Developing & Integrating Tools In Eclipse/PCTE

A Dissertation Presented in Fulfillment

of the Requirement for the M.Sc. Degree

15th May 1990

Sean P. MacRoibeaird B.Sc.

School of Computer Applications, Dublin City University

Supervisor : Prof. A. Moynihan

Declaration

This dissertation is based on the author's own work. It has not previously been submitted for a degree at any academic institution.

Sean P. MacRoibeaird

15th May 1990

Acknowledgements

I would like to thank my supervisor, Prof. Tony Moynihan, and my project manager, Mr. Barry Walsh, for their guidance and advice throughout this work. I would like to thank my parents without whose help I would never have stayed on the straight and narrow. I would particularly like to thank my fellow post-graduate students, Steve, Jimmy, Katharine, and the others, for trying to take me off the straight and narrow.

Sean P. MacRoibeaird

15th May 1990

Abstract

"Developing and Integrating Tools in Eclipse/PCTE"

by Sean MacRoibeaird

The whole area of software engineering environments is an emerging one. Such environments have become necessary due to the rapid changes which have occurred in the software industry in the last twenty years. The desire is to produce products of high quality and at a reasonable cost. Unfortunately history shows that, in general, software systems rarely met the specific need for which they were developed and were often unreliable, inefficient , poorly documented and required considerable maintenance. One of the main areas of research into increasing both the productivity and the quality of software has been the use of software engineering environments. The area of software engineering environments is a changing one with evolving definitions. What can be stated is that a key objective of software engineering environments is the support of software process from requirements definition through to system maintenance. Such support can only be provided through the development of integrated sets of tools each supporting various aspects of the software development process. In order for tools to be fully integrated and have the same 'look and feel' it is necessary that they are developed on a common platform, providing all the facilities needed for tool development and integration. Such a platform is the Eclipse tool builder's kit based on the Portable Common Tool Environment (PCTE).

The work in this thesis was based on an evaluation of this development platform for developing and integrating software tools, particularly real-time telecommunications software tools. The work in this thesis was carried out as part of the European Community's RACE programme. The project was called SPECS [1]. The SPECS project is outlined in chapter one of this thesis along with a brief history of the research into software engineering environments to date. The work which I was responsible for involved both the integration of existing toolsets and tools, developed by other partners in the SPECS project, as well as the development of new "native" tools within Eclipse/PCTE. This work was necessary so that the SPECS project could produce an integrated set of tools at the end of its research. It was my job to evaluate the potential of Eclipse/PCTE as a basis for this integration.

[1]SPECS - Specifications and Programming Environment for Communications Systems

Table of Contents

Chapter 1

Background to the Evaluation of Eclipse/PCTE

1.1. Introduction

The work described in this thesis was carried out as part of the SPECS[2] project, which is part of the European Community's RACE[3] programme. Dublin City University (DCU) is a partner in the SPECS project and one of its allotted tasks was to evaluate the Eclipse tool builder's kit. It was for this purpose that I joined the SPECS team at DCU. What exactly Eclipse is, and what it is used for, will be discussed at length throughout this thesis.

In chapter 2 and 3 I discuss the architecture of PCTE and ECLIPSE in detail because of the lack of widespread knowledge about the whole area of SEEs and the tool builder's kits upon which they are developed. I spent a considerable amount of time, at the start of my work with the Eclipse tool builder's kit, trying to come to terms with the architecture of both PCTE and Eclipse. This was a non-trivial task as the standard of documentation was very poor. A lot of experimentation was required and it was through this experimentation and the subsequent tool development and integration that I was able to extract the salient architectural features of PCTE/Eclipse. In order to discuss the work performed using the Eclipse tool builder's kit it is necessary to have some familiarity with the underlying PCTE/Eclipse architecture. It was with this intention in mind that I wrote these chapters.

In chapter 4 I discuss the usefulness of Eclipse in integrating tools which were not developed specifically for Eclipse. I discuss the integration which I performed on two tools with widely differing characteristics. For each tool I discuss the features which either aided or hindered the tool's integration.

In chapter 5 I discuss the usefulness of Eclipse as a platform for developing integrated toolsets. In order to do this it was necessary to develop a totally new Eclipse tool. I developed a tool to perform the rigorisation of informal network specifications. This tool used all the major features of Eclipse and, thus, it was possible to assess the usefulness of Eclipse as a tool building platform.

---

[2]SPECS - Specification and Programming Environment for Communications Systems

[3]RACE - Research into Advanced Communications for Europe

In chapter 6 I outline the conclusions I came to when working with Eclipse over a long period. I detail the areas which I feel are of a high standard as well as those areas which need improvement. I also discuss the future enhancements which are envisaged in furture releases of Eclipse.

1.2. Overview of SPECS

The pre-competitive RACE programme is designed to lead to integrated broadband communications (IBC). The software for IBC is expected to be evolutionary, complex, extensive but conforming to standards to allow the partitioning of IBC into components [Reed]. Because telecommunications systems are becoming more complex, the trend has been to use more formal languages at higher levels of design to enable the development of these systems. Formal software engineering requires a change in approach and in particular an integrated environment including methods and supporting tools covering the process from requirements and specifications through design, implementation, test, execution, design error detection and correction, modification and enhancement.

Within the RACE programme there are three interrelated projects ,ARISE, SPECS, and IOLE, which will develop a Programming Infrastructure (PI) to provide a consistent set of methods and tools to increases programming productivity and quality. The SPECS project has the objective to provide the maximum automation of the whole software process from requirements right through to the maintenance of the developed system. The basis for the methodology is the application of formal methods. SPECS is a 300 person year project due to finish in December 1992.

The outcomes of using the SPECS methodology are as follows :

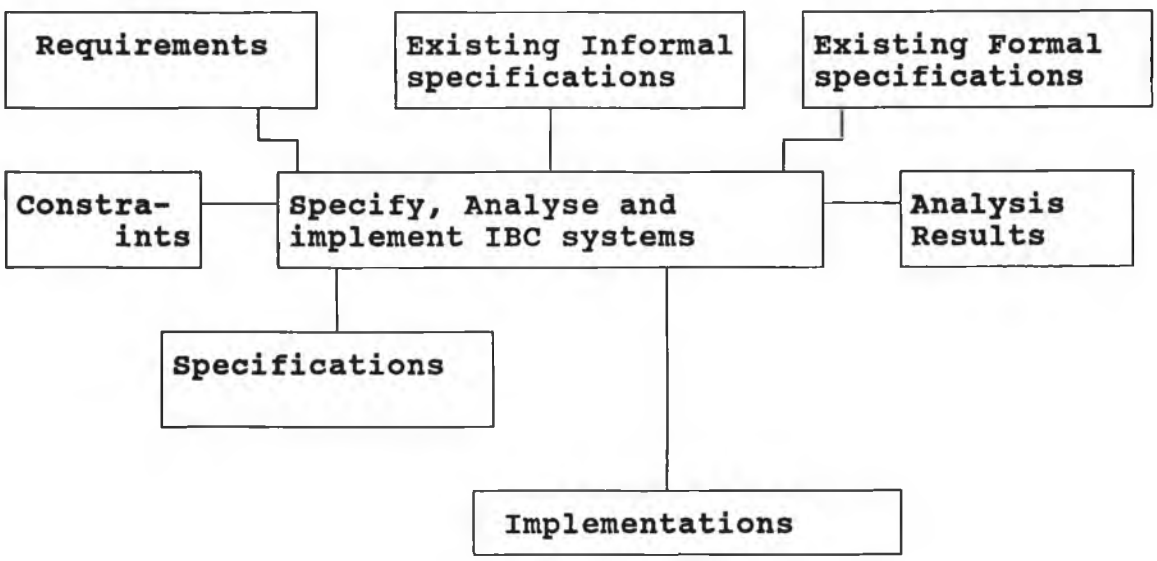
[1] ... System specifications, and reusable system specifications parts in a mixture of formal and informal languages.

[2] ... Implementations of systems to be executed on IBC nodes.

[3] ... Analysis results associated with specifications and implementations. These are analysed properties of specifications, and results from testing and certification of systems.

Diagram 1.1. shows an abstract functionality view of SPECS.

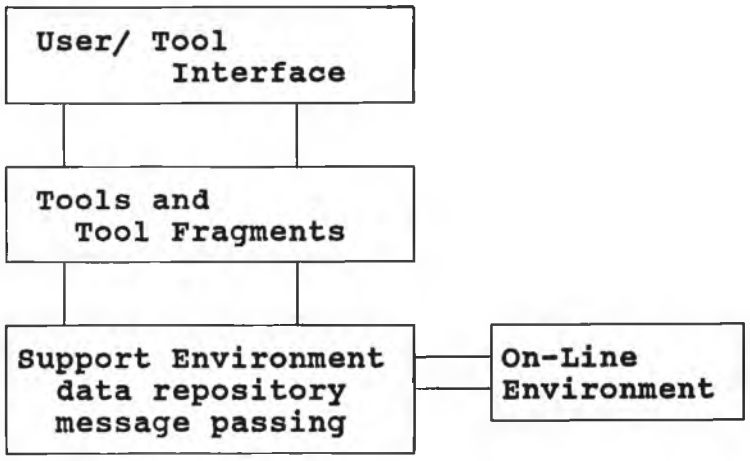
Diagram 1.1.



SPECS Abstract Functionality View

The SPECS methodology is supported by tools. The SPECS tools, themselves, depend on the PI[4] support environment for interfacing with the user, communication between the tools and interaction with a data repository. The primary function of such an environment, from the point of view of the SPECS tools, is to provide user interfacing and environment support as shown by the three layers shown in Diagram 1.2.

Diagram 1.2.



Architecture Layers

In SPECS there are two support environments under evaluation, Eclipse and a Lisp-based environment called Concerto. I was responsible for the evaluation of Eclipse to see how well it could provide the underlying functionality required by the SPECS tools. The SPECS environment is envisaged as providing the following facilities.

[1] ... Users interact with the system through a uniform user interface layer which creates the appearance of a single coherent environment rather than a hetrogeneous tool box.

[2] ... A collection of interacting tools and tool fragments interacts with the user interface and the data repository.

[3] ... All data relevant to one or more projects resides in a common data respository which has the form of a fine-grain database.

---

[4]PI - Programming Infrastructure

These requirements meant that I was responsible for developing new tools and integrating foreign tools, i.e. tools developed previously on a different platform, e.g. UNIX. These tools were of direct relevance to the SPECS project. This work provided valuable information on the usefulness of Eclipse and formed the basis of this thesis.

1.2.1. Methodology and Activities

Within SPECS formal languages will be used as much as possible and not just during specification. The user view should be at the highest level possible so that the objective is that the user works, essentially, at the formal specification language level. The use of formal methods, on a large scale, requires tools to support them. The primary objective of SPECS is to develop a methodology based on formal methods for telecommunications software. In SPECS, a methodology is characterised by -

[1] ... the perspective (functional, object-based ...)

[2] ... the language used

[3] ... the tools used

[4] ... the organisation of activities and roles.

SPECS uses formal languages to support the following activities -

Specify - This activity turns functional requirements and constraints into functional specifications.

Analyse - This activity involves analysis of both static and dynamic properties of specifications.

Implement - This activity involves both design of an implementation and actual code production.

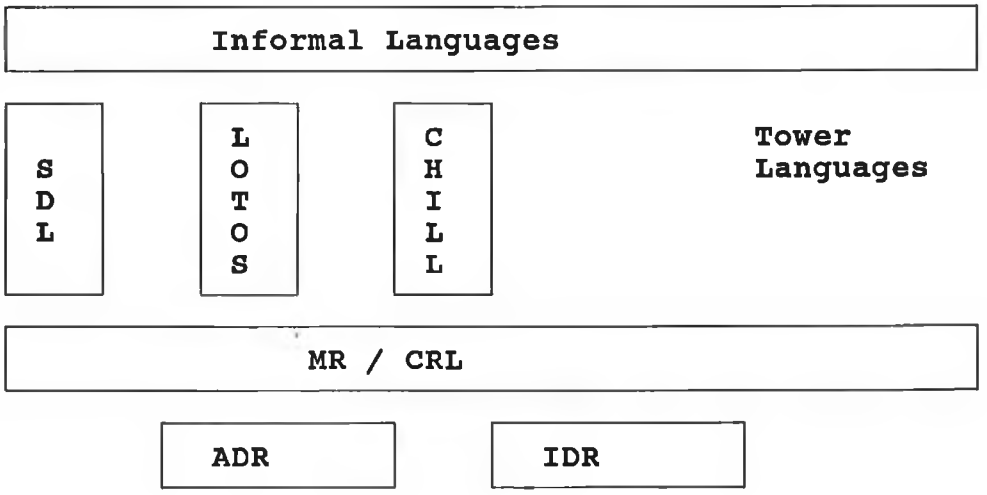
1.2.2. Architecture

The foundation of the whole architecture is a mathematically sound model to represent user languages. User specifications are translated into a machine representation of the mathematical model. Target code generation, simulations, analyses and comparisons are performed on the

mathematical model.

Formalisation tools give support to the method of translating the input into the languages supported by SPECS. Initially the languages supported by SPECS are CCITT SDL, LOTOS and a subset of CHILL. These are called the **tower languages**. These languages are translated into a mathematical representation (MR), which is a superset of the tower languages. The MR is implemented as a machine readable form called the Common Representation Language (CRL). CRL also contains **hooks** to information which cannot be represented in MR. The CRL can be analysed for consistency, deadlocks, etc. CRL can be translated either into an Analysis Derived language (ADR), e.g. Petri-Nets, for which analysis tool exist, or into an Implementaion derived Language (IDR), e.g. C, so that target code can be derived. Diagram 1.3., below, shows the layered relationship between the SPECS languages.

Diagram 1.3.



The SPECS Languages

Thus the starting point in the development of IBC using the SPECS methodology is a set of requirements. Intially these requirements will be stated in free-format language. These requirements will form the input to tools which will rigorise them into a more usable form, e.g. Context Diagrams, Data-Flow Diagrams, etc. These rigorous specifications will then form input

to tools which will generate formal specification languages, i.e. the Tower Languages. Again these Tower Languages will form input to tools which will generate the SPECS developed Common Representation Language. CRL will form the input to tools which either generate ADRs, for analysis purposes, or IDRs for implementation purposes. It is the intention of the SPECS methodology to make this whole process going from requirements through to implementation, as automatic as possible , requiring the minimum amount of user-intervention.

1.3. Background to Software Engineering Environments

The software industry has changed significantly in the past twenty years. Because the software industry is basically a manufacturing industry [Sten], which produces anything from individual programs to complete standalone systems, the desire is thus to produce products of high quality and at a resonable cost. By the mid 1960's software systems began to reach a size and complexity which made them hard to understand [Pene1]. This resulted in a situation where software systems rarely met the specific need for which they were developed. They were often unreliable , inefficient , poorly documented and required considerable maintenance for the duration the system's use.

Thus the overall objective is to achieve some combination of higher quality, lower cost, and greater predicatability, i.e. the ability to deliver a product that meets its requirements on time and on budget [Pene1]. In the 1970's several areas were studied with a view to attaining the above objectives.

[1]	...	New Techniques	,e.g. top-down design
[2]	...	New Technical Methods	,e.g. *Jackson Structured Programming*
[3]	...	New Management Methods	,e.g. *Usage of Software Life Cycle*

At the time there was little attention paid to integrating the various tools which were developed in these areas. Thus the tools developed were standalone and automated specific parts of the

software development process. There was no concept of support for the whole software development process effectively coordinating the various major aspects of the process. Indeed , until very recently software has been developed predominantly on large centralised computer systems using a collection of tools bearing little or no relationship to one another. Thus there was little general support given to the software engineer during the software development process. It was in the mid-1970's that the concept of software engineering environments (SEE)[5] appeared in an attempt to focus on the benefits that could be achieved by combining /integrating the above techniques.

It is recognised that the availability of such integrated environments is crucial for improving the productivity of the software industry. It is also recognised that the environments should not be tied to a particular hardware platform or operating system. The ultimate goal for a user is a hetrogeneous environment where the presentation and usage of all tools and methods is the same, regardless of the hardware platform or operating system.

The rest of this chapter will outline, in more detail, the exact nature of SEEs, the current status of SEE developments and a taxonomy of conceptual SEE architectures. The second chapter will then discuss the architectures of both PCTE and Eclipse and how they relate to the conceptual architectures discussed in this chapter.

1.4. Definition of an SEE

The environments research area covers many topics such as structure-oriented editing, graphics tools, incremental algorithms, distributed tools that support software development and mechanisms for handling system evolution [Notk]. Given such a broad spectrum of topics and the fact that the concept of an SEE is a changing one, in that it is changing with the software engineering community's understanding of the tasks involved in the development of computer systems, it is necessary at this point to define what we mean by an SEE. What follows is a

[5]The terms software engineering environment (SEE) and environment will be used interchangeably in this document.

discussion of the definitions which are currently used by various researchers in this field.

The word **environment** is synonomous with the following terms [Tull] -

integrated project support environment (IPSE)

integrated programming support environment

programming environment

programming support environment

software environment

software engineering environment

support environment.

Over the years there have been various meanings associated with SEE's. In [Dart] an environment refers to the collection of hardware and software tools a system developer uses to build a software system. The distinction is made between a **Programming Environment** and a **Software Development Environment.** A Programming Environment is one which supports programming-in-the small activities, i.e. the coding phase. This encompasses activities such as editing and compiling. A Software Development Environment is one that augments all the activities comprising the software development life cycle, including programming-in-the-large tasks, e.g. configuration management, programming-in-the-many tasks,e.g. project management, as well as supporting large scale, long-term maintenance of software.

In [Tull] an SEE is defined as being a collection of computer-based facilities to support the activities of programmers, software engineers, systems designers , project managers, etc. to achieve higher productivity and higher product quality. It is more than just a collection of tools in that it should be possible to pass information between tools , it should be possible to pass information among project members and support should not be confined to a small range of project activities. Indeed in [Pene2] it is stated that a key objective of SEE's is to support software projects in the generation, management and control of the vast amount of data and associated information which is generated and used during the project life-cycle.

[PCTE88] defines an SEE as an environment in which -

[1] ... the user can hold , in computer processable form, all of the managerial, administrative, and technical information describing and defining the current state of a system and to the extent required , its past history; and

[2] ... tools can integrated that can support the processing of this information, in order to carry out the work of all phases of the development and maintenance of the system.

In [Sten] the fundamental role of an environment is to support the effective use of an effective process. The environment must provide coordination between the various major aspects of the process. The four main aspects regarded as critical are -

[1] ... Technical Development

[2] ... Project Management

[3] ... Configuration Management

[4] ... Quality Assurance

In [Luba] it is stated that an SEE should support the user in the most mundane and mechanical aspects of development, automating those activities where possible. This is particularly pertinent with regard to the acceptance of an SEE since the users must want to use it rather that to try to escape its clutches. They should also provide assistance in the more challenging aspects of design, e.g. reuse of common solutions to subproblems. What follows is a list of seven identified forms of support which should be considered.

Clerical Support - e.g., recording designs with editing facilities for creating, modifying and browsing graphical design representations.

Interface Support - facilitates the communication between the user and the environment, e.g. graphics windows , menus.

Analysis Support - for such things as evaluating design quality and metrics.

Testing Support - for constructing test cases , prototypes.

Organisational Support - for keeping track of design goals, objectives ,etc.

Knowledge-based Support - for providing expert design knowledge to less experienced system designers.

Intelligent Support - for assisting in exploration and other creative design activities.

Common to all definitions is a concern for supporting individuals, task groups, and project teams. The SEE should support project management as well as software process activities; encourage, and perhaps enforce, good practice; increase the general quality of software systems; increase the productivity of software related personnel and provide management and customer visibility into a project's progress and an insight into the properties of the project's eventual results.

1.5. Current Status of SEE Developments

In order to better understand the context within which the evolution of SEE's is taking place it may be helpful to briefly outline the present state of development of SEEs. A suggested taxonomy might look as follows [Dart] -

[1] ... Language-Centred Environments

[2] ... Structure-Oriented Environments

[3] ... Toolkit Environments

[4] ... Method-Based Environments

1.5.1. Language-Centred Environments

Language-centred environments are ones in which the operating system and tool set are specially built to support a specific language. An example would be *Rational* for Ada. Language-centred environments encourage an exploratory style of programming to aid the rapid production of software. Programs can be developed interactively in increments, allowing the user to experiment with software prototypes. Because high-level languages do not adequately support the activities involved in constructing large systems, language-centred environments have added facilities to support **programming-in-the-large**.

1.5.2. Structure-Oriented Environments

Structure-oriented environments incorporate techniques that allow the user to manipulate structures directly. Examples of structure-oriented environments are *Cornell* and *Mentor*. The initial motivation for structure-oriented environments was to give the user an interactive tool - a syntax directed editor. Such an editor allows the user to enter programs in terms of language constructs. It is this editor which forms the core of such environments, providing the interface through which the user interacts and through which all structures are manipulated. The contributions of such environments are as follows -

[1] ... Direct manipulation of program structures.

[2] ... Incremental checking of static semantics.

[3] ... Semantic information is available to the user.

[4] ... Ability to describe the syntax and static semantics of a language.

1.5.3. Toolkit Environments

Toolkit environments consist of a collection of small tools and are intended to support the coding phase of the software development cycle. This approach starts with an operating system and adds coding tools such as a compiler, editor, etc., as well as tools to support large-scale software development tasks such as version control. The original motivation was the need to be language-independent while supporting programming-in-the-large activities. Examples are *PCTE* and *Arcadia*.

1.5.4. Method-Based Environments

Method-based environments incorporate support for a broad range of activities in the software development process. Each one supports a particular method for developing software. These methods are generally either development methods for particular phases of the software development cycle or methods for managing the development process. Different methods exhibit different degrees of formality, i.e. a method may be informal, as in written text; semiformal, as in textual and graphical descriptions; or formal, with an underlying theoretical model against which a description may be verified. A number of tools for single users have become available. Instead of encoding particular methods in these tools, their developers have engineered them to be more general purpose. Instances of design tools can be created through a tailoring and generation process. In general, method-based environments do provide support to most software development phases, however they can be cumbersome to use and integration across phases of the development process is poor.

Combining these technologies to produce an industrial scale SEE which will support all phases of software development together with programming-in-the-large is a research goal.

In [Stre] there is a description of a fifth type of environment, namely an **Environment Framework** that can create a variety of environments, each tailored to the needs of a particular software development project. Through this framework many of capabilities provided by the four previously described environment types are supported.

1.6. Conceptual SEE Architectures

The previous section outlined a taxonomy of current trends in SEEs. It is also possible to present an architectural taxonomy for SEEs [Pene1]. The main architectures which have been identified are as follows -

[1] ... Virtual Machine Architectures

[2] ... Network Architectures

[3] ... Data-Centric Architectures

[4] ... Control-Centric Architectures.

1.6.1. Virtual Machine Architectures

These organise the components of an SEE into layers of implementation support with the lower layers supporting the implementation of the higher ones ,e.g. the "onion skin" architecture of PCTE. Because this model is very pertinent to the architecture of Eclipse/PCTE it will be discussed in more detail than the other three architectures. The proposed layers for such an architecture are -

[1] ... User Interface Layer -

This layer provides **environment adaptation mechanisms** used to generate project-specific environments. It also provides **project user support capabilities**, e.g. information which defines user views of the process and data based on roles and/or expertise.

[2] ... Tool/Capability Layer -

This layer contains the components which provide an environment's functionality. It is typically populated with many more tools than will be needed in any particualr environment. There will be **functional tools** to automate the various methods and techniques in support of the software development process. There will also be **tool-building tools** used by environment builders to prepare environment components.

[3] ... Environment Support Layer.

This layer is part of any project-specific environment and provides the infrastructure upon which the environment is constructed. This layer simplifies the construction of the components in the tool layer by providing a set of commonly needed facilities. This layer comprises the following -

[a] ... A Virtual Operating System for providing portability to allow the environment to run on a wide variety of hardware.

[b] ... An Object Management System which supports the storage and retrieval of data, process and tool objects.

[c] ... A User Interface Management System which provides the basic mechanisms for defining user interfaces (windows, graphics, menus, etc.) and associating objects presented at the interface to objects within the environment.

[d] ... An Environment Management System which provides specialised control facilities needed to map user requests to internal activity.

[4] ... Hardware and Native Operating System layer

This is the lowest layer and forms part of any project-specific environment. It consists of the underlying hardware and native operating system and may consist of a hetrogeneous collection of processors and workstations and peripherals.

1.6.2. Network Architectures

Network architectures organise the components of an environment as a network of processes that typically interact by passing messages between one another.

1.6.3. Data-Centric Architectures

In data-centric architectures the environment's data repository forms the core of the environment and the components are organised in terms of flows of data into and out of the database. Most environments based on database management technology display this type of architecture. Such environments structure and organise data through the use of database schemas. Tools take the form of data transformers. Control is effected through the explicit and implicit flow of data among the tools with the database serving as a central data repository.

1.6.4. Control-Centric Architectures

In control-centric environments the internal supervisory subsystem forms the core of the environment and the components are organised in terms of flow of control among them. Most expert system-based environments display this type of architecture. Such environments use knowledge bases to hold information about both the products being produced and the processes used to produce them. Tools take the form of interpreters of rules specifying characteristics and constraints upon products and processes.

1.7. The Central Data Repository

Given the complexity of the software development process, the number of individual items of information that are involved, and the relationships between these items, any environment by definition must have a database[6] [Sten]. The information which is generated during the course of large software projects is complex and varied. The complexity is compounded by the fact that diverse tools are required to act in a cooperative fashion on these objects and the relationships

[6]The terms database and central data repository will be used interchangeably in this section.

between these objects. Thus the choice is not whether or not to have a central data repository but what form it should take.

The data model of such a central data repository needs powerful type constructors to model such objects as programs and program versions as well as fine-grained data such as individual modules and statements [Huds]. Unique SEE database requirements have been and are continuing to be identified, which means that traditional database management systems (DBMS) are unsuitable to act as the central data stores for SEE's. Some of the differences between traditional DBMS's and SEE DBMS's are -

[1] ... The objects being manipulated are typically not as easily represented as simple, flat records.

[2] ... SEE databases do not have the traditional low schema to data ratio of business applications. This means that there are fewer objects of the same type.

[3] ... Identifiable transactions on SEE databases tend to be much longer than typical transactions on a commercial database. A typical transaction might be a program bug fix, which could involve a long, interactive period of update, recompilation and reconfiguration.

[4] ... There tends to be a logical locality of reference, with each user requiring access to only a small part of the complete database over some significant period of time.

Thus it is essential to understand the data structure to be held in the database before the initial installation and subsequent evolution of the database [Sten]. This data structure comprises the types of the data items (which must have the ability to model unusual forms of data), their attributes (which may be structured or complex), their relationships and the rules for consistency. These rules should be explicitly specified in a **conceptual schema** which is at the heart of the SEE [Sten]. For a fully integrated environment it is essential that the conceptual schema is

sufficiently rich, encompassing all information of relevance to the tools.

An essential part of the definition of conceptual schema is the ability to define types and treat persistent objects as instances of these types. From this ability numerous benefits accrue for the accessability of the stored information and for the protection against accidental and malicious application of inappropriate operations to objects. In [Pene2] a hierarchy of object types is proposed. The model is as follows -

level 1 - **Data Model**

Defines the overall data model ,e.g. the *Entity-Relation Model (ERA)* [Chen] which serves as the common framework over which subsequent levels are built.

level 2 - **Specialised Data Model**

Augments level 1 with perhaps some rudimentary semantics built in, e.g. the *CAIS ERA model*.

level 3 - **Data Description Language (DDL)**

Defines the language in which the user-supplied type definitions are expressed.

level 4 - **Schema definition(s) in DDL**

Utilises the DDL to provide the schema definitions that describe the specific types of objects, their properties, and interrelations.

level 5 - **Instances of types**

Deals with the representation of objects as instances types defined at level 4.

The central data repository forms the basis of tool integration; the various tools and facilities must all operate on a single common data structure as defined by the conceptual schema.

Another related topic which is extremely important for SEEs is the ability to accommodate new facilities and services. Thus the SEE must be **open**. If the tool was developed outside the SEE

it must be able to be integrated to some extent into the SEE otherwise it would be necessary to re-implement all foreign tools of interest. The ability to incorporate **foreign tools** can ease an organisation's transition into use of the SEE since familiar tools can remain available [Sten]. Also important to this transition is the ability to transfer existing data into the SEE database.

