.[Sha84, Bis86] In the last few years, work in both programming language development, and Database Management Systems (DBMS's) has taken note of the need to provide mechanisms for abstraction.

4.2.1. Abstraction in Programming Languages

Most modern programming languages provide facilities to combine the primitive operators of the language into higher level or more abstract operations which are called procedures. By making it possible to call one procedure from within another, they have effectively provided a way of extending the primitives of the language, and define operators which are increasingly abstract.

Programming languages also contain a set of primitive data types such as "integer", "character", and "real". However, programmers seldom wish to work with such low level types. For example, if a problem involves complex numbers, it would obviously be helpful if the language offered such a type together with the operations with which to manipulate instances of that type. Since it is not possible to provide, as a primitive data type, all the possible types which programmers might require, modern languages such as Ada provide a mechanism for defining new data types derived from existing types.

Again, by allowing one abstract data type to be derived from another, a hierarchy of such types can be defined, each at an increasingly abstract level.

However, a data type is normally considered to comprise not only the data structure itself but also the operations which manipulate it. For example, the data type, "integer" in Pascal, incorporates not only the structure for containing all integers in the range of the machine being used, but also the operations, add, subtract, etc., for manipulating instances of this type. Therefore, a language such as Ada makes it possible to associate a set of operators with a particular data type as part of its definition, and forbids access to the underlying representation of that data type except through such operators. Such a type, with its associated operators, is usually called an Abstract Data Type (ADT).

Some important points concerning ADT's are as follows:

- If new ADT's can be defined in terms of existing ADT's as well as primitive data types, the new mechanism provides a very powerful means of extending the data types that can be worked with.
- Procedural abstraction is vital to the mechanism of ADT's, since it provides the power to create operators which manipulate the new data type.
- Inherent in the ADT mechanism is the concept of enforcing the new type. This means that only the operations defined with the type can be used to manipulate it. However, in Pascal, for example, although one can simulate the creation of a new data type (eg. using a linked list to simulate a first-in-last-out queue), it is not possible to enforce the type as a true ADT since there is nothing to stop the queue being accessed as though it were a linked list.

- The ADT mechanism also provides an appropriate degree of "information hiding" in the sense that if we have defined a queue ADT, the users of that ADT can only see the data structure as a queue and know nothing about how it is implemented (e.g whether as an array or a linked list or whatever). This allows the writer of an ADT and its users to be independent, with a controlled interface between them for communication. This in turn greatly facilitates decomposition of a program into separate tasks, which may be carried out by different people.

4.2.2. Database Views

The database community has long realised the need for providing mechanisms to support access to data at different abstract levels. Indeed, it has been said that "the main reason for the existence of DBMS's is to provide independence between the physical representation of data and a user's view of it".[Tsi82]

A database records information in a machine dependent way on secondary storage. This is often called the **internal level** view of data. However, not only will this internal representation of data differ from one implementation to another, it is also expected that it will need to be amended as the volume of data recorded increases, and storage organisation and access methods need to be tuned. It is clearly desirable to shield users of the database from these changes, and particularly to ensure that end-user application programs are not affected when such low-level changes take place. The property of insulation from changes to the internal level of a database is often called **physical data independence**, and can be defined as the immunity of

applications to change in the storage structure and access strategy of data.[Dat86]

This implies that we have a logical definition of the data, independent of the data's physical implementation. This logical data description, often called the **conceptual schema,** provides the basis on which the data can be analysed and discussed independent of the method of physical storage. In this way, the **logical level** can be seen as an abstraction from the internal level in the same way that a data type such as "integer" in the programming language Pascal is an abstraction from a machine's particular binary representation for integers.

However, the conceptual schema too can be expected to change over the lifetime of a database. There are two main reasons for this. Firstly, the data structures within the database will evolve over time as the needs and expectations of the organisation maintaining the database themselves evolve. Secondly, the schema will grow and be added to as new types of data need to be recorded, and additional properties of existing data types are recognised.

Again, it is desirable to shield end-users from such change so that applications have to be amended as little as possible. The property of insulation from changes to the conceptual schema is often referred to as **logical data independence.**

Hence, a further level of abstraction is required, to a level at which applications can be written. This is most often known as the **external level,** and while there is a single physical view of the database, and a single conceptual view, it is possible for each application to hold its own external view, suited to the application's needs. This three level architecture of databases is summarised in figure 4.1 below.[Tsi78]

A typical example of the need for multiple external views of the same data can be seen in a database maintaining information about program specification and design documents. It would be quite feasible for one

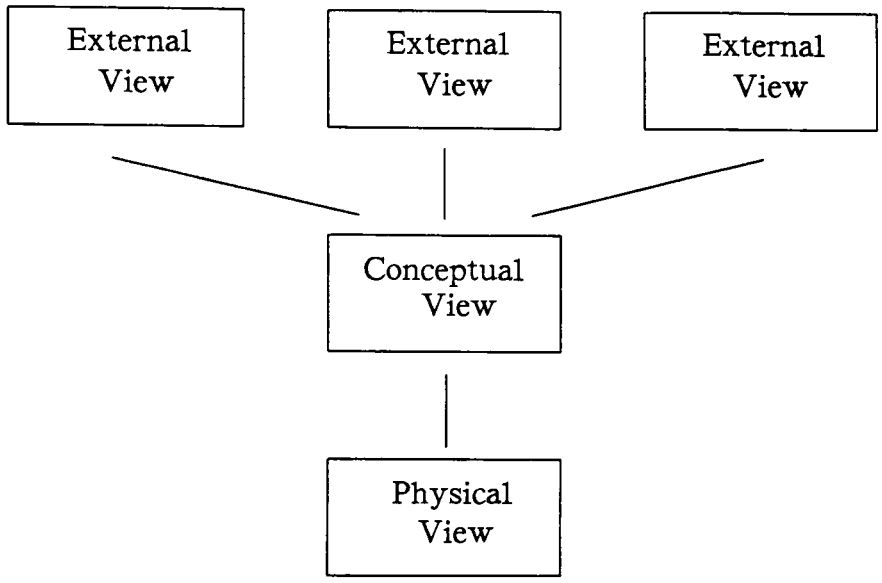

Fig 4.1 Three Level Architecture of a Database.

application to require full details of a particular specification document, while another may be only interested in whether the document has been finished or not, and who was the last person to change it. The ability to provide multiple concurrent views of the same underlying information is vital to the usefulness of a DBMS, and means that applications can be written to a view of the data that is suited to that application's particular needs. This makes the applications easier to write, understand, and maintain.

4.2.3. Summary

In conclusion we can say that the need to support the abstraction process which is so vital to a user's perception and understanding of a problem has naturally led to the development of mechanisms within both programming languages and databases which attempt to support it. While the underlying goals are the same, the approaches taken differ in a number of ways. These differences are discussed later in the chapter.

4.3. Project-Level Abstraction in an IPSE

Having seen the need for supporting different abstract views of data in programming languages and DBMS's, we can now examine The importance of a view mechanism for an IPSE.

4.3.1. An IPSE as a Extended Programming Language

One way in which we could view an IPSE is to say that it is an extended programming language in which we have shifted emphasis from the programming level alone, concerned with only a small part of the complete life-cycle of a project, to the **project level,** where issues such as requirements analysis, specification of design, and post-implementation maintenance, as well as management control of the development process, become important. Hence, while a programming language provides operators to manipulate fine-grain data items such as "integers" and "reals", an IPSE requires operators to manipulate much more coarse-grained items such as specification documents, programs, and management reports.

A project-level system will consist of a set of data types, and operators to manipulate instances of those types, both at a predefined abstract level. For example, we may have specification documents as one of our types, of which there may be many instances corresponding to particular versions of documents. Operators must be provided to create a new specification document, edit an existing document, display a document, and so on.

However, in the same way that applications in a programming language require different abstract views of the basic data types and operators, we also need to provide these at a project-level within an IPSE. Indeed, the need may be greater because of the diversity of users accessing the information contained in an IPSE. For example, a designer will want to see detailed information about the specification document he/she is designing, and have operators available to create a new document, change an existing document, analyse and check the consistency of a specification document, and so on. A manager, on the other hand, would only be interested in finding out which documents have been completed, and which are outstanding, and perhaps having summaries of their contents.

As one of the principal objectives of an IPSE is to reduce the amount of redundantly stored information in order to maintain the consistency of the information, it is inappropriate to maintain multiple copies of the same data at different abstract levels to suit each user's needs. It is much more desirable that where possible data is held in a single canonical form, with different views provided to suit each end-user's needs. Similarly, to ensure integrity of the operations on the data, it is desirable that new abstract operators on the data are defined in terms of those at a lower abstract level, in a similar fashion to procedural abstraction in a programming language.

One of the main aims of this thesis is to describe how we can provide a mechanism to support project-level abstraction within an IPSE to

accommodate the diverse set of users who will make use of the IPSE, while preserving the integrity and consistency of the data.

4.3.2. An IPSE as an Extended DBMS

An alternative way to view an IPSE is to focus on the data that is recorded in the IPSE, and in particular on the relationships and interdependencies that exist between data items.

For most programming languages, the only way in which they can interact with data that persists across program invocations is at the file level. As a result, application programs often spend a great deal of their time reading in the data from files, re-creating the structured information which has been written away in these files, and writing structured information back out to files in a flattened, unstructured form. This is not only wasted effort on behalf of the program, but must also be duplicated in many other application programs which want to see the data in a structured form, with the inherent consistency problems that this implies. This has naturally led to the use of persistent, structured repositories from within programming languages for recording data in the form of a database[Was79a, Row79] (see section 4.5 for an overview of the techniques which have been tried). Hence we have fine-grained control of data and the explicit recording of relationships between the data. This work has taken place within the context of providing a persistent, structured repository for data at a programming level, without regard for the wider issues of software development and the support that is needed in these areas. We briefly review some of this work below, with particular emphasis on the mechanisms they provide for supporting multiple concurrent views of the data at different abstract levels.

Recent work in IPSE development has also led to the use of a database as a structured repository for the data generated throughout the complete life-cycle of a software project. What is interesting, then, is to see how the notion of views, which was seen as so important in commercial database systems, can be used in an IPSE built around a database. In particular, we wish to extend the commercial database notion of a view to support the many different abstract levels of interaction which the diverse user base of an IPSE will require.

4.4. Support for Abstraction in an IPSE

Having examined in some detail how the process of abstraction has been supported in both programming languages and databases, we can now draw together the features of ADT's and views which we see as important within an abstraction mechanism for an IPSE.

4.4.1. A Comparison of Views and ADT's

To summarise the above discussion, it will be useful to directly compare the programming language notion of an ADT with the traditional database notion of a view. From this we may draw out those features which we consider to be necessary in an abstraction mechanism for an IPSE.

We can note at least the following points of comparison:

- both mechanisms are designed for a similar purpose: to provide better support for the user by allowing him/her access to information at an abstract level suited to his/her particular needs. In this way the user is more likely to understand the process which is being modelled, and hence errors are less likely.
- In relational database systems, a view is defined by a relational query written against underlying relations and views. The result is a single relation which acts as a type defining properties of its instances. An ADT definition consists of a set of type definitions derived from lower level types, together with a set of permitted operations on instances of those types.
- The definition of a view determines which update operations are allowed on the view. For example, with views involving a join of two relations, it is often the case that no view update operations are permitted because the effect of the view update on the constituent relations is not clear. The ADT approach is much more flexible in that any operation on a data type may be defined as an integral part of the ADT. This allows much greater control of the method and extent of data access.
- A basic difference between the mechanisms is one of persistence. Databases, and hence views of them, persist between program invocations allowing individual view object instances to persist between database accesses. For ADT's to use a persistent store, they normally have to do so at the file level requiring a manual conversion of the ADT instances to and from a file representation.

4.4.2. Features of an Abstraction Mechanism for an IPSE

Examining the mechanisms provided to support abstraction in both programming languages and databases, it can be seen that for an IPSE it would be desirable to combine certain features from both systems. In particular, we would like to have the flexibility of defining arbitrary abstract operators on a data type as provided in ADT's, together with the structured, persistent repository which is provided by a database. This would allow us to define abstract interfaces to IPSE information which consisted of operators suited to a class of user's needs, while maintaining the advantages of a single, underlying structured representation of data, shared between different groups of user.

The rest of this thesis examines the design issues in building such a mechanism, and goes on to design, specify, and implement a mechanism within the context of the ASPECT IPSE.

4.5. Related Work

We now briefly discuss related work with particular emphasis on how multiple, concurrent access to data can be provided.

We first of all look at how attempts have been made to combine programming languages with a database of persistent objects, then look at how databases are currently being extended with the ability to define ADT's as new domains from which values in the database can be drawn. We then discuss work in structured programming environments to present views of the single, canonical program representation maintained in a database of program structures at which tools can be directed. Then, we look at a recent proposal to combine a database with an object-oriented programming system to obtain a

persistent object store. This is built on the notion of a "view-object".

Finally, we see how existing IPSE's have allowed their facilities to be tailored to suit individual users' needs by examining three representative systems.

4.5.1. Combining Programming Languages and Databases

One of the problems with many programming languages which deal with data as complex objects is that there are no facilities in the language to record the objects in a data store that persists between program invocations other than at the granularity of a file.[Atk78] As a result, much time and effort is spent within application programs converting from structured, object format to flat, unstructured file format. Attempts to address this problem have progressed in three ways:

- the addition of new constructs in an existing programming language in order to support a database model. For example, the addition of relational constructs into the programming language Pascal.[Sch77]
- design of programming languages specifically to interact with data in a persistent store. For example, the languages PLAIN[Was79b] and RIGEL[Row79] are specifically designed for the construction of database applications.
- developing techniques to support persistent data which use the constructs already available in an exising programming language. PS-Algol is an example of such an approach,[Atk81] with the PISA project[Atk87] continuing the work to investigate machine architectures capable of supporting a persistent language approach.

This work is only interested in accessing a database at an individual program level, and does not concern itself with providing multiple, concurrent views of the underlying data, nor the problems of sharing this data between users. Hence, very little use is made of views in any of these languages, though RIGEL does have facilities for creating view modules which resemble the "packages" which can be defined for ADT's in Ada.

4.5.2. Abstract Data Types in Databases

The trend in the last few years has been to use databases for an increasingly varied range of applications. This has greatly influenced current work in the area, with many researchers proposing new database architectures and mechanisms better suited to a particular application.[Lor81, Buc84] One proposal has been to add abstract data type (ADT) facilities to a relational database.[Osb86, Ong79, Row87, Kem86] Such a mechanism allows users to create their own domains consisting of abstract data types which can be defined in terms of the lower level existing types. In this way, the complex data objects which are commonly found in design applications can be more closely modelled. For example, types can be defined to represent geometric shapes, polar and cartesian co-ordinates, and so on. The user can now define operators on these types by writing functions which perform the desired actions on the underlying data types. Finally, relations can be created which have attributes whose values are drawn from these abstract types. Now, queries on the relation can use the abstract operators to select those instances required.

For example, in the extension to INGRES which allows abstract data types for domains,[Ong79] we can define a relation "ComplexNums" to have a single attribute (field) which is of type "complex". This is a user-defined type

which contains two integers to represent the real and imaginary parts of a complex number. If we have defined an operator "Magnitude" which, given a complex number as a parameter, returns its magnitude, then we can write a query to retrieve all complex numbers recorded in the relation whose magnitude is greater than the magnitude of the complex number 3+4i as:

```
RANGE OF C IS ComplexNums
RETRIEVE (C.field1)
    WHERE Magnitude C.field1 > Magnitude "3,4"
```

In many ways this work resembles closely the kind of mechanism we would require for an IPSE. However, by allowing just the attributes of a relation to be abstracted we have only improved the way in which individual relations can model real-world entities. This mechanism can be seen as very much complementary to the kind of mechanism we would like for an IPSE, where we are also interested in providing multiple views of data objects and the controlled definition of view operators.

4.5.3. Garlan's Views

Interesting work is being carried out by Garlan as an extension to the programming support environment GANDALF.[Gar83] This system integrates a set of co-operating program development tools on a uniform structure database containing programs in the form of abstract syntax trees. However, although many of the tools wish to access a program in some structured form, as held in the database, it has been found that a structure appropriate to one tool may be inappropriate for another.

Garlan's solution is to generate abstract views of the database structure which are suited to the needs of individual tools. This provides each tool with a suitable view of a program, while maintaining consistency between the various representations.

It is clear, then, that the requirements within an IPSE context have been similarly recognised within a structured programming environment context, and indeed, the solutions offered by Garlan mirror closely those investigated in this thesis.

4.5.4. Wiederhold's View-Objects

In a recent paper by Wiederhold,[Wie86] an analysis is made of the problems of adding a persistent store to object-oriented programming systems, and an architecture is proposed which exploits the commonality of objects in programming languages and views in databases. The aims of this work are very closely related to those presented in this thesis.

The proposed architecture consists of three basic components. Firstly, a set of base relations serve as the persistent database for applications, containing all the data needed to create specified objects. Then, a set of view-object generators extract the data from the base relations in relational form, and convert it into sets of objects. The final component will be a view-update-generator, which will commit updates made against view-objects by performing their equivalent updates against the base relations.

Many of the ideas presented in this paper appear to be very appealing as regards the needs of an IPSE view mechanism. Unfortunately, it is difficult to make much comment on its suitability as only outline details of a proposed architecture are presented, without any information on how such a

mechanism can be implemented as part of a larger development environment. It will be interesting to see how this work develops in the future.

4.5.5. View mechanisms in IPSE's

It has been recognised from the original development of IPSE's that there was a pressing need for tailorability and adaptability of the facilities provided. Support for abstraction in IPSE's has been varied, but we now illustrate the mechanisms used by examining three different systems.

Firstly, we look at an example of what has been called a first generation IPSE,[Mai86] the UNIX development environment, and examine the way in which tailored access to data is possible through flexible mechanisms for tool creation and integration. Then, we examine the PCTE notion of a working schema to restrict individual tool access to subsets of the total schema. Finally, we look at the ISTAR mechanism of workbenches, which allow a set of tools to be selected by a user to carry out a particular unit of work.

4.5.5.1. UNIX Tool Sets

One of the strengths of the UNIX development environment[Dol76, Ker81] lies in the ability to create and integrate new tools in a flexible, convenient way. Then, through appropriate access controls, individual or groups of users can be given access to these tools. This in itself is not unique. What is important, however, is the way in which a user can make use of a tool within his/her defined workspace. Facilities are provided to:

- make access to tools easier through path name lists, which provide a list of directories to be searched whenever a tool is called.
- allow new tools to be easily created by the availability of an interpreted shell language which provides a simple interface to the predefined UNIX tools.
- embed the UNIX system calls into C programs by defining the system calls as a set of C language functions.
- be able to connect together tools by redirecting input and output of tools, and the creation of "pipelines" of tools. This is made possible through the single format used for all data communication - a file composed from a simple byte stream of characters.

Many IPSE's build on the simple, yet powerful facilities provided by the UNIX system, in particular by improving the facilities for security and control of data access, and by the addition of a database as a structured repository for development data.

4.5.5.2. PCTE's Working Schemas

Recognising that individual users and tools will only wish to access subsets of the information available in an IPSE, the Portable Common Tool Environment (PCTE) associates each process with a working schema to provide the context in which that process can execute.[Bul85, Sys87] Then, the process can access the underlying information through the working schema, which acts in an analogous way to a subschema definition in CODASYL databases,[Kay75] restricting access to some parts of the database.

Each data object is defined in a Schema Definition Set (SDS), and these are grouped together to form working schemas. In this way, data can be shared between processes by two working schema importing the same SDS.

Although the PCTE addresses the problems of data sharing and concurrent access through these mechanisms, only very limited facilities are available to tailor the environment to suit individual users needs, particularly with regard to the operations they are allowed to perform.

4.5.5.3. ISTAR's Workbenches

Illustrative of more recent work in IPSE's, the ISTAR IPSE is built around a model of the software process in which specific units of work, or "contracts", are defined and co-ordinated to simulate the controlled decomposition of a complex software project, and maintained within contract databases.[Dow86, Ste86] Within an ISTAR IPSE, tools are grouped together into meaningful sets (eg. Pascal tool set, VDM tool set, and so on) called workbenches, and an ISTAR user invokes a selected workbench to work on a particular contract.

The workbench idea allows a limited amount of tailorability of an ISTAR environment to suit the particular task at hand. This facility is further enhanced by including a set of tool building and tool integration tools as part of an initial ISTAR workbench. In particular, certain classes of foreign tools (ie. not written specifically to run in ISTAR) can be integrated into an ISTAR workbench through the use of an ISTAR tool that packages foreign tools in an "envelope". Then, the packaged tools interact with the user and the contract database via the envelope, which converts input and output expected by the tool to that supported within the ISTAR environment.

Certainly, the facilities offered in ISTAR go a long way towards achieving the aims of an abstraction mechanism for an IPSE, by allowing users to select a suitable set of tools when carrying out a piece of work. What we would like, however, is to extend these facilities to enable users to

augment and tailor their own environments under controlled conditions, and to try to move these facilities away from being hidden within tools, to being controlled and co-ordinated at a finer grain within the IPSE itself.