Chapter 2

Architecture of PCTE

2.1. Introduction

As indicated earlier, software development and maintenance has been and continues to be, an expensive activity, prone to costly delays and failures, with results that rarely fulfill the true requirements of the customer. Industry, in general , needs to reduce and control these costs and risks.

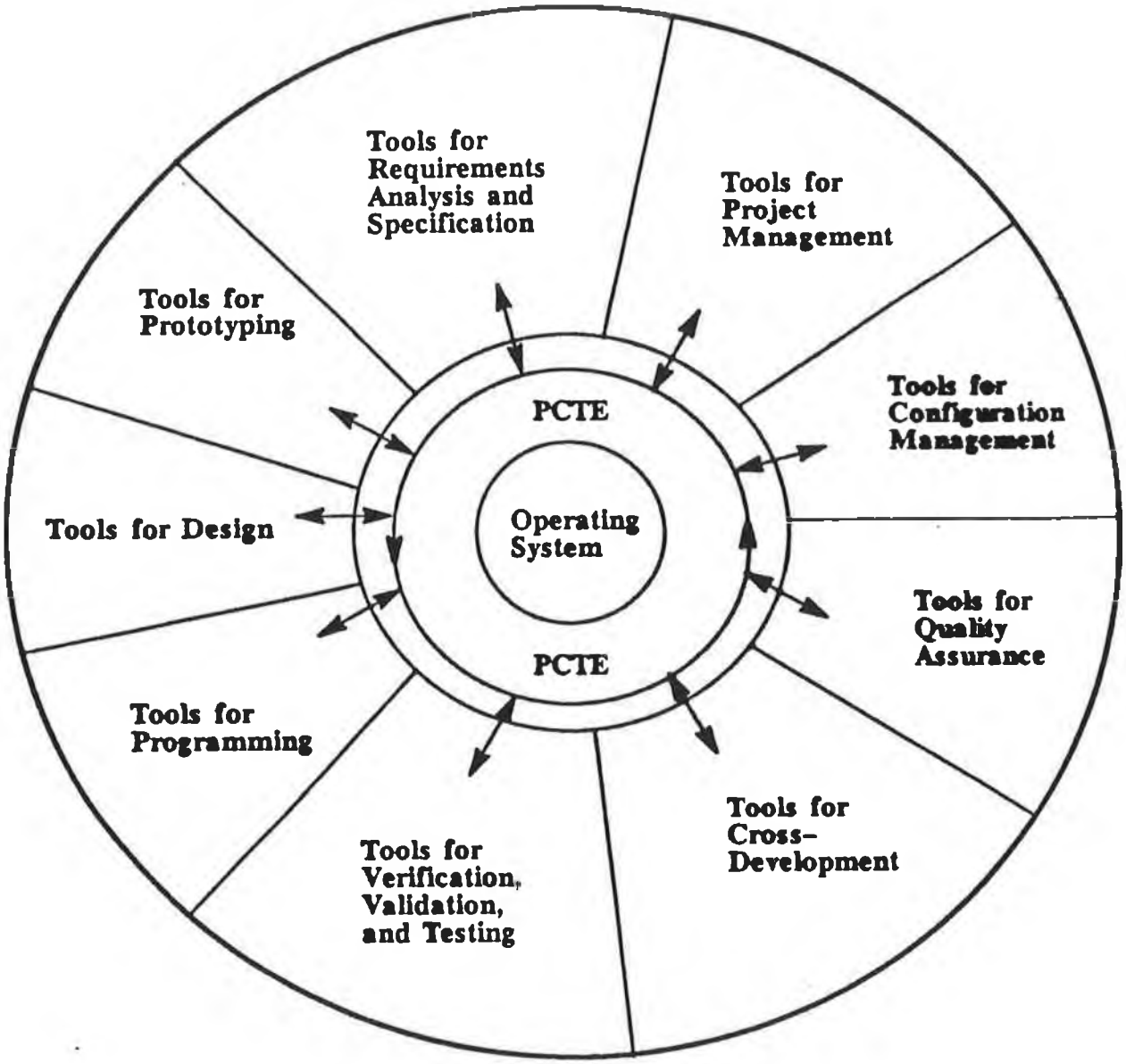
A hardware configuration of powerful, distributed workstations with graphics capabilities has been determined to be a productive development platform. In addition , it has been recognised that better control and productivity gains can best be achieved by increasing the number of **CASE**[7] tools available, and by establishing relationships between tools [PCTE88]. It was against this background that the PCTE project was launched in October 1983 under the auspices of the **ESPRIT** research and development programme.

The PCTE program approached the problem of efficient tool integration for CASE by factoring out those common features required by most tools for information management and interaction with the tool users [Thom]. The original PCTE project was intended to produce a definition of a public tool interface and to prototype implementations of different aspects of this interface. Version 1.4. of the Functional Specifications of PCTE interfaces [PCTE86] was published in August 1986. The actual implementation used was the Emeraude developed, industrial quality implementation of the PCTE interfaces. There were two versions used; the 10/3 version being replaced by the 10/4 version in May 1989.

[7]**CASE -** Computer Aided Software Engineering

Diagram 2.1., below, presents a conceptual view of the structure of an SEE developed on PCTE.

Diagram 2.1.



2.2. General Overview of PCTE

It is clear that PCTE itself is not an SEE but provides a platform upon which an SEE can be constructed by the integration of the appropriate tools supporting all aspects of the software development process.Before defining the architecture of PCTE the following statements can be made about the exact nature of PCTE [PCTE88] -

[1] ... PCTE is a kind of Operating System

PCTE hides the details of the underlying operating system and hardware, and provides a set of useful, commonly needed facilities to all CASE tools. Tools which run upon one implementation of PCTE will run upon any other, even where such implementations run on different hardware and operating systems.

[2] ... PCTE is a kind of Database Management System

PCTE implementations contain an **Object Management System (OMS)** which is a specialised database for holding all kinds of technical, administrative, and managerial information describing the state of the system being developed or maintained. The OMS will hold this information in a form which allows it to be accessed and manipulated by cooperating CASE tools.

[3] ... PCTE is distributed

Data and tools are distributed around the network. Data moves from workstation to workstation or is accessed from other workstations as the need arises. All of this is transparent to the users. Tool writers need not be aware of the characteristics of the particular network, nor do they need to concern themselves with the manner in which the distributed facilities are constructed. They write their tools as if all the data were held on a single machine.

In the model architecture of PCTE there is a clear distinction between the tools and the

underlying structure that hosts them [PCTE86]. The public tool interface to the PCTE services is defined by a set of program-callable primitives which support the execution of programs in terms of a virtual, machine independent level of comprehensive facilities.

The contents of the PCTE Interface Definition is broken down into two logical sections,i.e

[1] ... Basic Mechanisms

[2] ... User Interface

Both of these sections contain several distinct sub-sections which will be discussed in the following sections. Because the theme of this thesis is the Development and Integration of Tools into the Eclipse Tool Builder's Kit[8] it would serve no real purpose to go into a detailed discussion on the aspects of PCTE architecture which are not relevant to Eclipse and so only those areas that are pertinent to the tool-builder using Eclipse will be discussed in detail.

The PCTE basic mechanisms correspond to the functionalities required to manipulate the various entities that can exist in a development context [PCTE86]. These entities are essentially programs that can be executed and the various objects that are manipulated by the programs,e.g. the various representations of the programs being developed, documentation, input and output data, etc.

There are six categories within the PCTE basic mechanisms. They are as follows -

[1] ... Object Management System (OMS)

[2] ... Execution (EXE)

[3] ... Inter-Process Communication (IPC)

[4] ... Concurrency and Integrity control (ACT)

[5] ... Communication (COM)

[6] ... Distribution (DIS)

[8]The terms Eclipse Tool Builder's Kit and Eclipse will be used interchangeably in this document.

From the point of view of tool development and integration in Eclipse [1] and [2] above are the most important and will be discussed in more detail than the other categories.

2.3. The Object Management System (OMS)

2.3.1. Introduction

The OMS is the data repository of PCTE. A key aspect of an SEE is the set of functions that are provided to manipulate the various objects in the system [PCTE85]. Because the OMS is the most important feature of PCTE with regard to Eclipse it will be discussed in more detail than the other PCTE features. The various agents in the SEE (users and programs) operate on a number of entities that are known to the system and can be designated in it. These entities are globally referred to as **objects** . These objects could be files in the traditional sense, e.g. a document, a structured object like a library of software components or something more abstract like a project. The OMS can be seen as an evolution from the traditional **File Management System**, e.g. the hierarchic structure of UNIX, to a structure that can be adapted to the needs of different environments [PCTE86].

2.3.2. Schemas

A key concept in the OMS is that of the **schema**. The schema is a means of integrating tools around commonly accessed data structures. The type definitions making up the overall OMS schemas are organised into a collection of small sets of definitions called **Schema Definition Sets (SDS).** Each SDS is a partial view of the whole OMS schema. Changes to the schema are always carried out as updates to an SDS. PCTE provides a **Data Definition Language (DDL)** as a schema definition formalism [PCTE86]. Examples shown in the following sections will demonstrate the use of DDL.

Another key concept is that of the **Working Schema** each of which is made of one or more SDS. A working schema is a partial view over the overall schema. Working schemas allow the coexistence of several, otherwise conflicting views of the database.

2.3.3. Objects, Attributes, Relationships and Links

The OMS provides facilities for managing a distributed object base based on an **Entity-Relationship (E-R)** model [Chen]. From this model PCTE defines **objects** to be E-R entities which can be distinctly identified, e.g. a file, tool, program library. PCTE objects can be optionally characterised by -

Contents - a repository of unstructured data implementing the traditional UNIX file concept.

Attributes - a set of primitive values which can be named and typed individually.

Relationships - allow the representation of logical associations/dependences between objects.

Objects are classified into different object types where objects of a given type have the same characteristics , as defined above. PCTE defines a **type hierarchy** in which an object can be defined to be a subtype of another one, thus inheriting all the parent object's characteristics. An object type is characterised by -

[1] ... a name

[2] ... a parent type

[3] ... a set of attribute types

[4] ... a set of link types for which it is a valid origin

[5] ... a set of links for which it is a valid destination

In the example SDS in **Appendix A** there are a number of examples of object typing. The idea of the SDS shown is to define a schema to provide a simple C programming environment within the OMS. Example 2.1. below shows the DDL code required to define a C source object which has certain attributes and links. Thus **c_source** object is defined to be a subtype of **file,** i.e. instances of this object can have contents.

Example 2.1.

```
c_source    :       subtype of file;
    .
    .
extend c_source
            with
            attribute
                    f_time;
                    f_size;
            link    cc;
                    inc;
end c_source;
```

Attributes are defined for both PCTE objects and PCTE links. A **link** is a uni-directional association between an origin object and destination object. An object attribute defines an intrinsic property of an object. Thus in the example above the C source object is defined to have two attributes **f_time** and **f_size**. The types of these attributes can be seen in Appendix A. A link attribute qualifies a link and indirectly the object which is the destination of this link. An example from Appendix A of a link attribute is shown below in example 2.2.

Example 2.2.

```
cc      :       reference link (name) to object_file
        with
        options;
         optimize;
         debug;
end cc;
```

In this example it can be seen that the reference link cc has three attributes which are relevant to a specific compilation. The types of these attributes can be seen in Appendix A.

An attribute type is characteristics by -

[1] ... **a name**

[2] ... **a value type** (integer, boolean, date, string)

[3] ... **an initial value**

A link can be used to create a simple reference from an object to another one, or to model a particular structure, e.g. a hierarchical directory structure. Designations of links is the basis of designation of objects [PCTE85] since the principal means of accessing objects is to navigate

the OMS object space by traversing a series of links in the form of a **pathname**. The link type defines the characteristics of all its instances, i.e. the links.

A Link is characterised by -

[1] ... **a name**

[2] ... **a cardinality** - one-to-one, one-to-many links

[3] ... **a category**

[a] Composition Link - The creation of an object requires the creation of a link starting from an origin object and leading to the new object. This link must have the category **composition.**

[b] Reference Links - This category of link is used to refer to any pre-existing object. As long as the link exists the object cannot be deleted.

[c] Implicit Links - This category is specified for one direction of a relation (a pair of links such that the origin of each is the destination of the other)type and is created/deleted as a side effect of the creation/deletion of a reverse link.

[4] ... **a stability property** - As long as the link exists, the destination object can neither be deleted nor updated.

[5] ... **a set of origin object types**

[6] ... **a set of destination object types**

[7] ... **a list of key attribute types** - A key is necessary for cardinality many links.

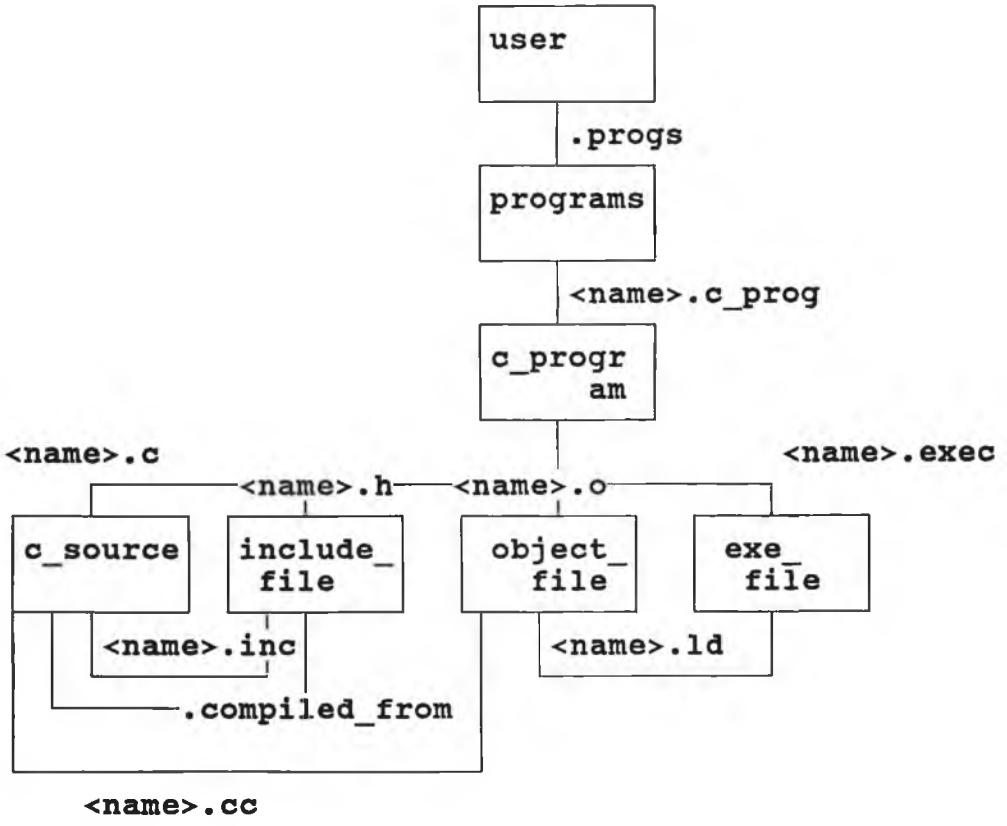
[8] ... **a set of attribute types** (see Example 2.2.)

A PCTE object is deleted when there are no more composition links leading to it.

2.3.4. Description of Example Schema Definition

As stated previously, the purpose of the DDL example in Appendix A is to model an SDS of a simple C programming environment. A diagrammatic representation of the underlying data model would be as follows -

Diagram 2.2.



The Data Model for the C Programming Environment

The schema definition shown in Appendix A can be compiled and automatically installed as a special PCTE object within the OMS. This means the it can be included in a working schema as follows -

+# setsch c_sds env

If this SDS is included in a working schema it makes all the link and object types defined within it visible. It is thus possible to create instances of these link and object types. An example of how one would create a **c_source** object would be as follows -

+# crobj -t c_source new_file.c

The command **crobj** creates an instance of the object type following the **-t**. In order to create the required object one must supply the path to the object, i.e. **new_file.c**. If the schema definition is checked it can be seen that a **c** link is a cardinality-many (keyed) composition link to a c_source object. The above command presumes that we have created the parent objects of the c_source object and that we have navigated to a **c_program** object, c.f Diagram 2.2. Such a navigation would look as follows -

+# cd _/.users/eclipse.usr/.progs/test_project.c_prog

This navigation path is broken down as follows -

_/.users/eclipse.usr	-	This is the path to a particular **user** object (Diagram 2.2.) as defined by the **env** system supplied schema.
.progs	-	A cardinality-one (one-to-one) link from a user object to a **programs** object.
test_project.c_prog	-	**c_prog** is defined to be a cardinality-many link to a **c_program object**. This link has to be keyed by a string so as to navigate to a particular c_program object, i.e. the test_project.c_prog link when traversed will lead to a particular c_program object.

2.3.5. Description of Example Tool

Appendix B contains the source code of the small tool written to use the schema definition shown in Appendix A. All the PCTE interfaces implemented by Emeraude are supplied in one library called **libemer.a** which must be linked in with the tool-builder's code. As well as operating on the link and object types defined in the schema definition shown in Appendix A, the tool also operates on the set of system attributes which are define for all object types in the OMS and which can be retrieved by using the **getobjstat** library function. Such attributes would hold information such as the current size of a object or the date of last update.

The tool stores the time of last compilation in the **f_time** attribute of a c_source object. It stores the size of the c_source object at the time of the last compilation in the **f_size** object. Thus, the value for f_size is compared with the system attribute containing the current size of the c_source object. If they are different then the source file has been modified since the last compilation. The other check requires that each include file referenced by a source file is checked to see if it has been modified since the last compilation. This is done by navigating all the **.Inc** reference links starting from the c_source object. The f_time attribute is compared with each include file's system attribute showing the time of its last update. If any of the include files have been more recently updated than the source file then a recompilation is required.

2.4. PCTE Execution Mechanisms (EXE)

The PCTE execution mechanisms cover the various notions and functionalities that deal with active entities, i.e. programs and processes [PCTE86]. PCTE defines the notion of the **static context** for executing programs. The basic constituent of a program is its load module. In PCTE a load module is not confined to be binary, compiled code, but can include any internal or external representation that can be executed by a suitable program.

Two levels of execution starting must be distinguished in PCTE, i.e. starting another process independently of what that process has to do or starting the execution of a given program either within the current process, or as a separate process.

The Emeraude implementation is based on UNIX and as such provides upward compatability for UNIX tools. Thus, PCTE preserves the UNIX primitives for spawning new processes and for starting the execution of a given program in a process. In addition PCTE defines two additional primitives which are -

CALL - creates a new process that executes the program and blocks the calling process until completion of the execution of the invoked program.

START - This is similar to CALL but does not wait for the called program to terminate. The parameters are the same as in CALL. An example of a START call can be seen in the **compile_program** function in Appendix B. The call to **startl** invokes the UNIX C compiler with the C source file object as a parameter.

2.5. PCTE Communications Mechanisms (COM)

These mechanisms deal with the primitives to operate on the contents of a given object, be it normal, i.e. a file, or special, e.g. a pipe. The primitives here are exactly as those provided by UNIX. Given an OMS object specified by a pathname, one can open the object contents for reading, writing or both returning a file descriptor as in UNIX. This area of PCTE is totally changed under ECLIPSE which allows the definition of object types which can have structured contents, i.e. one can specify a schema for the contents of an object which results in a **two-tier entity-relational database.**

2.6. PCTE Interprocess Communications Mechanisms (IPC)

The mechanisms used in PCTE are derived from primitives used in UNIX system V. There are three primary IPC mechanisms -

[1] ... **Pipes -** establish cooperation between two processes which may not be programmed specifically as cooperating processes, by appearing as normal files.

[2] ... **Signals -** send asynchronous signals to a given process, and indicate a response to a given signal as a procedure to be executed when that signal is received.

[3] ... **Messages -** Message queues are considered as the primary means to achieve interprocess communication. The basic concept is that of a **message queue**. Each queue has an identifier which can be distributed over the LAN.

The problems which the IPC mechanisms must address are as follows [PCTE86] -

[1] ... they must allow allow close cooperation between related, cooperating processes.

[2] ... they must be general, i.e. provide a system-wide facility even when the system is distributed.

[3] ... they must be efficient, both for implementation and use.

2.7. PCTE Concurrency and Integrity Control Mechanisms (ACT)

The aims of the PCTE Concurrency and Integrity controls are as follows [PCTE86] -

[1] ... to ensure consistency of data access operations.

[2] ... to support the maintenance of integrity of accessed information by the facilities allowing the grouping of a sequence of logically related actions as a single, atomic transaction.

[3] ... to work transparently for users and tools not concerned by concurrency control.

The elementary information item seen by the concurrency control mechanisms is called a **Resource** which is either an OMS object with its contents and its set of attributes or a link and its attributes. A resource is operated on when the resource is an object whose contents are open or when any other, self-contained, operation is in progress. An **Activity** is the framework within which a set of operations takes place and an operation is always carried out on behalf of one activity. There are three classes of activities in PCTE -

[1] ... **Unprotected Activities** - Do not require their data accesses to be protected from other concurrent activities.

[2] ... **Protected Activities** - Require their data accesses to be protected from other concurrent activities. However their effect is not required to be atomic.

[3] ... **Transactions** - A transaction starts from a consistent database state and leads to another consistent state.

PCTE defines **Lock** mechanisms to ensure the consistency of OMS data access operations by supporting the synchronisation of concurrent operations. It is a liaison between an activity and resource.

2.8. PCTE Distribution Mechanisms (DIS)

The user sees a community of compatible workstations, of possibly differing types, which comprises the PCTE environment as a single environment. Little of the function of the distribution is visible to the user. Its task is not only to allow the user access to the resources, software and hardware supplied by his own workstation, but also in a totally uniform way to all the resources, software and hardware, of the entire environment constituted by the different interconnected workstations [PCTE86].

2.9. The PCTE User Interface

What follows in this section is a brief description of the PCTE user interface. Because Eclipse has its own, self contained user interface, it would be of little use to have a detailed discussion of the PCTE user interface in this thesis.

In general, it can be stated that the design of a user interface is becoming more and more important. The provision of better user interfaces is being aided by the development of hardware which is able to drive graphic output. The emphasis these days is on systems which can be adapted to the user's requirements. The user can now interact with several tools concurrently, usually through **windows** on the screen. Thus the user interface provides the user with a communication port to the different applications in the user's working configuration, interprets user input, and channels it to the corresponding applications.

A problem which has existed has been the inconsistency in the meaning of various instructions depending on the context/mode. This is overcome in the PCTE user interface by the adoption

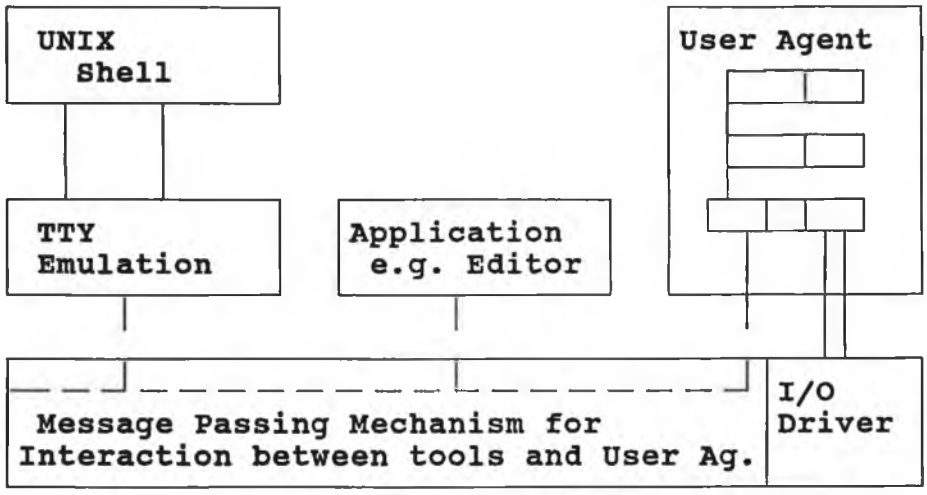
of an **Object Oriented** approach. The entities in the user interface are, as far as possible, handled in the same way, i.e. there is a standard set of operations. This is achieved by the following two steps -

[1] ... select the required object.

[2] ... perform an operation on that object ,i.e. generic commands which can be applied to different objects.

PCTE provides the standard functionality of a window management system [PCTE85]. Windows can be moved and can overlap or overlay on the screen. Every application can be accessed by the user via its associated window. It is also possible to iconise windows.

Diagram 2.3.



The Basic Structure of the User Interface

Diagram 2.3. illustrates the basic structure of the user interface model. The **User Agent** is in charge of translating the intention of the user to the system, e.g. display and management of windows, control of input from various input devices. **Applications** represent tools to the user.

Existing tools use an **OS-Application**, which emulates the host operating system in a window acting in a standard terminal mode [PCTE85]. They are not, however, able to take advantage of the PCTE user interface's advanced facilities.

2.10. Evolution Of PCTE

At least two implementations of the PCTE interfaces definition are currently running. One was developed by Olivetti within the original Esprit project. The other implementation was developed by the Emeraude consortium and it is upon this version that Eclipse was developed. Ada bindings for the PCTE basic mechanisms have been implemented on this version and an Ada compiler has been validated on Emeraude running on a Sun 3 and Bull SPS7 machines [Boud]. Version 10/4 of the Emeraude implementation was delivered in May 1989.

In 1987 the **Independent European Programme Group (IEPG)**, a grouping of European members of NATO, decided to finance the **PCTE+** project to examine how to build on the experience of the interface definition to define a basis for environments suitable for military as well as civil applications. IEPG felt that PCTE 1.4. was not suitable for military use without further development, notably in the areas of security and operating system independence [Tedd]. PCTE+ has two phases. The **definition phase** began in 1987; its major deliverable was the EURAC, the requirements and criteria to be used for the definition of PCTE+. The definition phase has now reached issue 3 and will form the basis of the **assessment phase** which will design and develop a layered implementation of PCTE+ on top of both UNIX and VMS.

The formal standardisation of PCTE ,i.e. PCTE 1.5., is also underway, under the guidance of the **European Computer Manufacturers Association (ECMA)**. It is hoped that this will lead to an ISO standard tool support interface, based on PCTE, in 1990.

In mid-1988 the threat of schism hung over PCTE [Tedd]. The ECMA version 1.5. was to be updated to a new verion 1.6. while PCTE+ was based on the 1.4. version and developing

separately. Fortunately PCTE+ has retrofitted the improvements of version 1.5. and the ECMA have decided to base their work on PCTE+.

Chapter 3.

Architecture of Eclipse

3.1. Introduction

Eclipse is the result of the Alvey Eclipse project initiated in the latter half of 1983. The intent of this project was to develop an integrated project support environment (IPSE)[9] of the kind described in [Stone], to populate it with various tools in support of some software engineering methods and in support of Ada [Alde1]. There were two versions of Eclipse used during the course of the research for this thesis; version 2.1 was replaced by version 2.2. in October 1989. The TBK[10] contains documents, libraries and header files, compilers and generators, example text files, initialisation scripts, and icons and cursors. The TBK provides all the facilities required by a tool builder, e.g. schema definition and database, user interface, graphical editing, system related facilities.

The features which characterise an IPSE, as envisaged by the Eclipse consortium, are its use of a database to hold all the data relating to the project and the provision of a kernel of facilities which provide all a tool's requirements with respect to execution, inter-process communication, input-ouput and database access. The develop of such a kernel was considered to be beyond the scope of the original Eclipse project. For this reason PCTE was chosen as the basis of the kernel and a layer of software was developed on top of PCTE. Thus tools are developed on a composite Eclipse/PCTE kernel.

The tool-builder's kit (TBK) is the key to the power of Eclipse. The idea behind Eclipse is to provide a base upon which integrated toolsets can be built rapidly. The TBK forms this base and it consists of libraries of functions and of meta-tools.