4.6. Summary

Views within database systems, and ADT's in programming languages, are important mechanisms for organising and controlling the complexity inherent in large software systems. The main difference of approach can be seen as one of scale. In particular, programming languages are concerned with manipulation in main memory of fine-grained objects, while sharing persistent data at the large-grained file level. Database systems, on the other hand, allow complex objects to be manipulated in main memory, and to be shared via persistent store. We have examined other systems which attempt to transfer the object-based persistent store to a programming language, or to add the fine-grained manipulation capabilities of a programming language to a database system. For an IPSE based on a database, the data structuring capabilities of a database view mechanism are vital to the integration of services provided, and hence an IPSE view mechanism must be built using these. However, the addition of ADT-like facilities to such a system will provide many useful capabilities, particularly the ability to define abstract operators. The next chapter examines how an IPSE view mechanism can be designed to take advantage of both of these approaches.

CHAPTER 5

A MODEL OF AN IPSE VIEW MECHANISM

5.1. The Aims of the Model

Earlier in this thesis particular problems and requirements of an abstraction mechanism for an IPSE were identified and discussed. We now describe a model of a mechanism designed specifically to support project level abstraction within an IPSE. In particular, the model we define is designed with the following explicit aims in mind:

- to be a flexible mechanism capable of supporting user and tool interaction with IPSE data at different abstract levels suited to individual user's needs, and hence make such interaction more meaningful, simpler, and less error prone.
- to provide facilities for a user to carry out a task in a working environment which that user can tailor and adapt in a controlled fashion to suit his/her individual style of interaction.
- to be able to use the mechanism as a management technique for controlling access to data by restricting the data that each user can access, and by explicitly defining the operations on that data which users can perform.
- to use the model to investigate the possibilities of using such a mechanism to aid with the integration of tools foreign to that IPSE environment by providing abstract interfaces through which such tools can access IPSE data. This can be achieved by providing an abstraction of the IPSE which closely mirrors a foreign tool's native environment, and so minimises the changes necessary to the tool to enable it

to operate in the IPSE environment.

- to integrate such a mechanism within the wider context of facilities offered by an IPSE. In particular, an IPSE will be concerned with controlled sharing of data between users, maintaining the integrity of the recorded data, and controlling allocation and execution of individual tasks in the development of a software project.

We now develop a simple model of view mechanisms in their most general form, examine two existing database view mechanisms using this model, and then extend the model to suit the stated needs of an IPSE abstraction mechanism.

5.2. A Basic View Model

We start with the basic definition of a view as a subset of a set of pre-existing objects, which we shall call "base objects". For our basic model we limit the base objects to data so that the view mechanism is only allowed to view data items. In this way we can think of the base objects as a collection of files, a set of relations, and so on.

For example, consider a set of base objects consisting of the items shown in figure 5.1.

Base Objects
A
B
C
D
E

Fig 5.1 A Set of Base Objects.

In figure 5.1, the base objects consist of a set of data objects which we may think of as relations recorded in a relational database, but they could equally well be different versions of source programs, specification documents, management reports, or any other object we may wish to record. We can say that the base objects are each given an identifier unique within the set of base objects.

Now, we can define view objects to be data objects with identifiers distinct from those used for base objects. An example of this is given in figure 5.2.

View Objects
FRED
JANE
JOE

Fig 5.2 A Set of View Objects.

In figure 5.2, the view objects have been given different identifiers to those used for base objects.

We can now define the basis of a view mechanism as the recording of a mapping relating a view object to a base object. So, for the above example, the view mappings could be recorded in a table as in figure 5.3.

View Mapping Table	
View Objects	Base Objects
FRED	A
JANE	C
JOE	D

Fig 5.3 A Simple View Mechanism.

From figure 5.3, we can see that the mapping table relates each view object to a base object. More strictly, we can say that all the view objects have an entry in the left-hand column of the mapping table, while the objects in the right-hand column will be a subset of the base objects. This is consistent with our definition of a view as a subset of the base objects, and allows for the use of local view identifiers for base data objects.

We can say that when an entry is made in the view mapping table, recording the view object together with its associated base object, then we have defined that view object. Then, when reference is made to a view object, the mapping table is checked, and the corresponding base object can be accessed.

Using this simple model, any number of views of the parent system can be created, where each one will map a view object to a base object. A view mapping table entry must be constructed for each view in order to convert the local view object into a base object. Remember, it is only the base objects that physically exist; views of them are only private abstractions created to help a user in his/her work.

Hence, we have a basic model of a view which allows the creation of local abstractions of a global set of data. It is important, however, to understand that the basis of the model for any view mechanism is the creation of a simple mapping table relating view objects to their underlying base objects.

5.3. Two Simple Extensions

As the model stands, we only have a renaming facility for base objects. This may appear to be of little use, but there are many commercial database view mechanisms which can be modelled in this simple way, provided we add two simple extensions to the model.

5.3.1. Expressions

The first extension we must allow to the model is to remove the restriction that view objects must correspond to exactly one base object. In other words, we allow the creation of abstract objects which may be combinations of more that one base object.

If we consider our data objects to be relations, then, we can envisage the need to create abstract relations as the combination of a number of parent relations. (In SYSTEM R,[Ast76] the parent relations are called "base tables", and the abstract relations are called "derived tables").

To do this we must allow the entry in the right-hand column of the mapping table to be an expression rather than a single object identifier. We say that when the entry is made in the mapping table, the new view object is associated with an expression defining how the view object is derived from the base objects. Then, whenever a reference is made to the view object, we must evaluate the expression to create a new value for the abstract object. In the case when the data objects are relations, we can allow the expression to be any statement of the corresponding relational algebra (or relational calculus) as this will create a new relation which can be used without distinction from the original relations. As the relational algebra provides a convenient and powerful way to manipulate relations, we have available an

elegant method for providing view objects as arbitrary combinations of base relations.

If the data objects are not relations, for example they could be files, then we must have some other way of expressing a combination of base objects as a view object. An interesting example is provided when we consider the objects to be programs, or program fragments. We can then envisage the expression resembling some form of UNIX "makefile", which we can think of as a recipe for constructing a larger program unit.

Note that we must extend the simple notion of a view as a subset of a set of base object to allow the use of operators for combining and manipulating base objects to produce a new abstract view object. In addition, we say that the operators form a *closed algebra* in that any view objects which are derived are of the same type as the base objects.

5.3.2. Nesting

The other important notion to add to the model is the idea of nesting. So far we have only talked about a view of base objects as a one-level mechanism - ie. a single parent system, with a number of views of that system. However, this simple mechanism will allow views to be nested to any level. This is possible because we have now defined a view object to be created within a closed algebra, so it will have the same characteristics as the base objects, and can hence be used as the parent for some further abstraction.

This situation can be considered analogous to the defining of types in a programming language such as Pascal. For example, a declaration of a subtype is a restriction of the parent type. However, it is still possible to apply

the rules and operations of the parent to the subtype, as the parents properties have been inherited.

Now, if an operation takes place on a view object, we may have to follow a number of mappings as in figure 5.4.

View Mapping 2		View mapping 1	
View 2	View 1	View 1	Base Objects
SID	e(FRED)	FRED	A
DAVE	e(JANE,JOE)	JANE	C
		JOE	D

Fig 5.4 An Example of Nested Views.

In figure 5.4, view objects have been defined at two levels, referred to as "VIEW 1" and "VIEW 2". For clarity, these have been recorded in two separate view mapping tables, one for each level. The objects at level 2 are defined as expressions involving the objects at level 1 using the notation "e(FRED)" to mean an expression involving the object FRED. Now, an operation referring to a view object would involve finding the object in the correct view mapping table, evaluating the expression associated with the object, and performing these steps recursively for each of the lower-level objects in the expression. Eventually, an expression involving base objects only will be obtained, and can be evaluated to derive the desired view object.

Although, in the simple example we have used above we have carefully separated the objects at two different abstract levels, in most existing database view mechanisms no such split occurs, and view objects can be defined which involve any other existing view object or base object. We maintain the distinction between abstract levels to more closely model the programming language concept of abstraction.

5.4. Applying the Basic Model

It is useful at this point to examine the model that has been developed thus far. This seems to be a good place to review the model as the facilities that have been described are quite sufficient to illustrate the view mechanisms of many existing database systems. The additions to the model that are described later in the paper can be thought of as enhancements to the basic view model.

We now examine two existing database systems - SYSTEM R[Ast76] and Ingres[Sto86] - to see how they have implemented a view mechanism. This is followed by a discussion on the problems of updating through views, leading to the extensions proposed for a view mechanism for an IPSE.

5.4.1. SYSTEM R and Ingres

Both SYSTEM R and Ingres take the same approach to providing a view mechanism, which can easily be modelled by the basic view mechanism that has just been described.

A view is defined to be a single derived table constructed from one or more base tables by an expression in the available relational calculus.† In both systems, any derivable table can be defined as a view. In fact, in these systems a view can be thought of as a "named query", as the result of any relational calculus query is a derived table. This is consistent with the model we have defined, as we have the ability to insert expressions into the view mapping table, where the expression can be anything that will produce a valid

† We shall use the terms "derived table" and "base table", as in SYSTEM R, where Ingres uses the terms "virtual relation" and "actual relation".

view object.

Also, as a derived table can be considered to be of the same type as a base table, it is possible to define a view in terms of other views, to create an arbitrary nesting of views. Again, we have allowed for this in the basic model, as we can relate operations on view objects to operations on actual data by reference to more than one mapping table.

The view objects that are defined act as additional abstract objects which can be manipulated along with the lower level objects. There is no concept of data hiding in either system; a user can see all objects at all times.

5.5. Update Through Views

One of the problems that exists with view mechanisms is that, although we have allowed any derivable table to be defined as a view, it is not possible to allow update operations on every derived table. We can best explain this by considering an example.

Assume that a database contains information about suppliers, parts, and the cities in which suppliers are located. Suppliers are instances of the relation S, parts are instances of the relation P, and the information that a particular supplier supplies a particular part is held as an instance of the associative relation SP. We could define a view which gives each part number, P#, together with the location, CITY, in which the supplier, S#, of such a part lives. This is illustrated in figure 5.5.

```
DEFINE VIEW PART_CITY AS
  SELECT P#, CITY
  FROM SP, S
  WHERE SP.S# = S.S#
```

P#	CITY
P1	PARIS
P2	ROME
P3	LONDON
P1	ROME

Fig 5.5 An Example of a View PART_CITY.

In figure 5.5, a view PART_CITY has been defined in SQL, and alongside it is the derived table that is produced at some point in time when the view expression is evaluated. There are no problems in using the view in data definition statements, but it is not possible to perform any update operations on the view. For example, if we wanted to add a new tuple to say that P1 is also supplied in London, we would just give the values involved, P1 and LONDON. However, as this is a derived relation, we must define a mapping that tells us what to do to the underlying base tables to reflect this change. Do we create a new supplier who is located in London? Do we use an existing supplier? If so, which one, there may be quite a few?

In summary, as is clear from this example, the problem of updating through a view is quite a complicated one. For a more detailed discussion of the problems of updating through a view see.[Ban79]

5.6. Extending the Model

To overcome the update through views problem, we can envisage three different approaches:

1. No view updates are allowed. Obviously, this will be the easiest method to implement, but severely restricts the use of a view mechanism.

2. View updates are only allowed when the update operations can be automatically and unambiguously converted to operations on the parent objects. An update translation algorithm must be available to do this (eg. SYSTEM R, and Ingres).

3. View updates may be allowed in all cases, and the view definer must explicitly define the operations to be performed on the parent objects for each view update operation

In terms of our existing model of the view mechanism, we can give each of the above methods an intuitive interpretation.

For the first alternative, clearly the model can remain unaltered, with the interpretation that whenever an expression is used in the right-hand column of the mapping table, then the corresponding view object cannot be updated. This interpretation will allow updates to take place if the view object is no more than a renaming of a parent object, as the object identifier will appear in the table, not an expression.

The second alternative does not have a particularly obvious interpretation in our model. The best we can say is that when a view object is mapped to an expression, the expression must be examined to see if it is sufficiently simple to allow update operations on the view. In SYSTEM R, for example, the expression can only involve one base table, and a restricted set of operators on that table (ie. There must be a one-to-one mapping

between view tuples and base table tuples).

The final alternative can be interpreted in the model by considering the addition of operator objects, as well as data objects, as components of a view. Hence, defining a view will involve defining both data and operator objects in terms of expressions involving parent objects.[Vel83]

When the decision between the alternatives was made for a view mechanism for an IPSE, the first alternative, not allowing updates through views, was rejected as this severely restricts the attraction of providing such a mechanism, and as a starting point we at least wanted to investigate how we could provide within an IPSE support for more than just read-only views of the underlying data.

The second alternative, automatic conversion of updates against the view to updates on the underlying data, is the method employed in most existing commercial database systems. We rejected this option, however, due to both the inherent problems that exist with this approach in existing systems, and the additional requirement which exist in an IPSE context. Existing view mechanisms using this approach have found that defining and implementing a general algorithm for translating updates against a view to the corresponding updates against base relations is a non-trivial task. Methods vary from defining a simple algorithm,[Cha75] which forbids updates to many views for which valid translations are theoretically possible, to defining a complex algorithm,[Kel85] which is much more difficult to implement and debug, but which permits many more updates. Also, users often find it difficult to determine *a priori* whether updates specified against a view are permitted, especially where the translation algorithm is complex. Sometimes the only option is to attempt the update and see if the translation takes place or not.[Dat86]

In addition, within an IPSE we would like to extend the notion of a view from the existing database concept of a single derived relation, to encompass the idea of an environment in which a user can work with a tailored set of facilities allowing multiple, concurrent access to project data. This led to us adopting the third alternative, which we can call the "abstract data type approach", in which each abstract data object is defined together with those operators which are permitted on that object. These operators could be for both manipulation and update of those data objects, and are defined by specifying exactly their effect on the parent data objects from which this data object was derived. Although the lack of an automatic update translation algorithm will require that the user must do more work when specifying a new view, we believe that this additional overhead to the user will be accepted due to the increase in control that is gained. We make the following observations on this approach:

- a view definer has much greater control over the operations permitted through a view, when the derived data objects are accompanied by the operations which can be performed on them. In particular, if a read-only view is required, then data objects will be defined with no update operators for those objects.
- update operators are no longer limited to the simple "insert", "update", and "delete" operations which are usually found in database systems. Abstract update operations can be defined using the same mechanisms. For example, in an IPSE database containing information about programmers and their assignments to carry out change requests, a particular view could contain a data object showing all change requests with status "completed", and an update operator "delete_all" could archive this information by deleting the appropriate entries, and adding them to a suitable archive relation. A user of this view would see nothing

of the underlying operations, being aware of the system only at the abstract level of completed change requests and their deletion.

- both manipulation and update operations are defined using the same mechanism. In the same way that the "delete_all" operation was define above, manipulative operators can be defined. For example, we could define an operator to return the number of completed change requests by writing a suitable operator that returns an integer result.
- the analogy between this mechanism and ADT's in programming languages is a close one. This not only means that we are using a mechanism with which many users are already familiar, but we are also easing the transition between the programming level of a project, interested in individual users and providing facilities for manipulating data local to that user, and the project level, where we are concerned with interaction between users and their sharing of resources.

5.6.1. Effect on the Model

As a result of the above decision, we now need to adapt the existing model to include the notion of view operators as an intrinsic part of a view. We do this in two steps. Firstly, we expand our notion of an expression from being simply a statement in the relational algebra, to the much more powerful notion of a statement in a programming language in which the relational algebra has been embedded. As discussed in,[Hit76] this not only gives us the power of sequence and iteration of statements, but also gives us the extensive manipulation facilities which will enable us to define general operators. The second step is to allow the left-hand side of a view mapping table to contain operator definitions, as well as data object identifiers. Now, for

example, we can consider the left-hand side of the mapping table as the defining expression for an operator, giving the operator name and the types of its parameters, and the right-hand side as the describing expression which is the sequence of operations which implements the operator.

As an example, consider the situation in figure 5.6 below, in which we have added the notion of operators as possible view objects. In this example, we have such operators, "op1" and "op2", where the notation "op1(t1,t2):t3" represents an operator "op1" with parameters of types t1 and t2, returning a result of type t3.

View Mapping Table	
View Objects	Base Objects
FRED	A
JANE	C
JOE	D
X():t2	e(op1)
Y(t1,t2):t3	e(op1,op2)

Base Objects
A
B
C
D
E
op1(t1):t3
op2(t1,t2):t3

Fig 5.6 An Example of Operators as View Objects.

In figure 5.6, abstract operators "X" and "Y" have been defined to accompany the abstract data objects which already exist. It is possible to define such operators with any number of parameters, and to indicate this, operator "X" has been given no parameters, and operator "Y" parameters of

types t1 and t2. In reality, such parameters will be formal arguments representing their types, analogous to the way in which functions and procedures are defined in a programming language.

Associated with each abstract operator in the view mapping table is an expression involving base objects. Hence, when an abstract operator such as "Y" is executed, the actual parameters of the call bind to the formal one in the definition, and the expression associated with the operator in the mapping table will be evaluated. This process is recursively applied until an expression exists involving base objects only. This resultant expression can then be evaluated to achieve the desired operation against the view. A more detailed examination of this process is deferred until later in this thesis, when a formal specification of the process is given.

5.6.2. Abstract Environments (AE's)

The final component of the model is to associate particular meaningful sets of data and operator objects together into an enforcible unit which we call an **Abstract Environment (AE).** We can consider an AE as an abstract interface to the base objects composed of a set of data and operator objects which can act as a user environment. This closely mirrors the ADT concept of packages and modules in programming languages such as ADA and MODULA-2. We can relate AE's through a hierarchy, or more strictly a connected acyclic graph, with the base objects, which we call the **Base Environment (BE),** as the root node of the graph. Hence, all the objects available in one AE will be defined purely in terms of the objects in its parent AE's in the graph. The lowest level AE at the root of the graph being the BE, while each AE which is created to extend the graph provides an interface to the

base objects at an increasingly abstract level. The objects in an AE will have been explicitly inherited from one of the parent AE's, or will be a combination of existing parent objects.

5.6.2.1. An Example

To illustrate the AE concept, consider the simple example given below.

Suppose we were interested in information concerning programmers, change requests for software components, and assignments of programmers to carry out these change requests. Represented in simple relational terms, at some point in time we may have the following data:

PROGRAMMERS		
PROG_ID	NAME	AGE
p1	fred	32
p2	joe	21
p3	jane	26

CHANGE REQUESTS			
CR_ID	DESCRIPTION	REPORT DATE	STATUS
cr1	fault in output	12-10-85	outstanding
cr2	crashes on input	8-11-85	completed
cr3	update documentation	23-6-86	outstanding
cr4	add new option	15-7-86	outstanding

ASSIGNMENTS		
PROG_ID	CR_ID	DEADLINE
p1	cr1	18-3-86
p2	cr2	20-8-86
p2	cr4	7-10-86

Coupled with this data will be a set of operators to insert, delete, and update individual tuples of each relation.

This lowest level view, or Base Environment (BE), will be used by the project controller who is responsible for all project data, and for initially setting up the AE hierarchy. For example, the project controller may wish to define an AE for the chief programmer which shows all assigned change requests with the programmers assigned to them, and also an operator to change the status of a change request to "completed" when the change has been carried out by junior programmers.

To do this, the project controller, who is working within the Base Environment, will define the data objects and operators necessary for the chief programmer in terms of the data and operators available in the Base Environment. When the chief programmer works in this AE, it may contain the following data :

CR_ID	DESCRIPTION	REPORT DATE	STATUS	NAME	DEADLINE
cr1	fault in output	12-10-85	outstanding	fred	18-3-86
cr2	crashes on input	8-11-85	completed	joe	20-8-86
cr4	add new option	15-7-86	outstanding	joe	7-10-86

Also, an operator "change-status" will be available, which, given the identifier of a change request in the data object, will change the status of that request from "outstanding" to "completed". This operator will have been

defined by the project controller as a sequence of lower level operations to insert, delete, and update individual tuples of relations using the operators provided in the BE.

Now suppose the chief programmer wishes to define an AE in which a junior programmer can work that only allows that programmer to see his/her own outstanding change requests, and not be given any operators to amend them. The chief programmer would do this by defining the AE in terms of the data and operators he/she has available in his/her own AE. In effect, the newly defined AE would be a further abstraction on what he/she has available. For example, this may result in the data shown below, with no operators being defined to manipulate or update this data :

CR_ID	DESCRIPTION	DEADLINE
cr4	add new option	7-10-86

The final result, then, will be a simple hierarchy of AE's with the Base Environment at the root, and the chief programmer and junior programmer AE's at subsequent levels.

Clearly, this is an over-simplified example to give a flavour of the view mechanism in operation. Without much difficulty, it is possible to envisage how this example could be extended to support a much larger project.

5.7. Summary

We have developed from a very simple model of a view mechanism to a much more sophisticated architecture which we hope is able to attain the aims stated at the beginning of this chapter.

After a brief description of the wider facilities of an IPSE developed as part of the Alvey-funded ASPECT project, we continue by examining a formal specification and implementation of a view mechanism based on the ideas of this chapter within the context of the ASPECT IPSE.

CHAPTER 6

THE ASPECT PROJECT

6.1. Introduction

Having described a model for an IPSE view mechanism in the previous chapter, and before we can discuss the specification and implementation of such a mechanism, we must describe the wider context of IPSE facilities within which the view mechanism must be integrated. In this chapter we describe the work of the ASPECT project, which is attempting to define and implement IPSE facilities based on the principles discussed earlier. We later go on to define and implement a view mechanism which can be used as part of this system.

6.2. An Overview of the ASPECT Architecture

ASPECT is an Alvey-funded project aimed at research into, and prototype development of a distributed host, multi-target IPSE.[Hal85] Its work has been directed towards two principle objectives: firstly, to provide an open environment in which new tools, methods, and techniques can be easily supported, and secondly, that the services provided should be integrated to provide consistent, secure transition between the many tasks performed in the development of a large software project.

As a result, an ASPECT IPSE offers to tools a powerful set of common services for structuring, storing, and manipulating information, for communication between users and tools, and for manipulating remote targets. Access to all of these facilities is offered through the ASPECT Public Tool

Interface (PTI) which provides a set of primitive operators to enable tool writers to make use of ASPECT services. This architecture is summarised in figure 6.1.

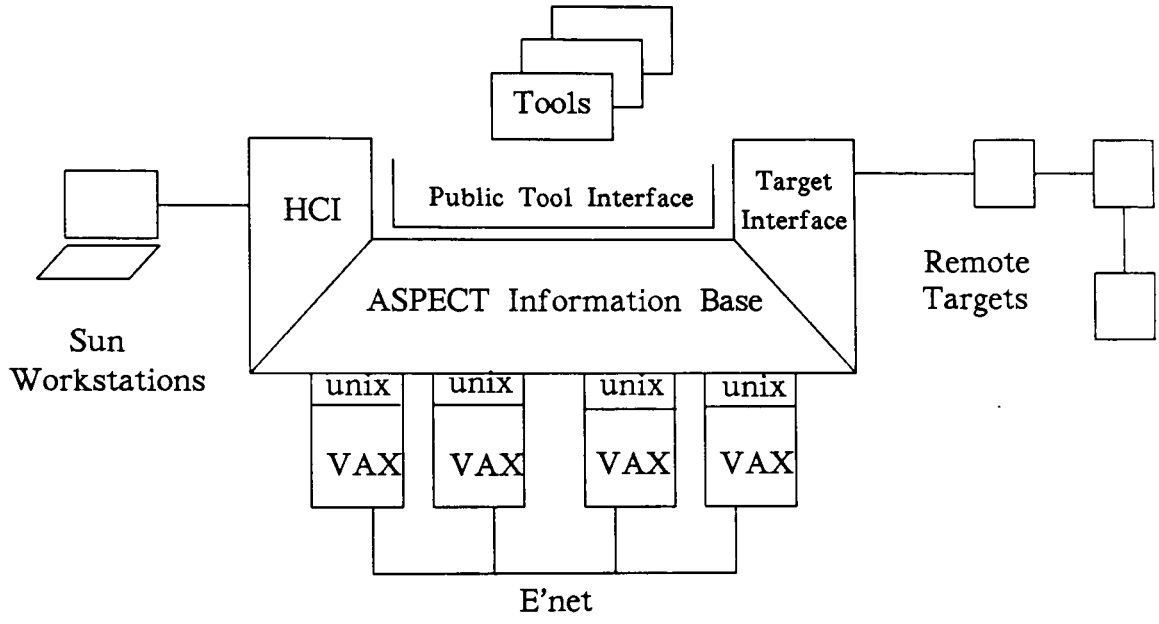

Fig 6.1 The ASPECT Architecture.

Here, we look in some detail at the facilities available for recording and controlling data which has been generated during the life-time of a software project. Details of other parts of the ASPECT architecture can be found elsewhere.[Too86, Hut87]

6.3. The ASPECT Information Base

Central to ASPECT is the information base, and it is the information base which achieves the integration between tools by providing a central, structured repository for all the information they manipulate.[Bro86b, Bro86c]

The information base is a database but contains in addition:

- Its own definition. A data dictionary of the structures maintained in the database is held as an integral part of the database, and this structural information can be accessed via the PTI in the same way as any other information.
- A rules mechanism. These support not only the integrity constraints of the basic data model, but also user-defined rules which may, for example, be the rules governing the use of a particular development method.
- Built-in structures to support software engineering. In keeping with the aim of integration, many of the structures (for example version identification and sharing of data) supporting development in the large are provided at the information base level.

We now examine the information base facilities in more detail.

6.3.1. The Data Model

Due to the enhanced semantic capabilities provided, the conceptual model for the ASPECT database was defined using the Extended Relational Model (RM/T).[Cod79, Dat83] We briefly describe the RM/T formalism with particular emphasis on the feature which made its use fundamental to ASPECT.

6.3.1.1. Entities and Entity types

In RM/T a database is considered to consist of a set of entities representing entities of interest in the real world. Every entity in the database is an instance of at least one entity type, with all entities of a type sharing the common properties of that type. An entity can be an instance of more than one type as types are related through a type hierarchy, and an instance of one type also inherits all the properties of its supertypes. For example, a programmer may be an instance of the programmer-type, but also inherit properties of the engineer and employee types which are superior types in the hierarchy. Associated with every entity is a unique, system-generated identifier called a surrogate. This can always be used within the database to uniquely identify the entity, although a separate mechanism exists for associating external names to entities.[Hit85]

6.3.1.2. Entity Classification

All entity types are classified to be one of either kernel, associative, or characteristic, and each type may be designative in addition. Through this classification integrity constraints can be applied over and above those normally enforced in the pure relational model, and as a result there is greater control over what constitutes a valid instance of a particular type. Looking at the classification in more detail, an entity can be classed as:

- Associative. An *association* is a potentially many-to-many relationship between two otherwise independent entities. For example, an assignment of a programmer to a project can be an associative entity. Associative integrity says that an association can exist only if all the entities participating in the association also exist. (ie. we could not assign a programmer to a non-existent project).

- Characteristic. The function of a *characteristic* entity is to describe some other entity. The characteristic entity is existence-dependent upon the superior entity. For example, individual commands which must be executed to keep a target object up-to-date can be considered characteristic of the target they recreate. Characteristic integrity says that a characteristic entity can only exist if the entity it describes also exists. (ie. a command line could not be characteristic of a non-existent target object).
- Kernel. A *kernel* entity is one which is neither characteristic, nor associative. For example, programmer and department may be kernel entities. These are the basic, independent objects of interest.
- Designative. A *designative* entity is a potentially many-to-one relationship between two otherwise independent entities. For example, a programmer who can belong to only one department is said to designate that department. Designative integrity says that a designative entity can only exist if all the entities it designates also exist. (ie. a programmer could not designate a non-existent department).

More complete descriptions of RM/T are available elsewhere,[Cod79, Dat83] however, the following subsections describe some of the facilities of RM/T which were considered important in choosing RM/T for use in ASPECT.

6.3.1.3. The Catalog

One of the most important features of RM/T is that an integral part of the model is a form of data-dictionary defining the internal structure of the database, known as the catalog. This meta-data is held in the same way as all other data, and can therefore be queried and manipulated using the same mechanisms. For example, queries can be posed concerning which entity types exist, and what properties are associated with an entity type.

6.3.1.4. Relational Basis

Although at a conceptual level RM/T deals with the more semantic notion of entities and entity types, at a lower level the model can be represented in terms of the pure relational model, with the mapping between the two having been defined. This has the advantage that the relational view of data is simple, well defined and understood, and is supported by a powerful general query language.

6.3.1.5. Extended Operators

With the relational query language available at one level, additional operators have also been defined at the entity level which extend the algebra of operations. For example, create, update, and delete are available at the entity level rather than at the relational level. Hence, it is not possible to by-pass the more semantic entity model to manipulate the data in its underlying relational form.

In addition, further data manipulation operators are available with RM/T. In particular, transitive closure of a directed graph relation is possible using the "close" operator which greatly eases querying of data in parent-child

relationships.

6.3.2. Activities

The development of a software engineering product is usually through the incremental refinement and implementation of a set of project plans. Typically, a project plan develops from an informal abstract description through to a detailed collection of inter-related specification documents, which can then be implemented. It is important to recognise that product development is as much about the coordination and monitoring of these activities as it is about the technical development of the system which these activities identify. In order to control this process, ASPECT supports the concept of an *Activity* as a unit of work that can be defined and executed.

Through supporting this process we not only have a method for recording product development as a series of related refinements, we are also able to monitor the complete life-cycle of a product as a natural progression of Activity definitions and executions. This provides a convenient and accurate way to model the state of a project from both the high-level management view, and for the individual project members.

6.3.2.1. Activity Definition

The only way to change the state of an ASPECT System is through the definition, and subsequent execution, of an Activity (ie. an ASPECT Activity is the software engineering database equivalent of a commercial database transaction). An Activity definition is essentially a set of rules governing the execution of a sequence of operations on the information base. Hence, by ensuring that all operations on the information base are carried out within the bounds of an Activity definition, we are able to restrict the allowable changes to the state of the information base to only those that are considered to be semantically correct.

We introduce the notion of constraining an Activity definition to deal with the situation where an Activity is decomposed into a sequence of smaller sub-Activities. This is to allow for the common mode of project development whereby a number of distinguishable areas of work within an Activity can be divided into separate smaller Activities, each with its own definition. In this way, a tree of Activity definitions is created in classical top-down development style.

Once this basic structure has been modelled, we can then introduce a number of special cases of Activity constraint. The most important of these is notion of refinement, when an Activity definition is constrained by a single more detailed Activity definition.

6.3.2.2. Activity Execution

The interface to the ASPECT system contains a collection of primitive operators and a mechanism for combining primitive operators and previously defined Activities to form new Activities. In this way, each execution of an Activity definition is said to form an atomic information base operation, and can be recorded as an Activity execution. However, because an Activity definition will usually be an element of an Activity definition hierarchy, represented in ASPECT by a tree of Activity definitions, we will also have the notion of an Activity execution hierarchy, which ASPECT models as a tree of Activity executions. The Activity execution tree can be used as the basis for recovery management following failure. In this way, we can have many instantiations of the same Activity definition by having a number of nodes in the Activity execution tree for each instantiation, and recording links between nodes in both trees.

After defining an Activity, we can say that each execution of the definition can be in one of three states:

1. Started. (ie. an Activity execution node exists).
2. Dormant. (ie. an Activity execution node exists but is currently not flagged as active).
3. Completed. (ie. the Activity execution node is flagged as complete).

By recording the execution of Activities in this way, and identifying the three possible execution states, we have a method for representing, controlling and monitoring all changes to the information base, relating Activity executions to their definitions, and for keeping a historical record of work that has been carried out.

In summary we can say that of key importance to ASPECT is the ability to monitor and control the development and maintenance of a software

product. It is also equally vital to maintain the semantic integrity of the information base by only allowing controlled operations to take place. The ASPECT Activity mechanism takes care of both of these requirements through the separation of the Activity definition and its execution, and with the ability to record Activity interdependencies in a flexible and convenient way.

6.3.3. Rules

In addition to supporting general integrity of an information base system by virtue of the modelling formalism adopted, an ASPECT IPSE will support the definition of rules which are specific to either an Activity, a project, or an organisation.

Typically, any operation which takes place in an ASPECT system does so under the control of a set of rules which govern the allowable states of the information base. These rules can be used in a number of ways: to constrain the operations a user is allowed to perform, to enforce an organisational policy, to conform to a particular development methodology, and so on.

6.3.3.1. Classification of Rules

Within an ASPECT system, a rule is defined to be one of two types:

- Static. A static rule is a function which maps all possible states of the information base to "True" or "False".
- Transition. A transition rule is a function which maps all possible transitions between two states of the information base to "True" or "False".

For example, in an information base containing data about programmers, a static rule could be that a programmer cannot have a salary greater than 50,000 pounds, while a transition rule could be that a salary increase for a programmer cannot exceed 20% of his/her former salary.

6.3.3.2. Consistency of Rules

Rules are organised into sets which are to be applied at particular points in time. For both static and transition rules the notion of a set of consistent rules is important. For a set of static (transition) rules, a consistent set is defined to be one for which a possible state of the information base (transition between states of the information base) exists for which all the rules of the set map to "True".

Hence, when a set of rules is defined, it must be a consistent set for its application to be meaningful. As the problem of deciding if a set of rules is consistent is semi-decidable, this process can only be applied to certain classes of rule.

6.3.3.3. Application of Rules

Rules are applied, or invoked, at particular points in time, to ensure that the state of the information base satisfies those rules (ie. they all evaluate to "True"). Two separate mechanisms are available in ASPECT to do this. Firstly, a manual mechanism allows a user to apply a set of rules at any time, and these will be checked against the current information base state and return the value "True" or "False". The second mechanism is automatic, and occurs on every relevant change of state of the information base. The rules applied consist of a set of global rules, which have been defined on

configuring an ASPECT system to always be applied, together with a set of applicable rules which are associated with the particular Activity that is currently being executed.

6.3.3.4. Rules and Activities

When defining an Activity, a user is able to specify:

- a set of pre-conditions (ie. a set of static rules) which must be satisfied before the Activity can commence.
- a set of post-conditions (ie. a set of static and/or transition rules) which must be satisfied before the Activity can terminate successfully.
- a set of applicable rules (both static and transition) which must be enforced during changes which take place within the Activity.

For example, consider a high level Activity that has been defined with rules as mentioned above. The Activity could be to write an overall design from an agreed requirements document, with a pre-condition that an approved requirements document exists, and a post-condition that some quality control authority must have approved the design. A set of rules may also be defined to apply during execution of this Activity.

We now chart briefly how the rules system will operate when such an Activity takes place. Firstly, before the Activity is started, the information base must conform to the set of global rules of the ASPECT system. Then, in addition to the global rules, the pre-conditions of the Activity must be met before the Activity can start. Once the Activity has started, the applicable rules to be enforced on the information base while the Activity lasts become those specified in the Activity definition and not those globally applicable outside the Activity. This allows for the situation within an Activity when the information base is temporarily inconsistent with respect to the

global rules. Finally, before the Activity can end successfully, the new state of the information base must conform to the post-condition of the Activity, in addition to the global rules which come back into force once the Activity is complete.

Therefore, using this mechanism, we have a way of controlling changes to the data at an Activity level, and the ability to apply constraints to the information base state at particular points in a project's development.

6.3.4. Views

This work is reported in the next chapter.

6.3.5. Shared Objects

Unlike conventional commercial applications, in an IPSE it is rare that objects are updated once they reach a stable state. Instead, new versions of objects are created. Examples are program sources and objects, management reports, and design documents, to name but a few. Typically, a single user is responsible for developing an object, and once in an acceptable state, the object is shared between a group of users.

This situation, coupled with the need for supporting long-lived transactions, has led to the development of a mechanism for sharing of objects in ASPECT in which each user is allowed to develop objects independently within their own work space. Then, when other users require access to an object, a "publish" operation is invoked which freezes the current state of the object, and provides read-only access to that object for those users. Each user must then "acquire" the object to be able to access it.

If for some reason the object later needs to be amended, for example due to error in the original, then a new version of the object is developed, and this again is "published" to the necessary users, perhaps accompanied by "withdrawing" the original version. The effect of the "withdraw" operation is to prevent other users from "acquiring" the object, and to indicate to those who have acquired the object that a new version is available. The users of the object can "dispose" of the old version before "acquiring" the new one. The action of "disposing" prevents the user from further access to the object.

This situation has many advantages as far as the database is concerned. If objects being changed are never shared, they can be locked into very lengthy transactions without penalty. This will also make recovery and distribution easier. We can afford to follow this update model because the rate of change of data objects will be slow compared with that of a commercial database, a positive feature of long transactions.

Ensuring that the correct versions of objects are chosen when a system is being built, or when a delivered system is being reconstituted, is known as configuration control. This will be built into an ASPECT IPSE together with the ability to define the version model that will be used. Clearly, the view mechanism could play an important part in giving users the versions required and in screening them from the underlying complexity of the version model. Investigating such issues is an important goal of the ASPECT project.

6.4. Summary

To provide a context for the work that is to follow, we have described in some detail the information base facilities of an ASPECT IPSE. We now go on to discuss the specification and implementation of a view mechanism to integrate with this system, and to analyse its use as a component of a larger development environment.

CHAPTER 7

SPECIFICATION OF AN IPSE VIEW MECHANISM

7.1. Introduction

Following the model that has been described informally in chapter 5, the next step is to specify the model in a formal notation before attempting a particular implementation. In this chapter, we begin by briefly describing the specification language used, the notation Z, before providing a fairly detailed summary of the formal specification itself. This is a shortened version of the full specification of the view mechanism given in.[Rob87c] For those readers unfamiliar with the Z notation, as well as the short introduction to Z given below, a summary of Z symbols and conventions used within this document is provided as Appendix A, and Appendix B contains a theory of trees, both as a self-contained example of the use of Z, and because a number of graph and tree structures are used as part of the specification of the view mechanism.

7.2. Specification

An important decision within the ASPECT project was that all ASPECT services should be specified in the formal notation Z, a specification language developed by the Programming Research Group (PRG) at the University of Oxford.[Abr82, Suf85, Fli87] Therefore, the view model has been developed, and is described below, using that notation. Indeed, the use of Z had a profound influence on the project, its ways of working, and its deliverables. In

particular, the clarity and precision encouraged by the use of a formal language led to the identification of errors, inconsistencies, and omissions which may otherwise have been undetected.

In the next section, we briefly describe the principles behind the Z notation, before summarising the formal specification of the view mechanism which has been developed for the ASPECT system.

7.2.1. An Introduction to the Z Specification Language

The Z notation consists of a mixture of English description and formal text. Because the formal parts of the specification may be inaccessible to many readers, the English text of the document is readable on its own as an informal definition of the facilities available and what they do. This description is supplemented by the formal text which gives precise and unambiguous meaning to each of the operations (where operation is the name given to the Z form of a facility).

Using a formal notation, rather than, for example, Ada package definitions, is preferable for at least the following reasons:

- The formal text defines the semantics of each operation, not just the syntactical conventions for its use.
- The formal text is independent of any implementation language and can be used as a specification of implementations in any programming language. This is particularly important within ASPECT which provides a set of services for tools which could be implemented on a number of different machines in a number of different programming languages. We will therefore not inadvertently build in to ASPECT restrictions based on the constructs of any one particular programming language.

- The formal text uses abstract notions which define a system at a high level, free of implementation detail. It therefore imposes precision at this abstract level, rather than leaving decisions at this level to be made by the implementor.

For a more detailed discussion of the benefits of this style of specification see.[Mey85]

7.2.1.1. The Z Notation

The formal language of Z should be readable by anyone with a knowledge of elementary set theory and first order predicate calculus. These two form the basis of the mathematical language embedded in Z. Also included within Z is a notation for manipulating pieces of this mathematical text; this is known as the schema calculus.

The basic method of specification is as follows:

A system is considered to have, at any one time, a certain state. The components of this state are described using set theory based on certain "given" sets: users, entities and devices, for example. The state components then consist of objects composed from the given sets: subsets, relations between them, functions and so on. The state also has certain invariants: predicates which are always true. Such a state can be expressed in a schema. For example, we may define a particularly simple database as below.

[E]

DB

$$
\begin{array}{|l}
\hline
\text{contents} \quad : \mathbb{F}\ E \\
\text{relationships} : E \leftrightarrow E \\
\hline
\text{dom relationships} \subseteq \text{contents} \\
\text{rng relationships} \subseteq \text{contents} \\
\hline
\end{array}
$$

First of all this introduces a set E. The structure of this set is not defined - it may be finite or infinite and we know nothing about its members. We are going to use it to represent the set of all possible entities. It is analogous to the Ada definition

```
type E is private;
```

Then we introduce a schema called DB. This can be thought of as representing the state of a database.

Following the schema name there are two parts to a schema. The first part, the signature part, defines the components, or objects, by giving the signature of each: that is, its name and type. The name can of course be anything; the type is expressed in terms of sets already known. So in this example the database has two components. The first one is called "contents" (we are using it to represent the set of all entities known about in the database), and it consists of a finite set of entities - that is, a finite subset of the set E of all possible entities. The second component in the example is called "relationships" and its type is a relation between entities. We can use it to represent the relationships between entities stored in the database. (Now we see how simple this model is in that it only allows a single relation between

two entities).

The second part of the schema is the predicate part: it contains a statement of facts which are true about the objects in the schema. In the case of a schema representing a state, this predicate is an "invariant" - it represents facts which are true about any instance of the state. In the example, two facts are true about the database. The first line of the predicate says that any entity which is related to another entity must be in contents: that is it must be known about in the database. The second line says the same for any entity which has other entities related to it.

The meaning of such a schema is important to the two classes of reader. The tool writer can absolutely rely on the fact that there are contents and relationships in the database, and that any entity which appears in "relationships" is also in "contents". The implementor is obliged to implement all operations including the initial state of the database in such a way that the invariant can never be violated. He/She is free, on the other hand, to choose ANY representation of contents and relationships that is considered suitable.

Having defined a state, usually by a schema, we can then specify the operations available in terms of their effects on the state. To do this we introduce a schema for the operation. This schema includes a schema for the state before the operation and a schema for the state after. (By convention unprimed names refer to the state before and primed names refer to the state after). The operation schema also includes signatures for any inputs and outputs of the operation. The predicate part of an operation schema states the semantics of the operation by stating things which are true about it. These may be preconditions - things which must be true before the operation is carried out; or postconditions - facts which define the results of the operation. Postconditions may simply be predicates about the final state (including

outputs) or may involve the relationship between the final state and the initial state and inputs.

For example, consider the operation to create a relationship:

create_RELATIONSHIP

```
DB
DB'

e1?, e2? : E
```

```
e1? ∈ contents
e2? ∈ contents
(e1?, e2?) ∉ relationships

contents' = contents
relationships' = relationships ∪ (e1?,e2?)
```

This states that creating a relationship takes an initial state of the database (DB) and turns it into a new state (DB'). It takes two input parameters, the first and second entity in the relationship to be created (e1?, e2?), where we use the convention that all input parameters are tagged with a question mark.