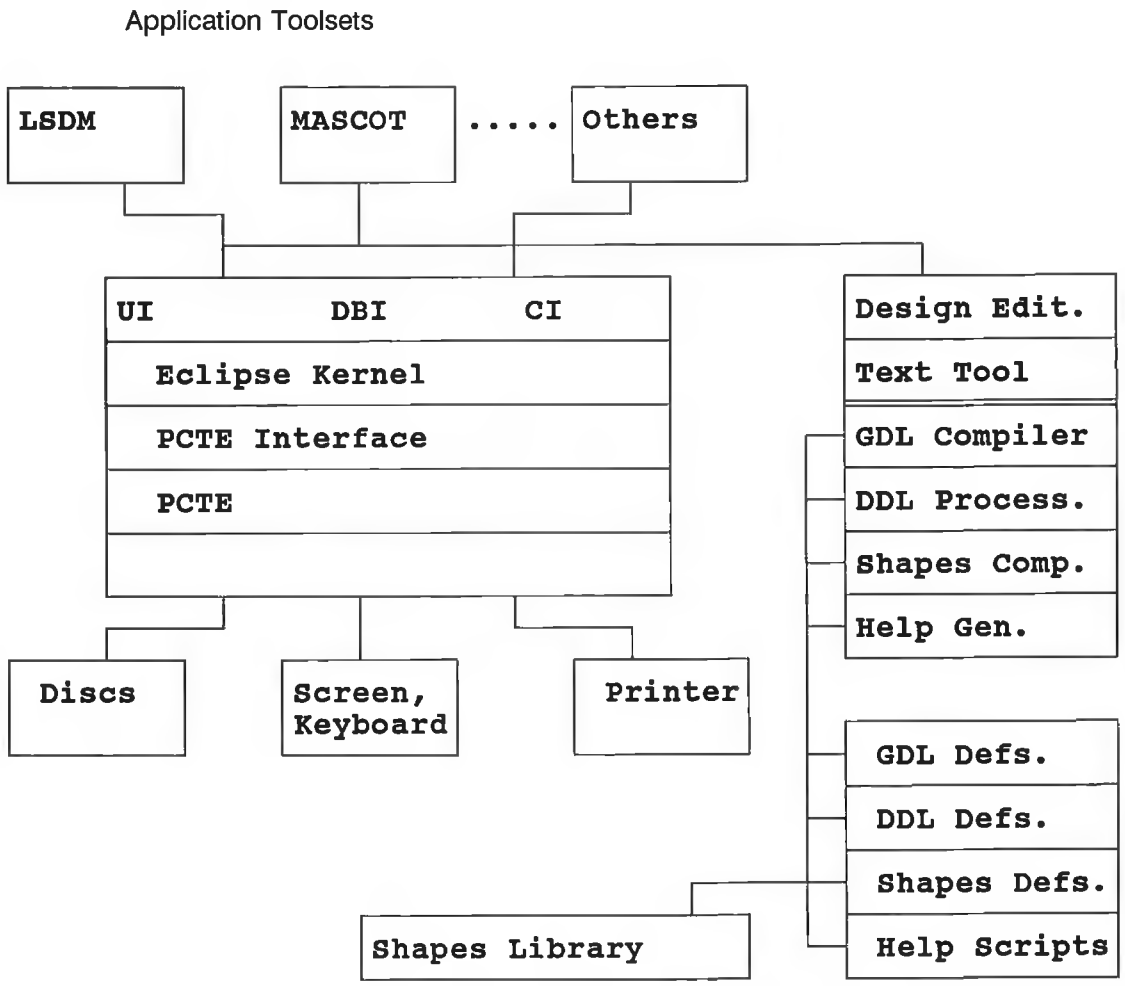
The Eclipse architecture can be seen in diagrammatic form in Diagram 3.1. This diagram shows the logical structure of the Eclipse kernel, toolsets and tool-builder's facilities.

---

[9]The terms integrated project support environment (IPSE) and software engineering environment (SEE) will be used interchangeably in this document.

[10]TBK - Tool Builder's Kit

Diagram 3.1.



UI - Eclipse User Interface
DBI- Eclipse Database User Interface
CI - Eclipse Configuration Interface

The individual parts shown in Diagram 3.1. will be explained in detail in the following sections. The discussion will focus on the major building blocks of the Eclipse TBK architecture, namely the following -

[1] ... The Eclipse Database.

[2] ... The Eclipse User Interface.

[3] ... The Design Editor

3.2. Forms of Tool Integration in Eclipse

The idea of tool integration is a central theme of this thesis and thus, before discussing the Eclipse architecture in detail, it is important to discuss the different types of integration recognised in Eclipse. At this stage, the details of how each level of integration would be implemented will not be discussed. The levels of integration available within Eclipse are as follows -

[1] ... Non-Integration

This level would be limited to running the tool concurrently with Eclipse tools. The foreign tool must have a compatible interface, i.e. any tools with user interfaces based on X windows cannot share the workstation monitor with Eclipse tools since the Eclipse user Interface is still based on SunView. At this level foreign tools would be invoked via a console window.

[2] ... Loose Integration

This level would be limited to invoking the foreign tool from an Eclipse tool ,i.e. "master tool", and running the tool concurrently with the Eclipse tools. This level has a number of benefits over non-integration. The availablity of the foreign tool could be controlled by the master tool. The master tool could check that all other tools had terminated before allowing the user to logoff. The foreign tool could be described in the Eclipse Help system and its use could be logged

using the Eclipse message system. The whole user interface would be improved since all tools would be invoked in the same way.

[3] ... Data Integration

At this level the foreign tool would be integrated with other tools in the sense of manipulating the same data structures. Sharing of data has to be considered from both the static and dynamic aspects. From a static point of view, the schema of the data needs to be defined. From a dynamic point of view, the degree to which concurrent access to the data is allowed has to be defined. Sharing data has the benefit of reducing duplication and increasing the reliability of information. The integration of data means that new tools can be designed to extract more useful information from the combined database. As this is the most important form of integration it will discussed at length throughout this thesis. A lot of the work carried out as part of this thesis dealt with this form of integration.

[4] ... User Interface (UI) Integration

At this level the foreign tool would be integrated with other Eclipse tools in the sense of having an Eclipse-style UI. A uniform UI increases ease of use, reduces learning time and results in fewer user errors. The user impression of the toolset as an integrated set of related tools is greatly enhanced by the provision of a uniform UI. The Eclipse UI provides a portability platform for tools between different windowing/manufacturers standards.

[5] ... Full Integration

This level includes both data and UI integration. The tool is designed and implemented in order to take full advantage of the Eclipse kernel facilities, to share data with other tools and to utilise a common UI style.

3.3. The Eclipse Database

3.3.1. Introduction

The PCTE Object store is an extension of the UNIX filestore concept. In this object store files are treated as entities with attributes and relationships. This was seen as a major advance by the Eclipse consortium but they regarded the fact that PCTE treated the contents of its entities as normal file store as a major failing. The primary means of integrating tools should be through access to common structures in Object Store, but in PCTE the majority of data is treated as having no structure [Alde1]. In reality this is not the case. Contents often have a well defined structure, e.g. a design diagram consisting of nodes and arcs. The Eclipse database provides a means of expressing the structure of the content by providing two tiers of data.

The first tier is at the level of PCTE objects. First tier information, generally, names and versions second tier data. Its purpose is to organise and control access to the second tier data. Each module of second tier data can be seen as a database in its own right and thus is known as a **Fine Structured Database (FSD)** [Cart1]. The Eclipse database has a unified data model in which both tiers of data are represented. In this section the Eclipse data model will be described and it will be shown how it refines the coarse-grain structure of PCTE's data model.

The purpose of having the two tier database is to integrate data. This is done by providing the following features -

[1] ... A uniform interface to physically and schematically different data stores.

[2] ... Linking the different data stores together by **inter-body pointers.**

3.3.2. The Eclipse Data Model

Just as with PCTE, Eclipse follows the hierarchical model presented in [Pene2] and discussed in Chapter 1. Thus the lowest level of the hierarchy is the definition of the overall data model. The data model used is a refinement of the ERA model presented in [Chen] and which is used

in PCTE. This data model is the two-tier model already mentioned. The design of the Eclipse data model and database interface was greatly influenced by [Bune] and [Ship] with their development of a function data model [Cart2]. The characteristics of such a model will be outlined in the following discussion.

The first tier of data is implemented by the PCTE OMS, the details of which are described in the previous section. There is, in addition, a predefined Eclipse SDS, **eclipse**, which provides the definitions required by the message system, the database, Eclipse users and basic Eclipse tools. Database transactions and recovery are not covered by the Eclipse specification and users must protect the integrity of their data using the facilities provided by PCTE.

Certain first tier objects are known as **fine structure objects**. These objects lead to second tier data and are defined to be subtypes of **fine_structure_object** which is a predefined type in the eclipse SDS and is itself a subtype of file because a fine structure object has contents. In Eclipse the contents of objects as well as the objects themselves are described by schemas. If a schema defines second tier data then it will be defined to be a subtype of **fine_structure_sds**.

Using the functional model of data , data is modelled in terms of **types** and **functions**. In Eclipse data is organised into **entities**. An entity is something which has attributes. Each entity is typed and it is the type of the entity which determines what attributes an entity has. This model thus follows the ERA model described in [Chen].

The roles of entity types in Eclipse are twofold -

[1] ... They determine the structure of entities as lists of named attributes.

[2] ... They classify entities, of which there are two types in Eclipse -

[a] ... **Subtype** - Allow an entity type to be defined as a **specialisation** of some other entity type.

[b] ... **Class** - Allow an entity type to be defined as a **generalisation** of other entity types.

Attributes in Eclipse are typed as follows -

[1] ... Printable Attributes - These are either integer, boolean, string, date, or user-defined enumeration type.

[2] ... Attributes of Type Entity - The purpose of these attributes is to link entities together. Instances of attributes of type entity are known as **links**. They are equivalent to cardinality-one links in PCTE [McLe].

In either case they are understood as **functions**. For example an entity value **f** is understood as a function with as signature [Cart2] -

f : A -> B

where A and B are entity types.

Entity attributes are defined to be **single-valued** or **many-valued**. Many-valued attributes have their values organised in two possible ways -

[1] ... Keyed Bag Attributes - A bag is an unordered collection of values in which the same value may occur many times. For keyed bags each value is associated with a key, which need not be unique. In Eclipse keyed-bags are always keyed bags of entities. They generalise the PCTE cardinality-many link type.

[2] ... Sequence Attributes - A sequence of values has order and may have multiple occurences of the same value. These attributes may be printable, e.g. sequence of integer, or they may be entity-valued, i.e. a sequence of links. A sequence-of-string attribute **h** of entity type B is shown as having the signature -

h : B -> Seq Of String.

Given this data model it can seen that database navigation is actually **functional composition**. Thus the functional expression -

husband/name : person -> String

specifies the name of the husband of the specified person.

It can also be seen that database iteration through entities of a given type is actually iteration through the values of a many-valued function.

3.3.3. Eclipse Data Definition Language (DDL)

Like PCTE and conforming to the [Pene2] hierarchy model, Eclipse has a DDL which is used to define Eclipse two-tier schemas and which may be compiled and installed using DDL compiler tools. Eclipse DDL defines both tiers of data in the Eclipse database. The first tier data is defined by a single notation referred to as **first tier DDL**, while the second tier DDL is given in terms of the IDLE[11] language. Examples of Eclipse DDL will discussed throughout this thesis , with regard to the various tools that were integrated into and developed for Eclipse.

First Tier DDL

Eclipse first tier DDL defines data in terms of sets of schema definitions referred to as Schema Definition Sets (SDS). These are the same as the PCTE SDS's described earlier in this document. The first tier DDL of Eclipse is logically equivalent to the PCTE DDL since both describe the PCTE OMS. The language syntax is different from PCTE DDL. Appendix C contains a example of an Eclipse DDL file and gives a short explanation of how it is used.

Second Tier DDL (IDLE)

Second tier schemata define the internal structure of a first tier entity type [McLe], i.e. its **fine structure**. The IDLE schema defines a data structure as a set of attributed directed graphs. IDLE is an extension of the **Interface Definition Language** [Nest] which was originally used in the generation of user interfaces. An IDLE schema consists of definitions of the various types of nodes and their attributes. In addition, a collection of classes of node types may be introduced in which an attribute of a class will also be an attribute of all the node types of that class. The example DDL source shown in Appendix C shows what an IDLE schema definition looks like.

3.3.4. The Eclipse Database Interface (DBI)

[11] *IDLE* - Interface Definiton Language for Eclipse

The database software will support many separate tool fragments running as separate processes running within a single user-id. Each tool fragment must have the database interface (DBI) bound into it [Cart1]. Thus each tool fragment must **logon** to the database but once a set of tool fragments have logged on then the combined effect on their calls is as if were interleaved through a single instance of the interface. It is important to note that when tools access common data in Eclipse it is up to the tools to access this data in a consistent way. Such a group of processes is called a **cooperating group** each of which has an access permission to each fine structure database (FSD).

Different cooperating groups are in competition with each other if they try to access data from the same fine structure database. Multiple readers are permitted but only a single writer. If competing processes need to share a fine structure object, e.g. a data dictionary which may be updated by several users simultaneously, then it must be defined to be of type **shared_fine_structure_object**. There are locking and unlocking facilities provided by the DBI. These must be used in order to ensure that data does not get corrupted.

When using the Eclipse DBI the notion of an **iterator** is used to iterate through one or more values of a many valued attribute. Iterators are tokens handed out by the database and having an internal state maintained by the database. There are several functions defined within the DBI which manipulate iterators, e.g. **next**.

The Eclipse database interface consists of twenty five functions with a maximum of three parameters. The exact nature of these functions will be discussed with respect to the various tools that I integrated into and developed for Eclipse.

3.3.5. The Eclipse Database Attributes

The reason that the Eclipse DBI can present a simple and uniform interface to both tiers of data is because of the rich and extensible set of attributes supported by the interface. The set of standard attributes associated with PCTE objects are maintained in Eclipse for the first tier objects. A full description of these attributes is found in [PCTE86].

Eclipse also defines several **virtual attributes** which return schematic and system related

information about an entity. Examples of such attributes are -

[1]	...	**Entity Type Name ($TYPE)** -	returns the entity type name of an entity.
[2]	...	**Textual Representation ($TEXT'x)** -	returns the text representaion of an integer, boolean or date attribute x.
[3]	...	**Destination Types ($DEST'x)** -	If x is an attribute typed by some attribute A then this derived attribute returns the names of all the entity types classified by A.

A special virtual attribute which is defined in IDLE is **!A**, for every entity type A. The meaning associated with this is "all A's which exist in this fine structure object" ,i.e. all entities of type A. It is an attribute of the local root entity, **UNIT**, of a fine structure object. !A is of type Seq Of A an is the only form of composition link supported at the second tier. Thus all entities of type A are created in relation !A to the local root.

Eclipse also supports **derived attributes** which support searches of IDLE structures enabling entities to be identified from the values of their attributes by exact matches and by pattern matching [Cart2]. Thus , given the extract of IDLE below, an example of a derived attribute would be the selection of a particular employee from the many employees in a department.

Example 3.1.

ENTITY ::= UNIT | department | person;

department => employees : Seq Of person;

person => name : String;

This example defines two entity types ***department*** *and* ***person****. The department entity type has an attribute which is of type sequence of type person. The person entity type has an attribute* ***name*** *which is of type string. Thus the selection of a particular employee,* ***smith from employees***

by name, *could be stated as the derived attribute -*

smith.employees^name

which has the signature

smith.employees^name : department -> person.

In order to retrieve all the smiths from employee by the derived attribute would be as follows -

!smith.employees^name : department-> Seq Of person

where ! is the virtual attribute from UNIT meaning **all.**

3.4. The Eclipse User Interface (UI)

In designing Eclipse, the integration of tools via the database was obviously essential, but as well as this, the Eclipse consortium considered that the provision of a fully integrated end-user interface, along with support for the construction of such an interface, was also essential. It was decided that the Eclipse user interface required a unifying metaphor [Pott1] which could be used consistently across all tools. An example of such a metaphor is the **desktop metaphor** used by the Apple Macintosh. Such a metaphor was required because the number of modes of interaction possible on the bit-mapped workstations, upon which the UI is supplied, is enormous and decisions had to be made to limit the number used so as to avoid confusion. The desktop metaphor was rejected because the objects that it represents are office documents with a limited set of allowed operations such as move, delete, etc. This metaphor was too simplistic because the objects which a software engineer is concerned with, e.g. programs, designs, etc., require a much greater set of possible operations, e.g. compile, link, etc., than those allowed in the desktop metaphor. On the other end of the scale the command languages of UNIX and PCTE were too terse and inconsistent, especially for inexperienced or occasional users. The need was thus for a compromise between the easy to understand but simplistic desktop metaphor and the terse command language offered by systems such as UNIX [Pott1].

The metaphor selected was the **control panel metaphor**. An interface definition layer was used to isolate the UI from the underlying architecture. This layer allowed the UI structure to be described in a definition language called the **Format Description Language**, which could be translated into the appropriate representation for the underlying window manager. The UI is not tied to any particular machine or operating system since all that is required to port it is the redefinition of the translation.

In order to have a fully integrated UI there must be a consistent treatment of messages to the user. It would be confusing to the user if every tool could make arbitary decisions about where and how messages were displayed. Inconsistency is a major irritation to users and thus Eclipse provides a unifying mechanism for all messages passing between tools and the end-user.

Another important aspect which relates to the UI is the provision of help. This is particularly pertinent for complex on-line systems. As with messages, it is important that the help facilities provided with each tool are of the same style so that the user does not have to learn a multiplicity of different ways to obtain help. In Eclipse, there is a context sensitive help system which relates to the tasks currently being undertaken by the end-user, thus allowing the user to browse through the help frames.

3.4.1. The Eclipse House Style

As far as the UI is concerned the Eclipse tool is one that conforms to the Eclipse presentation standards and provides a standard set of facilities in a standard way, i.e. Eclipse tools have a similar look and feel about them; they present the Eclipse **house style** [Pott2]. This style is provided via the building blocks which comprise the control panel metaphor. Control panels are built using five basic types of object -

[1]	...	**Button**	-	An object which, when selected, always initiates a single action.
[2]	...	**Menu**	-	A menu displays a list of objects which may be chosen by the user to initiate some an action.

The menu elements are not static and may depend on the context in which the user is working.

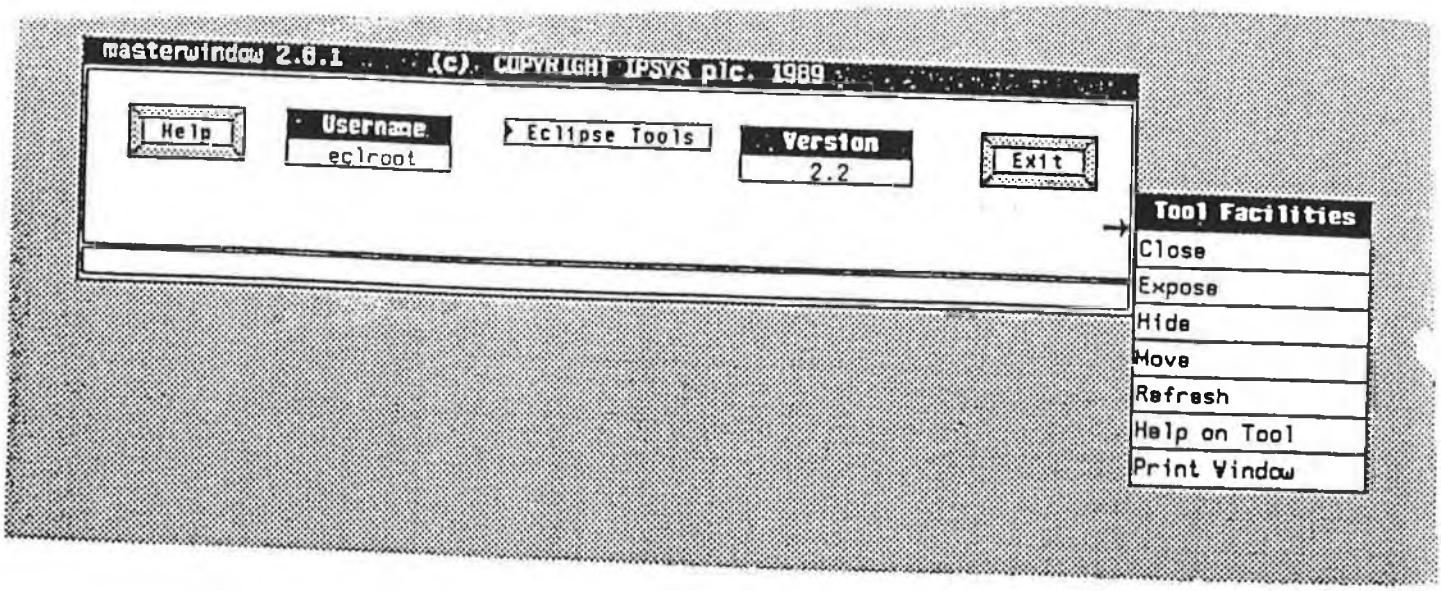
[3] ... **State Selector** - A composite symbol consisting of a menu and a value. The value displays the current state which may be changed by the selection from the menu.

[4] ... **SIgn** - A two part object with a fixed title and a value. The value may be static, dynamic user-changeable.

[5] ... **LIght** - A binary status indicator, e.g. tools arrange for a light to 'flash' during time consuming operations to assure the user that the system is still alive.

An example of a typical control panel is shown in Example 3.2. This is the standard control panel displayed when a user logs on to Eclipse.

Example 3.2.



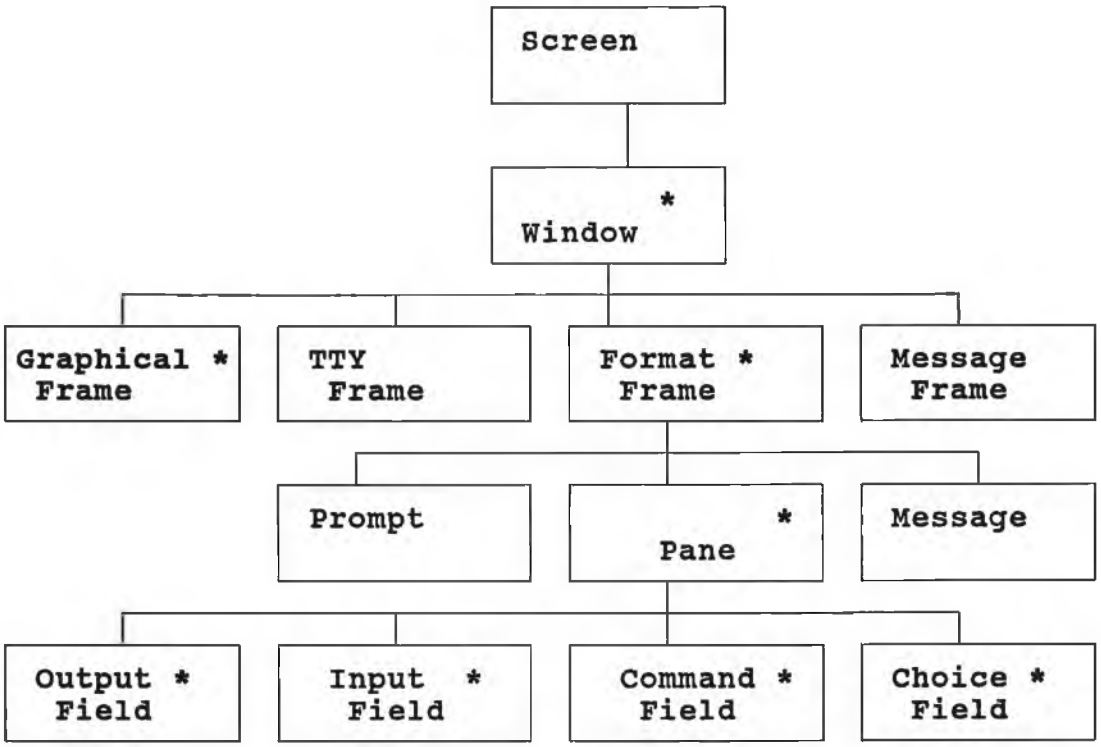
3.4.2. The Applications Interface (AI)

Software is needed to assist in the implementation of UI standards. The use of common software is the only way to guarantee identical interactions across all tools. The AI is the piece of software that implements the Eclipse house style and, in particular, it implements the control panel metaphor [Alde2]. The user is presented with a set of high-level, abstract primitives which are used to construct the tool interface and which hide the actual device characteristics while allowing powerful, low level graphics operations to be used when the need arises. By using the AI a tool will automatically inherit a style of presentation which is consistent with all other Eclipse tools - this is the Eclipse house style [Pott2].

The AI is defined as a communications channel between the user and the tool. A tool is not concerned with the manner of presentation of data. It needs to provide values for the user to see and it need to get values from the user. The precise nature of the interaction is not important to the tool. The basis of the high level abstract UI supported by the AI is a hierarchy of objects representing the various classes of images that a tool may use to construct its UI [Pott1]. Diagram 3.2. shows this hierarchy.

The root interface object is called the **screen** and represents the whole output area available on the workstation. Within the screen a number of **windows** can be defined, one of which may be visible at any one time. A window may contain one or more **frames** which may be aligned vertically or horizontally within the window. The frames are not constrained to be either as wide or as deep as the window. If frames overlap they do so destructively. With the exception of **formatted frames**, frames may not be subdivided. Frame subdivisions are called **panes** and consist of a set of **fields** which the tool wishes to manage as a group. Fields are generalisations of the control panel elements, e.g. buttons, signs, etc.

Diagram 3.2.



The User Interface Object Type

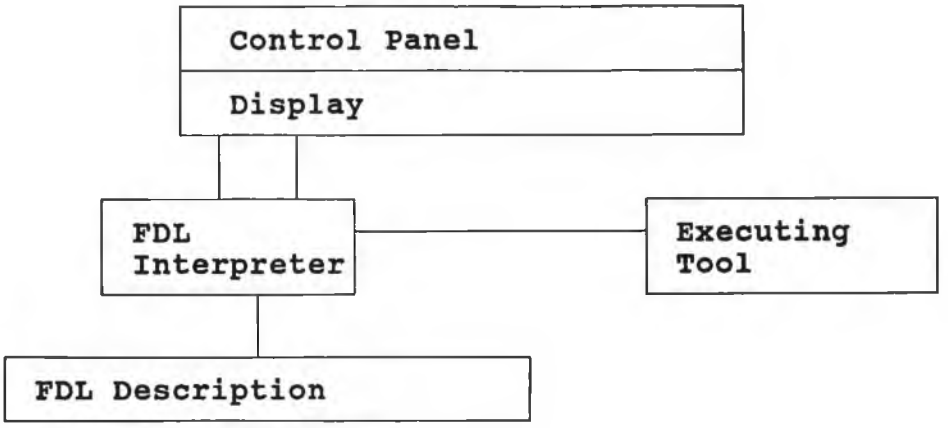
Graphic Frame - Supports the simple object-oriented graphical primitives for diagram drawing.
TTY Frame - Emulates a character terminal.
Formatted Frame - Used to represent frames which have sub-divisions - This form of frame is used to represent control panels, i.e. contains buttons, signs, etc.
Message Frame - Used to display text messages generated by the message system.

3.4.3. Format Description Language (FDL)

The actual presentation of the UI to the user can involve a lot of experimentation. The Eclipse consortium chose to describe the interaction style and layout by means of a language called **format description language (FDL)**. An FDL file contains a description of the control panel interaction as well as the windows, frames and panes used by a tool ,i.e. the hierarchy of tool user interface objects and their relative positions and extents. The layout of a control panel or

the window organisation of a tool may be changed without the requiring changes to the tool code. At run-time the FDL is interpreted to provide the control panels and the control panel actions. Diagram 3.3. shows the interaction between a tool and FDL.

Diagram 3.3.



FDL and Tool Interaction

In chapter 5 the use of FDL in presenting a UI for building a database to hold a classified specification of a network node is discussed.

3.5. The Eclipse Message System

In Eclipse, a message is regarded as information received by the **message handling subsystem** for routing to zero, one or more than one of the **message display area, console, and session event logs** [Hays]. The key aspects of the message handling system are as follows [Pott1] -

[1] ... Message Text and Code Separation

This means that different messages can be created for different user populations and can be represented in different languages. These message sets are read at run time and thus can be changed at run time without software rebuilding.

[2] ... **Structured Message Texts**

All messages are typed and fall into three classes, c.f. Appendix D. A message is structured into a **type**, a **text**, a **parameter list** where the parameters normally represent the user's context, and a link to the help system.

[3] ... **Help System Integration**

Each message is linked to a help system frame. To display the associated help frame the user selects the help button which is standard on every Eclipse tool.