The first part of the predicate is the three preconditions for the operation: e1? and e2? must both already be known in the database, and the relationship between them must not already exist. The second part of the predicate defines the meaning of the operation: after it has been carried out "contents" in the new state is the same as it was in the old, while "relationships" is the same as before but with a new relationship between e1? and e2? added.

A more detailed introduction to the Z language has been developed in,[Bro86d] while a collection of case studies are given in.[Fli87]

7.2.1.2. Relationship with Implementation

To implement facilities which have been specified in Z, it is obviously necessary to give concrete form to the operations in some programming language. This is straightforwardly done by defining concrete types to represent all the parameters of the operations and providing a procedure or function for each operation. In some cases generic subprograms may be provided. To continue the simplified example, the concrete form in Ada of create_RELATIONSHIP might be:

```
package DB is
   type E is private;
     :
     :
   procedure CREATE_RELATIONSHIP ( E1, E2 : in E );
     :
```

The full Public Tool Interface for the Aspect system was specified in the language Z, and can be found in.[Rob87c] The role of this specification was as a formal communication tool for the project members and as a precise way of describing their ideas. It guided the subsequent implementation, but there was no formal refinement from specification to implementation.

We now give an overview of the formal specification of the view mechanism as the complete specification is rather lengthy and interwoven with the other parts of the definition. The objective of this is to give an overall flavour of the approach that has been taken. For a full understanding of the details of the specification refer to.[Rob87c]

7.2.2. Overview of the Formal Specification of the View Mechanism

There are a number of inter-related elements of the model, all of which must be specified. We begin the formal specification by defining the relationships between Abstract Environments (AE's) as a graph structure, controlling the relationships between defined views of the Information Base. We then continue by specifying how each AE in the graph is constructed and controlled. Finally we move on to define operations to both manipulate the graph structure, and the AE's which are the nodes of the graph. Of particular importance is the operation to evaluate an AE object, which performs the changes of state defined by that object.

It can be seen in the specification that separate operations are given for the cases when the object in an AE is a data item, and when it is an operator. Strictly, these two could have been coerced and manipulated with single operations, but for reasons of clarity within the specification, and because a user of an AE may indeed want to distinguish the two types of object, we have not done so.

The concept of a tool is also important. Through an AE a user has available a set of tools which he/she may invoke. We specify operations to register and invoke a tool, and to allow an existing tool to be made available to an AE.

7.2.2.1. Abstract Environments

The first thing we do is define the basic structure which controls the relationship between AE's. To allow complete flexibility, this is a connected acyclic graph structure (CAG). This implies that a node in the graph can have multiple parents, but that there is a distinguished node called the "root" from which all other nodes in the graph can be reached by traversing the parent/child arcs.

We distinguish the root of the graph as being the **base environment (BE)** from which all other AE's are derived, containing a set of base objects. Note that the structures and operators for defining and manipulating graphs and trees are specified separately in[Rob87c] and summarised in Appendix B. Within the specification of AE's we make use of these structures and operators.

So, we define the set of all possible AE's to be AB_ENV, and define a connected acyclic graph (CAG) whose nodes are elements of this set.

[AB_ENV]

AB_ENV_GRAPH

CAG [AB_ENV]	
base_env : AB_ENV	
known_AEs : $\mathbb{F}$ AB_ENV	
{base_env} = roots	a_env.1a
nodes $\subseteq$ known_AEs	a_env.1b

a_env.1a the single root of the graph is the base environment.

a_env.1b all the nodes of the graph come from the set of known AE's.

Once we have the basic structure, we define a function which maps each AE to the set of mappings which define the objects of the AE. To allow for the definition of operator objects in the AE, we say that a mapping relates an expression which describes the object, to another expression that is its definition. For example, a simple addition operator for integers could have a mapping of the form

+(a, b) <===> ...exp...

We will call the left-hand side expression the **declaration expression** and the right-hand side expression the **defining expression.**

We can see a mapping as similar to the definition of a "procedure" in a programming language such as Pascal. The declaration expression acts as the

procedure declaration, with the name of the procedure and the types of the operands to that procedure (where the formal arguments are acting as representatives of the types). The defining expression acts as the procedure body; a description of the operations performed by the procedure. Within the body of the procedure, use of the formal arguments will be made.

The predicates of the schema ensure that these mapping entries are of the correct form. In particular, the declaration expression for operator objects will be a two-level tree (operator name and parameters), while for data objects it will be a single object name only.

Note that the specification of expressions is to be found in a separate document.[Bro86a] In this we find the specification of the set of expressions, EXP, the elements of an expression, OBJ, and the subsets of expression elements which are operator objects, OP_OBJ, or leaf objects, LEAF_OBJ. The functions "e_op" and "e_leaf" map an expression element, OBJ, to an operator object, OP_OBJ, or leaf object, LEAF_OBJ, respectively. We make use of these specifications below.

MAPPING $\hat{=}$ EXP $\rightarrow\!\!\!\!\!+$ EXP

AB_ENV_GRAPH

AB_ENV_GRAPH

env_to_map : AB_ENV $\rightarrow\!\!\!\!\!+$ MAPPING

---

a_env.2a

dom (env_to_map) = nodes

a_env.2b

$\forall$ e1: EXP; o1: OBJ; ae:AB_ENV;
| e1 $\in$ dom(env_to_map(ae))
{o1} = e1.roots
• ((e1.nodes = e1.roots $\cup$ e1.leaves)
$\wedge$ (e_op(o1) $\in$ OP_OBJ))
$\vee$
((e1.nodes = {o1}) $\wedge$ (e_leaf(o1) $\in$ LEAF_OBJ))

a_env.2a every node has an entry in env_to_map - including the base environment.

a_env.2b the left-hand side of mapping entries consist of a root node and children for operators, or a root node only, for data objects.

We now specify some additional functions which are auxiliary to the model, and are used to help with the predicates that are needed to constrain the model. These auxiliary functions are :

1. env_to_objs. For each environment, this gives the set of objects which are available to the environment user (ie. have been declared in the environment). These will be the roots of the declaration expressions for objects.

2. env_to_uses. For each environment, this gives the set of objects which are used in defining the environment objects.

In addition, we need a function that maps some objects to tools, as later in the specification we need to determine which objects are associated with a tool, and be able to invoke that tool. The set of all possible tools in an ASPECT system is the set TOOLS, defined in another section of the full ASPECT Specification.

AB_ENV_GRAPH

AB_ENV_GRAPH

env_to_objs : AB_ENV ⇸ 𝔽 OBJ
env_to_uses : AB_ENV ⇸ 𝔽 OBJ

tool_map : EXP ⇸ TOOLS

base_objs : 𝔽 OBJ

---

a_env.3a

dom(env_to_uses) = nodes

a_env.3b

dom(env_to_objs) = nodes

a_env.3c

∀ ae1, ae2 : AB_ENV; e1,e2 : EXP
| e1 ∈ dom(env_to_map(ae1))
e1 ∈ dom(env_to_map(ae2))
• (e1,e2) ∈ env_to_map(ae1)
⇒ ((e1 = e2) ∧
(ae1,ae2) ∈ parent)

∀ ae: AB_ENV; es: 𝔽 EXP
| ae ∈ nodes ∧
es = dom(env_to_map(ae)) •

a_env.3d

env_to_objs(ae) = { o1:OBJ; ex:EXP
| ex ∈ es ∧
{o1} = ex.roots
• o1 }

a_env.3e

env_to_uses(ae) = { e:OBJ; ex:EXP
| ex ∈ rng(env_to_map(ae)) ∧
e ∈ ex.nodes • e }

a_env.3a all environments use objects in their definition.

a_env.3b all environments have an associated set of objects.

a_env.3c the same expression appears on the lhs of two AE's only if we are inheriting that expression from one of its parent AE's. In such cases, the mapping is to itself.

a_env.3d the objects defined for each environment are those for which a mapping exists.

a_env.3e the objects used by an environment consist of those objects that have been used in object definitions.

7.2.2.2. Initialising the model

The first thing to do is to define the initial state of the model. This will be when there is a base environment and base objects only, and no other environments have yet been defined. Note that the set ET represents the set of all possible entity types in an ASPECT system, and the set "entity_types" are those that exist in an initial ASPECT system.

init_AB_ENV_GRAPH

Φ AB_ENV_GRAPH

---

init.1

env_to_objs' = { (base_env ↦ base_objs) }

init.2

∀ m : MAPPING; e : EXP
| (e,e) ∈ m
e.roots ⊆ base_objs
• env_to_map' = { (base_env ↦ m) }
env_to_uses' = { (base_env ↦ base_obj) }

init.3

parent' = {}
nodes' = {base_env}
known_AEs = base_env

init.4

obj_returns' = obj_returns ⊕
{ s:ℙ LEAF_OBJ; b:OBJ; l:LEAF_OBJ
| b ∈ base_obj
e_leaf(b) = l
l ∈ s
• (b ↦ s) }

op_expects' = op_expects ⊕
{ s:seq ℙ LEAF_OBJ; b:OBJ;
| b ∈ base_obj
b ∈ dom(e_op)
• (b ↦ s) }

init.5

data_obj_map' = { o:OBJ; ets:ℙ ET | o ∉ dom(e_op)
o ∈ base_objs
ets = {et}
et ∈ entity_types
• (o ↦ ets) }

| entity_types | = | data_obj_map' |

init.1 the initial set of objects is the set "base_obj".

init.2 the initial set of mappings is for the base environment, which has all the base objects mapping to themselves. The initial uses set is the base_objs.

init.3 the AE graph is initialised to the base_env only.

init.4 all base objects have entries in "obj_returns" giving the type of object they return, and all operators have an entry in "op_expects" giving the types of the parameters expected by the operator.

init.5 the function data_obj_map records the mapping between all view objects and the entity types from which they are eventually derived. The initial state of data_obj_map has all base data objects mapping to a known ET. All known ET's will have entries in the function.

7.2.2.3. Operations on the Model

The basic operations on the model are specified in order to be able to manipulate the graph of environments. For example, this includes defining a new environment, and adding a new object to an environment. Those specified form the minimum set of operations required to manipulate the model.

The full set of operators on the model allow new AE's to be created and deleted, data and operator objects to be added to, and removed from an AE, and for the definition of a data or operator object to be amended. Here, we only give the operations to add an object to an AE, and to create a new AE as these provide the models on which the other specifications have been based. The full set of operators are specified in,[Rob87c] and are informally

described in terms of their implementation in chapter 8.

7.2.2.3.1. Adding an Object to an Environment

We have two separate add operators, one for operator objects, and the other for data objects. This is necessary because we need to record extra information, and do a little more work when we add operator objects than for the others.

Data Objects

Given an environment, an expression to define the object, and the type returned by the object, then the object is added to the environment, and the left-hand mapping expression returned.

add_DATA_OBJ

Φ AB_ENV_GRAPH

env? : AB_ENV

exp? : EXP

ret? : $\mathbb{P}$ LEAF_OBJ

ret_exp! : EXP

---

add_D.1a

env? $\in$ nodes - roots

add_D.1b

ret_exp!.roots $\cap$ $\bigcup$ env_to_objs (| nodes |) = {}

add_D.1c

exp?.nodes $\subseteq$ $\bigcup$ (env_to_objs (|parent(env?)|))

add_D.1d

env_to_map' = env_to_map $\oplus$
{(env? $\mapsto$ (env_to_map (env?) $\oplus$
{(ret_exp! $\mapsto$ exp?)}))}

add_D.1e

$\forall$ r1: OBJ
| ret_exp!.roots = {r1}
• obj_returns' = obj_returns $\oplus$ {(r1 $\mapsto$ ret?)}

add_D.1f

$\forall$ r1: OBJ; s : $\mathbb{F}$ ET;
| ret_exp!.roots = {r1}
s = $\bigcup$(data_obj_map(| exp?.nodes |))
• data_obj_map' = data_obj_map $\oplus$ {(r1 $\mapsto$ s)}

add_D.1a the given environment is one of the environments in the graph (not including the root).

add_D.1b the object to be added is not already in an environment.

add_D.1c all objects in the expression to be added must come from the parent environments.

add_D.1d the definition of the new object is added to the mapping table.

add_D.1e the new data object is added to "obj_returns".

add_D.1f the entry in data_obj_map is from the new data object to the collection of ET's held for each of the objects in the defining expression.

Operator Objects

Given an environment, an expression to define the operator, the type of object returned by the operator, and the type of the operator's parameters, then the object is added to the environment, and the left-hand mapping expression is returned.

add_OP_OBJ

Φ AB_ENV_GRAPH

env? : AB_ENV

exp? : EXP

ret? : $\mathbb{P}$ LEAF_OBJ

params? : seq $\mathbb{P}$ LEAF_OBJ

ret_exp! : EXP

---

add_O.1a

env? $\in$ nodes - roots

add_O.1b

ret_exp!.roots $\cap$ $\bigcup$ env_to_objs (| nodes |) = {}

add_O.1c

exp?.nodes $\subseteq$ $\bigcup$(env_to_objs (|parent(env?)|))

add_O.1d

env_to_map' = env_to_map $\oplus$
{(env? $\mapsto$ (env_to_map (env?) $\oplus$
{(ret_exp! $\mapsto$ exp?)}))}

add_O.1e

$\forall$ r: OBJ | {r} = ret_exp!.roots •
obj_returns' = obj_returns $\oplus$ {(r $\mapsto$ ret?)}

op_expects' = op_expects $\oplus$ {(e_op(r) $\mapsto$ params?)}

add_O.1a the given environment is one of the environments in the graph (not including the root).

add_O.1b the object to be added is not already in an environment.

add_O.1c all objects in the expression to be added must come from the parent environments.

add_O.1d the definition of the new object is added to the mapping table.

add_O.1e the type of the returned object, and of the parameters, are recorded in "obj_returns" and op_expects".

7.2.2.3.2. Create a New Environment

There are two steps to creating a new environment. First of all, we add a new environment to the set of known AE's, then, given the new environment, and the parent environments, we must put this new environment in the graph with a link to each of the parents.

create_AB_ENV

Φ AB_ENV_GRAPH

new_env? : AB_ENV

---

create.1

new_env? $\notin$ known_AEs

create.2

known_AEs' = known_AEs $\cup$ new_env?

create.1 the new AE is not in the known set of AE's.

create.2 add the new AE to the known set of AE'S.

link_AB_ENV

Φ AB_ENV_GRAPH

p_env? : $\mathbb{F}$ AB_ENV

c_env? : AB_ENV

---

link.1

p_env? $\subseteq$ nodes

link.2

(c_env? $\notin$ nodes) $\wedge$ (c_env? $\in$ known_AEs)
$\vee$ (c_env? $\in$ nodes)

link.3

parent' = parent $\cup$ {p: AB_ENV | p $\in$ p_env?
$\bullet$ (c_env? $\mapsto$ p)}
nodes' = nodes $\cup$ {c_env?}
(c_env? $\notin$ nodes) $\wedge$ (env_to_map' = env_to_map
$\oplus$ {(c_env? $\mapsto$ {})})

link.1 the parent environments already exist in the graph.

link.2 the new environment does not exist in the graph, but is a known AE; or it is an existing node on the graph.

link.3 add a new node to the graph in the correct place, and make a new null entry in the env_to_map function. Note that if the node already exists in the graph, then this operator allows us to add new links to other parents.

7.2.2.4. Evaluating an Object

The process of evaluating an AE object is the response that occurs when a user or tool within an AE asks for the AE operator to be invoked, or the AE data object to be materialised. Whenever this happens, we must go through two distinct steps:

- a compilation step. The AE object is defined as an expression involving other AE objects. We must map this expression to an equivalent one involving base environment objects only. Remember, the base objects are the only ones that we can invoke to carry out some action, or are the only "real" pieces of data that exist in the system.
- an invocation step. Once we have an expression involving base objects, we must then obtain from this expression the sequence of operations that we are to perform and actually carry them out, executing the state changes that they imply.

In the specification we make these steps explicit, combining them in the final operation.

It can be seen from the full specification[Rob87c] that the Z specification for this operation is complicated and difficult. However, we can summarise what the Z is specifying in simple terms with the following informal description.

Evaluating an AE object consists of the following three operations:

1. compile_AE_OBJ. Takes an expression defining an AE object within any AE, and returns an equivalent expression which contains base objects only.

2. map_to_OPS. Takes an expression involving base objects only, and returns an equivalent sequence of PTI operations that can be invoked (a tool execution), or a single PTI operation that can be called.

3. execute_OP. Takes a tool or single PTI operation as input, and performs the state changes that are implied, recording the individual operations in the execution tree. (The operations to "invoke" a tool, and to "call" a PTI operation, are defined in the "Activities" section of the full ASPECT Specification).

So, we combine the steps of compilation, mapping to operations, and execution to form the complete evaluate operation. This is the operation that will be available at the PTI. Note that the operation " ; ≫ " is described as forward relational composition with piping, and has the effect of both of the individual operations.

evaluate_AE_OBJ ≙ compile_AE_OBJ
; ≫ map_to_OPS
; ≫ execute_OP

7.2.2.5. Tools and AE Objects

When a user is working within an AE, he/she has available a set of objects which define the operations and data that can be accessed. However, we want the AE to be extensible in the sense that we allow a user to add new objects to their AE, which we call "tool objects". In many ways this is similar to the way in which the original objects have been added to the AE, but with a number of differences. Firstly, it is not possible for a user to use the add operators for objects, as these require the user to know about objects in the parent AE's as well as their own. We want the new object to be a combination of existing objects in the AE. Secondly, the step of evaluation which is necessary for the primitive operators of an AE will not be used for tools. We can think of tools as being compiled once, and then for invocation direct access can be made to this compiled version. Finally, the way in which we record tool invocation is different from the way in which primitive operators are recorded. Tools are tracked separately from PTI calls, so that, for example, recovery can be handled differently.

So, the first operator we need is to allow us to create a new tool object that is accessible within the AE in which it is defined. This operation is referred to as "registering a tool". As part of this operator we need to associate the new tool object with a member of the set TOOLS, which represents all possible tools. We are not able to say which particular tool is associated with the tool object, only that there must be such an association.

register_TOOL_OBJ

Φ AB_ENV_GRAPH

env? : AB_ENV

tool? : EXP

ret? : $\mathbb{P}$ LEAF_OBJ

params? : seq $\mathbb{P}$ LEAF_OBJ

ret_exp! : EXP

---

reg.1

env? $\in$ nodes

reg.2

ret_exp!.root $\cap$ $\bigcup$ env_to_objs $(\!|$ nodes $|\!)$ = {}

reg.3

tool?.nodes $\subseteq$ env_to_objs(env?)

reg.4

env_to_map' = env_to_map $\oplus$
{(env? $\mapsto$ (env_to_map (env?) $\oplus$
{(ret_exp! $\mapsto$ tool?)}))}

reg.5

$\forall$ r: OBJ | {r} = ret_exp!.roots •
obj_returns' = obj_returns $\oplus$ {(r $\mapsto$ ret?)}

op_expects' = op_expects $\oplus$ {(e_op(r) $\mapsto$ params?)}

reg.6

$\exists$ t : TOOLS
• tool_map' = tool_map $\oplus$ {(tool? $\mapsto$ t)}

reg.1 the given environment is one of the environments in the graph.

reg.2 the object to be added is not already in an environment.

reg.3 all objects in the tool object to be added must come from the current AE.

reg.4 the definition of the new object is added to the mapping table.

reg.5 the type of the returned object, and of the parameters, are recorded in "obj_returns" and op_expects".

reg.6 the tool object is associated with an element of the TOOLS set.

7.2.2.5.1. Invoking a tool

When a user wishes to invoke a tool object from within an AE, we must first check that the tool is accessible from the AE, and then we can find the instance of the TOOLS set associated with the object and call the operator "invoke_tool" which performs the state change expected of the tool, and records that the execution of the tool has taken place. The operation "invoke_tool" is specified as part of the "Activities" section of the full ASPECT Specification.

check_TOOL_OBJ

Φ AB_ENV_GRAPH	
ae_name? : AB_ENV	
tool_obj? : EXP	
tool! : TOOLS	
tool_obj? ∈ dom(env_to_map(ae_name?))	chk_tool.1
tool! = tool_map(tool_obj?)	chk_tool.2

chk_tool.1 the tool object must be in the given AE.

chk_tool.1 return the tool associated with the tool object.

We now specify "invoke_TOOL_OBJ" as the composition of the "check_TOOL" and "invoke_tool" operators.

$$invoke\_TOOL\_OBJ \hat{=} check\_TOOL \; ; \gg invoke\_tool$$

7.2.2.6. Publishing AE Objects

When a user is working within an ASPECT system, that user will have available a set of data objects, and a set of operators which manipulate instances of those objects. The view mechanism is responsible for defining the objects and operators available, while the instances of the objects that can be accessed are determined by the ASPECT notions of ownership and Publication.

Because an ASPECT system will be used by many classes of user, each with different AE's, we want to enable users to be able to share information between AE's. Broadly speaking, we could allow a user of one AE to permit users in other AE's to:

- see data objects that they have available.
- execute operators and invoke tools that they can use.
- see particular instances of data objects that are common to both AE's.

The first two of these options involve adding new objects to the receiving AE which define the data objects, operators, or tools that are now available, while the third will mean that underlying instances of entity types will be published from one AE to another. It is expected that the latter operation will be carried out much more frequently than the former ones.

Underlying the notion of AE's, ASPECT has the concept of a Domain as a mechanism beneath the ASPECT PTI to control fine-grained data sharing between different users. Operators are provided to Publish entities between Domains. However, as we do not want users to be aware of the existence of Domains at the PTI level, we re-cast the operators provided for Publication between Domains as operators which publish AE object instances between AE's.

To allow us to specify the Publishing of instances of AE objects, we define an operation called "ae_to_dom_map1" that takes the object instance,

its type, and the AE's which are involved, and returns the corresponding entity and Domain to call the lower-level "publish_ENTITY" operator. We also specify a very similar operation called "ae_to_dom_map2" that takes the instance, its type, and the single AE involved, and returns the entity and Domain to pass to "acquire_ENTITY", "withdraw_ENTITY", and "dispose_ENTITY" operators.

We now define the AE publication operators as the composition of the mapping operators with the lower-level publication operators that operate on a single entity between two Domains. The new operations will be the ones that are available at the ASPECT PTI.

Note that we do not need any additional predicates on these operations as these have been specified at the lower-level, and will be checked there. For example, "acquire_ENTITY" checks that the Domain doing the acquiring has previously had that entity published to it.

publish_AE_ENT $\hat{=}$ ae_to_dom_map1 ; $\gg$ publish_ENTITY

acquire_AE_ENT $\hat{=}$ ae_to_dom_map2 ; $\gg$ acquire_ENTITY

withdraw_AE_ENT $\hat{=}$ ae_to_dom_map2 ; $\gg$ withdraw_ENTITY

dispose_AE_ENT $\hat{=}$ ae_to_dom_map2 ; $\gg$ dispose_ENTITY

7.3. Summary

In this chapter we have given an implementation independent description of the view mechanism for ASPECT by developing the model using the formal specification language Z. We now discuss a particular implementation based on this specification, before analysing how the mechanism integrates within the wider context of facilities provided in the ASPECT prototype.

CHAPTER 8

IMPLEMENTATION OF AN IPSE VIEW MECHANISM

8.1. Introduction

Although the above specification has defined an IPSE views mechanism independent from consideration of other IPSE facilities, for implementation of the mechanism it is vital that it is seen as a component of a larger system, and fits in with that system's needs.

From this, and earlier discussion of the expectations of a view mechanism, we can define the following aims of the implementation:

- to provide a set of primitives at the ASPECT Public Tool Interface (PTI) which allow Abstract Environments (AE's) to be created, added to, manipulated, and so on.
- to control individual users' access to data, operators, and tools, and hence aid integrity of the system.
- to allow tailored interfaces for both users and tools to access the IPSE facilities at an abstract level suited to their particular needs.
- to integrate these facilities within the wider ASPECT concepts for control and co-ordination of software development in a large project.

8.2. Constraints on the Implementation

As part of a larger project, the implementation of the IPSE view mechanism was built on, and integrated with, a number of existing software systems. A simplified implementation architecture is given in figure 8.1.

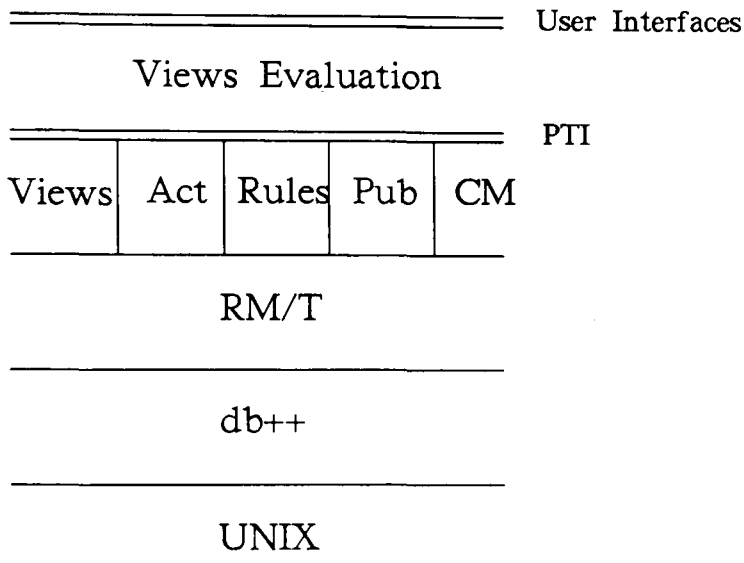

Fig 8.1 A Simplified Implementation Architecture.

At the lowest level of the implementation is the UNIX operating system, running on a SUN workstation. On top of this we use a commercially available Database Management System (DBMS) called db++[Con84] as the internal, or physical representation of the conceptual model of the ASPECT Information Base. The reasons for choosing this DBMS as opposed to the many others available were mainly pragmatic: it is specifically written to interface well with both the UNIX operating system environment, and the C programming language in which it is implemented. (The decision for us to use the C

programming language to implement the ASPECT prototype was taken early on in the project). However, it is also a fairly pure implementation of the Relational Model, accessible through a well-defined relational algebra, and this too makes it an attractive system for our needs.

Above this, the conceptual level of the Information Base, the Extended Relational Model (RM/T), was implemented. This is often referred to within ASPECT as the Information Base Engine (IBE). As discussed earlier, this provides not only an extended algebra of operations to the services built above it, but also provides enhanced semantic integrity through the entity interface that it supports. As far as we are aware, this is one of the first major implementations of RM/T that has been attempted.

The facilities and services provided above the RM/T interface, collectively known as the Information Base Superstructure (IBS), provide a set of primitive operators which together make up the ASPECT Public Tool Interface (PTI). This is the lowest level at which any user can gain access to the services provided by an ASPECT IPSE. Therefore, part of the IBS will be a set of operators which allow users to create and manipulate views of the PTI, as well as for Activity management, rule creation and enforcement, sharing of information, and version identification and control. A major part of the view mechanism implementation is to provide the relevant primitive operators at the PTI.

However, the view mechanism differs from the other IBS components in that all interaction with a user to an ASPECT Information Base must take place through a view. Therefore, a separate component of the view mechanism is to provide, above the level of the PTI, the necessary facilities to accept a user request to perform an operation on some data items, and to not only validate that the operation requested, and the data to be accessed are available to the user through the particular view of the ASPECT system which he/she

has, but also to then translate the request to a series of lower level PTI operations which when executed will implement the request. This is a further important component of the view mechanism implementation.

8.3. Details of the Implementation

We now look in detail at some of the issues which arose during the implementation of the view mechanism.

8.3.1. Underlying Structures

In providing a set of operators as part of the ASPECT PTI, structures must be defined within the IBE to contain information that can be queried and manipulated when these operations take place. These will, necessarily, be in the form of entities and entity types, and be manipulated using the RM/T operators provided by the RM/T implementation. A representation of the entity types used to contain view mechanism information is given as figure 8.2.

In figure 8.2 the boxes represent entity types, each of which is given a name. Also within the box is a letter to represent the classification type of the entity type. This will be one of kernel (K), associative (A), or characteristic (C), and may be designative (D) in addition. Where one box is contained within another box, the entity type represented by the inner box is a subtype of the entity type represented by the outer box. In fact, a box enclosing the whole diagram has been omitted for clarity; this would represent the fact that all entity types in an ASPECT system are subtypes of a root entity type. To represent the relationships between types, an arc is

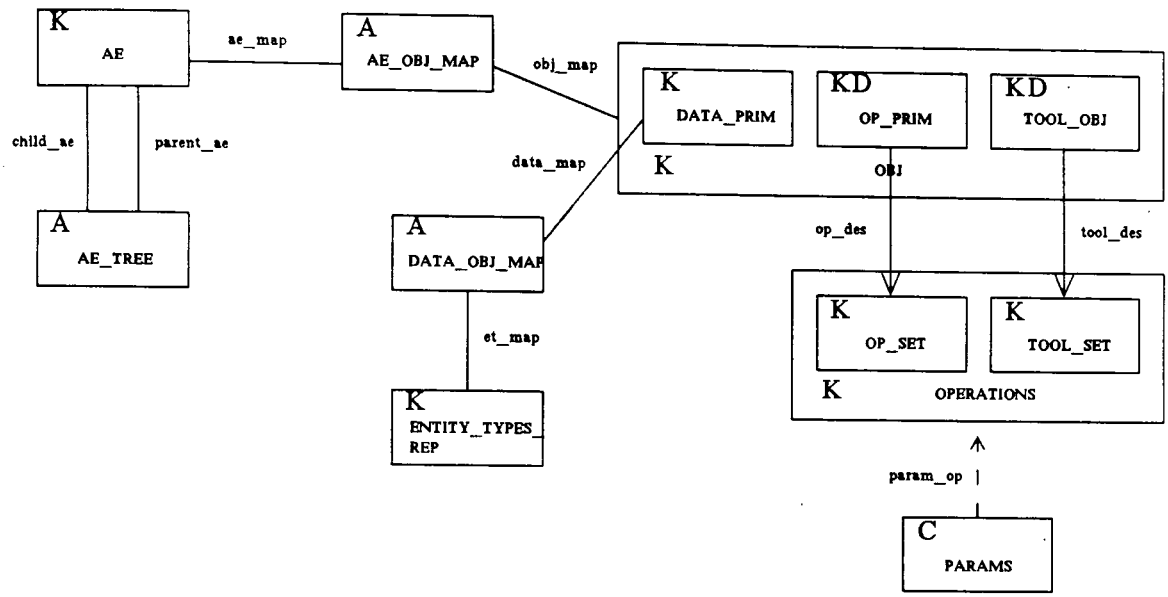

Fig 8.2 Entity Types Used to Represent View Mechanism Information.

drawn between entity types which are associated, and a name is given to that associative attribute. Similarly, a solid arrow is drawn between a designative entity type and its designated type, with the name of the designation labelling the arrow. The dashed arrow represents a characteristic link between two entity types.

Interpreting the contents of the diagram, we can say that all Abstract Environments (AE's) are represented as entities in the entity type "AE", and are related in a graph structure through the entity type "AE_TREE". Each AE is associated through "AE_OBJ_MAP" with a set of view objects held in "OBJ". The view objects are classified as data, operator, and tool objects

through the subtypes "DATA_PRIM", "OP_PRIM", and "TOOL_OBJ" respectively. Data objects are further associated through "DATA_OBJ_MAP" to the particular entity types from which they are eventually derived. Operator and tool objects, on the other hand, designate an "OPERATION" entity through the subtypes "OP_SET" and "TOOL_SET", and each operation is characterised by a set of parameters held in "PARAMS".

In figure 8.3, each of the entity types is given with the properties associated with that type, where the notation

A[p1(d1), p2(d2)]

defines an entity type "A" whose instances have properties "p1" and "p2" drawn from the domains "d1" and "d2" respectively. Each property is named, and the domain for values of the property is given, where "etype" is the domain of all possible surrogate values. In addition to the properties shown here, every entity type has a property known as the "e-attribute". For each entity instance this hold the unique surrogate value for the entity drawn from the etype domain. Note that any entity can be associated with a name within a particular namespace through a separate naming scheme.

8.3.2. Expressions

An important concept in the previous discussions about an IPSE view mechanism, and within the specification, is that of an expression. Even though we have formally specified in Z what is meant by an expression, with a particular refinement to specify the special case of RM/T expressions, it is at an abstract level such that a number of possibilities exist with regard to its implementation. Here we discuss some of the possibilities, and examine

```
AE[ ae_lib(file) ]
AE_TREE[ child_ae(etype), parent_ae(etype) ]
AE_OBJ_MAP[ ae_map(etype), obj_map(etype) ]
OBJ[ defn_src(file), defn_exec(file), ret_type(string) ]
DATA_PRIM[ ]
OP_PRIM[ op_des(etype) ]
TOOL_OBJ[ tool_des(etype) ]
OPERATIONS[ ]
OP_SET[ ]
TOOL_SET[ ]
PARAMS[ position(int), param_type(string), param_op(etype) ]
DATA_OBJ_MAP[ data_map(etype), et_map(etype) ]
```

Fig 8.3 View Mechanism Entity types with their Properties.

the consequences of the decisions made.

8.3.2.1. First Prototype

A view mapping is specified to be a function from one expression to another. The left-hand mapping expression defining the view object, while the right-hand mapping expression describes the lower level operations which are executed to implement the view object. However, in the specification of expressions we specify a general expression to consist of a set of data and a set of operator objects related in a graph, without saying any more about what those objects might be. The decision to leave the specification at such an abstract level was made to ensure that a single specification could be made use of throughout the ASPECT project. A particular refinement of the

specification is made for the special case where we have an expression in the extended relational algebra of RM/T.

For a preliminary prototype implementation of the view mechanism, it was decided to interpret an expression as equivalent to an RM/T expression, and to investigate the advantages and limitations of this approach. These are summarised below:

- a view object was defined using the extended relational operators of RM/T. This provided all the operators of the relational model (select, project, join, and so on), together with some additional operators (close, apply, ptuple, and so on).[Cod79]
- the relational operators implemented formed a closed algebra (ie. the result of any relational operation is itself a relation), and hence the only objects that could be defined in a view were data objects of type relation.
- as data objects only could be defined, an AE was defined to consist of those data objects specifically assigned to the AE, together with the full set of PTI query operators which we assumed were inherited by each AE.
- the describing expression for a view object was read in as a text string, parsed to check its validity, and recorded in string form ready for evaluation when required.
- evaluation of a view object in this context had an obvious interpretation. The describing expression associated with the view object was mapped to the lower level objects by the act of textual substitution of the describing expressions of each of the constituent components. For example, if we define "A" to have the describing expression "+(B,C)", where this is the expression with root operator "+" and leaves "B" and "C", and "B" and "C" themselves have describing expressions "-(X,Y)"

and "*(X,Y)" respectively, then evaluation of the view object "A" will lead, via textual substitution, to the expression "+(-(X,Y),*(X,Y))". This process can be recursively applied to obtain an expression involving base objects and operators only, which can then be executed.

- it is not obvious how this implementation of an expression can be easily extended to allow both data objects of different types, and operator objects, to be defined as part of an AE.

8.3.2.2. Second Prototype

Aware of the limitations of the first prototype implementation, in particular the inability to define operator objects as components of an AE, it was decided that the second prototype implementation should take a much wider view of an expression. Indeed, it was recognised that the only way to obtain the expressive power required was to use the RM/T algebra embedded within a programming language.[Hit76] In this way, not only are we able to access the data restructuring primitives which are vital to the success of the mechanism for data objects, but we can also make use of the comprehensive control facilities available in a programming language, together with the ability to combine the update operations for data with the arithmetic operators of a programming language.

For example, if we wanted to define an operator which updated the salary property of an entity by 10% of its current value, then we need the arithmetic operators, and the control and sequencing operations of a programming language to express the operation, together with the update operators of RM/T. In the first attempt, not only were we unable to access the update operators (as they are not considered to be part of the extended relational algebra), but we were also unable to perform arithmetic operations on data

values, nor to carry out sequences of operations.

The decision made was to use the C programming language, and allow the describing expressions of a view object to be equivalent to a C language function. This seemed the most appropriate choice given that all other implementation work was to use this language, and that the RM/T operations were specifically written as a C library of functions to be linked in with other C routines.

We now discuss the details of how each of the operators at the PTI for the view mechanism were implemented, and then deal with some of the problems that we encountered, some of which are directly related to the decisions made concerning expressions.

8.3.3. Operators at the ASPECT PTI

The operators for the view mechanism which are available at the ASPECT PTI have been implemented as a library of C programming language functions which a user of the PTI, an ASPECT tool writer, would link in to a main program which he/she was developing. This would make available all the operators implemented.

Although a summary of the specification in Z of the operators available at the PTI has been given in the previous chapter, here we briefly describe each of the primitive operators implemented with particular emphasis on their implementation, the decisions and problems involved, and their relationship to the formal specification.

Note that the full set of operators have not been implemented for this prototype. It was felt that the functionality of the mechanism would not be impaired by omitting certain secondary operations which are required for completeness of the model, but which otherwise serve no research purpose. In

particular, these are the operators to delete an AE, remove an object from an AE, and to modify an AE object.

8.3.3.1. A Note on Error Handling

Within each of the operators offered at the ASPECT PTI, the constraints and integrity checks defined in the specification are implemented as part of the operator. Therefore, when an operator is executed, if any of these checks fail, then the operator returns a null value to indicate failure, and the predicate in the specification which has been invalidated is recorded within an error variable. A user of the operators can now examine this variable and can relate the error precisely to some point in the specification document. This direct correlation between implementation and specification will clearly be a major benefit to the system's use, and is a prime example of the advantages of formally specifying software before implementation.

8.3.3.2. Creating and Linking AE's

The basic framework for the creation and control of an AE graph was simple enough to implement; each AE is represented as an instance of the entity type "AE", and can then be linked in the AE graph to existing AE's using the "link_ae_obj" operator.

8.3.3.2.1. create_ab_env

Creates a new AE, and returns the surrogate of the new AE. A single entity of type "AE" is created.

8.3.3.2.2. link_ab_env

Given a set of parent AE's which already exist in the AE graph, a link is made between these parents and the given child AE. For each link, an instance in the "AE_TREE" entity type is created.

8.3.3.3. Adding Objects to an AE

To add an object to an AE, a set of operators are available. The choice of which one to use depends on whether the object is a data, operator, or tool object, and on whether it is a new object or an existing one. In all cases we associate an object entity with an AE entity, but for new objects the object entity must first be created.

8.3.3.3.1. add_data_obj

Adds a new data object definition to an existing AE. The describing expression of the object is contained in a file as the compiled form of a C function implementing the expression. A new surrogate is generated to represent the defining expression for the new data object.

To assist in type checking, and as a display aid, the return type of the object, and a list of the lower level objects used in the describing expression, are given as text strings. Each of the lower level objects used in the describing expression is then checked to ensure that it is a valid object, and it

is an object available in a parent of the AE to which this object is being added.

8.3.3.3.2. add_op_obj

This is similar to the previous operator in that a compiled C function implements the operator, but an additional set of parameters must be given which specify the parameters to the operator. This is no more than a set of text strings containing the types of the parameters.

In addition to making the association between an AE and the new object, a new operator entity is created, and a set of parameter entities characterise that operator. The new object then designates the operator entity.

8.3.3.3.3. register_tool_obj

This is similar to the previous operator, except that all objects used in the describing expression must come from the AE in which the object is being made available, not from its parents.

8.3.3.3.4. inherit_ae_obj

If a view object already exists, and is associated with an existing AE, then a child of that AE can inherit the object using this operator. On giving the operator to be inherited, a check is made that this object is available to a parent of this AE, and if so a new association to this AE is then made.

8.3.3.3.5. inherit_tool_obj

A tool object can also be inherited in an analogous way to the previous operator.

8.3.3.4. Displaying Objects in an AE

To enable users to determine what is accessible within an AE, display operators are available to show them. A separate operator exists for data and operator objects.

8.3.3.4.1. show_data_objs

For a given AE, a list of surrogates of data objects available through that AE is returned, together with the types of each object. This is implemented by querying the view mechanism structures to find which objects are associated with an AE, see which ones are data objects, and to return them with their types.

8.3.3.4.2. show_op_objs

This is similar in all respects to the previous operator, but in addition the parameters expected by a view operator object are also returned.

8.3.3.4.3. show_tool_objs

As in the previous operator, all tool objects of an AE can be displayed together with the expected types of parameters.

8.4. Accessing ASPECT Through a View

A second component of the ASPECT view mechanism, over and above the set of operators provided at the ASPECT PTI, must deal with the interactions between users and view objects.

When a user accesses the ASPECT services, it must always be to carry out a given Activity. In logging into this Activity, the user is then given a particular view of the ASPECT services through an AE which provides the set of data, operator, and tool objects which enable the user to perform the task in hand. Through the view mechanism, we must ensure that a user working within an AE is only able to access the facilities through that AE, and that when a view operator is executed, for example, then the correct sequence of lower level operations take place to implement the operation.

8.4.1. Providing the Correct Objects

The problem of ensuring that the users of an AE are only given access to the specified objects for the AE can be handled in a number of ways. One of the most obvious solutions is to insist that a parameter of every view object is the AE from which the object is being accessed. Then, a command shell around all operations can accept a request to perform an operation on certain data items, and can immediately check that the given AE has access to both the operator and data objects specified. The identifier of the AE need

not be explicitly given for each operation, but can be bound into a users environment on logging in to the Activity (in UNIX systems through "environment variables").

So, the front line to any user command is a simple command shell which accepts the requested operation, determines which AE is making the request, and queries the view mechanism structures in the information base to ensure that the operator and data items accessed are available to that AE.

8.4.2. Evaluating View Objects

In the specification of the view mechanism, and in the preliminary prototype implementation discussed earlier, the process of evaluating a view object was described as an interpretive process; when a request for a view object is made, either for a data object or execution of an operator object (we deal with tool objects later), then the describing expression for the object was found, and evaluated by replacing all the objects within that expression by their describing expressions, and repeating this process recursively until an expression involving base objects only remains. This base object expression can then be directly executed using the operators and data objects of the ASPECT PTI.

However, one of the consequences of the decision to implement expressions as RM/T operators embedded within a compiled form of C language function, is that we now have view objects as pre-compiled units which can be directly executed by instantiating the function with suitable parameters. This approach has obvious performance advantages, as the time taken to evaluate a view object is much less for a compiled object against an interpreted one, but has the disadvantage that changes in lower level view objects are not automatically reflected in higher level objects derived from them.