Messages which relate to the same tool are held in files and are called **message sets**. Each message within a message set is identified by an associated **message number**. Eclipse provides a C library to allow tools to display message texts in a message frame.

3.6. The Eclipse Help System

The prime objective of the Eclipse help system is to give users the information they need quickly and to assist them to relate that information to the task at hand. Example 3.3. shows the layout of a typical help frame.

The salient features of the help system are as follows [John] -

[1] ... Provides rapid access to information stored on the Eclipse database, displaying the information quickly with the minimum of interaction.

[2] ... Help text is generally displayed in a single frame without the need for scrolling.

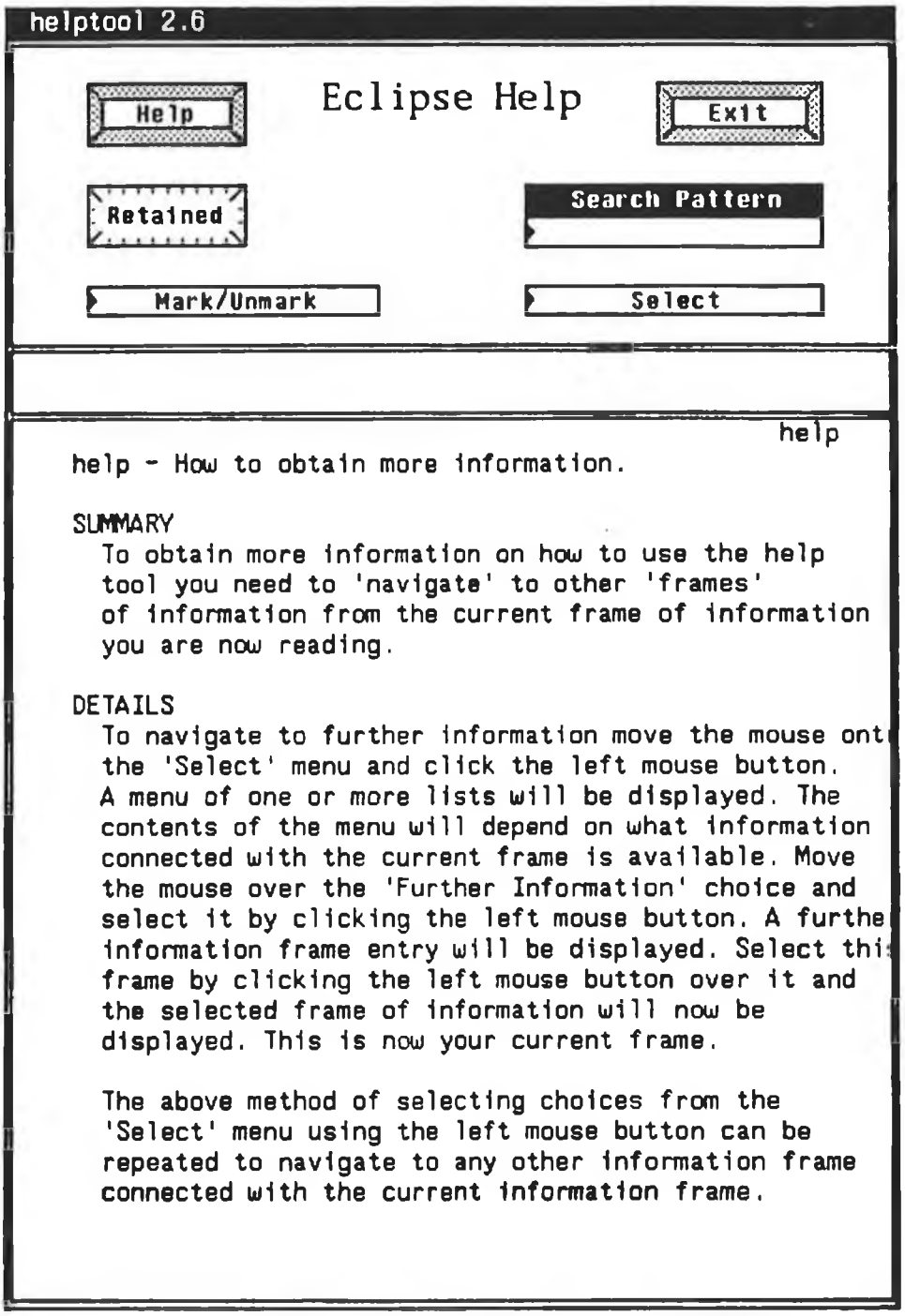
[3] ... Help information can be viewed alongside the component upon which help is requested.

[4] ... Any number of helptools can be invoked at any time.

These features are provided via the **helptool** which runs asynchronously with the invoking tool.

Each help entry is specified by a help frameset name and a help frame name. Each help entry is unique. Help entries are normally linked together to allow navigation around the help information.

Example 3.3.



3.7. The Design Editor (DE)

What follows is an overview description of the DE. In chapter 4 the use of the DE in the generation of graphical designs will be discussed in detail. Graphical approaches to software design, such as Data Flow Diagramming, Jackson Structured Design, etc., have become very popular. The Eclipse was required to allow the incorporation of support tools for such methods and to allow the designs produced with such tools to be stored in the Eclipse database. Because Eclipse is an open IPSE it was impossible to predefine which methods would be supported. The Eclipse consortium came to the conclusion that the most effective way to provide a design support tool for graphical design methods was to produce a generic tool which could be tailored by the system builder for whatever methods were supported in any single release of Eclipse [Beer]. This requirement went beyond the provision of a graphical editing system. It required the provision of a tool which would allow the user to create and maintain software designs in a diagrammatic form and to capture both the diagrammatic representation and the underlying semantics in the Eclipse database [Robs1]. A design definition language was developed in order to support arbitrary symbolism. This language is called the **Graph Description Language (GDL)**. GDL provides a notation for defining the syntax and partial semantics of software designs which are expressed as directed graphs.

The normal procedure for using the Design Editor to implement graphical design methods is as follows [Beer] -

[1] ... The Eclipse tool builder defines the syntax and the semantics of the design method to be supported using GDL.

[2] ... The GDL compiler generates tables for input to the DE.

[3] ... The symbols associated with a method are defined by the Eclipse tool builder and the tables are generated for the DE.

[4] ... The DE uses these generated tables to provide an interface which is tailored to whatever design method is in use.

[5] ... The generated designs are stored in the Eclipse database.

Chapter 4.

Integrating Foreign Tools Into Eclipse

4.1. Introduction

A major part of my work for this thesis involved the evaluation of the Eclipse database as a basis for integrating tools. In the next chapter I will discuss the development of a new tool using Eclipse/PCTE. I selected two tools which had been developed by two other partners in the SPECS project. The first tool was developed by PTT Research - Neher Laboratories in Holland. They developed a LOTOS tool environment for the integration of LOTOS tools. The second tool was a C code generator, developed as part of the SPECS project by Bell in Belgium. This tool took the Common Representation Language (CRL), developed by the SPECS project, as input and generated C source code from it. I found that the degree of integration into Eclipse which was possible differed greatly for these tools. In this chapter I will discuss the steps taken in the integration of these tools, and will highlight the problems encountered.

4.2. The LOTOS Tool Environment (LOTTE)

4.2.1. Introduction

LOTOS (Language of Temporal Ordering Specification) is one of the two **formal description languages** developed within the International Organisation for Standardisation (ISO) for the formal specification of open distributed systems, and in particular for those related to the Open Systems Interconnection (OSI) computer network architecture [Bolo]. LOTTE is an environment which aims to give a uniform access to tools with each tool being accessible with a special command in the environment. LOTTE was chosen as a first attempt at tool integration into Eclipse. My initial idea was to take tools from LOTTE and examine the extent to which they could be integrated into the Eclipse environment. What follows in this section is an account of the tools used and the knowledge gained in trying to integrate them into Eclipse.

For the purposes of integration into Eclipse two LOTTE tools where chosen. These were a Syntax and Static Semantics checker (SSS) and a Gate-Sortlist Report Generator (GSR). In

LOTTE all interaction between these tools occurred via intermediate files. It was hoped to replace this file-based tool integration with an Eclipse two-tier database implementation.

4.2.2. The Syntax and Static Semantics Checker (SSS)

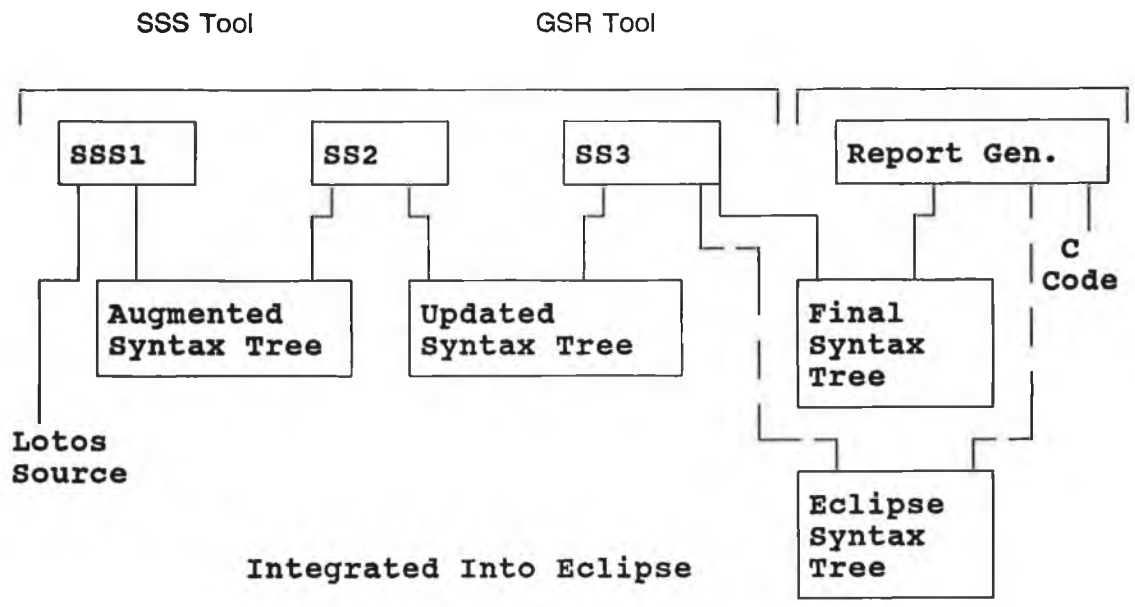
The Purpose of the SSS

The purpose of the SSS is to verify that a specification, written in LOTOS, complies with the formal syntax and static semantics defined in the ISO LOTOS standard. This assurance of correctness is usually a precondition for further analysis/simulation of the specification (as is the case with the GSR). The checker's central purpose is to provide clear error messages. Another purpose of the checker is to produce a dump of its internal data structures, i.e. the abstract syntax tree. It was through reading these dumps that other tools, e.g. the GSR, could avoid having to scan, parse and analyse the LOTOS text all over again.

The Structure of the SSS

The tool is a multiple pass checker, invoked by the **sss** command in LOTTE. As with the Gate-Sortlist Report Generator this tool was developed using GAG , a compiler generator system based on attributed grammars. Whereas the use of GAG had a number of advantages for the tool developers ,e.g. shorter development times, it also caused a number of problems for integration into Eclipse and meant that the tool could not, realistically, be completely integrated into Eclipse. The SSS consists of three programs which are called in sequence. These exchange information using intermediate files which contain abstract syntax trees augmented by attributes or information added to relevant syntax nodes.

Diagram 4.1.



The three programs perform the following functions -

[1] ... The first program reads the specification source and performs the lexical analysis and syntax checking. It assigns some of the attributes for the second program and writes the augmented syntax tree to an intermediate file.

[2] ... The second program uses the file holding the abstract syntax tree generated by the first program as its only source input. It performs analysis of data types and writes a syntax tree with updated and added attributes into a second intermediate file.

[3] ... The third program uses the intermediate file generated by the second program as its only source input. It performs the analysis of LOTOS behaviour expressions and writes a special syntax tree with the attributes for the report generator in a special result file.

I decided that it would be the last tree which would form the basis of the integration into Eclipse, i.e. replacing the intermediate file holding the syntax tree with an Eclipse two-tier database holding the same information. Instead of writing out to a result file the third program would store its data in the Eclipse database. I developed a two-tier schema which would provide the required structure for such an operation and this schema would be part of the working schemas of both the SSS and the GSR.

4.2.3. The Gate-Sortlist Report Generator (GSR)

The Purpose of the GSR

The purpose of the GSR is to give static information that is helpful in understanding the behaviour of a LOTOS specification. It can be used to check for possible deadlock situations in the specified system. The interactive report generator gives the possibility of generating the possible interactions in which a behaviour expression can participate.

The Structure of the GSR

The GSR consists of two programs invoked by the **gsr** command. The non-interactive part uses the augmented syntax tree produced by the third program of the SSS. It is this program that forms the second part of the integration into Eclipse. Instead of constructing its internal structures from an intermediate file it would, instead, build its internal structures from the Eclipse database created by the last pass of the SSS.

4.2.4. Integration Into Eclipse

The Scope of the Integration

An initial form of integration, which I implemented, was the integration of LOTTE as an Eclipse foreign tool. This meant that LOTTE could be invoked from within Eclipse object space ,e.g. from an Eclipse menu. Such an implementation had the effect of making LOTTE look as if it is an

Eclipse tool,due to being invoked from within LOTTE, though it was unable to interface to any other Eclipse tool via Eclipse facilities.

As stated before , the fact that GAG was used presented some major problems for the degree of integration which was feasible. The first problem I encountered was the fact that GAG produced PASCAL code for which Eclipse does not provide an interface. This problem was overcome through the use of external calls to C functions. When integrating into Eclipse, at the data level, one must know the structure of the permanent data used by the tools that are being integrated. For the purposes of the SSS and GSR this was not a factor because such data structures were an internal concern of GAG alone. Thus the whole area of tree input and output was a black box to LOTTE. I was thus faced with the prospect of examining large quantities of GAG-generated PASCAL code in the hope of extracting the required data structures. I was by no means certain that I would be successful in this pursuit due to the very large data structures declared within the GAG-generated code.

This made the tools less than ideal for the purposes of integration into Eclipse.

I decided to limit the integration to the interaction between the last pass of the SSS and the non-interactive GSR program and see if this could form the basis for further integration at some later date.

Integrating LOTTE as a Foreign Tool

Eclipse provides a generic tool called **texttool** which enables TTY tools to run in a windowed environment. Texttool creates an Eclipse window containing one TTY frame in which the TTY tool ,i.e. LOTTE, is invoked. All the options and parameters of the tty tool can be supplied at invocation time. For LOTTE certain pathnames in the invocation scripts had to be changed since they presumed invocation from an owning directory and not from within Eclipse volumes space. Otherwise, I found that the invocation of LOTTE as a foreign tool was a simple matter.

Extracting the Data Structures

This was one of the major problems areas in integrating LOTTE. The data structures used in each program were very large and inter-twined and the fact that they were generated by GAG meant that they had very confusing names. An example of a small section of the data structures used by SSS and GSR is shown in Appendix E. As can be seen from this small section, which declares the structure of a node in the syntax tree, the code is far from readable.

The job of extracting the data structures used by the SSS and GSR involved tracing through every possible path of the tree output noting the data structures stored in the intermediate file. Even though the internal structures were very large, with each node in the parse tree having possibly many different types of attributes, it was possible to narrow the output structure to a very manageable size since, through the tracing operation, I found that only a subset of the data structures declared in the program were actually written to the intermediate file.

Building a Two-Tier Database

Once I had extracted the structure of the data, the next step was to map this structure onto an Eclipse two-tier schema. I decided to define first tier objects to hold the syntax trees, while individual node types held in the syntax trees would be defined as second tier entities.

The schema that was developed mirrors the structure of the intermediate file quite closely. This schema structure was chosen as a way of limiting the effort required to attain an initial level of integration. In the intermediate file, nodes are stored in a byte stream with a rule value differentiating the nodes. In the schema that I developed, nodes are stored in sequences , which are a built-in form of data representation in Eclipse. The syntax tree structure could have been mapped onto an Eclipse schema , exactly , without requiring any extra Eclipse functionality. Such a schema would facilitate the total integration of the GSRs data. This would mean that the GSR would access data from the tree-structured schema in an equivalent way to accessing the internal syntax tree read from the intermediate file. Whereas the creation of such a schema was possible it would have been infeasible to replace the many hundred data structure accesses, which occur throughout the program, with Eclipse DBI calls.

Storing the Data in the Database - SSS

Storing the data in the database required the modification of the last pass of the SSS as well as the development of C functions to actually store the data. I developed/modified the code so as to have a clear distinction between the GAG generated code and the Eclipse interface code. The basic SSS algorithm used for writing the data was maintained. Initially, the first tier object was created. It is within this object that the various nodes were stored. The function which performs a pre-order walk through the tree was modified to the extent that writing out a node actually involved calling a C function which created the correct second tier entity, in the Eclipse database, based on the rule value associated with the node.

For the following nodes there were no further attributes to be stored -

[1] ... Empty Node

[2] ... List End Node

[3] ... List Node

Terminal nodes had two attributes associated with them -

[1] ... Selector Value

[2] ... Encoded symbol table values.

For attributed grammar nodes the exact form of the attributes stored was determined on a per node basis, based on the value of a node type selector attribute. The individual functions within the last pass of the SSS, which originally wrote the individual attributes to file were modified to copy them into a structure. This structure was then passed to the C code which stored the values in the correct entity attributes.

The exact form of the nodes can be seen in the schema as shown in the Appendix F. The coding required to store the data was straightforward. The only real problem encountered was when the Eclipse libraries were linked in with the LOTTE code. One of the Eclipse User Interface

library functions appeared to be called for some reason even though no such call existed. It was found that the library in question contained a **main ()** and the linker was picking this up instead of the PASCAL program's mainline. This sort of problem can be very exasperating, especially for an inexperienced user of Eclipse. In order to find out what the problem was, it was necessary to have the original database library source code consulted.

Retrieving the Data from the Database - GSR

This constituted the mirror image operation of the write operation. As in the write operation , the basic GSR algorithm for constructing the syntax tree structure was maintained. Basically , when reading a node , a call was made to a C function which retrieved the next node entity in the sequence from the database. The rule value was returned to the GSR and based on this value the GSR knew the node type. Subsequent reading of node attributes , for attributed grammar nodes,involved the reading of the values from the database and the storing of the values in a structure. I modified the GSR so that the functions which would normally have read the values directly from the file read them, instead, from the correct fields in the structure. Appendix G contains details of the integration of the LOTTE tools along with some example C code.

4.2.5. Possible Further Integration Into Eclipse

In the following sections, I outline the sequence of possible steps that could be taken to more fully integrate the tools into Eclipse.

Remove All Intermediate Files

As discussed above , the main output file from the SSS, i.e. the file containing the abstract syntax tree, was integrated into Eclipse. As well as this file, there were other files passed to the GSR from the SSS. The files passed to the GSR are as follows -

[1]	...	INFILE	- Source Text
[2]	...	STABFILE	- Symbol Table
[3]	...	VSFILE	- Visit Sequence Table
[4]	...	MSGFILE	- Error Messages

To allow further integration, these files could be examined and their structures extracted. As in the case of the syntax tree, schemas could be developed for these files, allowing the last pass of the SSS to store their data in an Eclipse database. As a next stage of integration, the GSR could read in these structures from the database.

Total Data Integration

Because the tools were GAG generated and the underlying permanent data structures were unknown, I decided to see if a limited degree of data integration could be achieved. This limited degree of integration was achieved . A next stage in the integration of the tools would be to integrate their run-time data into the database. This would mean that, instead of reading from the database and building the internal syntax trees, the tools would retrieve their run-time data from a database whose schema has the same structure as the syntax tree used within the GSR. This would be implemented by altering the way the SSS writes out the permanent data. The schema shown in the Appendix F would allow the SSS to store its data in an Eclipse database with the equivalent structure of its internal syntax tree.

The next step , which would involve a large amount of work, would be to modify the GSR so that everywhere it currently accesses its internal syntax tree it , instead , retrieves the equivalent node/data from the database. This means that the GSR would no longer need to read in the data and build its internal structures. Where it currently has a pointer to a node structure it would , instead , have an Eclipse database entity token for the the equivalent node in the database. Where it currently accesses a pointer from one node to another node in the internal structure it would , instead , traverse the equivalent second tier link.

Because the GSR does not have clearly defined functions for tree manipulation the operation above would be a huge task. In the GSR, there is lots of tree navigation, update and retrieval embedded into the code. For such an operation to be feasible it would be necessary to have the tree handling operations separated into clearly-defined functions which could be replaced with equivalent functions operating on an Eclipse database.

Extract SSS and GSR from LOTTE

So far, the integration that has been discussed has been slightly unusual in that the tools being integrated are still part of the LOTTE environment. It is, thus conceivable to invoke LOTTE as a foreign tool, which contains within it tools which actually use the Eclipse database. A next step could then be to make the individual tools Eclipse tools ,i.e. remove the tools from LOTTE and invoke them as Eclipse tools. As LOTTE stands , invocation of the tools is tied up in shell scripts which are called from C programs. To separate them out would require the replacement of these scripts with built-in-functions which could be invoked from a tool menu defined in FDL. It should be possible to add the tools to an Eclipse tool menu and invoke them from FDL.

Integrate the User Interface

A last stage of integration could be to integrate the whole of the LOTTE user interface into Eclipse. At the moment the various functions provided by LOTTE are presented in a scrolling menu. The user enters a command string at the prompt in order to perform the required function.Integrating the user interface would involve the development of an FDL file which would present in Eclipse house style the options currently available in the LOTTE main menu. Invocation of the various options provided by LOTTE could then be done from the FDL file based on a mouse selection from the menu.

The last two options are feasible, but were not as important as data integration, from the point of view of the SPECS project.

4.3. The CRL to C Tool

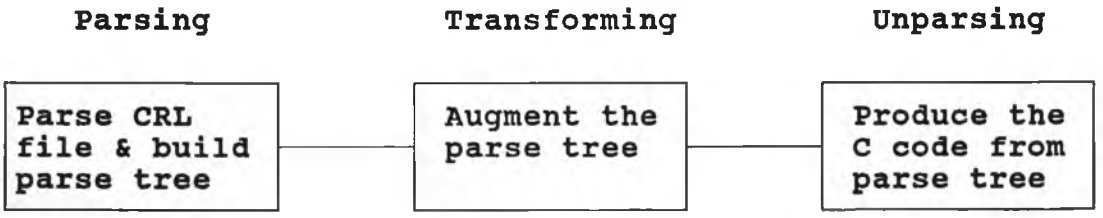
4.3.1. Introduction

Bell in Belgium, developed a tool to translate the Common Representation Language (CRL), which was developed as part of the SPECS project, into the C programming language. CRL plays a central role in the SPECS architecture. It is the target for translations for **tower languages**[12]. It should also be the source from which an implementation of a specification should be achieved. It is this latter reason which prompted Bell to develop a tool which would perform the automatic translation from CRL into an implementation language, which for the SPECS project is C. From the point of view of this thesis, the CRL-C tool provided a contrast with LOTTE for data integration of a foreign tool into the Eclipse database.

4.3.2. Structure of the Tool

The CRL-C tool had certain similarities in structure to the SSS and GSR in LOTTE in that it was composed of a number of phases. Diagram 4.2., below, shows the structure of the tool.

Diagram 4.2.



[12]Tower Languages - In SPECS the languages which constitute tower languages are SDL and LOTOS.

The first pass was responsible for parsing the input CRL file. This was facilitated by the use of the YACC and LEX UNIX tools. LEX was used to scan through the CRL source file and return tokens based on the CRL keywords found in the file. YACC is a parser generator in which the grammar of CRL was defined. A parse tree is built using a set of programmer-supplied tree-building functions. Which functions are called and what values are passed to them can be defined as part of the grammar productions defined in the YACC file. An example CRL file can be seen in Appendix G.

Unlike the tools in LOTTE, there were no intermediate files used in the CRL-C tool since all the passes were linked together at compile-time. The parse tree, built in memory by the first part of the tool, was used by the second two parts.

4.3.3. Scope Of Integration

Because there were no intermediate files used in the tool, the form of integration possible for this tool automatically differed from that in the LOTTE tool. The tool itself was much smaller than either the SSS or the GSR and was much easier to understand. The code was organised in a way which leant itself to integration. All tree navigation, update and access were separated into a set of functions. Because of this, it was possible to fully integrate the tool's run-time data into the Eclipse database.

This operation required the development of an Eclipse schema definition for storing the data. The schema, that I developed, was logically equivalent to that used by the original tool. The parse tree would not be built using the standard internal tree data structure , but would instead build the tree by storing the node values in, and creating the required links between, Eclipse second tier entities, as defined in the schema definition. The original data structures can be seen in Appendix H.

4.3.4. Building a Two-Tier Database

The extraction of the data structures used by the tool was a very trivial operation as the data structures used were quite simple. They were also held separately in an include file rather than being embedded in the code as was the case with the LOTTE tools. In order to facilitate full data integration of the tool, I replaced the original data structures with an Eclipse schema definition which had a logically equivalent structure. The schema definition can be seen in Appendix J. This schema defines a fine-structure-object, **bell_tree**, within which the parse tree schema is defined.

The actual building of the database required a modification to the YACC file used to parse the input CRL file. The actual grammar productions defined within the YACC file remained the same since the input CRL remained the same. What did change were the functions which were called to build the parse tree. This greatly helped the integration since it meant that I could concentrate on rewriting the tree handling functions without the need to have a deep understanding of the syntax and semantics of CRL.

Example 4.1. shows a small extract of the modified YACC file.

Example 4.1.

```
crl : '(' sigdeclaxioml inf ')'
        { $$=Tree_CreateNode(AS_CRLDATA) ;
          Tree_AddFirst($$, $2) ;
          yy_root=$$ ;
        }
;

sigdeclaxioml: axiomlactdecl
        { $$=Tree_CreateNode(AS_SIGDECL) ;
          Tree_AddNext($$, $1) ; }
| sigdecl axiomlactdecl
        { $$=$1 ;
          Tree_AddNext($$, $2) ; }
;
```

It can be seen from this extract how the YACC source would call user-developed functions upon the recognition of a LEX token, e.g. **crl**. When this LEX token was recognised the

Tree_CreateNode function was called with an explicit value parameter. The **Tree_Addfirst** function would then be called to create the **first** link, as per the schema definition, between the crl node and the node generated when the **sigdeclaxioml** token was recognised. This function takes two parameters; the first one is the Eclipse entity token returned from Tree_CreateNode while the second is the entity token returned by the parsing of sigdeclaxioml. Both the LEX and YACC were processed to produce the required scanner and parser. The C source files, generated, would then be compiled and linked in with the rest of the CRL-C source code.

In Example 4.2., below, is a listing of the Tree_CreateNode function.

Example 4.2.

```
#define CHECK_RESULT( call )                                    \
        if ( (call) != EI_OK ) { DBI_logoff (); return( FATAL_ERROR ); }

/*************************************************************************
 *                                                                       *
 *  Procedure name  :      Tree_CreateNode                               *
 *                                                                       *
 *  Purpose         :      This routine creates a nood in the Eclipse    *
 *                         tree and stores the node's name.              *
 *                                                                       *
 *  Input           :      name - the node's name                        *
 *                                                                       *
 *  Output          :      root - the node's token                       *
 *                                                                       *
 *  Written By      :      Sean Mac Roibeaird, DCU                       *
 *                                                                       *
 *  Date Written    :      August 3rd 1989                               *
 *                                                                       *
 *************************************************************************/

DBI_ENTITY Tree_CreateNode(name)
int name ;
{
/*pNode root ;

  root = (pNode)malloc(sizeof(Node)) ;
  root->name = name ;
  root->next = Tree_NULL;
  root->first = Tree_NULL ;
  root->back = Tree_NULL ;
  root->value = AN_NULL;
  root->anot = AN_NULL ;
  return(root) ;
*/
```

```
DBI_ENTITY          root;
DBI_ATTRIBUTE_VALUE     avalue;
char                space[MAX_ATTRIBUTE_LENGTH];

/*************************************************************************
 *                  Create the node entity                               *
 *************************************************************************/

    CHECK_RESULT (create_entity (
                            "node",
                            "!node",
                            &root));
/*************************************************************************
 *                  Store the name value                                 *
 *************************************************************************/

    avalue.attr_type = DBI_INTEGER_SCALAR;
    avalue.attr_value.v_entity = &name;
    CHECK_RESULT (attribute_becomes (
                            root,
                            "name",
                            avalue));

    return (root);
}
```

In the example above, both the original code and the Eclipse DBI code are shown. In the original code a memory block is reserved for each node to be created. The name string is then stored in the **name** field of the node structure. All the other elements of the node structure are initialised to null.