The higher level objects must be re-compiled. As a result, we lose some of the data independence properties which we found attractive in this mechanism.

For this prototype implementation, though, we accept this shortcoming, but note that if we were to implement view objects as RM/T operations within an interpreted language such as Lisp or Prolog, then we may well alleviate these problems. Similarly, we could devise a scheme which determines which objects are affected when another object is amended, and automatically re-compile such objects. This has not been attempted for this implementation.

8.4.3. Invoking Tool Objects

The process of invoking a tool object is considered separately from evaluating data or operator objects for two main reasons.

Firstly, the concept of a tool differs from other view objects in a number of ways. In scope, for example, it differs because a user can register a tool object in an AE in which they are currently working, but cannot add new data or operator objects to that AE. Tools also differ in their construction, as it is expected that they will be written in a programming language as a separate program, and then compiled. It is this compiled program object that can be inherited by child AE's.

The second way is through the Activities mechanism which controls and co-ordinates operations on the information base. Within the Activity mechanism, tool invocations are tracked as recoverable atomic operations. Hence, it should be possible to abort the invocation of a tool and the system will automatically roll-back changes that have been made by that tool invocation. View operator objects, on the other hand, are treated as atomic (they either complete or do nothing) but are not recorded by the Activities

mechanism.

Hence, a separate operator available in an AE will be to invoke a tool object. This will, first of all, check that the tool object invoked is accessible through this AE, before calling an operator implemented as part of the Activities mechanism which records the start of the tool as a node in an execution tree, and forks a new process (in the UNIX sense) to invoke the tool.

8.5. Physical Characteristics of the Implementation

As a prototype for further experimentation and analysis, the implementation of the view mechanism described above has been developed with the aim of assessing the functionality of the mechanism without particular regard for performance issues. However, for completeness, it is appropriate at this point to make a brief comment on the physical characteristics of the view mechanism implementation.

The software was developed on a SUN Workstation running UNIX 4.2 BSD, and the view operators were written in the C language as a set of C functions maintained in a library. The current version of this library occupies 75K bytes of memory.

Due to the layered architecture of the ASPECT system shown in figure 8.1, the performance of the view operators is dependent on the underlying RM/T implementation, which itself has been developed as a test vehicle for ideas rather than for high performance. As a result, execution times for the view operators range from 3 seconds elapsed time to create a new AE, to over 30 seconds elapsed time to show the operators in an AE. The issue of performance is discussed further in the "Future Work" section of chapter 10.

8.6. Summary

In this chapter, we have taken the specification of a view mechanism given in chapter 7, and refined this to an implementation integrated with the other facilities provided by the ASPECT IPSE. The main emphasis of this discussion has been the implementation problems that existed, the choices made to resolve these problems, and analysis of the relationship between the implementation and the formal Z specification. In the next chapter we examine, through examples and discussion, how this mechanism performs within the wider context of IPSE facilities, and see how the stated aims for ASPECT of integration and openness have been achieved through this mechanism.

CHAPTER 9

EXAMPLES, ANALYSIS, AND EVALUATION OF THE VIEW MECHANISM

9.1. Introduction

To illustrate how the view mechanism can be used within an ASPECT IPSE to support the process of large software systems development, the ideal would be to allow an ASPECT IPSE to be used in a project of realistic size and complexity, and to investigate its performance in terms of its ease of use, physical performance statistics, and so on. In trying to achieve this, we are faced with at least the following problems:

- the aim of ASPECT is to provide an infrastructure for users and tools. The evaluation of services provided by an ASPECT IPSE can only take place when the IPSE has been populated with tools, and customised to support a large project. This work will be costly and time consuming to carry out.
- for an organisation to invest time and resources into moving its software development to an ASPECT IPSE, there must be some proven record of success before such a commitment will be made. This problem, experienced by IPSE vendors in general, and only recently being aided by the availability of metrics and statistics for IPSE performance, is particularly applicable to a research project such as ASPECT, producing a prototype implementation as a test vehicle for ideas rather than for rigorous testing.

- as a prototype implementation, many of the development decisions taken within the ASPECT project have resulted in an implementation which allows analysis at a functional level, at the expense of areas such as performance optimisation, robustness, and error recovery. Consequently, a certain amount of re-engineering would be needed to bring the implementation to production standard.

Having said this, however, work is progressing in three areas which, in the long term, will help in the analysis and evaluation of the work reported here.

Firstly, under the Alvey Software Engineering Programme, a project is currently underway specifically aimed at a practical evaluation of the ASPECT IPSE.[Sef86] A collaborative project between British Aerospace and Lancashire Polytechnic, the work is designed to follow three routes:

- evaluation and assessment of the ASPECT project through reading and commenting on the working papers and reports produced by the project.
- applying the prototype ASPECT IPSE to suitable practical examples.
- examining the training requirements for organisations which would wish to introduce ASPECT-type IPSE's into their software development strategy.

This is an on-going project, which will take many more months of work before presenting the results of its evaluation.

Secondly, work on the ASPECT project itself has been extended with the particular aim of consolidating the results achieved thus far. Amongst other things, this involves applying the ASPECT IPSE to a number of simplified examples. In analysing its performance for these cases we can infer how an ASPECT IPSE might handle a larger, more complex situation. It is

this work that forms the bulk of this chapter.

Finally, some of the fundamental principles of ASPECT were derived from an IPSE product called *Perspective*, produced by the company Systems Designers (SD). These ideas formed the basis of the work carried out on ASPECT and, of course, brought the project quantifiable results showing how productivity gains of up to 600% can be achieved, as well as a 90% reduction in errors, using IPSE technology.[Pri87] As the project's ideas developed a number of these have been taken up by SD and incorporated in their development of *Perspective* currently known as *Perspective Kernel (PK)*. Product development is continuing by incorporating more of the ideas in planned future releases. It will be interesting to see how this IPSE develops, and how it performs in forthcoming projects.

So, as a precursor to the more stringent analyses which are to be carried out in the near future, the analysis and evaluation of the ASPECT view mechanism reported here rests on its use in a number of simplified examples. We examine how an ASPECT system could deal with these examples, discuss an implementation in practice, and suggest how we can extrapolate from this work to use an ASPECT IPSE for large scale software development.

9.2. An Example

Of the many problems which arise throughout the lifetime of a large software project, one of the most complex and time consuming is that of maintaining the finished product. We therefore select a realistic example of the particular operations and tasks which occur at this phase, and then examine one way in which an ASPECT IPSE could be used to support these activities.

For a typical scenario of the maintenance phase of a large software project we can envisage a situation in which a project manager receives error reports from users of software. When the manager evaluates these reports, he/she may decide that changes to the released versions of software are needed and can formulate change requests which are approved by a designer before being assigned to programmers, and subsequently carried out to effect the change. For one particular change request, the course of events will typically proceed as follows.

Firstly, the manager will create the new change request, describing the fault that has occurred and the software modules involved. This request will then be issued to the designer for approval before being passed on to a programmer who is instructed to deal with the problem. Now the programmer can carry out the change by creating amended versions of some of the software modules involved, and testing the new components to ensure the changes are effective.

Once this unit testing has been completed, the new software will be passed on to the quality control department, who will then run rigorous system testing with the new software before approving a new release for customers.

9.2.1. The ASPECT Approach

Consider the use of an ASPECT IPSE to support the development of a large scale software project, and the maintenance phase of the development in particular. Part of the data in the Information Base may concern change requests to amend released versions of program modules, and we would like to record the assignment of programmers to attend to these change requests, and to monitor the status of the requests. This is typical of the problems which are encountered in developing large software systems, and in most cases very poor ad hoc solutions exist. Let us consider how we might support the maintenance process of a project as outlined above through the use of an ASPECT IPSE, with particular emphasis on the use made of the view mechanism.

Suppose that structures are defined in the Information Base to record the necessary data, as shown in figure 9.1 using the notation described in chapter 8.

In figure 9.1 change request entities are recorded as instances of the "Change_Request" entity type, and programmers as instances of the "Programmer" entity type. Assignments of individual programmers to work on particular change requests are represented by instances of the associative entity type "Assignment". Each change request is associated with an initial source code file through the entity type "Change_Request_Old", and when dealt with, will also be associated with a new version of the source code through the entity type "Change_Request_New". The contents of a file are classified as either Ada source or Ada object through the entity types "Ada_Src" and "Ada_Obj", which are subtypes of the entity type "File". The information about which programmers can access each file is represented in the entity type "Permissions". Finally, the particular tools used to create the files are represented by the entity type "operation", which associates a file with the tool that was

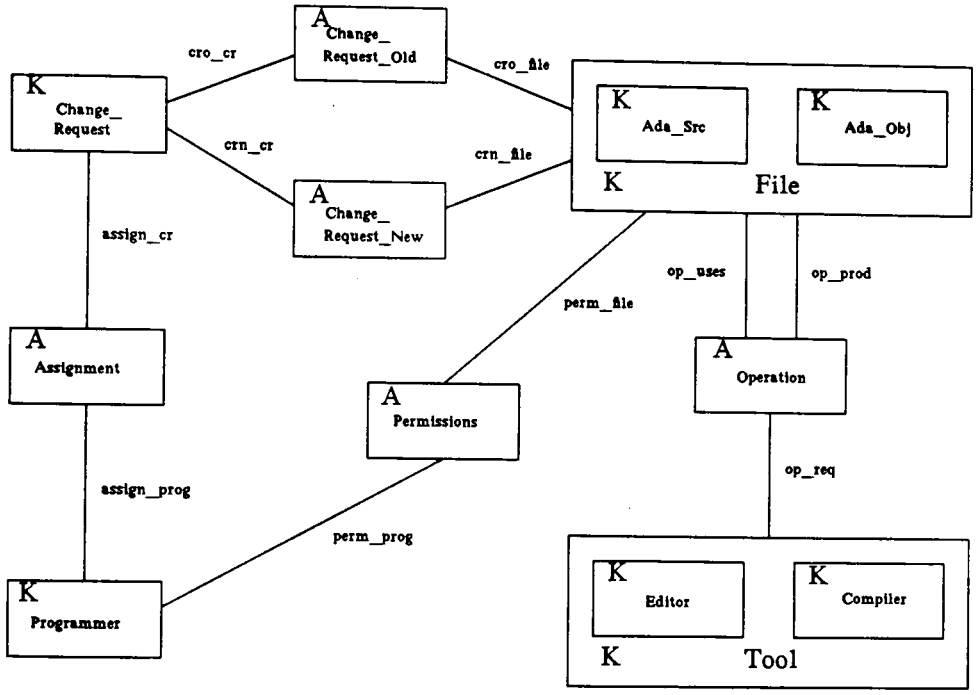

Fig 9.1 Entity Diagram for Simplified Example.

used to create it.

Each of the entity types shown will have a number of properties appropriate to the information in which we are interested. For example, file entities may be recorded with their size, owner, and the date they were last changed. Hence, by recording the data in a form which allows arbitrary queries to be defined we are able to easily extract the information of interest.

To illustrate how an ASPECT IPSE can be used to help maintain control, let us consider the creation of a new change request, its assignment to a programmer, and the subsequent creation of an amended version of the code to effect the change.

We can envisage a manager who is responsible for creating and assigning new change requests from error reports which have been submitted from customers. When the manager has evaluated an error report, and decided that a new change request needs to be created, then he/she will log in to the existing "CREATE NEW CR" activity to carry out this task. This activity will already have been defined for the purpose of allowing new change requests to be created and assigned. It will consist of a set of pre-conditions which define when it is valid to create a new change request, a set of post-conditions which define when the task has completed satisfactorily, and a set of rules which govern all operations which form part of the execution of the activity. When the activity was created, it was associated with a user who is allowed to carry out the activity, in this case the manager, and an Abstract Environment (AE) which defined how the user executing the activity views the Information Base in terms of the data he/she can access, the operators that can be executed, and the tools that can be invoked. For example, for this activity, the AE may allow access to programmer, change request, and assignment information, provide operators to create a new change request and assign a change request to a programmer through the "publish" operator.

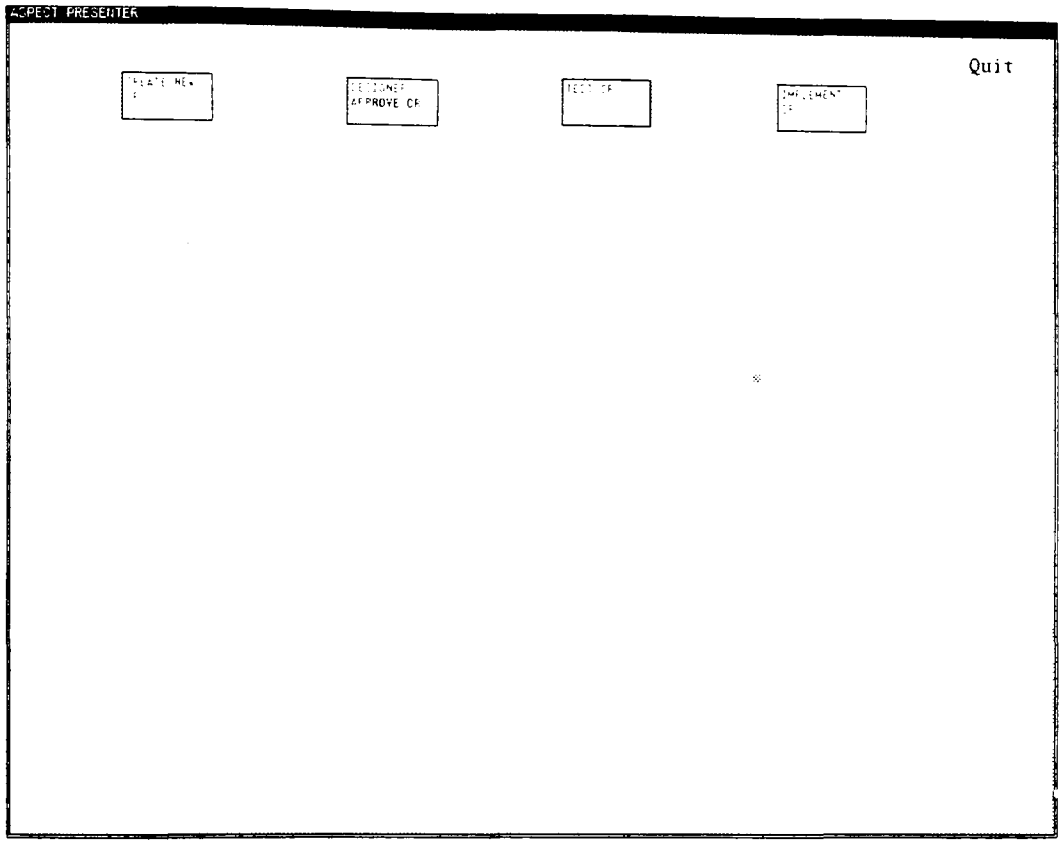

Fig 9.3 Screen Display on Entering the ASPECT System.

In figure 9.3 we see the screen display from an implementation of this example using the graphical interface facilities to an ASPECT IPSE built using the facilities of the ASPECT HCI services, collectively known as *The Presenter*.[Too86] The display shows the state of the system when a user enters the ASPECT system, with four named regions along the top of the screen. Each region refers to an available Activity which the user can attempt to log into to carry out some work. So, for example, by selecting the left-most region the user can attempt to log into the Activity to create a new change request "CREATE NEW CR".

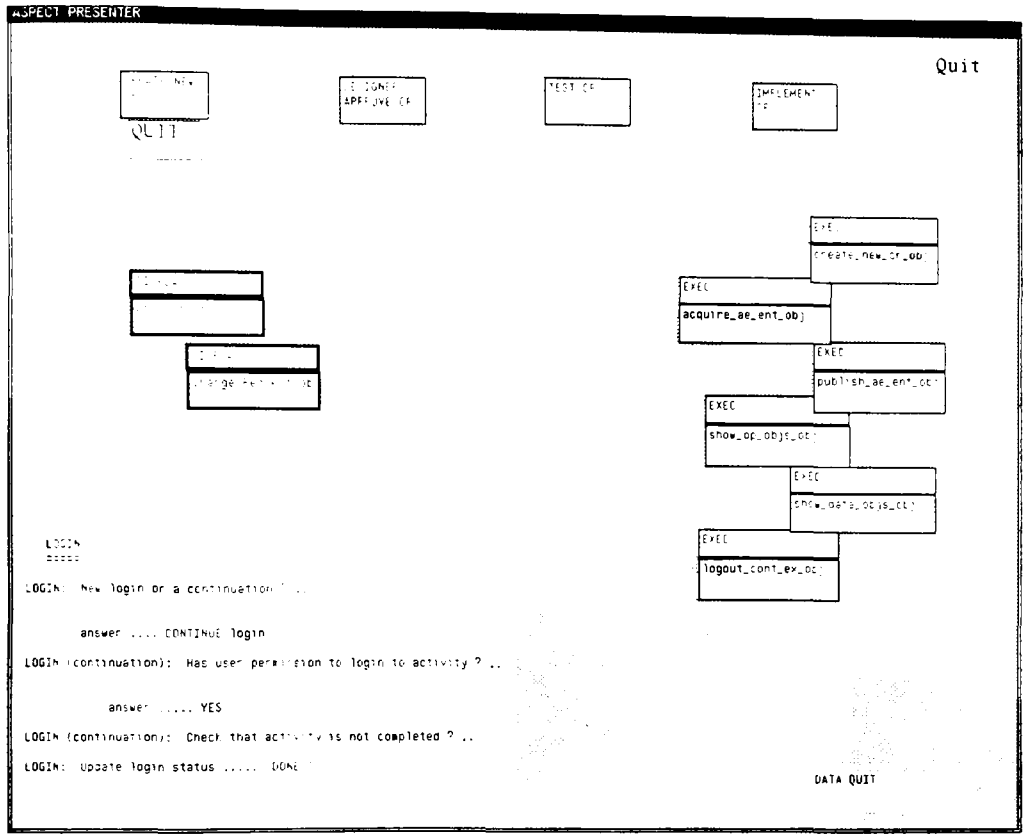

Fig 9.4 Screen Display on Logging into the "CREATE NEW CR" Activity.

In figure 9.4, the manager has logged into the Activity to create a new change request. The region on the lower left of the screen contains trace information which indicates to the user which actions are taking place, reports errors and illegal actions, and so on. On logging into this Activity, a number of regions have been displayed on the right of the screen which show the different operations that the user is allowed to perform within this Activity. These are obtained by querying the ASPECT Information Base to determine which AE has been linked with this Activity, and then displaying a region for each of the operators within that AE. So, for example, one operator available is to create a new change request, "create_new_cr_obj", and is

defined, perhaps unknown to the user of the operator, as a sequence of lower level operations to create a change request instance with appropriate property values.

Also available within this Activity are a set of data objects shown on the left of the screen which the user can display at any time. These may have been explicitly inherited from a parent AE, such as "Change_Request_obj", or be abstract data objects such as "prog_cr_obj" which is a combination of lower level objects. In this case the object contains all change requests together with the programmers assigned to carry them out. For all the data objects available, only the entities which are owned by that user, or have been explicitly published to, and acquired by that user, are displayed.

Now let us suppose that the manager executes the operators to create a new change request and publishes that request to the designer to approve the request. If the programmer, impatient to carry out his/her work, attempts to login to the Activity to implement the change request, "IMPLEMENT CR", as shown in figure 9.5, then the login operation fails because the pre-conditions associated with the Activity have not been met. In this case, one of the pre-conditions set for the Activity was that the programmer could not start the Activity until a designer approved change request was available. As this is not yet the case, the request to initiate the Activity has failed.

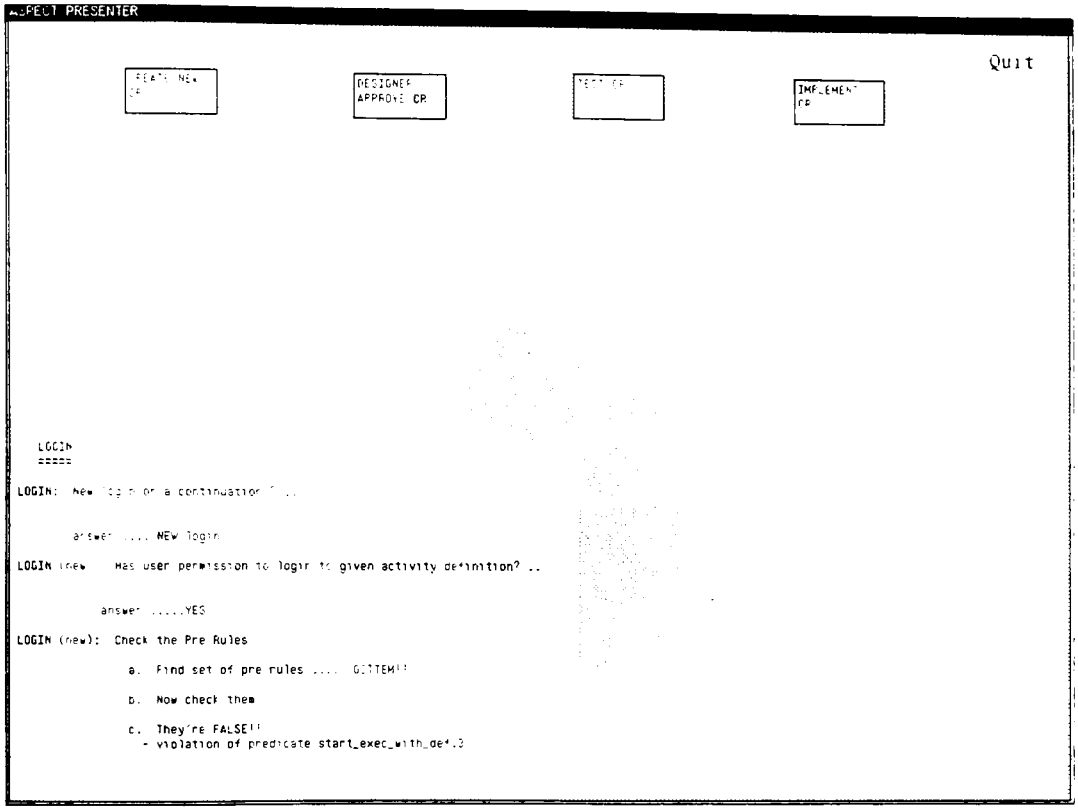

Fig 9.5 Screen Display on Attempting to Login to the "IMPLEMENT CR" Activity.