In order to create a node entity in the Eclipse database it is necessary to call the **DBI_create_entity** [Cart1] function. It is recommended practice, when using Eclipse library functions, to envelope them in higher level functions which contain all the necessary error checking and message display code. In the above code the **create_entity** and **attribute_becomes** functions contain the DBI_create_entity and DBI_attribute_becomes Eclipse DBI functions. This greatly reduces the amount of code required when using the Eclipse DBI functions since all error-checking and message display can be performed in one place rather than everywhere a DBI function is called.

The create_entity function takes three parameters as follows -

node - This is the type of the entity to be created, c.f. Appendix I.

!node - All entities are created relative to UNIT, the local root of the parse tree fine structure database. !node is a virtual attribute of UNIT which is of type **sequence of node**.

&root - The entity token of the created node entity.

In order to store the node's name in the database one must call the **attribute_becomes** function, which updates a given attribute of an entity. The attribute_becomes function takes three parameters as follows -

root - The entity token of the created node.

"name" - The name of the attribute to be updated.

avalue - This structure holds the type and the value which will be used to update the attribute.

In the original code, all the pointer elements were initialised to null. In the Eclipse code there is no need for an equivalent initialisation. This is due to the fact that any attempt to traverse a non-existent link in Eclipse will return an exception condition which can be trapped. Such navigations are implemented by using the **DBI_value_of_attribute** function. This function, when passed an entity-type attribute will return the entity token of the attribute, if it exists. An example would be the navigation from a node entity to its brother node entity.

Example 4.3.

```
DBI_value_of_attribute (root,
                        "brother",
                        &buffer,
                        &avalue);
brother = *avalue.attr_value.v_entity;
```

An explanation of the parameters expected by this function are explained in Appendix G.

It can be seen from the above examples that full data integration of the CRL-C tool involved similar code re-writes for all the tree building, navigation, access and update functions. The algorithms used in these functions were, to a great extent, maintained in the Eclipse version.

4.4. Conclusions

By performing the integration of the LOTTE and CRL-C tools I noted some key factors which I feel must be considered when integrating foreign tools into the Eclipse database. In this section I will outline these factors.

[1] ... Integration of LOTTE as a foreign tool was simple, but was of little use.

If a tool is considered unsuitable for integration into the Eclipse database, then an initial level of integration can be achieved by invoking the tool from an Eclipse menu, as one would invoke native Eclipse tools. The parameters required by the tool could be entered in FDL fields instead of being typed at the UNIX shell prompt. This would increase the compatibility of the look and feel of the tools to be used. For LOTTE , I found that this was a relatively simple exercise , but once invoked the tool is incapable of availing of the extra functionality provided by Eclipse.

[2] ... The ease with which the data structures used by foreign tools can be mapped onto Eclipse schema definitions.

In the two cases dealt with in this chapter, there were great differences in the visibility of their data structures and, hence, there was a large difference in the degree of integration which was feasible. The LOTTE tools had very large and difficult to understand data structures, of which only a subset was used. The CRL-C tool, on the other hand, used a simple data structure which could be easily mapped onto an Eclipse schema definition.

[3] ... The quality of the source code in foreign tools.

By quality of code I mean the degree to which the accesses to data structures were separated from the rest of the code in the tools. In LOTTE, the code was generated by GAG and accesses

to the elements in the syntax tree data structure were embedded in the source code. It was infeasible to replace all these data structure accesses with Eclipse DBI function calls. The CRL-C tool, on the other hand, was well designed with a clear separation between the code which accessed the data structure elements and the code which performed the CRL to C translation. It was thus possible to replace this part of the tool, en-masse, without having a major impact on the rest of the code in the tool. A crucial factor was the fact that it was possible to modify the YACC source file and then rebuild the system. The job of integration would have been much more difficult, and somewhat similar to LOTTE if one had to try to integrate the YACC generated source code.

[4] ... Size of the Foreign Tool

There was a vast difference in the sizes of the two tools. Each of the LOTTE tools were an order of magnitude bigger than the CRL-C tool. When this factor is combined with the fact that the CRL-C tool had a clear separation of the code which manipulated its data structures it can be seen that it was a far easier task to integrate the CRL-C tool , and to a far higher degree, than the LOTTE tools. Thus, the larger the tool ,and the more embedded the data structure accesses, the more code which has to be developed/modified in order to integrate the tool into the Eclipse database.

Chapter 5

Developing a New Tool in Eclipse

5.1. Introduction

As well as integrating foreign tools into Eclipse, I considered it important to evaluate Eclipse capabilities when developing a completely new tool, particularly a tool which required graphical output to the screen. The Eclipse TBK contains a graphical design editor tool which provides design support for graphical design methods. The whole focus of Eclipse is the support for the development of integrated toolsets which implement design methodologies. The need for a graphical design editor is obvious when one is implementing graphical methodologies such as MASCOT or HOOD.

5.2. Background

Within the SPECS project, a vital part of the architecture is the area of informal specification rigorisation and formalisation. In the SPECS architecture, an informal specification of an IBC network is regarded as the starting point in the development of the network. Informal specifications are expressed in natural language and free diagrams. SPECS has developed a **conceptual model** for mapping from informal specifications to formal specifications. In order to build formal specifications from informal ones, two intermediate modelling levels have been identified, namely **classified specifications** and **rigorous specifications**.

The purpose of classification is to get a complete understanding of the system as described in the informal specification, by classifying the elements of interest. The objectives of classification are as follows -

[1] ... To identify ambiguities and missing information.

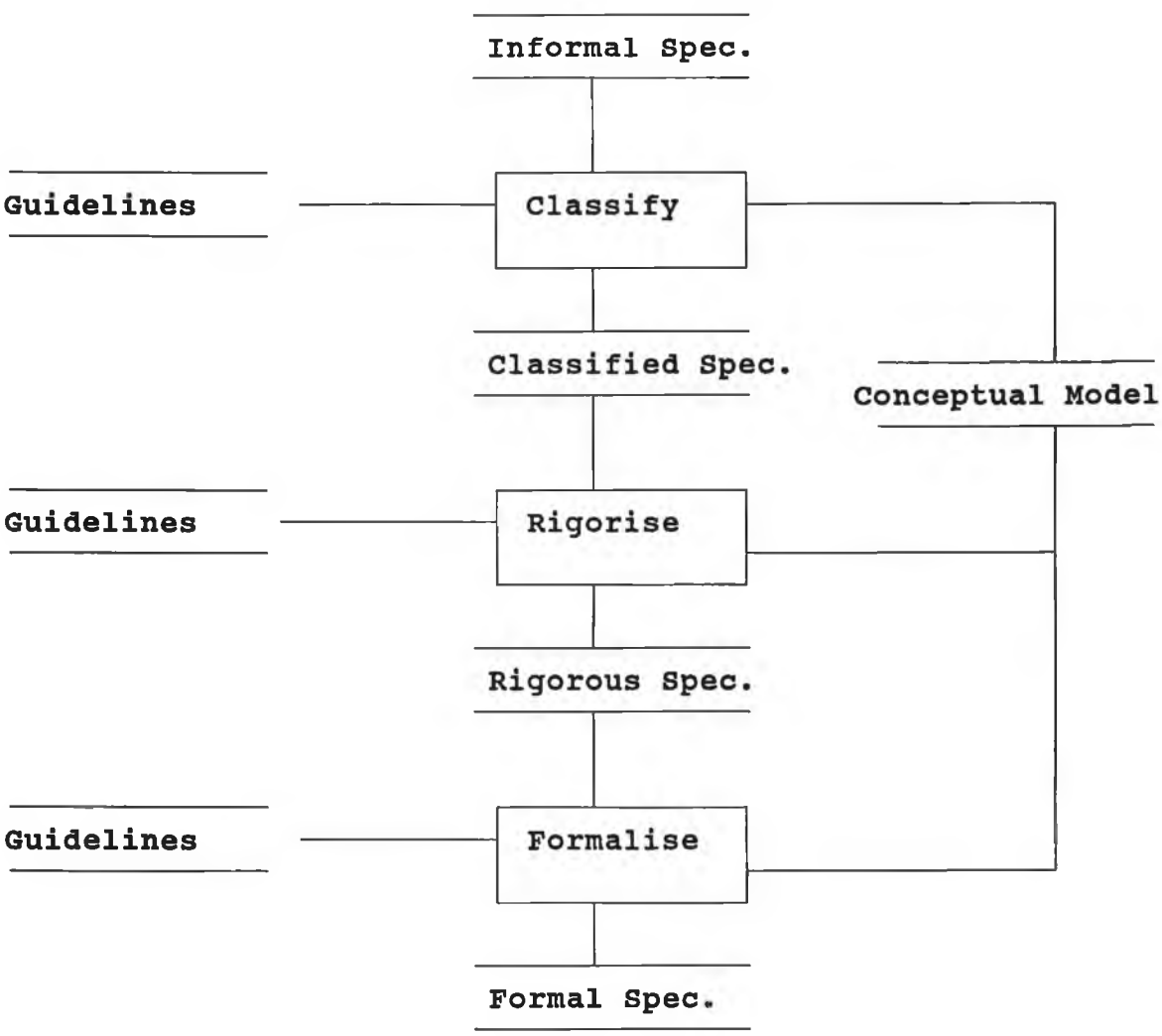
[2] ... To be a first step towards formal specification.

[3] ... To provide a basis for automated tool support.

[4] ... To ease the process of modifying the specification.

The conceptual model developed in SPECS presents a consistent set of IBC concepts, e.g. action, function, interaction-point. The classified specification is a document which presents the information contained in the informal specification in terms of these conceptual model concepts. The rigorisation process concentrates on the system to be specified. Its objective is to analyse the system described in the classified specification. The rigorisation analysis is based on the widely known non-application oriented techniques such as **entity-relationship, data-flow, state transition** techniques. Diagram 5.1., below models the transformation from informal to formal specifications.

Diagram 5.1.



From Informal To Formal Specification

The objectives of the tool were as follows -

[1] ... It should be able to store the classified specification concepts in the Eclipse two tier database.

[2] ... It should allow the user to develop rigorous specifications, i.e. nested levels of Data-Flow Diagrams. The user would be prompted at each stage with the various relevant elements of the classified specification.

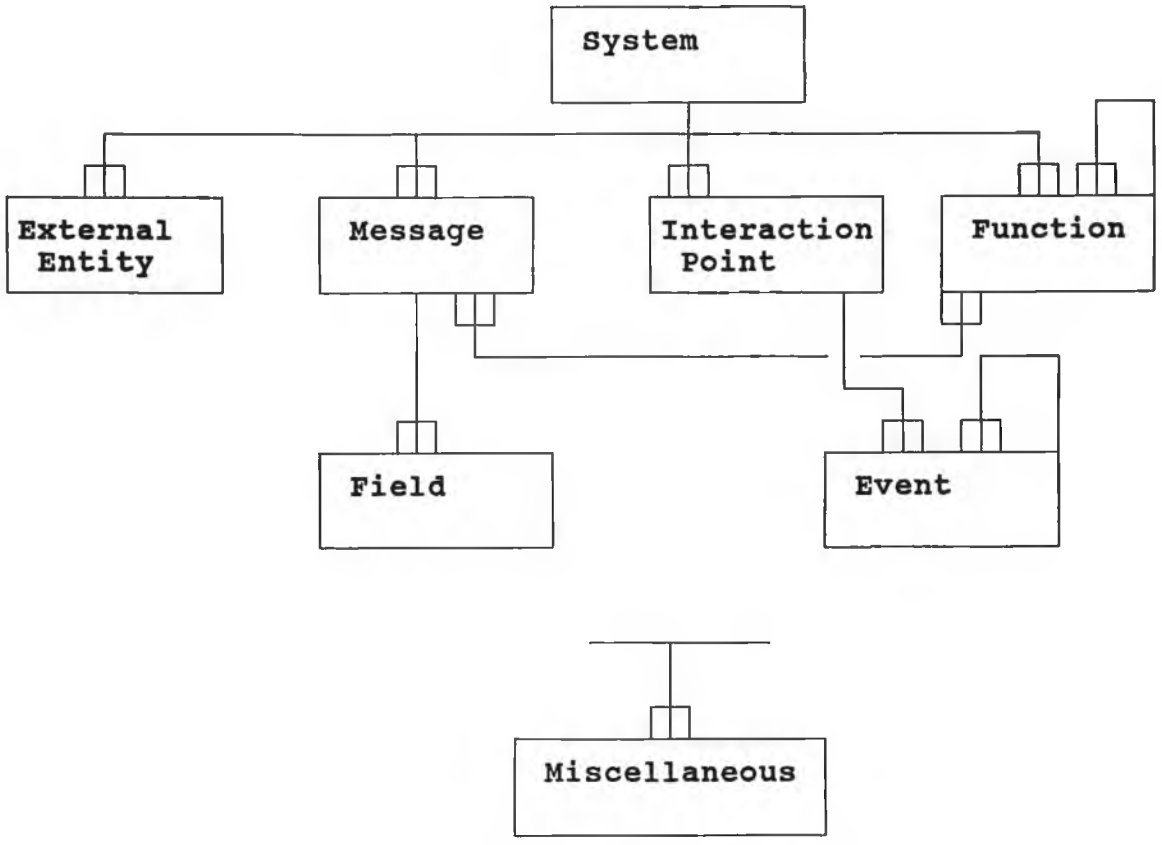
One of the partner members in the SPECS project produced a document [Lari] which was supposed to be the classified specification for an example network node called **Al's node**. I had hoped that this document would be in a form which would allow me to develop a parser, using YACC and LEX, to build a classified database consisting of the conceptual model concepts. Unfortunately the classified specification was not in a form which permitted parsing as it contained many inconsistencies and ambiguities. The only alternative was to build the database by hand. Before this could be done, I had to go through the classified specification and extract the conceptual model concepts along with their attributes and relationships. It was this operation which formed the basis of the schema design for the classified database.

The rigorous specification to be developed would consist of a high level **context diagram** and nested levels of **data-flow diagrams**. Each **system** object in the context diagram could be exploded into nested levels of data-flow diagrams.

5.3. Developing the Classified Database

As was already stated, the design of the classified database schema was facilitated by the extraction of the conceptual model concepts from the classified specification. I modelled each specification as a first-tier object. Within the specification object, each concept was modelled as a second-tier database entity. The concepts which were extracted from the classified specification and the relationships that existed between them are shown in diagram 5.2., below.

Diagram 5.2.



Eclipse Classified Database Model

The actual schema that was developed can be seen in Appendix K. From this schema it can be seen that every entity (concept) has a **name** attribute, which is used to distinguish entities of the same type. Every entity also has a link to a sequence of **miscellaneous** entities. The miscellaneous entities hold supplementary data as well as data which cannot be formalised. Every entity, with the exception of **external entities**, has additional attributes.

A **system** entity has a link to a sequence of external entities, a link to a sequence of **interaction point** entities, and a link to a sequence of **function entities**.

A **function** entity has a link to a sequence of other function entities, links to sequences of input and output message entities, a link to a sequence of event entities, and a string attribute which

holds a text description of the function, called **descr**.

An **interaction point** entity has two integer attributes **max** and **initial**.

A **message** entity has a link to a sequence of **field** entities. It has origin and destination links which can be of type external entity, system entity or interaction point entity.

An **event** entity has a link to a sequence of other event entities, a **rule** attribute which is an enumerated type signifying if the event is part of a choice, iteration or sequence, a string attribute, **choice**, which defines a condition for choosing the event, an integer attribute, **type**, which shows the message type, e.g. reception of a message, a string attribute, **details**, which is a text description of the event.

A **field** entity has a boolean attribute, **optional**, which indicates whether or not the field is optional, an enumerated type attribute, **data-type**, which indicates the data type of the field, i.e. string, integer or boolean, and a string attribute, **value**, which holds the the text value of the field.

A **miscellaneous** entity contains an enumerated type attribute, **type**, which indicates the type of the miscellaneous entity, and a string type attribute, **descr**, which contains the text description of the miscellaneous entity.

5.4. Storing the Data in the Classified Database

5.4.1. Introduction

Having designed and created the classified database schema, the next step was to store the classified specification concepts in the database. This required the development of an FDL program to implement the required user interface. In chapter 2 the structure of the Eclipse User Interface (UI) is discussed. In this chapter, the discussion will centre on the development of a UI, following the Eclipse house-style guidelines, which allowed the user to input classified specification concepts along with their associated attributes. As well as discussing FDL there will also be a discussion of the functions written which interface to the database and the UI. These

functions retrieve data from the screen and store it in the database and vice versa.

5.4.2. Developing the Tool's User Interface

FDL is designed to provide the tool writer with a means of describing the structure and content of a tool's UI in a device independent fashion and in a form which does not require building the UI into the application code [Gree]. The use of FDL ensures that it is possible to modify the UI in the light of experience without the need to rebuild the tool. The fact that FDL is device independent ensures that the UI is portable and that the tool is not tied to any specific workstation. Diagram 2.5. (page 51) shows the hierarchy of UI objects supported by the Eclipse Appilcations Interface (AI). FDL describes this hierarchical structure together with the contents of these objects. Each object definition specifies the lower level objects it contains in terms of -

[1] ... The class of the object

[2] ... The presentation style of the object

[3] ... The position and extent of the object relative to its parent object (container).

In Example 5.1. there is an example of a declaration of a screen, a window and several frames.

Example 5.1.

```
---------------------------- SCREEN

.SCREEN [0]
        DEAD_FONT)$EI_DEAD_FONT
        LIVE_FONT)$EI_LIVE_FONT
        VALUE_FONT)$EI_VALUE_FONT
        SIZING_FONT)$EI_SIZING_FONT
W1:     .WINDOW [1]             (51,124) (0,0)143,48
        GRAPHIC)#/design_editor.icon
.end [0]


---------------------------- WINDOW W1

.WINDOW [1]
Tmenu:   .TOOLMENU [10]    (0, 0)          | Tool Facilities |
F1:      .FRAME [11]       (0, 0)->,25
F2:      .MSG_FRAME [12]   (25, 0)->,1
F3:      .GRAPHIC [13]     (26, 0)->,++
.end [1]
```

In this example the **SCREEN** object contains a window **W1**. This window is declared to have an origin at 0,0 of the containing screen. It is 143 columns wide and 48 rows down. Its icon, which is held in a file called design_editor.icon, will appear at character position 51,124. Within the W1 object the standard tool facilities menu is declared. Three frames are also declared. **F1** is a formatted frame, i.e. one which can be sub-divided into panes and fields. It is positioned at row 25, column 0 of window W1 and extends to the right boundary and 25 rows down in W1. **F2** is a message frame, i.e. all messages will be routed by the AI to this frame. It is positioned at row 25 of window W1 and extends to the right boundary and one row down in W1. **F3** is a graphics frame which cannot be controlled by FDL. It is this frame which is controlled by the design editor (DE). F3 is positioned at row 26 , column 0 in W1. It extends to the right and lower boundary of W1 and one row down.

In order to build the classified database it was necessary to present a UI to the user which would allow him/her to enter the classified specification concepts. In Appendix L there is an extract of the FDL file used to create a pane for entering data for a particular sub-system within the classified specification.

A lot of the power in FDL is provided by **built-in functions (BIFs)**. BIFS are functions that are linked in with the AI and can be called from the FDL. Thus when building the classified database, BIFs were developed for creating the various database entities and storing and retrieving their data. In Appendix L there are several calls to BIFs. Appendix M shows some of the C code I developed for these BIF calls.

When the user enters the name of the sub-system in the **System Name** sign, c.f. Example 5.2., the syntax of the data entered is checked. The data entered is compared with the **SYNTAX** attribute of the system sign. In the extract, below, this states that the data entered in the field must begin with an alphabetic character and then contain only alphanumeric characters.

Example 5.2.

```
.SIGN [631]
  SYNTAX)   ^[a-zA-Z][A-Za-z_0-9]*$
  UPDATE) ('WP3_create_entity &[/W1/F1/Sys_Pane] '.!system^name' 'system' '!system' '2'')
  ERROR)   'Must start with an Alphabetic character and then contain only AlphaNumerics')
.END [631]
```

The * repeats the previous character only, i.e. the alphanumeric characters. If there is an error in the syntax, the error message associated with the **ERROR** attribute is displayed in the message frame. If a valid name is entered the **WP3_create_entity** function is called. This function's code is shown in Appendix M. When a BIF is invoked from FDL its parameters are passed in argc/argv format in exactly the same way as the UNIX **main** process. This means that they can be tested independently of the AI and can have any number of parameters. The AI always passes the FDL name of the BIF as its first parameter.

The **get_field** function called from WP3_create_entity retrieves the value of a UI field. The parameters passed are the token of the containing pane, i.e. the sub-system pane, the name of the field in the pane, i.e. **name** (c.f. Appendix L), a buffer to hold a string value to be returned, an integer to hold the choice id value for state selector fields, and a boolean switch which indicates whether the field is a string field, e.g. a sign, or a state selector field.

The get_field function shows some of the possible calls to the AI C library. The **AI_get** function retrieves the value(s) of specified attributes of a given item. AI_get takes different, null terminated, parameter lists based on an attribute which always comes after the container token parameter. For example, the AI_CHILD_BY_ID attribute causes the token of the required child item to be returned, while the AI_VALUE attribute causes the value of a field to be returned.

The value for the name field in the sub-system pane is used by WP3_create_entity to form a derived attribute (c.f. chapter 2, The Eclipse Database Attributes) which is passed to the DBI function **DBI_value_of_attribute**. This function returns EI_OK if the sub-system's database entity already exists. If this is the case, a switch value, passed to the WP3_create_entity as part of the argv parameter list from FDL, is used to determine which function is called to display the values of the relevant entity's attributes.

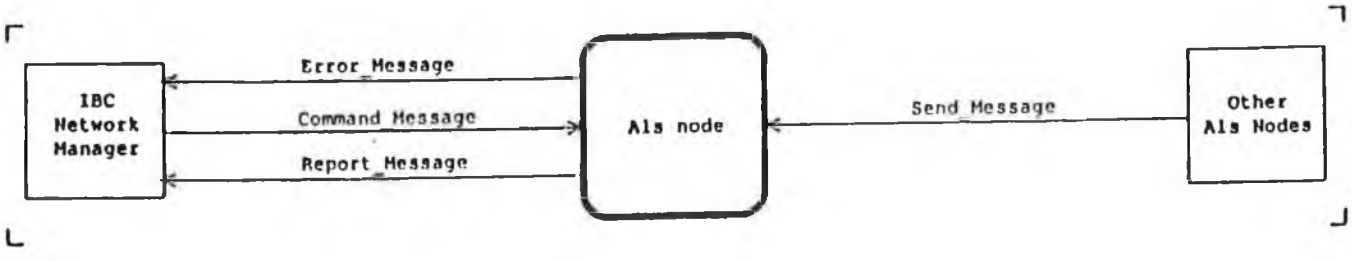
If the return value is not EI_OK, the **create_wp3_entity** function is called. This function creates a second tier entity. Two of its parameters are used in the **DBI_create_entity** function. These are the entity type, i.e. **system**, and the the attribute, i.e. **!system**. The token of the newly created node is then passed, along with the value of the entity's name, to the **DBI_attribute_becomes** function. This function store the entity's name in its **name** attribute (c.f.

Appendix K).

5.5. Rigorising the Classified Specification

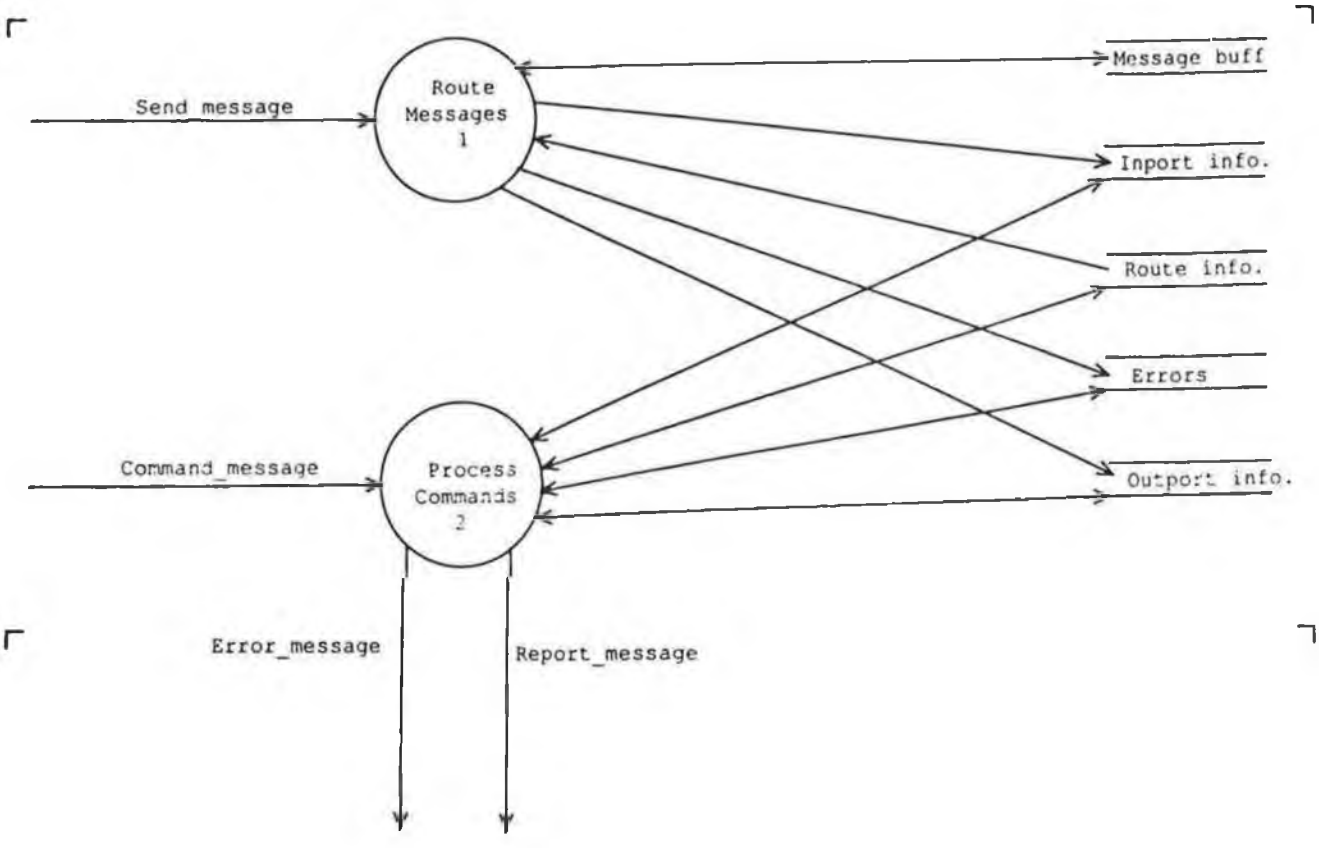
The tool that was developed had two distinct parts. The first part, the building of the classified database, has already been discussed. The second part allowed the user to rigorise the data in the classified database. Rigorisation, in this context, meant the generation of two types of diagrams, **context diagrams** and nested levels of **data-flow diagrams**. For the example network node, that was stored in the classified database, the context diagram to be generated can be seen in Example 5.1.

Diagram 5.1.



Each rectangle in the diagram constitutes an external entity in the classified database. Each rounded box constitutes a sub-system in the classified. The links between the external entities and the sub-systems are messages from the classified database. In the tool the user could explode a sub-system and generate a data-flow diagram as in Diagram 5.2., below.

Diagram 5.2.



The rounded boxes in the data flow diagram are functions from the classified database. The parallel lines are data stores and the data-flows are messages from the database. Before discussing how the diagrams were generated it is necessary to discuss the **graph description language** used by Eclipse to generate diagrams.

5.6. Graph Description Language (GDL)

5.6.1. Introduction

GDL was developed to provide a description language which would allow tool builders to specify the symbols and to describe the constraints of a particular design method [Robs2]. Diagrams can be expressed as **graphs**, i.e. a structure composed of nodes and arcs. Depending on the design which is used, the nodes and arcs in the graph have different symbols and semantics. All methods allow implicit partitioning of the design graph by using different diagrams to show the design in greater or lesser levels of detail.

In GDL **NODES** and **LINKs** are the fundamental types used and every object which is represented graphically should be sub-types of NODE or LINK. These are the base types with a type hierarchy derived from these. A type with no sub-types is known as a **concrete type.** Only concrete types are visible on the design graph.

GDL also allows the creation of **dependent nodes** and **links.** This occurs where a node or link may own a set of dependent nodes and links and each dependent node or link may have a reference back to its owner.

5.6.2. Type Declarations

Type declarations are used to associate a name with the fundamental objects in the design method. When a type is defined to be of base type NODE, a parameter list may be associated with that type. This list allows statements to be made about links to that node type. The parameter list may have an input part , where the node is the destination of a link, and an

output part, where the node is the source of a link. Bi-directional links are supported by allowing the same link to appear in both the input and output parts. When a type is defined to be of base type LINK, the parameter list may have two parts which indicate which node types the link is permitted to connect to. The **In_part** specifies the type of the source of the link and the **out_part** specifies the type of the destination of a link. In Appendices N and O one can see the GDL files that were developed for the context diagrams and data-flow diagrams. The type declarations in these files show the various NODE and LINK types used in these diagrams. Example 5.3., below, shows an extract of the type declarations in the context diagram GDL file.

Example 5.3.

```
Type PROC_ENV Is NODE
        (in_links: In Bag Of message;
         out_links : Out Bag Of message)

Type external_entity Is PROC_ENV

Type message  Is LINK
        (source_end : In PROC_ENV;
        dest_end    : Out PROC_ENV)
```

This type declaration declares a NODE type **PROC_ENV**. Its parameter list declares a BAG of input links, **In_links**, and a BAG of output links, **out_links**. A **BAG** is a form of composite type which allows duplicate links to exist. The **external_entity** type is a sub-type of PROC_ENV and thus inherits PROC_ENV's attribute list. As external_entity does not have any sub-types it will be displayable on the screen. The **message** type is declared to be of type LINK and thus links NODE types together. Its parameter list states that its origin, **source_end**, is of type PROC_ENV as is its destination, **dest_end**. This means that any NODE types which are sub-types of PROC_ENV can be origins and destinations of **message** LINK types.

5.6.3. Use Declarations

Use declarations are the means by which the user defines how a particular type is to be represented on a diagram. The representation of a particular type is defined as a **symbol** plus a set of named labels. Example 5.4., below, shows an extract from the use definitions in the context diagram GDL file.

Example 5.4.

```
For external_entity Use
        {
        SYMBOL(EXT_S)
        ++ EXT_NAME(STRING) : "attribute = ext_name"
        }
```

The use declaration is used to specify which defined shape should be used to represent a GDL type, and what other labels the type may have. In the above expression, two labels are declared. A **picture label** is represented by a shape. The special picture label **SYMBOL** is mandatory for concrete types. In the above expression the concrete NODE type, external_entity, has a SYMBOL picture label which is represented by the defined shape **EXT_S**, c.f. Shape Declarations, Example 5.8. A **text label** has a value which is visibly represented as a text string. Text labels can be of type **STRING, INTEGER, DATE** and **ENUMERATION**. In the above expression a text label, **EXT_NAME**, is declared which is of type STRING. There is an additional attribute string associated with the external_entity NODE type. This is used as a link with the Eclipse DDL file which will be used to generate a schema to allow the shapes created in the diagram to be stored in the Eclipse database. The **ext_name** attribute will, thus, be declared as a string type attribute of the second tier entity used to hold an external_entity in the database.

5.6.4. Shape Declarations

Shape declarations are the means by which the user specifies how a particular symbol should be drawn. Two distinct types of shapes exist within GDL - one type for link symbols and one type for all other shapes. Node symbols use a **SHAPE** type while link symbols use a **LINKSTYLE** for link symbols. In Example 5.5., below is an extract from the context diagram GDL file.

Example 5.5.

```
Shape EXT_S Is
        {
         Box        0,0 : 90,90
        }
Linkstyle MESS_L Is Start Arrow
Linkstyle MESS_L Is End Arrow
Linkstyle MESS_L Is Start Arrow End Arrow
```

In the above the node symbol **EXT_S** is defined to be composed of the **box** primitive. There are several pre-defined primitives in GDL such as point, line, box, rounded box, diamond, triangle and ellipse. The position of a shape is relative to the shape's top left hand corner which is at position (0,0). A linkstyle is basically a line with optional arrows at either end. In the above example, **MESS_L** is defined to be a link symbol with three alternative settings, i.e. Start Arrow, End Arrow or bi-directional.

5.6.5. Compiling the GDL File

Method descriptions written in GDL are compiled into tables which are used to drive the DE. The compiled GDL definitions are accessed by the DE from a **method directory** object. This object is of type **design_method_directory** and is accessed by the DE via the path -

_/eclipse.tools/<diagram_type>.design_method

where <diagram_type> is the parameter specified on the command line invocation. The GDL table are accessed from the method directory via an unkeyed link of type **.gdltable**. The installation of a method needs to include the setting up of appropriate method directories, i.e. one for the context diagram and one for the data-flow diagram. For the data-flow diagram the following method directory was created -

_/eclipse.tools/dfd_diagram.design_method

5.7. Generating the Rigorous Diagrams Using the DE

5.7.1. Introduction

The GDL file only forms one of the three essential parts which are necessary to have a complete DE application. It is necessary to have an SDS to facilitate the storage of the generated shapes in the Eclipse database. The DE also expects its own FDL file which has certain standard menus and state selectors which must be modified for the particular diagrams to be displayed.

5.7.2. The DE DDL Files

The DDL files developed for the context diagram and the data-flow diagram can be seen in Appendices P and Q respectively. The context diagram DDL defines a fine_structure_object_type of **drawing**, which has all the attributes required by the DE. It also has method-specific data through the modelling of nodes, links and labels. The attributes required by the DE are imported from the **mmi** SDS. The **diagram_object** IDLE definition contains the generic attributes required by the DE. Nodes and links are defined as abstract types which have method specific subtypes. In the DDL two class types can be seen, i.e. **NODE** and **LINK**. The concrete object types must have names which are identical to those specified in the GDL description of the method. In the context diagram DDL file both **external_entity** and **system** are declared to be sub-types of NODE, while **message** is declared to be sub-type of LINK. This is logically equivalent to the declarations in the GDL file. Each of these object types have attributes associated with them. Example 5.6., below, shows extracts from the context diagram's GDL and DDL files showing equivalent DDL and GDL attribute declarations.

Example 5.6.

FDL ----> external_entity =>
ext_name : String;

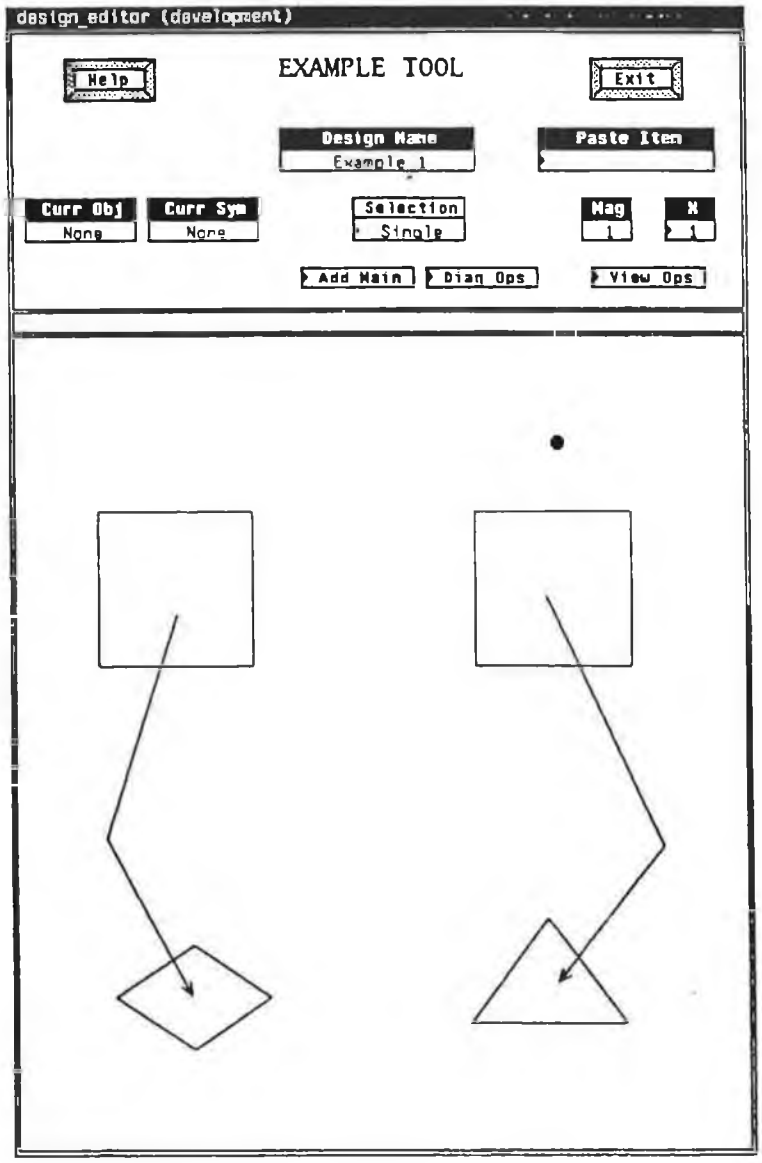
GDL ----> EXT_NAME(STRING) : "attribute = ext_name"

These object type attributes all have identical names to the equivalent attributes in the GDL file. For example the attribute string **attribute = ext_name** ties the GDL label **ext_name** to a physical value in the database. Since the label is a STRING text label, the attribute must be declared to be of type string. If the label has not been added or has been deleted then the attribute is set to the null string.

5.7.3. The DE FDL Files

Example 5.7, below shows the layout of a standard DE user interface. The layout of this UI and the hierarchy of object types are fixed and most of these cannot be altered.

Example 5.7.



The default FDL file associated with an instance of the DE defines several panes which must exist. The **Common Pane (P1)** is permanently visible in the control panel. It contains the Exit and Help buttons, a light to indicate read-only invocation of the DE and an output sign to contain the name of the design. The descriptions of these buttons and signs cannot be changed. The **Normal Pane (P2)** occupies the lower part of the control panel frame, and is visible after initialisation. This pane contains the main fields used to drive the DE. Some of the more important ones are as follows -

[1] ... Current and New Object

The current object sign displays the type of the currently selected object, while the new object displays the type of a new object being added to the diagram. Both of these signs are maintained by the DE itself. Both signs occupy the same position in the pane and their visibility is controlled by the DE.

[2] ... New Symbol and Current Symbol State Selectors

The new symbol state selector appears during an add operation. The possible choices indicate the allowed alternative symbols for the object being added. In Appendix R this state selector is set up with the alternative symbols to be used for a **message** symbol in the context diagram. It is the DEs responsibility to make unavailable those choices which are not applicable to the particular object type being added. The DE needs to understand which symbol each of the choices represents in order to provide the correct set of alternatives. This is achieved by utilising the FDL **DATA** attribute of each alternative. The format of this attribute is as follows -

<shape_name>.<alternative>

An example of such an attribute, from Appendix R is -

(1)Start Arrow
DATA)MESS_L.1

The <shape_name>, i.e. **MESS_L**, is as defined in the context diagram GDL, c.f. Appendix N. The alternatives are numbered consecutively starting from 1 in the order they appear in the GDL

file. The text **Start Arrow** is the option value which will indicate to the user what the symbol is. The **current symbol** state selector appears when a single object is selected. The possible choices should be the same as in the new symbol state selector.

[3] ... Add Subordinate Menu

This menu contains the choices which allow the adding of subordinate objects to the currently selected object. In a similar fashion, the new symbol state selector must contain all possible choices for subordinates. The DE ensures that only the relevant choices are made available at any particular time. This is done by putting the encoded string in the DATA attribute associated with each choice, c.f. Appendix R. ***Such an encoded string would look as follows for the context diagram.***

(1)External Entity Name
DATA)+T/EXT_NAME:External Entity Name

The **+T/** form of encoding is used for a text label subordinate of the GDL name **EXT_NAME**, c.f. Appendix N. The **External Entity Name** string following the **:** is the **usertype** part of the encoding which is set as the value of the current object. Both the usertype and the choice string displayed in the menu need not be the same as the type name specified in the GDL and are expressed in end-user terms. The add subordinate menu should always contain choices for box, line and text annotations.

[4] ... Add Primary Menu

The add primary menu is similar to the add subordinate menu except that it is used to add **primary** object to the design, i.e objects which are not subordinate to another object, e.g. for the context diagram these would be **system** and **external entity**. Primary objects are limited to nodes and annotations, hence the appearance of the message link object in the add subordinate menu, c.f. Appendix R.

[5] ... Current Object Operations Menu

This menu has choices which are the allowable operations on the current selected object. This menu must contain all the possible choices and the DE subsets the choices using the encoding in the DATA attributes of the choices. For the context diagram a method-specific operation was implemented. This **Open** operation allowed method-specific code to be written which allowed a system object in the context diagram to be exploded into a level 0 data-flow diagramming tool. In the GDL description of a system an **open** attribute was associated with a system. This allowed the open operation to be available when a system was selected.

In Appendix R. the following choice can be seen in the current object operations menu -

(6)Open
UPDATE)('DE_open')
DATA)OPEN

On selection of the choice the procedure DE_open was called. How the exploding was performed is discussed in the section dealing with the invocation of the data-flow-diagramming tool.

5.7.4. Prompting the User from the Database

At all stages of the development of the rigorous specification the user is prompted with relevant information from the classified database. In this section I will discuss the various types of data which are retrieved from the database to aid the generation of the diagrams. The generation of the rigorous specification actually required the development of two tools, one for each type of diagram.

The context diagram to be generated for the example network node can be seen in Diagram 5.1. The classified specification concepts relevant to the context diagram are as follows -

[1] ... Systems

[2] ... External Entities

[3] ... Messages

When the user wishes to enter a System he/she can select the system sign and view the list of available systems in the classified database. The list is generated using the **LIST** attribute in the FDL file. When the user chooses to view the list of available systems the **WP3_retrieve_system_fields** function is called, c.f. Appendix S. This function navigates the sequence of systems and sends the name of each one to **STDOUT** where it is trapped by the AI and entered into a drop-down menu. For systems the FDL entry in the FDL file is as follows -

```
.SIGN [824]
        L) WP3_retrieve_system_fields &[/W1/F1/Sys_Disp_Pane]
        U) sub_sys1 (^[sub_sys])
.end [824]
```

One of the major restricting factors encountered when using the DE is the fact that the tool developer is limited to using the frames defined in the DE's FDL file. The most one can do is juggle around with the frame sizes. This is particularly troublesome when one wishes to prompt the user for a lot of data. For the data-flow diagram the classified specification concepts which were required for prompting purposes were as follows -

[1] ... **Systems**

[2] ... **External Entities**

[3] ... **Messages**

[4] ... **Interaction Points**

[5] ... **Functions**

When the user wished to get details of a particular message to be entered into the diagram he/she selected the function sign in the sub-system pane. The user then selected a particular function from the drop-down menu consisting of all the messages of this system, c.f Appendix S - **WP3_retrieve_messages**. This caused the message pane to be overlayed on the bottom half of the control panel. The field values of the selected message entity are displayed on the message pane, c.f. Appendix S - **WP3_retrieve_message_fields**. The same procedure was

followed for interaction points and messages with the pane holding the relevant fields overlayed on the bottom half of the control panel.

5.8. Invoking the Data-Flow Diagram Tool

Each system entity within the context diagram was envisaged as containing a level 0 data-flow diagram. Each function within the level 0 data-flow diagram was envisaged as containing a level 1 data-flow diagram and so on. From this idea of containers came the idea of exploding an object on the screen in order to see the lower level diagram it contains. In the **DE FDL FILES** section there is an example of the **Open** option in the objects operations menu. This is a valid operation on a system object in the context diagram since, in the context diagram's GDL file there is an **open** attribute associated with the system object. When the Open option is chosen the **DE_open** function is called. This is a standard BIF which results in the DE's **Method Handler** being called. The method handler is called when method specific actions are required. The reason for entry is passed as a parameter together with the information required to perform the necessary action. The check for an open call in the method handler looks as follows -

```
case DE_REQUEST_OPEN:
        return_code =
                action_open( values[0], entity_ids[0]);
        break;
```

The action_open function can be seen in Appendix T. This function retrieves the entity token of the system entity which is to be exploded. The value of this entity's **ext_name** attribute is then retrieved. This name is used as a key to the cardinality-many link to a data-flow diagram first-tier object. This enables each system object to have an indirect link to a single data-flow-diagramming first-tier object. The data-flow diagramming tool is invoked using the PCTE **callp** function which allows an asynchronous process to be called. When the user exits from the data-flow diagramming tool control is returned to the context diagramming tool whose state will have been preserved.

5.9. Conclusions and Future Enhancements

In this section I will outline some of the conclusions that I came to when devloping the tool using Eclipse , and in particular, the conclusions I came to with regard to the DE and GDL. I will also outline the future enhancements envisaged for the tool.

5.9.1 GDL

I found GDL to be both easy to use and powerful. The syntax of GDL is concise and much less messy than FDL. When I became familiar with the GDL syntax I found that it took very little time to define diagrams.

One major benefit to using GDL is the fact that one can define diagram constraints. Using this feature one can define which objects have links between them, the types of the links and how many. As well as defining the links that may exist between diagram objects, it is also possible to define the objects to be subordinate to other objects ,i.e. one can define objects which own certain other objects.

The major drawback with GDL is the fact that when GDL source files are compiled they form static entities consisting of a number of tables to be used by the DE. Thus the usefulness of GDL is determined, not by its own features, but rather by the ease of use of the DE which operates on the GDL tables. It is thus impossible to see what a diagram would look like using the GDL definitions without first developing a schema definition for the diagram and then modifying the DEs FDL menus and state selectors. Thus, there is no facility for getting the look and feel of the diagram correct, before actually hooking it into the rest of Eclipse. I feel that such a prototyping facility would be an excellent addition to the Eclipse TBK. It could either take the form of a library which the user could use to develop prototyping tools or a generic prototyping tool could be developed which would act on the GDL tables, directly, without the need to create database objects.

5.9.2. Design Editor

I found that the DE was an extremely useful tool when used for very straightforward diagram development. Beacause it is a generic tool, it is inherently restrictive. I found it frustrating to find that when using the DE one is restricted to using only the three FDL frames declared in the DE's FDL file even though any new frames would have no impact on the operation of the DE. The scope for modifying this FDL file is very limited. Because the tool that was developed required the use of part of the screen for prompting the user with classified database information, it would have been nice to use completely new text frames, but this was forbidden. I envisaged, in my original design, that the tool would have the control panel on one side of the screen while the graphical area would be on the other side. Even such a seemingly minor change was not possible without developing two tools, one for prompting from the classified database, and an A5 version of the DE which would occupy half the screen and be used to develop the diagrams. I found the linkage between the DDL, FDL and GDL files ,required in order to use the DE, to be very involved and intricate. The syntax used in the menus and state selectors of the DE's FDL file is not very intuitive, though luckily one only has to modify the file in a small number of places. It is thus essential that any tool developer, wishing to use the DE to develop graphical tools, should be fully aware of the following -

[1] ... The files are required by the DE.

[2] ... The database entities need to be declared in the DDL file and what attributes are to be associated with these entities. These entities and attributes will be equivalent to the ones declared in the GDL file.

[3] ... The values are to be stored in the modifiable menus and state selectors in the DE's FDL file, i.e. create the link between the UI and the GDL declarations.

[4] ... The parameters are required in order to invoke the DE.

5.9.3. Future Enhancements

The tool itself will be greatly enhanced in the next phase of the SPECS project. One feature which will be added is a hypertext facility to allow the user to go from the original informal specification to a matching section of the classified database and onto the rigorous specification and back again. This will require a schema definition for the original informal specification and the definition of links between the various objects. It was not possible to implement this enhancement, in the current version, since I was constrained to use the control-panel metaphor only. This meant that it was infeasible to display large amounts of text. It was also impossible to select text from within any of the UI fields. Such a feature would be necessary in order to implement a hypertext-style facility. It is hoped that in future releases of Eclipse a text editing tool will be provided to enable text selection and that the restriction on the number of frames that can be declared in the DE's FDL file will not be as restrictive as it currently is. If this is not the case then it will be very difficult to display the required information.

Another enhancement which is enviaged is the translation from the rigorous specification to a LOTOS formal specification. One of the major benefits of the DE is the fact that diagram objects are stored as entities in the Eclipse database. The problem of mapping from the database , generated in the rigorisation process, to LOTOS is currently an active area of research within DCU.

Chapter 6.

Conclusions

6.1. Introduction

When I got the job of evaluating Eclipse I knew very little about software engineering environments. It was very obvious, from the start, that this was a relatively new area of study, but one in which widespread interest was growing due to the potentially large productivity gains which could be gained by use of SEEs. From the start I adopted a set of goals which I hoped would enable me to fully understand the capabilities and the weaknesses of Eclipse as a platform for the development of integrated tool sets. The initial set of goals was as follows -

[1] ... To fully understand the problem space which Eclipse was trying to address.

[2] ... To fully understand the architecture of Eclipse. This was a pre-requisite before I could embark on developing and integrating tools in Eclipse.

[3] ... To try to integrate different tools and toolsets into Eclipse, noting the various good and bad points of the system.

[4] ... To try to develop a completely new tool in Eclipse, hopefully one which required a graphical interface. As Eclipse is envisaged as a platform for developing full SEEs I considered it necessary to develop a native tool for Eclipse. Eclipse also aims to provide support for the development of graphical design methodologies. Thus by developing a graphical tool I hoped to be in a position to comment on the functionality provided by Eclipse to the tool-builder hoping to implement graphical design methodology tools.

In the rest of this chapter I will discuss the various areas of Eclipse/PCTE which I feel are of importance. I will discuss the conclusions which I arrived at,i.e. the various good and bad points and the paths which should taken or avoided if one is developing or integrating tools in Eclipse/PCTE.

6.2. The Architecture of Eclipse/PCTE

Both PCTE and Eclipse are new developments. As such there are few people with in-depth knowledge of these systems. Thus before I could embark on developing or integrating tools I

had to become fully familiar with the architectures of both PCTE and Eclipse. This was a very worthwhile experience and one which I would recommend to anyone intending to use Eclipse/PCTE. The standard of the documentation for both Eclipse and PCTE is pretty dreadful and a lot of experimentation was required in order to become familiar with the architectures. This was one of the main motivating factors for the in-depth discussion of both architectures in chapters 2 and 3.

The main features of the Eclipse architecture are its use of a database to hold all the data relating to a project and the provision of a kernel of facilities which provide for all requirements of a tool with respect to execution, inter-process communication, input-ouput and database access. It was PCTE which provided this set of kernel facilities. The architecture of PCTE is very clever in that it preserves nearly all the functionality of raw UNIX while replacing the rigid hierarchical file-system of UNIX with a highly flexible object management system (OMS). This architecture greatly reduces the need for intermediate files, signals and piping among co-operating tools.

The major problem with PCTE , as it exists in its present form, is the fact the it is totally entwined with UNIX and requires a modified UNIX kernel to be installed before it can be used. This means that one is tied to a particular version of UNIX kernel and one cannot avail of the greater functionality which may be provided by later versions of the kernel. This became evident when version 4 of Sun OS became available, but could not be used since PCTE currently only operates on a modified version of Sun OS 3.5. It is hoped that by mid 1990 a layered version of PCTE will be available which will not require a modified kernel. This will also allow for the portation of PCTE and hence Eclipse/PCTE to various operating systems since only the layer which interface to the operating system will be effected by such portations.

The design of the Eclipse two-tier database is also very clever in that it encapsulates within the one data-model both coarse-grained and very fine-grained data. If the tool developer is familiar with the design of the PCTE OMS then it is very logical and straightforward progression to use the two-tiers of data-modelling provided by Eclipse.

6.3. The Eclipse Two-Tier Database

The database is at the centre of Eclipse. The designers of Eclipse recognised that the concept of an OMS needed to be extended because when dealing with advanced software engineering tools the contents of objects, i.e. files, are highly structured. By extending database concepts to the contents of objects Eclipse removes one the major problem areas within PCTE.

Eclipse provides data schema facilities in which the database designer can prepare a schematic representation of data at both tiers using the Eclipse Data Definition Language (DDL). The Eclipse DDL is very straightforward to use and is a logical extension of the DDL provided for PCTE. The second tier schema definition, within the DDL source file, is expressed in IDLE. A compiler is provided for compiling DDL sources. When compiled, one can include the defined schema in ones working schema and create objects and links as defined in this working schema.

One of the nicest features of Eclipse is its relatively small database interface (DBI) library of functions. The same interface is used to provide uniform access to both tiers of data. The functions interfaces are well designed with relatively few parameters.

The major problem I confronted when using the DBI was the lack of visibility into the database. This meant that when one had created a database, there was no way of seeing the layout of the database contents without writing a program to navigate and browse the database. This problem will be addressed in future releases of Eclipse with the provision of a language called EASEL which is an interpreted language for accessing various aspects of the Public Tools Interface. Another, more powerful, tool called the Structure Editor (SE) will also be provided in future releases. The SE will allow tool writers to construct tools for editing structured representations of IDLE data structures. In these tools, end-users will be offered a textual representations of the data that reflects the underlying database structure. By manipulating the displayed text, users will be able to change existing structures and create new ones. The provision of such tools will greatly increase the ease with which databases are built, modified and browsed.

I can envisage that speed could become a problem for tools using the Eclipse database. If a fully integrated SEE supporting the whole software life-cycle was to be developed on Eclipse then there would be a need for integrated compilers, translators, browsers, etc. For the case where a compiler was developed the parser would have to build a parse tree within the Eclipse database. In standard tool development a parse tree would be devleoped in memory, but in Eclipse the parse tree would have to be modelled in an Eclipse schema and each node would conform to a second tier object. Thus each node or link creation would require a function call to the DBI which would be much slower that the equivalent pointer operation using UNIX , for example. In a trial case I developed, using YACC and LEX to scan and parse a large CRL dump file, I found that it took over an hour to parse the input file using DBI function calls. To make Eclipse useful, for this form of tool, new techniques may have to be developed to speed up the database building process.

6.4. The Eclipse User Interface

One of the requirements for an integrated toolset is a common look and feel for all tools. This means that the UI for all tools behave in a predictable fashion, e.g. exit button always performs the same act for all tools. One area of commonality should be in the way tools are invoked. If tools are invoked in the same way it has the effect of giving a toolset a more complete feel. Eclipse adopts a 'control panel' house style which all Eclipse tools must conform to. Thus the tool developer is restricted to using a limited number of fields, e.g. light, sign, menu, etc., when developing a UI. A tool developed using the Eclipse house style is immediately recognisable as such and attains a degree of predicatibility in its behaviour in the user's mind.

The placement of windows and frames is specified using the Format Description Language (FDL). This is high level langauge which is simple to use and permits the rapid development of UIs. Unfortunately FDL does not fit nicely into the rest of the Eclipse system in that it is too verbose and the syntax checking tool is next to useless at finding even the most obvious synatx errors. Thus it is only at run-time that one finds out whether or not the FDL source defines the

required UI correctly. In its favour, it does provide a device independent interface for the definition of window contents and layout and the ability to change the layout of the window without needing to change the tool. Because UIs generally require a good deal of tinkering in order to get them as required, I found that the separation of the tool from the user interface was of great benefit in that the tool never needed to be recompiled when the layout of the UI changed.

The use of Built-In-Functions (BIFs) within FDL provides a way of invoking tool-specific code from FDL. I found this to be very straighforward and it provided great flexibility because one could change which functions were called from which field by just changing the FDL file, with no requirement to recompile.

One of the major problems with the version of Eclipse on the Sun workstations is that it does not conform to the X standard and instead uses Suntools. This has resulted in a certain degree of reluctance among prospective users to develop tools on Eclipse. It is hoped that a release of Eclipse will conform to the X standard by mid 1990.