So, in a similar fashion, the designer could log into the "DESIGNER APPROVE CR" Activity, and approve the change request, publishing it to the programmer. The programmer in turn can log into the Activity "IMPLEMENT CR" and will have available the data and operators necessary for the task. Most probably, besides the software modules involved in the changes, this will include the error report itself, a designer's summary of the problem, and the initial specification document for the modules. Tools and operators will also be available within the Activity such as editors, compilers, debuggers, and so on.

When the programmer has made the changes he/she thinks are necessary, and performed testing on the modules in question, the latest software will then be published to the quality control department, which will have an on-going activity to fully test amended versions of software published to it, "TEST CR". On receiving the new software, the quality control department will apply the test suite to the fully built system and analyse the results. If unsatisfactory, then a report of the problems would be sent to the programmer, who will then continue with his/her work. If, on the other hand, the tests are successful, then an acknowledgement will be sent to the programmer who is now able to terminate his/her activity. The manager will also be informed, and can update the status of the change request to "completed".

9.2.2. Summary

In this section we have discussed in detail a simple example, with reference to an implementation of the example built using the ASPECT HCI services. Without too much difficulty we can extrapolate from this example problem to envisage a larger system in which a similar process of the creation of Activities linked to AE's provides the mechanism for the controlled implementation of a task. In addition, part of an AE for an Activity may well be the operators for creating new Activities and AE's, and for assigning them to users. In this way the hierarchical nature of many software tasks can be modelled and controlled in a convenient and flexible way, allowing Activities to be created and decomposed to mirror the development process.

9.3. Tool Interfaces

One of the stated goals for the ASPECT view mechanism was to investigate the ease with which we could embed existing tools into an ASPECT IPSE with minimum changes to the tools themselves. The way in which we do this is to define a view of the Information Base, through an AE, which mirrors a tool's native operating environment. The tools can then run in this view without change, while the integrity and consistency of the Information Base is not compromised by any attempt to by-pass the enhanced semantic controls applied at the ASPECT PTI.

To illustrate how this mechanism could be applied in practice, we examine one particular example of the inclusion of tools foreign to ASPECT which is of particular importance, and suggest how we might use the view mechanism to facilitate their use in an ASPECT IPSE.

9.3.1. The ASPECT Open Tool Interface (OTI)

An important set of tools which we would like to be able to make use of in an ASPECT IPSE will be those available in UNIX systems. The ease with which new tools can be added to a UNIX system, coupled with the popularity of the UNIX operating system itself, has led to a particularly rich set of tools and utilities which operate in a UNIX environment. Naturally, where these tools have already been implemented it would be desirable to import them in to an ASPECT IPSE and not only save a great deal of time and effort in duplicating the implementation of existing tools, but also to provide access to ASPECT data through tools which are familiar to many users.

Of course, we have a simple solution to include UNIX tools in an ASPECT IPSE - because ASPECT is hosted on the UNIX operating system, we could allow UNIX tools to access data directly through this interface, by-

passing the higher-level ASPECT PTI which has been built above it. However, there are a number of reasons why this approach would be undesirable. Firstly, from a security and integrity viewpoint, having constructed a highly semantic interface to data access, in which users view data as collections of entities, or as relations, it would mean that we now allow them to break the constraints that exist at that level and deal with the underlying file representation itself. Maintaining data integrity in this environment would be significantly more complex. Additionally, taking this approach is dependent upon this particular implementation of ASPECT. If we were to re-implement ASPECT on another operating system, for example, then this solution would no longer be appropriate. A further reason for not adopting this solution is that although we have overcome the problem for these particular tools, we face a similar set of problems for importing tools built to other interfaces, and a uniform solution to the problem would clearly be desirable.

The approach taken in ASPECT is to provide a view of the ASPECT Information Base which supports the UNIX System Call Interface, and hence allows integration of the UNIX tools in a uniform way, without compromising the integrity provided at the ASPECT PTI. This approach is the uniform mechanism used to support integration of any foreign tool into an ASPECT IPSE.

9.3.2. An Example

As an example, consider a UNIX tool which, as part of its operation, opens a UNIX file, writes to the file, and then closes the file. One of the predefined types of ASPECT is the file type, and any entity can have an attribute whose value is drawn from the file type. Therefore, at the ASPECT PTI there are operations for opening, closing, reading, and writing of ASPECT files. So, for example, to support the UNIX file open operation, a mapping would be defined in the form of an operator within an Abstract Environment (AE) which converts the UNIX file open into an ASPECT file open. Typically, a UNIX operation to open a file "fred" would result in an ASPECT operation to open the file attribute associated with the ASPECT entity whose name is "fred". In a similar way it is expected that many of the UNIX System Call Interface operations could be supported in ASPECT, and consequently many of the UNIX tools could be used within an ASPECT IPSE.

Clearly, having established and demonstrated the principles behind the mechanism, practical issues such as performance would need to be examined. At this point we choose not to address these issues, other than making the observation that in a full production standard ASPECT IPSE the overhead involved in accessing UNIX operations in this way could be reduced through exploiting optimised software implementation techniques, and if necessary through the use of specialised hardware.

9.4. Other IPSE Interfaces

The IPSE field is still in a state of flux, and a number of approaches to IPSE design and philosophy are at present under consideration in an attempt to define the way forward in IPSE research. As a result, a number of IPSE's have been implemented over the past few years. (A discussion of some of these is presented in Chapter 2 of this thesis). Hence, work such as has been carried out within the ASPECT project must be fully aware of the wider context of IPSE development if it is to avoid becoming classed as irrelevant as both political and technical forces advance the field. In particular, work in providing an IPSE framework, as attempted by ASPECT, in which a tool writer can embed tools for use throughout the software development life-cycle, is likely to be directed towards a standard tool writer's interface, driven by the need for portability of tools between different IPSE's. Until this occurs, in the near or distant future, the ASPECT approach to tool integration will be particularly useful for allowing an ASPECT IPSE not only to allow tools to be added which have been specially written to the ASPECT PTI, but also to allow the addition of tools written to other IPSE PTI's. This use of the view mechanism makes the ASPECT system particularly attractive in the quickly changing field of IPSE development.

Here, we look at how an interface to data based on the entity-relationship approach to data modelling, as supported by the PCTE, could be offered as a view of the ASPECT Information Base through the creation of a suitable AE.

9.4.1. The PCTE as a View of ASPECT

The PCTE, in a similar way to ASPECT, is designed to be the basis for the construction of IPSE's which support a particular project life-cycle model through an integrated collection of tools and services.[Bul85, Sys87] The tools access PCTE services through a common Public Tool Interface (PTI) which provides a set of primitive operations that allow controlled interaction with the PCTE facilities. Hence, the basic philosophy of PCTE and ASPECT can be considered to be the same. What is significantly different, however, is the way in which each system attempts to provide the necessary support for meaningful tool interaction.

As in ASPECT, the services provided in PCTE are grouped into a number of areas, and for this work we concentrate on the Object Management System (OMS) of PCTE, which can be seen as analogous to the Information Base of ASPECT.

9.4.1.1. The Object Management System of PCTE

To record and manipulate the data generated during a project life-cycle, the PCTE contains an Object Management System (OMS) which records data items as objects which are involved in relationships with other objects.[Gal86] Where as the ASPECT Information Base uses RM/T to define the conceptual model of data, the PCTE OMS is based on its own particular derivation of the Entity-Relationship (ER) model in which PCTE objects resemble the entities of the ER model.[Che76]

In many ways the RM/T approach of ASPECT and the ER-based model of PCTE can be considered very similar at the conceptual level.[Hit87a] Where they differ greatly is in the representation of the model they employ, and in particular the data definition and manipulation languages they use.

The main differences can be summarised as:

- in PCTE two distinct types of object exist, objects and links, and separate operators exist for each type. For example, there is one operator to create an object, and another to create a link. A link is a uni-directional association between two objects, while a pair of links between two objects can be bound together to form a relationship between the objects. In ASPECT everything is considered to be an entity, which will be classed as one of kernel, associative, or characteristic, and in addition may be designative. A single set of operators are available for all entities.
- in PCTE a network-based implementation is used in which users navigate from one object to another via the relationships and links. In contrast, ASPECT employs a relational set-oriented approach with data manipulation via a relational algebra.
- in PCTE objects and links are identified by pathnames, as a direct extension of naming facilities in UNIX. ASPECT, on the other hand, identifies each entity with a unique system generated identifier, and allows the user the flexibility of defining the specific naming scheme required for a particular project or task.
- the meta-data describing, for example, which types of object exist, what their attributes are, and so on, is accessed in PCTE through a fixed set of operators. The structures containing meta-data information are not exposed, and hence user-defined queries on this data are not supported. In particular the query tools which manipulate user data cannot be applied to the meta-data. In ASPECT, however, the model is self-referential in the sense that all meta-data is held in the same form as user defined data, and can be accessed in the same way, using the same set of relational algebra operators.

9.4.1.2. Embedding a PCTE Tool in ASPECT

Having given an indication of the kinds of differences which exist in the information storage and manipulation facilities of PCTE and ASPECT, we now look at how we might approach the task of taking an existing PCTE tool which accesses and manipulates data, and supporting it within an ASPECT IPSE.

We begin by looking at an example of how we could support operations on a particular set of data structures already within ASPECT, before considering the more general case where any PCTE tool could be supported in ASPECT.

9.4.1.2.1. A PCTE View of Existing Structures

Given an ASPECT system in which the data schema has been defined, one of the actions we may wish to take could be to import an existing PCTE tool which makes access to those structures. To support the new tool we would define a view of the ASPECT PTI by creating a suitable Abstract Environment (AE) which supports the operations carried out by the tool. These operations would be mapped down to their interactions with ASPECT data using ASPECT PTI operations.

To illustrate how this could be achieved, consider the example described earlier in which the ASPECT Information Base is used to record information about programmers and their assignments to carry out change requests on program modules, shown in figure 9.3. Also, consider a PCTE tool which, amongst other things, creates new change requests and assignment instances. Following the C language interface to PCTE,[Bul85] particular calls to the operators to create objects and relationships could be

```
crobj( "mypathname", "change_request1", Change_Request, 01 );
crlink( "fred", "link1", "change_request1" );
```

which creates a new change request named "change_request1" within the context of "mypathname", and creates an assignment of the new change request to programmer "fred", giving the assignment the name "link1". Default initial attribute values will be assigned to the new object and link, which can be re-defined with calls to the operator "setattr" for an object, and "setlattr" for a link.

In order to support these operations within an AE, suitable AE operators would need to be defined which map these PCTE operations into their equivalent ASPECT operations. For example, the operation to create a new object could be defined as in figure 9.6.

```
int crobj( namespace, obj_name, obj_type, mode )
char   *namespace, *obj_name, *obj_type;
short  mode;
{
   surrogate  s_namespace, s_type, new_surg;

   /*  convert namespace and obj_name to surrogates  */

   s_namespace = surrogate_of( namespace, initial_namespace );
   s_type = surrogate_of( obj_type, s_namespace );

   /*  create an instance of obj_type  */

   new_surg = create_entity( s_type, attmap );

   /*  give name obj_name to new entity  */

   name_entity( new_surg, obj_name, s_namespace );

   return(OK);
}
```

Fig 9.6 ASPECT Implementation of "crobj".

In figure 9.6, a simplified description of the PCTE "crobj" operation in terms of its underlying ASPECT operations has been given. From this example we can see that a correspondence exists between a PCTE object and an ASPECT entity, and that through the operations "surrogate_of", which returns the surrogate of a name within a namespace, and "name_entity", which gives a name to an entity within a particular namespace, we can create a mapping between PCTE names and ASPECT surrogates. In a similar way we could define a mapping for the PCTE "crlink" operator.

9.4.1.2.2. Providing Support for any PCTE Tools

The simple example in the above section shows how the PCTE operators for creating objects and links could be supported in ASPECT. To enable any PCTE tool to be supported in ASPECT, an ASPECT AE would have to be defined in which all of the operators which are available at the PCTE PTI were available and had their expected functionality. It would clearly be a non-trivial task to define and implement such an AE, particularly with the major differences in data manipulation operations offered by the two systems. However, the way in which this could be achieved would be to define these mappings though analysing the operators of PCTE and producing a schema for the meta-data of the OMS in terms of RM/T entities. This schema could then be recorded within ASPECT, and the PCTE operators defined in terms of their RM/T operations on this meta-data, in an analogous approach to the way in which the RM/T catalog is used by the RM/T operators. Indeed, a first attempt at the definition of a meta-schema for PCTE has been made[Rob87a] and the step of recording this information in ASPECT and defining the appropriate operators is currently under consideration. Once this is achieved, it would be possible to support any of the PCTE operators, and

hence any PCTE-based tool, within an ASPECT system.

Again, practical issues such as the performance of supported PCTE tools will be important in a production standard ASPECT IPSE. Steps currently being taken towards addressing these issues are discussed in the next chapter.

9.5. Summary

In this chapter we have shown through a number of simple examples how a view mechanism can be used as a component of an IPSE to support the creation and use of external views of the ASPECT services to suit individual users and tools. Clearly, more work is required in this area to fully evaluate the mechanisms provided, and the means by which this is to be achieved have been outlined. Given the current state of of IPSE research, with a significant degree of uncertainty as to the best way to provide IPSE support through a PTI, the ASPECT approach of including within an IPSE the facilities for its own customisation to suit individual users and tools, and hence a number of IPSE PTI interfaces, should prove particularly useful as a vehicle for the experimentation and further study which is certainly needed.

CHAPTER 10

CONCLUDING REMARKS

10.1. Introduction

In the final chapter of this thesis we review the work that has been carried out, and discuss the areas in which it may continue in the future.

10.2. Review of Work

The original aims of this thesis, outlined in chapter 1, were to examine past and current methods for constructing IPSE systems, to evaluate software engineering databases as the central component of an IPSE, and to specify and implement a view mechanism as an integral feature of an IPSE infrastructure. In this section we review the work reported in this thesis with particular emphasis on these aims.

The early stages of the thesis were an analysis of IPSE components through examining the development of IPSE technology from the original development environments for software which consisted of a set of ad hoc tools supporting different stages of the development life-cycle. From this analysis, the importance of open IPSE's was advanced, providing flexible support for the integration of tools through a Public Tool Interface (PTI).

At the heart of an IPSE is a structured repository for recording all development data. We continued by analysing the advantages of using a database as this central component, and looking in detail at how a software engineering database exhibits particular requirements for which conventional

database technology is not always readily applicable.

One of the mechanisms generally available within a commercial database system allows multiple external views of the database to be defined. Such a view mechanism allows different abstract interfaces to the data to be defined, appropriate to different users needs. Within an IPSE system there is a similar need for supporting interaction with IPSE data at different abstract levels. However, the nature of IPSE use, accessed by a particularly diverse set of users from programmers to high-level managers, coupled with the need to support tool interaction with data where the tools may have been written to many different interfaces, provides additional requirements for an IPSE view mechanism. Indeed, by examining the abstraction mechanisms used within programming languages to support similar aims, we were able to define a model of a view mechanism for use in an IPSE which is much more powerful than those generally available in existing conventional database systems.

Having defined a view model, we then specified an IPSE view mechanism using the formal specification language Z. This provided a formal description of the model which was then used as the basis for a subsequent implementation within the constraints of a larger IPSE project.

Following a discussion of the implementation issues which needed to be addressed, an analysis of the mechanism was given by considering a realistic example on which the view mechanism, in conjunction with the other IPSE facilities, was used. From this example, the ways in which the view mechanism could be applied to other examples were discussed.

10.3. Future Work

Following on from the work reported in this thesis, there is scope for continued research in a number of directions. This work can be divided into three main areas.

Firstly, improvements to the implementation of the view mechanism will need to be made as a precursor to further analysis of its use. In particular, the time taken to perform some of the operations, for example, to link an Activity with an AE and a user, would at present not permit the realistic use of these mechanisms within a "live" system. The individual view mechanism operators themselves could be optimised, but the more fundamental reason for the performance problems lies in the use of a commercial database system, db++, as the internal level on which the conceptual base of ASPECT, RM/T, has been implemented. As a result, each RM/T query or update operation on an entity executed by a view operator performs a number of underlying db++ operations on relations. These in turn are mapped to underlying UNIX operations on files. This results, for example, in an execution time of the order of a second each time a name is converted to a surrogate, or vice-versa. As a typical view mechanism operator will require the conversion of names to surrogates many times in a single execution, we have a severe bottleneck on performance. Work currently under consideration is to re-implement RM/T directly on top of the UNIX operating system, by-passing one of the most time consuming layers in the implementation architecture. This would significantly improve execution times of view operators.

Secondly, there is wide scope for further investigation and analysis of the view mechanism in a number of areas. Most importantly, there is a need for continuing the initial evaluation of the ASPECT services for supporting large scale software development, in particular the role of the view mechanism in that process. This is seen as the most appropriate area for continued

ASPECT research, and is currently under investigation.

However, a wider analysis of the view mechanism is also possible. For example, there is currently a great deal of work going on in the area of Object-Oriented Programming Systems (OOPS). In this approach, a programming process is considered to be a set of objects, which can can be classed into object types. Each object type is related through a hierarchy, and has an associated set of methods which are operations that can be performed on instances of the type. Messages are passed between object instances instructing them to execute particular methods, and hence carry out actions. It is interesting, then, to compare the OOPS notion of an object with the ASPECT notion of an entity, where entities are members of an entity type, controlled through a type hierarchy, and through the view mechanism we associate particular operators with a set of entities. A much more detailed investigation of these issues may prove particularly instructive for the further understanding of IPSE systems.

Finally, the view mechanism facilities are made available through a set of C language functions held in a library, which a user must link into a main program to use. Clearly there is a need for tools to be built on top of this primitive interface which allow a user to define and use AE's in a much more convenient and natural way. The graphical interface to ASPECT built for the example in chapter 9 has given some indication as to the forms this can take, but much more work could be carried out in this area.

APPENDIX A

Z CONVENTIONS USED IN THIS THESIS

Throughout this thesis we use the following conventions:

For any schema S , Δ S represents the conjunction of a before-state S and an after-state S'. Formally:

$$\Delta S \mathrel{\hat{=}} S \wedge S'$$

Further, for any schema S, Ξ S represents the schema in which the before and after states are identical. Formally:

$$\Xi S \mathrel{\hat{=}} \Delta S \mid S' = S$$

We also use the notion of a framing schema. This is used to represent a change of state in which we wish to state some changes explicitly and say "everything else remains the same". We cannot define a framing schema in a context-free way: our use will always be in the context of some other schema. In general it is used as follows:

T

Φ S
other declarations and schemas
Predicate

Suppose S contains a set of variables V, of which a subset V1 are mentioned in the predicates of T or the other schemas and the remainder V2 are not. Then the above is to be read as:

T

Δ S
other declarations and schemas
Predicate
V2' = V2

For operations we always decorate input variables with "?" and outputs with "!".

We use the following case conventions:

- Names entirely in upper case are global types or state schemas - for example "USER", "IB".
- Names entirely in lower case are relations or names local to a schema - for example "subtree", "attributes".
- Names in mixed case are operation schemas - for example "init_IB".

A.1. A List of Z Symbols

$\hat{=}$	Syntactically equivalent to
$\neq$	not equals
$\leqslant$	less than or equal to
$\geqslant$	greater than or equal to
$\wedge$	logical and (conjunction)

$\rightarrowtail\!\!\!\!\twoheadrightarrow$	bijection
^	sequence catenaction
]	close bag bracket
)	closing image bracket
×	cartesian product
∩	Distributed set intersection
Δ	Schema state change
∪	Distributed set union
◁	domain restriction
⩤	domain subtraction
∃	there exists (existential quantification)
∀	for all (universal quantification)
;	forward relational composition
$\mathbb{F}$	Finite subset
→	total function
⊕	function over-riding
⇔	if and only if (equivalence)
⇒	implies
⇐	is implied by
$\rightarrowtail$	injection
$\mathbb{Z}$	the set of all negative, 0 and positive integers
$^{-1}$	function inverse
λ	lambda
∈	set membership
↦	maplet

μ	mu
$\mathbb{N}$	Natural Numbers
$\neg$	Negation operator
$\notin$	not set membership
$^{+}$	non-reflexive transitive closure
$\not\cap$	not set and
$[\![$	left bag bracket
$(\!|$	opening image bracket
$\vee$	logical or (disjunction)
partitions	partition
$\mapsto\!\!\!\!\!\!\parallel$ ⇻	partial finite function
⤕	partial finite injection
⤀⇻	partial finite surjection
⇸	partial function
Φ	phi
⤔	partial injection
$\gg$	Pipeline
$\mathbb{N}^{+}$	Positive natural numbers
$\mathbb{P}$	Power set
⤀	partial surjection
$\circ$	relational composition
$\leftrightarrow$	relation
$\rhd$	range restriction
⩥	range subtraction
*	reflexive transitive closure

$\cap$	set intersection
-	set difference
$\subseteq$	set inclusion
$\cup$	set union
$\upharpoonright$	sequence restriction
$\restriction$	sequence range restriction
$\subset$	strict set inclusion
$\twoheadrightarrow$	total surjection
θ	theta
$\vdash$	theorem
$\models$	universally valid
Ξ	Schema no change

APPENDIX B

TREES FOR ASPECT

This appendix provides a generic theory of trees, and other types of graph, for reference from within the thesis. It is a subset of the definition of trees and graphs for ASPECT that is defined in ASPECT's specification of the public tool interface.[Rob87c]

First, we build up the underlying data types using schema calculus.