6.5. The Design Editor

The Design Editor (DE) is a generic tool, that is table-driven and can be tailored to edit the graphical representation of a particualr design method , e.g. MASCOT, JSD, etc. The tables represent the syntax and semantics of the design method and are described in the Graph Description Language (GDL).

GDL is extremely powerful alllowing both syntax and sematics of designs to be expressed. The syntax is easy to use and because it is compiled, unlike FDL, it is possible to get the syntax completely right before running the tool. As stated in section 5.9. it would be nice to be able to see the layout of the diagrams which are described by the GDL without first having to develop a DDL source file to store the diagram objects and then modify the DEs FDL file.

The DE is a very useful tool given the scope of what it tries to achieve. It provides a quick way of developing graphical tools to implement design methodolgies. It would be much better if one

had greater scope to modify the DEs FDL file and tailor the appearance of tool. The linkage between the DDL, FDL, and GDL is to involved and the invocation of the DE, which requires several parameters, could be cleaned up.

6.6. Final Conclusions

The Eclipse/PCTE system is still a very new development which is constantly being updated. At the moment there is an unfinished look and feel to the system. Eclipse will need to be ported onto the layered version of PCTE and the UI will need to conform to the X standard before it can hope to be accepted outside the confines of research projects. The quality of the documentation is truly awful and it is be hard to see any commercial company using the TBK to develop SEEs without a vast improvement. On the other hand Eclipse/PCTE has a very sound design. The database, in particular, is excellently designed and when the Structure Editor becomes generally available it will greatly increase the ease with which the databases can be built, modified and browsed. Having used Eclipse for over a year, with all the frustrations which accompany it, I still do not hate it and can see the benefits it could hold in the future when it is in a more stable state.

Appendices

Appendix A.

```
--*************************************************************
--                                                            *
-- File Name        :    c_sds.w                              *
--                                                            *
-- Purpose          :    This DDL file describes an SDS       *
--                       to model a C programming database    *
--                                                            *
-- Written By       :    Sean Mac Roibeaird, Dublin City      *
--                                   University.              *
--                                                            *
--  Date Written    :    23rd February 1989                   *
--*************************************************************

new_sds c_sds is

--*************************************************************
--       Import needed definitions from other SDS's           *
--*************************************************************

import sys-file as file;
import sys-name as name;
import sys-sctx as sctx;
import env-user as user;

--*************************************************************
--                  Needed Definitions                        *
--*************************************************************

variant             :       string;
options             :       string;
optimize            :       boolean := false;
debug               :       boolean := false;
f_time              :       date    := "0101000070";
f_size              :       integer := 0;


--*************************************************************
--                  Define Object Types                       *
--*************************************************************

include_file        :       subtype of file;
object_file         :       subtype of file;
exe_file            :       subtype of file;
programs            :       subtype of file;
c_source            :       subtype of file;
c_program           :       subtype of file;
```

```
--*************************************************************
--                    Define Link Types                        *
--*************************************************************

progs               :       composition link
                            to programs;
c_prog              :       composition link (name)
                            to c_program;
c                   :       composition link (name)
                            to c_source;
o                   :       composition link (name)
                            to object_file;
h                   :       composition link (name)
                            to include_file;
exec                :       composition link (name)
                            to sctx;

cc                  :       reference link (name) to object_file
                            with
                                    options;
                                    optimize;
                                    debug;
end cc;

compiled_from       :       reference link to c_source;

ld                  :       reference link(name) to exe_file
                            with
                                    options;
end ld;

inc         :       reference link (name)
                            to include_file;

--*************************************************************
--                    Define the completed Objects             *
--*************************************************************

extend c_source
    with
    attribute
            f_time;
            f_size;
    link    cc;
            inc;
end c_source;

extend object_file
    with
    link    compiled_from;
            ld;
end object_file;
```

```
extend user
    with
    link progs;
end user;

extend programs
    with
    link     c_prog;
end programs;


extend c_program
    with
    link     c;
             exec;
             h;
             o;

end c_program;

end c_sds;
```

Appendix B.

```
/*************************************************************
*                                                            *
* Program Name    :     c_make.c                             *
*                                                            *
*                                                            *
* Purpose         :     This is a small tool to implement    *
*                       a make tool for C programs under     *
*                       PCTE. It was developed to            *
*                       demonstrate the OMS interface of     *
*                       PCTE.                                *
*                                                            *
* Written by      :     Sean Mac Roibeaird, Dublin City      *
*                                  University.               *
*                                                            *
* Date            :     24th February 1989                   *
*                                                            *
*************************************************************/


/*************************************************************
*                    Include  Files                          *
*************************************************************/

#include    <sys/types.h>
#include    <oms.h>
#include    <objstat.h>
#include    <linkstat.h>

main(argc,argv)
int  argc;
char        **argv;
{
int  i;
char        link_name[LINKSIZE];        /* name of a link          */
char        link[LINKSIZE];
char        path[64];
struct      objstat obj_status;         /* object status structure */
attrval     value;
short       d_kind;
int         size,num_links;
struct      linkstat current,list[20];
struct  defid typid;
time_t      mod_date;                   /* date of last modifications to C
                                           source file */
off_t       size_area;                  /* areas of storage for attrval*/
time_t      date_area;                  /*                         */
```

```
 if (argc < 2)
    {
    printf ("Usage = c_make filename \n\n");
    exit(-1);
    }
 if (getobjstat(argv[1],&obj_status) < 0)          /* get the object status */
    {
          perror("objstat");
          exit(-1);
    }

 value.v_integer = &size_area;

/* get the time of the last compilation */

 if(getattr(argv[1],
             "f_size",
             size,
             value.v_integer,
             &d_kind) < 0)
    {
          perror("getattr");
          exit(-1);
    }     /* has the file been modified since the last compile */
 if (obj_status.o_size > *value.v_integer)
    {
/*********************************************************************
 *                Update size attribute before recompilation          *
 *********************************************************************/

          *value.v_integer = obj_status.o_size;
          if (setattr (argv[1],
                       "f_size",
                       value.v_integer,
                       V_INT) < 0)
                  {
                        perror("setattr");
                        exit(-1);
                  }
          compile_program (argv[1]);
         exit (0);
    }
```

```
/*************************************************************
 * The source file has not been modified so check to see if
 * the include files it uses have been.
 *************************************************************/

/* Get a list of the links from the C source object */

if ((num_links = lslinks (argv[1],
                          "inc",
                          (char *) 0,
                          &num_links,
                          list)) < 0)
    {
        perror ("lslinks");
        exit(-1);
          }

if ( num_links > 0)
        {
/*************************************************************
  * Get the date of last compilation of the source file
  *************************************************************/
        value.v_date = &date_area;
        if (getattr (argv[1],"f_time",
                              size,
                              value.
                              v_date,
                              &d_kind) < 0)
                {
                        perror("getattr");
                        exit(-1);
                }

/*************************************************************
 * Get the type-id of the C-source object which will be used
 * to the get the names of the .inc links
 *************************************************************/

        if (gettype (argv[1],&typid) < 0)
            {
                perror("gettype");
                exit(-1);
            }
    /* Get the name of each link */
         for (i =0;i < num_links;i++)
             {
             if (linkname (&typid,
                           &list[i],
                           1,
                           link) < 0)
                           {
                                   perror("linkname");
                                   exit(-1);
                           }
        /* Form the full path to the include_file object */
```

```
                strcpy(link_name,argv[1]);
                strcat(link_name,"/");
                strcat(link_name,link);

                if (getobjstat (link_name,&obj_status) < 0)
                    {
                            perror ("objstat");
                            exit(-1);
                    }
            /* Has the include file been updated since last compilation */
                if (obj_status.o_ctime > *value.v_date)
                    {
                       *value.v_date = obj_status.o_ctime;
                    /* update the time of last compilation */
                       if (setattr (argv[1],
                                    "f_time",
                                    value.v_date,
                                    V_DATE) < 0)
                                    {
                                            perror("setattr");
                                            exit(-1);
                                    }
                       compile_program(argv[1]);
                       exit(0);
                    }
                }
            }
}
```

```
/*************************************************************
 *                                                           *
 *  Function        :     compile_program                    *
 *                                                           *
 *  Input           :     c_source - the program to be       *
 *                        compiled                           *
 *                                                           *
 *  Output          :     None                               *
 *                                                           *
 *  Purpose         :     This function invokes the UNIX C   *
 *                        compiler.                          *
 *                                                           *
 *  Written by     :      Sean Mac Roibeaird, Dublin City    *
 *                                  University.              *
 *                                                           *
 *  Date            :     28th February 1989                 *
 *                                                           *
 *************************************************************/

compile_program (c_source)
char *c_source;
{
    printf ("Recompiling %s\n",c_source);
    if  (startl ("/bin/cc","cc","-c",c_source, (char *) 0) < 0)
            {
                    perror("startl");
                    exit (-1);
            }
}
```

Appendix C.

Example Use of Eclipse DDL

The example will define a fine structure object called 'customer'. The data dictionary entry for customer might look like the following -

```
customer    =     { customer_name + customer_address +
                  customer_credit_balance }
```

The Eclipse DDL required to create this simple object would be as follows -

```
Sds customer_sds

Import eclipse-fine_structure_object;

-- define a customer to be a subtype of fine_structure_object

customer        <=     fine_structure_object;

-- define the layout of the contents of a customer object

customer        =>>
[
     ENTITY   ::=     UNIT | customer_details;

     customer_details  =>  customer_name       : String;
                           customer_road       : String;
                           customer_area       : String;
                           customer_city       : String;
                           credit_balance      : Integer;

];
```

In order to make the schema definition usable a number of steps have to be followed. Firstly, the source file has to stored within PCTE volume store. The link to the schema definition is a cardinality-many link **ddl**. Thus for the above schema definition a possible filename name would be -

customer.ddl.

The next step is to compile the source file using the **ddl_compile** command, e.g.

+# ddl_compile customer_sds.

This creates a customer_sds.ii file and a customer.ii file. The **ddl_install** command is used to

install the schema definition as a usable schema, e.g.

+# ddl_install customer_sds.

This command creates a **customer_sds.lsds** file in the **_/sdsdlr.sys** directory. In order to include the above schema definition in the working schema it is necessary to append it to the standard SDS's used in eclipse using the **setsch** command, e.g.

+# setsch customer_sds mml ui eclipse env.

Appendix D.

Message Class	Message Type	Description
Interaction Messages	Information,error, warning, fatal.	Interaction messages provide details of the status of an Eclipse interaction. Messages of this type are generated by tools for user information.
Input Messages	Comment	These messages allow the users to provide feedback to builders.
System Messages	Metric, diagnostic, tracking, console, security, system.	System messages provide system status information and are generated by tools for tool builders rather users.

Classes Supported by the Eclipse Message System

Appendix E.

Definition of a Node Generated by GAG in PASCAL.

```
NODE=
  record
    BROTHER:NODEPTR;
    case RULE:RULENRS of
    LISTRULE:(
        ELEMS:NODEPTR);
    R378GEN,R379GEN,R380GEN,R381GEN,R382GEN,R383GEN,
    R384GEN,R385GEN,R386GEN,R387GEN,R388GEN,R389GEN,
    R390GEN,R391GEN,R392GEN,R393GEN,R394GEN,R395GEN,
    R396GEN,R397GEN,R398GEN,R399GEN,R400GEN,R401GEN,
    R402GEN,R405GEN,R406GEN,R407GEN,R408GEN,R409GEN,
    R410GEN,R411GEN,R412GEN,R413GEN,R414GEN,R415GEN,
    R416GEN,R417GEN,R418GEN,R419GEN,R420GEN,R421GEN,
    R422GEN,R423GEN,R424GEN,R425GEN,R426GEN,R427GEN,
    R428GEN,R429GEN,R430GEN,R431GEN,R432GEN,R433GEN,
    R434GEN,R435GEN,R436GEN,R437GEN,R438GEN,R439GEN,
    R440GEN,R441GEN,R442GEN,R443GEN,R444GEN,R445GEN,
    R446GEN,R447GEN,R448GEN,R449GEN,R450GEN,R451GEN,
    R452GEN,R453GEN,R454GEN,R455GEN,R456GEN,R457GEN,
    R458GEN,R459GEN,R460GEN,R461GEN,R462GEN,R463GEN,
    R464GEN,R465GEN,R466GEN,R467GEN,R468GEN,R469GEN,
    R470GEN,R471GEN,R472GEN,R473GEN,R474GEN,R475GEN,
    R476GEN,R477GEN,R478GEN,R479GEN,R480GEN,R481GEN,
    R482GEN,R483GEN,R484GEN,R485GEN,R486GEN,R487GEN,
    R488GEN,R489GEN,R490GEN,R491GEN,R492GEN,R493GEN,
    R494GEN,R495GEN,R496GEN,R497GEN,R498GEN,R499GEN,
    R500GEN,R501GEN,R502GEN,R503GEN,R504GEN,R505GEN,
    R506GEN,R507GEN,R508GEN:
        (POS:POSITION;
        SONS:NODEPTR;
        case NTSEL:NTNRS of
        N199SpecificationText:(
            A199152inputSorts:T163SortListType;
            A199153inputOperations:T172OperationListType);
        N259SortName:(
            A259154outputSort:T158SortType);
        N260OperationName:(
            A260252sorts:T163SortListType;
            A260218operations:T172OperationListType;
            A260261variables:T196VariableListType;
            A260217noOfArguments:T87INT;
            A260170position:T164PositionType;
            A260263unique:T88BOOL;
            A260228operation:T167OperationType);
        N265ProcessName:(
            A265155outputProcess:T184ProcessType);
        N288Expression:(
            A288154outputSort:T163SortListType);
        N298ActualGates:(
            A298299expectedGates:T186GateListType);
        N313ParallelOperator:(
            A313156outputGates:T186GateListType);
```

```
        N320HideOperator:(
            A320156outputGates:T186GateListType);
        N335EquationSection:(
            A335152inputSorts:T163SortListType;
            A335153inputOperations:T172OperationListType);
        N350Equation:(
            A350261variables:T196VariableListType);
        N354Process:(
            A354252sorts:T163SortListType);
        N357Parameters:(
            A357185gates:T186GateListType;
            A357156outputGates:T186GateListType;
            A357261variables:T196VariableListType;
            A357187functionality:T188FunctionalityType);
        N367Local:(
            A367152inputSorts:T163SortListType;
            A367153inputOperations:T172OperationListType);
        N368Locals:(
            A368369allProcesses:T192ProcessListType);
        N256BehaviourSymbol,N257OperationIdentifier,
        N258GateName,N266VariableList,N269VariableItem,
        N270Variables,N271GateList,N272GateItem,N275Gates
        ,N276GateSet,N283ExpressionList,N290Term,
        N291SortIndicator,N293Atom,N294ExitParameters,
        N295ProcessCall,N296Behaviour,N297ExitParameter,
        N302ActionPrefix,N303Action,N305ExperimentList,
        N306Predicate,N307Experiment,N308Premiss,
        N309Guard,N310Choice,N311ChoiceOperator,
        N312Parallel,N314Disable,N315DisableOperator,
        N316Enable,N317EnableOperator,N319Hide,N321Par,
        N322Sum,N323SumDomain,N324Let,N325LetEquationList
        ,N327LetEquation,N328PExpression,N329TypeUnion,
        N330PSpecification,N331ActReplacement,
        N332Replacement,N333SortList,N334OperationList,
        N336SortPairs,N337OperationPairs,N338SortPair,
        N339OperationPair,N340NewSort,N341OldSort,
        N342NewOperation,N343OldOperation,N344Operation,
        N345OperationNameList,N346Domain,N347Result,
        N348OperatorName,N349EquationOfSortList,
        N351EquationListTail,N352PremissList,
        N353SimpleEquation,N358Block,N359Functionality,
        N360ExitSortList,N362DataType,N363LibraryList,
        N370Definition,N371ExternalLibrary,
        N372Specification,N375DataTypeList:());
    TERMRULE:(
        TERMSEL:TNRS;
        TERMAT:TERMATTYPE);
    LISTEND,EMPTYRULE:()

end;
```

Appendix F.

The Schema Defined for LOTTE

```
--*************************************************************
--*     Title   -       lotte                                 *
--*                                                           *
--*     Purpose -       To set up two-tier schema to store the *
--*                     abstract syntax tree used by the final *
--*                     phase of the SSS program and the GSR  *
--*                     program.                              *
--*                                                           *
--*     Written By-     Sean Mac Roibeaird, Dublin City       *
--*                                         University        *
--*                                                           *
--*     Date Started-   18th May 1989                         *
--*                                                           *
--*     Modifications -                                       *
--*             Initials    Date        Details               *
--*                                                           *
--*************************************************************

new_sds lotte is

--*************************************************************
--*             Imports from other SDSs
--*************************************************************

Import  eclipse-fine_structure_object as fso;
Import  sys-object as object;
Import  env-project as project;
Import  eclipse-summary_key;
Import  eclipse-summary_count;


--*************************************************************
--* Define the 'tree' to be a subtype of a fine_structure_object
--*************************************************************

tree            <=      fso;


--*************************************************************
--*         Define the layout of the Second tier
--*************************************************************

tree            =>>
[
        ENTITY  ::=     UNIT | NODETYPE | AG_NODETYPE;

        UNIT    =>      tree_type  : Integer,
                        with_attrs : Integer;

        NODETYPE ::=    list_node | list_end | empty_node | ag_node
                        | term_node;
```

```
AG_NODETYPE ::= name_type | sort_name| process_name
              | expression_details | parallel_operator
              | hide_operator | parameter_details;

NODETYPE =>     brother : NODETYPE;
              rule    : Integer;

list_end =>;

empty_node =>;

list_node =>    elem    : NODETYPE;

term_node  =>   termsel : Integer,
              termat   : Seq Of Integer;

ag_node =>      sons    : NODETYPE;
              ntsel   : Integer,
              line    : Integer,
              column  : Integer,
              sort    : sort_name,
              process : process_name,
              expr    : Seq Of sort_name,
              par_op  : Seq Of name_type,
              hide_op : Seq Of name_type,
              params  : Seq Of name_type;

name_type =>    name       :    Seq Of Integer;

sort_name =>    name       :    Seq Of Integer,
              type       :    Seq Of Integer,
              line       :    Integer,
              level      :    Integer

process_name => name       :    Seq Of Integer,
              gates      :    Seq Of name_type,
              arguments  :    Seq Of sort_name,
              discr      :    Integer,
              sort_list  :    Seq Of sort_name,
              noexit     :    Integer,
              line       :    Integer,
              valid      :    Boolean;

]; -- end of tree
```

Appendix G.

Details of LOTTE integration into Eclipse

What follows in this section are some examples of the Eclipse DBI code written to enable the LOTTE syntax tree data to be integrated into Eclipse. Before any data storage/retrieval can occur one must logon onto the database as follows -

DBI_logon (WORKAREA,"");

WORKAREA is a pathname identifying the location of an object of type **workarea** which will have been previously created.

The actual object which will contain the data must then be created. From the schema shown in Appendix F it can be seen that the fine-structure object will be of type 'tree'. As this is a first tier object one can use the PCTE command 'crobj' as follows -

```
crobj (LOTTE_ORIGIN,
       ROOT_LINK,
       NODE_TYPE,
       FLAGS);
```

The parameters used are as follows -

LOTTE_ORIGIN	This is the pathname to the directory in which the 'tree' object will reside.
ROOT_LINK	This is the name of the link from the LOTTE_ORIGIN directory to the **tree** object created.
NODE_TYPE	This is a string containing the name of the node type ,i.e. tree.

The **crobj** command can be replaced by the Eclipse interface function **DBI_create_entity**. The use of crobj is easier, in this case since DBI_create_entity requires that one has navigated down to the tree objects parent object before creating the tree object. The crobj command, on the

other hand, accepts a pathname to the tree object, as a parameter.

The basic algorithm used to access the nodes of the LOTTE syntax tree is a pre-order walk through the trees. Based on the rule value associated with the node , the program knows the node type and thus calls the appropriate external C function to create the correct node entity in the 'tree' object. The simplest example of this occurs for the following node types

Empty Node

List End Node

List Node

This is because these nodes have no attributes associated with them. The C function used to create these node types could be as follows -

```
create_noparm_node (rule , entity_type ,attribute )
int  rule;                              /* rule value for this node   */
DBI_ENTITY_TYPE_NAME    entity_type;    /* name of the entity type in */
                                        /* the  schema.*/
DBI_ATTRIBUTE           attribute;      /* Name of the link to this   */
                                        /* entity type */
{
DBI_ATTRIBUTE_VALUE     avalue;         /* Eclipse defined structure which */
                                        /* stores the type & address of the*/
                                        /* data item stored in/retrieved  */
                                        /* from the database  */
DBI_ENTITY              token;          /* The token of the node entity  */
                                        /* created. */

/*************************************************************************
 *  Create the node entity - Returns the token associated with the
 *  created ,i.e. token
 *************************************************************************/

     DBI_create_entity (entity_type,
                        attribute,
                        &token);

/*************************************************************************
 *      Store the rule value associated with this node in the 'rule'
 *  attribute of the node (consult schema to see all the attributes of
 *  nodes). This will distinguish nodes when they are retrieved.
 *************************************************************************/

       avalue.attr_type = DBI_INTEGER_SCALAR;  /* 'rule' is an integer */
       avalue.attr_value.v_int = &rule;
       DBI_attribute_becomes (token,
                              "rule",
                              avalue);
}
```

Both terminal nodes and attributed grammar nodes have attributes associated with them. Terminal node attributes are constant in type , but attributes associated with attributed-grammar nodes vary depending on the value of a selector attribute.

When the GSR is invoked , the first thing it does is build its syntax tree from the data stored in the database. This is almost a mirror image operation to the operation performed when storing the data. The type of node created in the internal syntax tree is dependant on the rule value associated with the equivalent node in the database. Thus a call which would retrieve the rule would be as follows-

```
DBI_value_of_attribute (token,
                        attribute,
                        &buffer,
                        &avalue)
rule = *avalue.attr_value.v_int;
```

The parameters used are as follows -

token The token of the node entity.

attribute The name of the attribute ,i.e. "rule".

&buffer The address of a block of memory used to perform the operation.

&avalue The address of an Eclipse-defined structure which contains the type and address of the value retrieved.

Once the rule value is known , the GSR knows the node type and thus the attribute types , if any , that should be read in from the database. Once the node type is known the GSR also knows which links should be created in the syntax tree.

Appendix H.

Example CRL Source.

```
; algebraic of natural numbers

typel int) int)
    (oper add (typel int int) int)
 )
 (axioml
    (axiom (opapp add int (expl (id x int) (opapp zero int (expl))))
           (id x int)
    (condl))
    (axiom (opapp add int (expl (id x int) (opapp succ int (expl (id y int)))))
           (opapp succ int (expl (opapp add int (expl (id x int) (id y int)))))
    (condl)) ; only data part !
 )
 (actdecl)
 (synchl)
 (setdecl)
 (rendecl)
 (objdecl)
 (acteffl)
 (pnamdecl)
 (createfl)
 (epsilon)
)
```

Appendix I.

The Original Data Structures Used in the CRL-C tool.

```
#define Tree_NULL 0

union hAnotType {
        int i ;
        char *s ;
          } ;

struct hValue{
         int name ;
         int type ;
         struct hValue *next ;
         union hAnotType val ;
            } ;

typedef struct hValue Value ;
typedef Value *pValue ;

struct hNode{
        int name ;
        struct hNode *next ;
        struct hNode *first ;
        struct hNode *back ;
        pValue value ;
        pValue anot ;
          } ;

typedef struct hNode Node ;
typedef Node *pNode ;
```

Appendix J.

The definition of a Parse Tree Used In the CRL-C Tool.

```
Structure bell_tree Is

    ENTITY ::=    UNIT | node | value | annotation;

    node    =>
                  name   :    Integer,
                  next   :    node,
                  first  :    node,
                  back   :    node,
                  val    :    value,
                  anot   :    value;

    value   =>
                  name   :    Integer,
                  type   :    Integer,
                  next   :    value,
                  val    :    annotation;

    annotation =>
                  int    :    Integer,
                  char_str :  String;

End;
```

Appendix K.

The Schema Definition for the Classified Database.

```
Structure Sds wp3 Is

Import From env
      Entity project;

Import From eclipse
      Structure IDLE idle_object
      String summary_key,name;


classified <= idle_object;

project =>> spec (name) : classified;


classified =>>
[
ENTITY  ::=     UNIT | OBJECTS | IPEPS;

OBJECTS ::=     system | interaction_point | message | field |
              env | function | event | miscellaneous ;

IPEPS ::=  interaction_point | env | system ;

Type action_rule Is (SEQUENCE,
                     PARALLEL,
                       CHOICE,
                       ITERATION,
                       NONE);

Type misc_type Is (PERFORMANCE,
                     CONFIGURATION,
                     REF_CONFIG,
                     SYS_RESOURCE,
                     SALES_RESOURCE,
                     SECURITY,
                     SERVICE_QUALITY,
                     DATA_PROTECT,
                     ORGANISATION,
                     POLICY,
                   DISTRIBUTION,
                     OTHER);

Type field_type Is (INTEGER,
                    BOOLEAN,
                    STRING);
```

```
OBJECTS =>
        name            :       String,
        misc            :       Seq Of miscellaneous;
system =>
        int             :       Seq Of interaction_point,
        func            :       Seq Of function,
        envs            :       Seq Of env;

interaction_point =>
        initial         :       Integer,
        max             :       Integer;

env => ;

message =>
        fields          :       Seq Of field,
        origin          :       Seq Of IPEPS,
        dest            :       Seq Of IPEPS;

function =>
        inputs          :       Seq Of message,
        outputs         :       Seq Of message,
        descr           :       String,
        funcs           :       Seq Of function,
        events          :       Seq Of event;

event =>
        rule            :       action_rule,
        choice          :       String,
        value           :       String,
        details         :       String,
        events          :       Seq Of event,
        type            :       Integer;


field =>
        optional        :       Boolean,
        data_type       :       field_type,
        value           :       String;

miscellaneous =>
        type            :       misc_type,
        descr           :       String;

];
End;
```

Appendix L.

Extract from the FDL which generates the UI for entering data into the Classified Database.

```
.PANE [630]
--
--      The only field to be displayed initially is the system name field
--
        U) misc (_off)
+           inter_pt1 (_off)
+           inter_pt2 (_off)
+           inter_pt3 (_off)
+           inter_pt4 (_off)
+           func1    (_off)
+           func2    (_off)
+           func3    (_off)
+           env1     (_off)
+           env2     (_off)
+           env3     (_off)
+           env4     (_off)

            .TRIM          (0,0)              |System Update|
name:        .SIGN       [631] (2,1)20       |System Name|
store:      .BUTTON      [632] (1,115)             |Store System Update|
exit:       .BUTTON      [633] (5,115)       |Exit System Update|
misc:       .BUTTON      [634] (9,115)       |Miscellaneous Data|
inter_pt1:  .SCALAR      [635] (7,1)20    (7,30)20    |Interaction Points|
inter_pt2:  .SCALAR      [636] (8,1)20    (8,30)20    |                  |
inter_pt3:  .SCALAR      [637] (9,1)20    (9,30)20    |                  |
inter_pt4:  .SCALAR      [638] (10,1)20   (10,30)20   |                  |
func1:      .SCALAR      [639] (7,55)15    (7,80)20   |Functions|
func2:      .SCALAR      [640] (8,55)15    (8,80)20   |         |
func3:      .SCALAR      [641] (9,55)15    (9,80)20   |         |
env1:       .SCALAR      [642] (12,55)15   (12,80)20  |Environments|
env2:       .SCALAR      [643] (13,55)15   (13,80)20  |            |
env3:       .SCALAR      [644] (14,55)15   (14,80)20  |            |
env4:       .SCALAR      [645] (15,55)15   (15,80)20  |            |
.end [630]

.SIGN [631]
        S)  ^[A-Za-z_0-9]*$
        U)  ('WP3_create_entity &[/W1/F1/Sys_Pane] '.!system^name' 'system' '!system' '2'')
+           misc (_on)
+           inter_pt1 (_on)
+           inter_pt2 (_on)
+           inter_pt3 (_on)
+           inter_pt4 (_on)
+           func1    (_on)
+           func2    (_on)
+           func3    (_on)
+           env1 (_on)
+           env2 (_on)
+           env3 (_on)
+           env4 (_on)
```

```
        E)  'Must start with an Alphabetic character and then contain only AlphaNumerics')
.end

.BUTTON [632]
        A) WP3_store_sys_data &[/W1/F1/Sys_Pane]
        U)   name ()
+              inter_pt1 ()
+              inter_pt2 ()
+              inter_pt3 ()
+              inter_pt4 ()
+              func1     ()
+              func2     ()
+              func3     ()
+              env1 ()
+              env2 ()
+              env3 ()
+              env4 ()
+              misc (_off)
+              inter_pt1 (_off)
+              inter_pt2 (_off)
+              inter_pt3 (_off)
+              inter_pt4 (_off)
+              func1     (_off)
+              func2     (_off)
+              func3     (_off)
+              env1 (_off)
+              env2 (_off)
+              env3 (_off)
+              env4 (_off)
.end

.BUTTON [633]
        U) /W1/F1/Sys_Pane (_off) /W1/F1/Tool_Pane(_on)
+            name ()
+              inter_pt1 ()
+              inter_pt2 ()
+              inter_pt3 ()
+              inter_pt4 ()
+              func1     ()
+              func2     ()
+              func3     ()
+              env1 ()
+              env2 ()
+              env3 ()
+              env4 ()
+              misc (_off)
+              inter_pt1 (_off)
+              inter_pt2 (_off)
+              inter_pt3 (_off)
+              inter_pt4 (_off)
+              func1     (_off)
+              func2     (_off)
+              func3     (_off)
+              env1 (_off)
+              env2 (_off)
```

```
+            env3 (_off)
+            env4 (_off)
.end

.BUTTON [634]
        U) /W1/F1/Sys_Pane (_off) /W1/F1/Misc_Pane (_on)
+           ('WP3_get_misc_iterator')
+           ('WP3_display_misc_data &[/W1/F1/Misc_Pane]')
.end
.SCALAR [635]
        U) inter_pt2 (_do)
.end
.SCALAR [636]
        U) inter_pt3 (_do)
.end
.SCALAR [637]
        U) inter_pt4 (_do)
.end
.SCALAR [638]
        U) func1 (_do)
.end
.SCALAR [639]
        U) func2 (_do)
.end
.SCALAR [640]
        U) func3 (_do)
.end
.SCALAR [641]
        U) env1 (_do)
.end
.SCALAR [642]
        U) env2 (_do)
.end
.SCALAR [643]
        U) env3 (_do)
.end
.SCALAR [644]
        U) env4 (_do)
.end
```

Appendix M.