B.1. The Basic Types

We start with a general directed graph which is generic in its node type, N.

[N]

DG

$$\begin{array}{l} nodes : \mathbb{P}\ N \\ parent : N \leftrightarrow N \\ \hline dom\ parent \cup rng\ parent \subseteq nodes \end{array}$$

In this and subsequent definitions, the name of the relation should be read as "has parent". So its domain is the set of all children, its range the set of all parents.

This is the basic definition, but it is useful to extend it with some auxiliary relations:

- **children** which is obviously the inverse of parent
- **ancestors** which relates a node to everything reachable via the parent relation
- **descendants** which is all the nodes reachable from a given node via the children relation
- **family** which is all the descendants of a node plus the node itself.

We also define two subsets of the nodes :

- **leaves** which are all the nodes with no children
- **roots** which are all the nodes with no parents

We define a relation leavesof which yields all the leaves in the family of a given node.

DG

$$
\begin{array}{|l}
\hline
\text{DG} \\
\\
\text{children,ancestors,descendants,family} : \mathbb{N} \leftrightarrow \mathbb{N} \\
\\
\text{leaves,roots} : \mathbb{P}\ \mathbb{N} \\
\\
\text{leavesof} : \mathbb{N} \leftrightarrow \mathbb{N} \\
\hline
\\
\text{children} = \text{parent}^{-1} \\
\text{ancestors} = \text{parent}^{+} \\
\text{descendants} = \text{children}^{+} \\
\text{family} = \text{children}^{*} \\
\text{leaves} = \text{nodes} - \text{rng parent} \\
\text{roots} = \text{nodes} - \text{dom parent} \\
\text{leavesof} = \text{family} \rhd \text{leaves} \\
\\
\hline
\end{array}
$$

We next define acyclic directed graphs, which are those that contain no loops in the parent relation.

AG

DG

ancestors ∩ Id N = {}

We can also, independently, define connected graphs which are those for which there is some node of which all other nodes are descendants.

CG

DG

∃ n : nodes . nodes = family (| n |)

A connected acyclic graph has the obvious definition. The UNIX directory structure is an example.

CAG

CG
AG

Each node in a tree has at most one parent. More generally, an acyclic graph in which this is true is a forest, and a connected forest is a tree.

FOREST

AG

parent ∈ N ⇸ N

TREE

FOREST
CAG

B.2. Labelled and Ordered Graphs

As we have a generic node type, we do not need to consider properties of nodes. However, it is useful to consider graphs whose links are labelled. For our purposes, we treat labelling in terms of sets of labels on the links to the children of each node.

The most general labelled graph is:

[L]

LG

DG
labels : $N \longrightarrow (L \leftrightarrow N)$

$\forall n : N \;.\; \text{rng labels } n \subseteq \text{children } (\!|\, n \,|\!)$

$\forall n : N \;.\; (\text{labels } n)^{-1} \in N \rightarrow\!\!\!\!\!+\!\!\!\!\!\rightarrow L$

We can now also consider other particular forms of labelled graph such as:

DLG Distinctly labelled graph, in which all links of a node are labelled, or none are.

CLG Connected labelled graph, which have all links labelled.

OG Ordered graphs, a special kind of DLG in which the labels are the positive integers.

COG Completely ordered graphs, which are CLG's with ordering.

COCAG Completely ordered connected acyclic graph, includes all of the above, and must be acyclic.

We can also add the notions of labels and ordering to trees.

B.3. Operations on Graphs and Trees

A large number of graph operations are required. These must be specified for both labelled and non-labelled graphs. This is not pursued here.

APPENDIX C

REFERENCES

Abr82. Abrial, J.-R., "The Specification Language Z: Basic Library," Internal Report, Oxford University Computing Laboratory, PRG (1982).

Ald85. Alderson, A., Bott, M.F., and Falla, M.E., "An Overview of the Eclipse Project," pp. 100-113 in *Integrated Project Support Environments*, ed. J. McDermid, Peter Peregrinus (1985).

Ast76. Astrahan, M.M. and others, "System R: Relational Approach to Database Management," *ACM TODS* 1(2)(June 1976).

Atk78. Atkinson, M.P., "Database Systems and Programming Languages," *Proceedings of 4th VLDB Conference*, pp. 408-419 (September 1978).

Atk81. Atkinson, M.P., Chisholm, K., and Cockshott, P., "PS-Algol: an Algol with a Persistent Heap," CSR-94-81, University of Edinburgh (December 1981).

Atk87. Atkinson, M.P., Morrison, R., and Pratten, G., "PISA - a Persistent Information Space Architecture," *ICL Technical Journal* 5(3) pp. 477-491 (May 1987).

AT&84. AT&T, *UNIX System V Documenter's Workbench Introduction and Reference Manual.* April 1984.

Ban79. Bancilhon, F., "Supporting View Updates in Relational Databases," in *Database Architectures*, ed. Bracci, North Holland (1979).

Ber87. Bernstein, P.A., "Database System Support for Software Engineering," pp. 166-179 in *Proceedings of the 9th International Conference on Software Engineering*, (March 1987).

Bis86. Bishop, J., *Data Abstraction in Programming Languages*, Addison-Wesley (1986).

Boe81. Boehm, B.W., *Software Engineering Economics*, Prentice-Hall (1981).

Bro85a. Brown, A. W., "Software Configuration Management in a Programming Support Environment," SRM/403, University of Newcastle upon Tyne (Sept 1985).

Bro85b. Brown, A. W., "The Problems of Software Configuration Management," SRM/402, University of Newcastle upon Tyne (August 1985).

Bro86a. Brown, A. W., Earl, A. N., and Weedon, R., "Specification of Expressions," aspect/wb/pub/sr/ibs/expr/spec.1.0, ASPECT Report (August 1986).

Bro86b. Brown, A.W., Earl, A.N., Hitchcock, P., Weedon, R., and Whittington, R.P., "The Use of Databases for Software Engineering," pp. 55-70 in *Proceedings of the Fifth British National Conference on Databases (BNCOD5)*, ed. E.A. Oxborrow, (14th - 16th July 1986).

Bro86c. Brown, A.W., Robinson, D.S., and Weedon, R.A., "Managing Software Development," pp. 197-235 in *Software Engineering '86*, ed. P. Brown, Peter Peregrinus (1986).

Bro86d. Brown, A.W. and Weedon, R., "An Informal Introduction to the Specification Language Z," SRM/435, University of Newcastle upon Tyne (July 1986).

Bro87. Brown, A.W., "A View Mechanism for an Integrated Project Support Environment," in *Proceedings of a Conference on Automating Systems Development*, , Leicester Polytechnic (April 1987).

Buc84. Buchmann, A. P., "Current Trends in CAD Databases," *Computer-Aided Design* 16(May 1984).

Bul85. Bull, ICL, Nixdorf, Olivetti, and Siemens, *PCTE: A Basis for a Portable Common Tool Environment - C Functional Specification.* 1985.

Bux80. Buxton, J. N., *Requirements for APSE - STONEMAN*, US Department of Defence (February 1980).

Cha75. Chamberlin, D.D., Gray, J.N., and Traiger, I.L., "Views, Authorisation, and Locking in Relational Database Systems," *Proceedings of NCC* 44(May 1975).

Che76. Chen, P. P., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems* 1(1) pp. 9-36 Massachusetts Institute of Technology, (March 1976).

Cod79. Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems* 4(4) pp. 397-434 (December 1979).

Con84. Concept_Asa, *A User's Guide to the db++ Relational Database Management System (First Edition).* 1984.

Dat81. Date, C. J., "Referential Integrity," in *Proceedings of IEEE 7th Conference on Very Large Databases*, (1981).

Dat83. Date, C.J., *An Introduction to Database Systems - Volume II*, Addison-Wesley (1983).

Dat86. Date, C.J., *An Introduction to Database Systems - Volume I*, Addison-Wesley (1986).

DeM82. DeMarco, T., *Controlling Software Projects*, Yourdon Press (1982).

Den86. Denning, D.E., Akl, S.G., Morgenstern, M., Neumann, P.G., Schell, R.R., and Heckman, M., "Views for Multilevel Database Security," pp. 156-172 in *Proceedings 1986 Symposium on Security and Privacy*, (April 1986).

Dol76. Dolotta, D. A. and Mashey, J. R., "An Introduction to the Programmer's Workbench," pp. 164-168 in *Proceedings of 2nd International Conference on Software Engineering, San Francisco*, (October 1976).

Dow86. Dowson, M., "ISTAR - An Integrated Project Support Environment," *Proceedings of 2nd SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 27-33 (December 1986).

Enc82. Encarnacao, J. and Krause, F. L., *File Structures and Databases for CAD*, North-Holland (1982).

Fel79. Feldman, S. I., "Make - A program for Maintaining Computer Programs," *Software Practice and Experience* 9 pp. 255-265 (April 1979).

Fli87. Flinn, B., Gimson, R., Hayes, I., Morgan, C., Sorensen, I.H., and Sufrin, B., *Specification Case Studies*, Prentice-Hall International (1987).

Gal86. Gallo, F., Minot, R., and Thomas, I., "The Object Management System of PCTE as a Software Engineering Database Management System," *Proceedings of 2nd SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pp. 12-15 (December 1986).

Gar83. Garlan, D.B., "Views for Tools in Software Development Environments," *Ph.D Thesis Proposal*, (May 1983).

Gla82. Glass, R.L., *Modern Programming Practices - A Report from Industry*, Prentice-Hall (1982).

Hal85. Hall, J. A., Hitchcock, P., and Took, R., "An overview of the ASPECT Architecture," pp. 86-99 in *Integrated Project Support Environments*, ed. J.McDermid, Peter Peregrinus Ltd. (1985).

Ham81. Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM TODS* 6(3) pp. 351-386 (1981).

Hau82. Hausen, H-L. and Mullerburg, M., "Conspectus of Software Engineering Environments," pp. 462-476 in *Tutorial: Software Development Environments*, ed. A.I. Wasserman, (1982).

Hay85. Hayes, I., "Specifying the CICS Application Programmer's Interface," Technical Monograph PRG-47, Oxford University Computing Laboratory, PRG (July 1985).

Hen76. Henderson, P., "The TOPD System," Technical Report No. 77, University of Newcastle upon Tyne (September 1976).

Hit76. Hitchcock, P., "User Extensions to the Peterlee Relational Test Vehicle," pp. 169-180 in *Systems for Large Databases - Proceedings of the 2nd Very Large Database Conference*, North-Holland (1976).

Hit85. Hitchcock, P., Whittington, R.P., and Robinson, D.S., "Modelling Primitives for a Software Engineering Database," in *Proc. of 4th British National Conference on Databases (BNCOD4)*, ed. A.F. Grundy, Cambridge University Press (10th-12th July 1985).

Hit87a. Hitchcock, P., "A Database View of ASPECT and the PCTE," in *Software Engineering Environments*, ed. P. Brereton, Ellis Horwood (1987).

Hit87b. Hitchcock, P., "An Introduction to Integrated Project Support Environments," *Information and Software Technology* 29(1) pp. 15-20 (Jan/Feb 1987).

Hof85. Hoffnagle, G.F. and Beregi, W.E., "Automating the Software Development Process," *IBM Systems Journal* 24(3) pp. 102-120 (1985).

Hum85. Humphrey, W.S., "The IBM Large-Systems Software Development Process: Objectives and Direction," *IBM Systems Journal* 24(2) pp. 76-78 (1985).

Hun87. Hunt, V.R. and Zellweger, A., "Strategies for Future Air Traffic Control Systems," *IEEE Computer* 20(2) pp. 19-32 (February 1987).

Hut87. Hutcheon, A.D. and Wellings, A.J., "ADA for Distributed Systems," *Computer Standards and Interfaces* 6(1)(1987).

Jac83. Jackson, K., "MASCOT," *Systems Designers Plc. Internal Paper*, (1983).

Kat83. Katz, R.H., "Managing the Chip Design Database," *IEEE Computer*, pp. 26-36 (December 1983).

Kat84. Katz, R. H., "Transaction Management in the Design Environment.," in *New Applications of Databases*, ed. Garadin and Gelenbe, Academic Press (1984).

Kay75. Kay, M. H., "An Assessment of the Codasyl DDL for use with a Relational Subschema," pp. 199-214 in *Data Base Description*, ed. B. C. M. Douque and G. M Nijssen, North Holland, University of Cambridge (1975).

Kel85. Keller, A. M., "Updating Relational Databases Through Views," Ph.D Thesis, Stanford University (Feb 1985).

Kem86. Kemper, A., "Abstract Data Types in CIM Databases," 4/86, University of Karlsruhe (March 1986).

Ker81. Kernighan, B.W. and Mashey, J.R., "The UNIX Programming Environment," *IEEE Computer* **14**(4) pp. 12-22 (April 1981).

Ket87. Ketabchi, M.A. and Berzins, V., "Modeling and Managing CAD Databases," *IEEE Computer* **20**(2) pp. 93-102 (February 1987).

Kim85. Kim, W. and Batory, D. S., "Modeling Concepts for VLSI CAD Objects," *ACM Transactions on Database Systems* **10**(September 1985).

Laf79. Lafue, G. M., "An Approach to Automatic Checking of Integrity in a Design Database.," Ph.D Thesis, Carnegie-Mellon University (1979).

Leb84. Leblang, D. B. and Chase, R. P., "Computer-Aided Software Engineering in a Distributed Workstation Enviroment," *Sigplan Notices*, (19th May 1984).

Lil84. Lilien, L. and Bhargava, B., "A scheme for batch verification of integrity assertions in a database system," *IEEE Transactions on Software Engineering* **10**(6) pp. 664-680 (November 1984).

Lor81. Lorie, R. A., *Issues in Databases for Design Applications*, IBM Research Report (1981).

Lyo86. Lyons, T.G.L., "The Public Tool Interface in Software Engineering Environments," *Software Engineering Journal* 1(6) pp. 254-258 (November 1986).

Mai86. Mair, P., *Integrated Project Support Environments - State of the Art Report*, NCC Publications (1986).

McD84. McDermid, J. and Ripken, K., *Life-cycle Support in the Ada Environment*, Cambridge University Press (1984).

McG80. McGuffin, R.W., Elliston, A.E., Tranter, B.R., and Westmacott, P.N., "CADES - Software Engineering in Practice," *ICL Technical Journal* 2(1) pp. 13-28 (May 1980).

Mey85. Meyer, B., "On Formalism in Specifications," *IEEE Software* 2(1) pp. 6-26 (January 1985).

Mor87. Morgan, D., "The Imminent IPSE," *Datamation* 33(7) pp. 60-68 (April 1987).

Myl80. Mylopoulos, J. and Wong, H. K. T., "Some Features of the TAXIS data model," pp. 399-410 in *Proc. 6th Int. Conf. Very Large Data Bases*, (1980).

Ong79. Ong, J., Fogg, D., and Stonebraker, M., "Implementation of Abstraction in the Relational Database INGRES," *ACM. SIGMOD* 14(1) pp. 1-14 (1979).

Osb86. Osborn, S.L. and Heaven, T.E., "The Design of a Relational Database System with Abstract Data Types for Domains," *ACM TODS* 11(3) pp. 357-373 (September 1986).

Ost81. Osterweil, L., "Software Environment Research: Directions for the Next Five Years," *IEEE Computer* 14(4) pp. 35-43 (April 1981).

Ost83. Osterweil, L. J. and Cornell, W. R., "The Toolpack/IST Programming Environment," in *IEEE Softfair*, (July 1983).

Pri87. Price, C., *SAFRA - A Debrief Report*, NCC Publications (1987).

Rob87a. Robinson, D.S., *Private Communication*. October 1987.

Rob87b. Robinson, D.S., *Private Communication.* December 1987.

Rob87c. Robinson(ed.), D.S., "ASPECT - Specification of the Public Tool Interface," aspect/wb/pub/pti/Zspec3.1, Systems Designers PLC. (August 1987).

Row87. Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," pp. 83-96 in *Proc. the 13th Int. Conf. on Very Large Data Bases,* (1987).

Row79. Rowe, L. A. and Shoens, K. A., "Data Abstraction, Views and Updates in RIGEL," *ACM. SIGMOD,* (1979).

Row83. Rowland, B. R. and Welsch, R. J., "Software Development System," *Bell Systems Technical Journal* 62(1)(January 1983).

Sch84. Schiel, U., "A Semantic Data Model and its Mapping to an Internal Relational Model," pp. 373-400 in *Databases - Role and Structure,* ed. P.M. Stocker and Others, Cambridge University Press (1984).

Sch77. Schmidt, J.W., "Some High-Level Language Constructs for Data of Type Relation," *ACM TODS* 2(3) pp. 247-261 (September 1977).

Sef86. Sefton, E., "A Proposal for the Evaluation of the ASPECT IPSE," BAe-WAA-R-RES-SWE-182, British Aerospace Plc. (March 1986).

Sha84. Shaw, M., "Abstraction Techniques in Modern Programming Languages," *IEEE Software,* pp. 10-26 (October 1984).

Som85. Sommerville, I., *Software Engineering (2nd Edition),* Addison-Wesley (1985).

Sta84. Standish, T. A. and Taylor, R. N., "Arcturus: A Prototype Advanced Ada Programming Environment," *ACM SIGPLAN Notices* 19(5) pp. 57-64 (May 1984).

Ste86. Stenning, V., "An Introduction to ISTAR," pp. 1-22 in *Software Engineering Environments*, ed. I. Sommerville, Peter Peregrinus Ltd. (1986).

Ste87. Stenning, V., "On the Role of an Environment," pp. 30-35 in *Proceedings of the 9th International Conference on Software Engineering*, (March 1987).

Sto86. Stonebraker, M., *The INGRES Papers*, Addison-Wesley (1986).

Stu83. Stucki, L.G., "What about CAD/CAM for Software ? - The ARGUS Concept," *SOFTFAIR '83*, pp. 129-144 IEEE, (1983).

Suf85. Sufrin, B., Morgan, C., Sorensen, I., and Hayes, I., *Notes for a Z Handbook Part 1 — Mathematical Language*. July 1985.

Sys84. Systems_Designers, *DEC/VAX Perspective Technical Overview*. November 1984.

Sys87. Systems_Designers, *PCTE: A Basis for a Portable Common Tool Environment - Ada Functional Specification*. 1987.

Too86. Took, R., "The Presenter - a Formal Design for an Autonomous Display Manager for an IPSE," pp. 151-169 in *Software Engineering Environments*, ed. I. Sommerville, Peter Peregrinus Ltd. (1986).

Tsi78. Tsichritzis, D.C. and Klug, A., "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems," *Information Systems* **3** pp. 173-191 (1978).

Tsi82. Tsichritzis, D. C. and Lochovsky, F. H., *Data Models*, Prentice-Hall (1982).

Ull80. Ullman, J. D., *Principles of Database Systems*, Computer Science Press (1980).

US.85a. US.DOD, "Common APSE Interface Set (CAIS)," Proposed MIL-STD-CAIS (January 1985).

US.85b. US.DOD, *Requirements and Design Criteria for the Common APSE Interface Set (RAC).* September 1985.

Vel83. Veloso, P.A.S. and Furtado, A.L., "View Constructs For the Specification and Design of External Schemas," pp. 637-650 in *Entity-Relationship Approach to Software Engineering*, ed. C.G. Davis, S. Jajodia, P.A. Ng, and R.T. Yeh, North-Holland (1983).

War86. Warboys, B.C., "IPSE 2.5," pp. 221-230 in *Proc. Joint IBM/University of Newcastle upon Tyne Seminar*, (2nd-5th September 1986).

Was79a. Wasserman, A. I., "USE: a Methodology for the Design and Development of Interactive Information systems," pp. 31-50 in *Formal models and Practical Tools for Information systems design*, ed. H. J. Schneider, North Holland, University of California, San Francisco (1979).

Was79b. Wasserman, A.I., "The Data Management Facilities of PLAIN," *Proceedings of ACM SIGMOD International Conference on the Management of Data*, (1979).

Wie86. Wiederhold, G., "Views, Objects, and Databases," *IEEE Computer*, pp. 37-44 (December 1986).

Wil80. Wilson, G.A., "A Conceptual Model for Semantic Integrity Checking," pp. 111-125 in *Proceedings of 6th Conference on Very Large Databases*, (1980).