Sample BIFs and functions developed to store sub-system data in the Classified Database.

```
/*************************************************************************
 *
 *      Procedure name  :       WP3_create_entity
 *
 *      Purpose         :       This BIF tries to create a second tier
 *                              entity
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *     Written By       :       Sean Mac Roibeaird, Dublin City
 *                                              University.
 *
 *      Date            :       12th October 1989
 *
 *************************************************************************/

WP3_create_entity (argc,argv)
int     argc;
char    *argv[];
{
    AI_ITEM_TOKEN       pane_token;
    AI_TEXT             name;
    DBI_ATTRIBUTE_VALUE         avalue;
    char                search_string[LINKSIZE];
    int                 pane_number;
    EI_BOOL             flag;
    int                 result;


/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);

/*************************************************************************
 *          Get the Values in the Name Field
 *************************************************************************/

        CHECK_RESULT(get_field (pane_token,"name",&name,(int *)0,FALSE));


    strcpy(search_string,name);
    strcat(search_string,argv[2]);
```

```
    if((value_of_attribute (DBI_ENTITY_SCALAR,
                            local_root,
                            search_string,
                            &avalue)) != EI_OK)
        {
        CHECK_RESULT (create_wp3_entity (argv[3],
                                         argv[4],
                                         name,
                                         &entity_token));
        }
else
        {
        entity_token = *avalue.attr_value.v_entity;
        pane_number = atoi (argv[5]);
        switch (pane_number)
                {
                        case    ENV:
                                        break;
                        case    SYS:
                                        display_system (pane_token);
                                        break;
                        case    INT_POINT:
                                    display_intpoint (pane_token);
                                        break;
                        case    MESSAGE:
                                        display_message(pane_token);
                                        break;
                        case    FIELD:
                                        display_field(pane_token);
                                        break;
                        case    FUNCTION:
                                        display_function(pane_token);
                                        break;
                        case    EVENT:
                                        display_event(pane_token);
                                        break;

                        default:
                                        break;
                }

        }
    return (EI_OK);
}
```

```
/*************************************************************************
 *
 *      Procedure name  :       WP3_store_sys_data
 *
 *      Purpose         :       This BIF tries to store the system data
 *                              into the two-tier database.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                              University.
 *
 *  Date                :       16th October 1989
 *
 *************************************************************************/

WP3_store_sys_data (argc,argv)
int  argc;
char        *argv[];
{
        AI_ITEM_TOKEN           pane_token;
        AI_TEXT                 interaction_point[6];
        AI_TEXT                 channel[4];
        AI_TEXT                 func[6];
        int                     i;
        char                    inter_pt[10];
        char                    funcs[10];
        DBI_ENTITY              int_token;
        DBI_ENTITY              func_token;
        DBI_ATTRIBUTE_VALUE     avalue;

/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);

/*************************************************************************
 *          Retrieve the names of the Interaction Points
 *************************************************************************/

    CHECK_RESULT (store_seq_attribute (4,
                                       "inter_pt ",
                                       pane_token,
                                       "interaction_point",
                                       "!interaction_point",
                                       "int",
                                       ".!interaction_point^name")
                  );
        CHECK_RESULT (store_seq_attribute (3,
                                           "func ",
                                           pane_token,
                                           "function",
                                           "!function",
                                           "func",
                                           ".!function^name")
                      );
```

```
        CHECK_RESULT (store_seq_attribute (4,
                                        "env ",
                                        pane_token,
                                        "env",
                                        "!env",
                                        "envs",
                                        ".!env^name")
                    );
    return (EI_OK);
}

/*****************************************************************************
 *
 *      Procedure name  :       WP3_get_misc_iterator
 *
 *      Purpose         :       This BIF gets the iterator for the 'misc'
 *                              attribute of the current entity.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                      University.
 *
 *      Date            :       20th October 1989
 *
 *****************************************************************************/

WP3_get_misc_iterator(argc,argv)
int     argc;
char    *argv[];
{
        DBI_ATTRIBUTE_VALUE     avalue;


       CHECK_RESULT(value_of_attribute(
                        DBI_ITERATOR_VALUE,
                        entity_token,
                        "misc",
                        &avalue));
        misc_iterator = *avalue.attr_value.v_iterator;
    return (EI_OK);
}
```

```
/*************************************************************************
 *
 *      Procedure name  :       WP3_display_misc_data
 *
 *      Purpose         :       This BIF gets the token of the next 'misc'
 *                              entity in the sequence and then displays
 *                              its text.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                              University.
 *
 *      Date            :       20th October 1989
 *
 *************************************************************************/

WP3_display_misc_data (argc,argv)
int  argc;
char        *argv[];
{

    AI_ITEM_TOKEN         pane_token;
    DBI_ATTRIBUTE_VALUE         avalue;
    DBI_BUFFER                  buffer;
    char                        space[256];
    int                         ret;

/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);

        buffer.bf_size = 256;
        buffer.bf_pointer = space;
        if ((ret = DBI_next (misc_iterator,
                             &buffer,
                             &avalue)) != EI_OK)
            misc_token = NO_MORE_MISCS;
        else
            {
            misc_token = *(avalue.attr_value.v_entity);
            display_dbi_entity (misc_token,
                                "descr",
                                pane_token,
                                "data");
            }
    return (EI_OK);
}
```

```
/*************************************************************************
 *
 *      Function name  :       create_wp3_entity
 *
 *      Purpose        :       This function creates a second tier
 *                             and stores a string in its name attribute.
 *
 *      Return         :       0 - if everything works
 *                             1 - if incomplete data
 *
 *      Written By     :       Sean Mac Roibeaird, Dublin City
 *                                                 University.
 *
 *      Date           :       11th October 1989
 *
 *************************************************************************/

create_wp3_entity(entity_type,attribute,name,e_token)
DBI_ENTITY_TYPE_NAME entity_type;
DBI_ATTRIBUTE        attribute;
char                 *name;
DBI_ENTITY           *e_token;
{
    DBI_ATTRIBUTE_VALUE      avalue;
    DBI_ENTITY               token;

    CHECK_RESULT (create_entity (
                        local_root,
                        entity_type,
                        attribute,
                        &token));
    *e_token = token;

/*************************************************************************
 *          Store the name value
 *************************************************************************/

        avalue.attr_type = DBI_STRING_SCALAR;
        avalue.attr_value.v_str = name;
        CHECK_RESULT (attribute_becomes (
                            token,
                            "name",
                            avalue));
    return (EI_OK);
}
```

```
/*************************************************************************
 *
 *      Function name  :        store_seq_attribute
 *
 *      Purpose        :      This function retrieves a sequence of
 *                          values from the user interface and stores
 *                          them in a sequence attribute in the
 *                          database.
 *
 *      Return         :       0 - if everything works
 *                          1 - if incomplete data
 *
 *      Written By     :       Sean Mac Roibeaird, Dublin City
 *                                              University.
 *
 *      Date           :       11th October 1989
 *
 *************************************************************************/

store_seq_attribute(seq_size,
                    field_name,
                    pane_token,
                    entity_type_name,
                    attribute,
                    seq_attribute,
                    derived_attribute)
int                         seq_size;          /* # elements in sequence    */
AI_TEXT                     field_name;          /* field name on UI pane     */
AI_ITEM_TOKEN               pane_token;          /* token of UI pane          */
DBI_ENTITY_TYPE_NAME        entity_type_name;    /* the type of 2nd tier entity */
DBI_ATTRIBUTE               attribute;          /* the attribute for storage */
char*                       seq_attribute;       /* sequence attribute        */
char*                       derived_attribute;   /* derived attribute for entity */
                                                 /* name                      */
{

    AI_TEXT                 field_value;
    DBI_ENTITY              token;
    DBI_ATTRIBUTE_VALUE         avalue;
    int                     i;
    char                    search_string[LINKSIZE];
    int                     length;
    char                    field[20];

/*************************************************************************
 *              Retrieve the values from screen
 *************************************************************************/

    strcpy (field,field_name);
```

```
        for (i = 1; i <= seq_size; i++)
            {
         if (i < 10)
              field[(strlen(field_name)) - 1] = (char)i+0x30;
         else
              {
              field[(strlen(field_name)) - 1] = '1';
                field[strlen(field_name)] = (char)(i-10)+0x30;
                field[(strlen(field_name)) + 1] = '\0';
              }
         CHECK_RESULT (get_field (pane_token,
                                  field,
                                  &field_value,
                                  (int *)0,
                                  FALSE));
         if (field_value[0] == '\0')
              strcpy(field_value,"null");

/*************************************************************************
 *          Does the Entity Exist ?
 *************************************************************************/

         strcpy(search_string,field_value);
         strcat(search_string,derived_attribute);
            if(value_of_attribute (
                              DBI_ENTITY_SCALAR,
                              local_root,
                              search_string,
                              &avalue) == EI_OK)
                        token = *avalue.attr_value.v_entity;

/*************************************************************************
 *             Create a New Entity
 *************************************************************************/
         else
                        {
                        CHECK_RESULT (create_wp3_entity (
                                        entity_type_name,
                                attribute,
                                        field_value,
                                        &token));
                        }
/*************************************************************************
 *          Add to the tail of the sequence
 *************************************************************************/

            avalue.attr_type = DBI_ENTITY_SCALAR;
            avalue.attr_value.v_entity = &token;
            CHECK_RESULT (add_to_head (
                                        entity_token,
                                        seq_attribute,
                                        avalue));
         CHECK_RESULT (release_entity_token (token));
            }
      return (EI_OK);
}
```

```
/*****************************************************************************
 *
 *      Procedure name  :       get_field
 *
 *      Purpose         :       This routine gets the value of a field
 *                              given its container,and it FDL name.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                                  University.
 *
 *      Date            :       18th April 1989
 *
 *****************************************************************************/

get_field (container,fdlname,buffer,choice_id,choice)
AI_ITEM_TOKEN       container;
char                *fdlname;
AI_TEXT     *buffer;
AI_CHOICE_ID        *choice_id;
EI_BOOL             choice;
{

        AI_ITEM_TOKEN   field_token;
        AI_CHOICE_ID        id;

        CHECK_RESULT (AI_get (container,
                              AI_CHILD_BY_ID,
                              fdlname,
                              &field_token,
                              0)
                      );
    if (choice)
            {
            CHECK_RESULT (AI_get (field_token,
                                  AI_VALUE,
                                  &id,
                                  0)
                          );
              *choice_id = id;
            }
        else
            {
            CHECK_RESULT (AI_get (field_token,
                                  AI_VALUE,
                                  buffer,
                                  0)
                          );
            }
    return (EI_OK);
}
```

Appendix N.

The GDL file defining the Context Diagram shapes and semantics.

```
Method context_diagram

--
-- Declare shapes to be used
--
--
-- An External Entity is represented as a Box
--
Shape EXT_S Is
{
        Box         0,0 : 90,90
}
--
-- A System is represented as a Rounded Box
--
Shape SYS_S Is
{
        Rounded Box     0,0 : 120,120 Bold
}
--
-- Three types Links are used
--
Linkstyle MESS_L Is Start Arrow

Linkstyle MESS_L Is End Arrow

Linkstyle MESS_L Is Start Arrow End Arrow
```

```
--
-- Declare forward types to be used in type declarations
--

Forward message Is LINK


--
-- Declare basic types
--

Type PROC_ENV Is NODE
        (in_links: In Bag Of message;
         out_links : Out Bag Of message)


Type external_entity Is PROC_ENV

Type system Is PROC_ENV

Type message  Is LINK
       (source_end : In PROC_ENV;
        dest_end   : Out PROC_ENV)
```

```
--
-- Use declarations
--
For external_entity Use
{
        SYMBOL(EXT_S)
    ++ Ext_name(STRING) : "attribute = ext_name"
}

For system Use
{
        SYMBOL(SYS_S) : "open"
    ++ SYS_NAME(STRING) : "attribute = sys_name"
}

For message Use
{
        SYMBOL(MESS_L)
    ++ MESS_NAME(STRING) : "attribute = mess_name"
}


-- End of EXAMPLE GDL
```

Appendix O.

The GDL file defining the Data-Flow Diagram shapes and semantics.

```
Method data_flow_diagram

--
-- Declare shapes to be used
--
--
-- Functions are represented by circles
--
Shape FUNC_S Is
{
        Ellipse         0,0 : 100,100
}
--
-- External Entities are represented by boxes
--
Shape EXT_S Is
{
        Box     0,0 : 100,90
}
--
-- Data stores are represented by parallel lines
--
SHAPE STORE_S Is
{

    Line        0,0 : 100,0
    ++Line      0,20 : 100,20
}

Linkstyle DATA_FLOW Is Start Arrow

Linkstyle DATA_FLOW Is End Arrow

Linkstyle DATA_FLOW Is Start Arrow End Arrow

--
-- Declare forward types to be used in type declarations
--


Forward data_flow Is LINK

--
-- Declare basic types
--

Type PROC_EXT_STORE Is NODE
        (in_links: In Bag Of data_flow;
         out_links : Out Bag Of data_flow)
```

```
Type external Is PROC_EXT_STORE

Type function Is PROC_EXT_STORE

Type data_store Is PROC_EXT_STORE

Type data_flow Is LINK
       (source_end : In PROC_EXT_STORE;
        dest_end   : Out PROC_EXT_STORE)


--
-- Use declarations
--

For external Use
{
       SYMBOL(EXT_S)
     ++ EXT_NAME(STRING) : "attribute = ext_name"
}


For function Use
{
       SYMBOL(FUNC_S) : "open"
     ++ FUNC_NAME(STRING) : "attribute = func_name"
}

For data_store Use
{
       SYMBOL(STORE_S)
     ++ STORE_NAME(STRING) : "attribute = store_name"
}

For data_flow Use
{
       SYMBOL(DATA_FLOW)
     ++ FLOW_NAME(STRING) : "attribute = flow_name"
}


-- End of Data Flow Diagram GDL
```

Appendix P.

The DDL file for the Context Diagram.

```
Structure Sds context_diagram Is

    Import From eclipse
     Entity dir
     String name;

    Import From mmi
        Structure IDLE diagram_object
        String drawing_name,
               check_status ;

    drawing <= diagram_object ;
         -- drawing inherits fine structure of diagram_object
--
-- Declare the first tier link to the context diagram object
--
    dir =>>
     dgm(name) : drawing;

    drawing =>
            drawing_name,
            check_status;

    drawing =>>
    [
--
-- Declare the node types in the diagram
--
        NODE ::=
             external_entity | system;


     external_entity =>
             ext_name : String;

     system =>
             sys_name : String;

--
-- Declare the link types in the diagram
--
        LINK ::=
            message;

     message =>
             mess_name : String;
    ] ;

End ; -- Context Diagram
```

Appendix Q.

The DDL file the Data-Flow Diagram.

```
Structure Sds data_flow_diagram Is

    Import From eclipse
     Entity dir
     String name;

    Import From mmi
        Structure IDLE diagram_object
        String drawing_name,
               check_status ;

    dfd_diagram <= diagram_object ;
         -- drawing inherits fine structure of diagram_object

--
--   Declare the first tier link to a data flow diagram object.
--
    dir =>>
     dfd(name) : dfd_diagram;

    dfd_diagram =>
           drawing_name,
           check_status;

    dfd_diagram =>>
    [
--
--  Declare the node types in the diagram
--
     NODE ::=
           external | function | data_store;

     external =>
           ext_name : String;

     function  =>
           func_name : String;

     data_store =>
             store_name : String;
--
--  Declare the link types in the diagram
--
     LINK ::=
           data_flow;


     data_flow =>
           flow_name : String;
    ] ;

End ; -- Data Flow Diagram
```

Appendix R.

Menus and state selectors changed in the Context Diagram FDL file.

```
.STATE [203]            ----- The New Symbol State Selector
        UPDATE)('DE_new_symbol ?[newsym]')
        (1)Start Arrow
                DATA)MESS_L.1
        (2)End Arrow
                DATA)MESS_L.2
       (3)Both Arrows
                DATA)MESS_L.3
.end [203]

.STATE [204]            ----- The Current Symbol State Selector
        UPDATE)('DE_current_symbol ?[currsym]')
        (1)Start Arrow
                DATA)MESS_L.1
        (2)End Arrow
                DATA)MESS_L.2
       (3)Both Arrows
                DATA)MESS_L.3
.end [204]

.MENU [212]             ----- The Add Subordinate Menu
        UPDATE)('DE_add_subordinate ?[addsub]')
        (1)External Entity Name
                DATA)+T/EXT_NAME:External Entity Name
        (2)System Name
                DATA)+T/SYS_NAME:System Name
        (3)Message
                DATA)+L/message:Message
        (4)Message Name
                DATA)+T/MESS_NAME:Message Name
        (5)Line Annotation
                DATA)+A/LINE:Line Annotation
        (6)Box Annotation
                DATA)+A/BOX:Box Annotation
        (7)Text Annotation
                DATA)+A/TEXT:Text Annotation
.end [212]

.MENU [214]             ----- The Add Primary Menu
        UPDATE)('DE_add_primary ?[addmain]')
        (1)External Entity
                DATA)+N/externa_entity:External Entity
        (2)System
                DATA)+N/system:System
        (3)Line Annotation
                DATA)+A/LINE:Line Annotation
        (4)Box Annotation
                DATA)+A/BOX:Box Annotation
        (5)Text Annotation
                DATA)+A/TEXT:Text Annotation
.end [214]
```

```
.MENU [213]          ----- Current Object Operations Menu
        (1)Add Waypoint
                UPDATE)('DE_add_waypoint')
                DATA)+W
        (2)Delete Waypoint
                UPDATE)('DE_delete_waypoint')
                DATA)-W
        (3)Edit Text
                UPDATE)('DE_edit_text')
                DATA)EDIT
        (4)Stretch
                UPDATE)('DE_stretch')
                DATA)STRETCH
        (5)Delete
                UPDATE)('DE_delete')
                DATA)DELETE
        (6)Open
                UPDATE)('DE_open')
                DATA)OPEN
.end [213]
```

Appendix S.

Example Code used for prompting the user during the rigorisation process.

```
/*************************************************************************
 *
 *      Function  name  :       WP3_retrieve_system_fields
 *
 *      Purpose         :       This function retrieves the field values
 *                              of a system object.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                                  University.
 *
 *      Date            :       2nd December 1989
 *
 *************************************************************************/

WP3_retrieve_system_fields (argc,argv)
int  argc;
char *argv[];
{
        AI_ITEM_TOKEN           pane_token;
        DBI_ATTRIBUTE_VALUE     avalue;
        AI_TEXT                 system;
        DBI_ENTITY              token;
        DBI_ITERATOR            iterator;
        char                    link_name[LINKSIZE];

/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);

/*************************************************************************
 *              Get the Sytem Name Field
 *************************************************************************/

        CHECK_RESULT(get_field (pane_token,
                                "system",
                                  &system,
                                 (int *)0,
                                 FALSE));
    strcpy (system_name,system);
    locate_at_root (WORKAREA,&root);
    strcpy (link_name,system);
    strcat (link_name,".spec");
    locate_at_local_root (root,link_name);
```

```
/*************************************************************************
 *              Get the Sub-Systems for this System
 *************************************************************************/

        CHECK_RESULT(value_of_attribute(
                        DBI_ITERATOR_VALUE,
                        local_root,
                        "!system",
                        &avalue));
        iterator = *avalue.attr_value.v_iterator;

    retrieve_field_names (iterator);  /* lists the names of the systems */
    DBI_release_iterator_token (iterator);
    return (EI_OK);
}


/*************************************************************************
 *
 *      Function  name  :       WP3_retrieve_messages
 *
 *      Purpose         :       This function retrieves the list of
 *                              messages in the classified specification.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                                  University.
 *
 *      Date            :       5th December 1989
 *
 *************************************************************************/

WP3_retrieve_messages (argc,argv)
int     argc;
char    *argv[];
{
        AI_ITEM_TOKEN           pane_token;
        AI_TEXT                 system;
        char                    link_name[LINKSIZE];
        DBI_ITERATOR            iterator;
        DBI_ATTRIBUTE_VALUE     avalue;


/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);
```

```
/*****************************************************************************
 *              Get the Sub-Sytem Name Field
 *****************************************************************************/

        CHECK_RESULT(get_field (pane_token,
                                "system",
                                 &system,
                                (int *)0,
                                FALSE));
    strcpy (link_name,system);
        strcat (link_name,".spec");
        locate_at_root (WORKAREA,&root);
        locate_at_local_root (root,link_name);

        CHECK_RESULT(value_of_attribute (DBI_ITERATOR_VALUE,
                                         local_root,
                                         "!message",
                                         &avalue));
        iterator = *avalue.attr_value.v_iterator;
    retrieve_field_names (iterator);
    return (EI_OK);

}

/*****************************************************************************
 *
 *      Function  name  :       WP3_retrieve_message_fields
 *
 *      Purpose         :       This function retrieves the list of
 *                              messages in the classified specification.
 *
 *      Return          :       0 - if everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                                  University.
 *
 *      Date            :       5th December 1989
 *
 *****************************************************************************/

WP3_retrieve_message_fields (argc,argv)
int  argc;
char *argv[];
{
    AI_ITEM_TOKEN           pane_token;
    AI_ITEM_TOKEN           mess_token;
    DBI_ATTRIBUTE_VALUE     avalue;
    AI_TEXT                 messages;
    AI_TEXT                 system;
    char                    search_string[LINKSIZE];
    char                    link_name[LINKSIZE];
    DBI_ENTITY              token;
    DBI_ENTITY              field_token;
    DBI_ITERATOR            iterator;
```

```
/********** Convert the token argument to integer ***********/

        pane_token = (AI_ITEM_TOKEN) atoi(argv[1]);
        mess_token = (AI_ITEM_TOKEN) atoi(argv[2]);

/*************************************************************************
 *              Get the Sub-Sytem Name Field
 *************************************************************************/

        CHECK_RESULT(get_field (pane_token,
                                "system",
                                 &system,
                                (int *)0,
                                FALSE));

/*************************************************************************
 *              Get the Message Name Field
 *************************************************************************/

        CHECK_RESULT(get_field (pane_token,
                                "messages",
                                 &messages,
                                (int *)0,
                                FALSE));

        CHECK_RESULT(set_field (
                                mess_token,
                                "message",
                                messages,
                                0,
                                FALSE));
        strcpy (link_name,system);
        strcat (link_name,".spec");
        locate_at_root (WORKAREA,&root);
        locate_at_local_root (root,link_name);
        strcpy(search_string,messages);
        strcat(search_string,".!message^name");

        CHECK_RESULT(value_of_attribute (DBI_ENTITY_SCALAR,
                                         local_root,
                                         search_string,
                                         &avalue));
        token = *avalue.attr_value.v_entity;
        entity_token = token;
        CHECK_RESULT(value_of_attribute(
                                DBI_ITERATOR_VALUE,
                                token,
                                "fields",
                                &avalue));
        iterator = *avalue.attr_value.v_iterator;
        retrieve_field_names(iterator);
        DBI_release_iterator_token (iterator);
```

```
    CHECK_RESULT(value_of_attribute(
                        DBI_ITERATOR_VALUE,
                        token,
                        "origin",
                        &avalue));
    iterator = *avalue.attr_value.v_iterator;
    CHECK_RESULT(next_token (iterator,
                        &field_token));
    DBI_release_iterator_token (iterator);
    CHECK_RESULT(value_of_attribute(
                        DBI_STRING_SCALAR,
                        field_token,
                        "name",
                        &avalue));
    release_entity_token (field_token);
    CHECK_RESULT(set_field (
                         mess_token,
                         "origin",
                         avalue.attr_value.v_str,
                         0,
                         FALSE));
    CHECK_RESULT(value_of_attribute(
                        DBI_ITERATOR_VALUE,
                        token,
                        "dest",
                        &avalue));
    iterator = *avalue.attr_value.v_iterator;
    CHECK_RESULT(next_token (iterator,
                        &field_token));
    DBI_release_iterator_token (iterator);
    CHECK_RESULT(value_of_attribute(
                        DBI_STRING_SCALAR,
                        field_token,
                        "name",
                        &avalue));
    release_entity_token (field_token);
    CHECK_RESULT(set_field (
                        mess_token,
                        "destination",
                        avalue.attr_value.v_str,
                        0,
                        FALSE));
    DBI_release_iterator_token (iterator);
    return (EI_OK);
}
```

Appendix T.

The function called from the method handler to invoke the Data-Flow diagramming tool.

```
/*************************************************************************
 *
 *      Function  name  :       action_open
 *
 *      Purpose         :       This function invokes the DFD tool.
 *
 *      Return          :       0 - If everything works
 *                              1 - if incomplete data
 *
 *      Written By      :       Sean Mac Roibeaird, Dublin City
 *                                                  University.
 *
 *      Date            :       5th December 1989
 *
 *************************************************************************/

static int action_open ( label_value, label_id)
char *label_value;
char *label_id;
{

int             result;
int             status;
int             i,j;
struct          dynp_ds context;
struct          dynp_ds *dynp;
char            *params[6];
char            name[64];
char            fso_object[64];
char            temp[64];
DBI_ENTITY      label_entity;
DBI_ATTRIBUTE_VALUE avalue;
DBI_ENTITY      token;

/*
 * This procedure is called when a system is opened .
 */


        CHECK_RESULT(
                entity_token_from_id( label_id, &label_entity ) );

        CHECK_RESULT(
                value_of_attribute(
                        DBI_STRING_SCALAR,
                        label_entity,
                        "sys_name",
                        &avalue ) );
```

```
        if (strlen (avalue.attr_value.v_str) == 0)
                {
                fprintf (stderr,"SYSTEM NEEDS A NAME \n");
                fflush (stderr);
                return (EI_OK);
                }
        else
                {
                i = 0;
/*****************************************************************
 *   Use the system object name as a key to the DFD diagram
 *   PCTE object.
 *****************************************************************/

               strcpy (temp,avalue.attr_value.v_str);
                strcpy(name,system_name);
                strcat(name,"_");
                j = strlen(name);
                while (temp[i] != LF)
                        name [j++] = temp[i++];
                name[j] = '\0';
                strcat (name,".dfd");
                CHECK_RESULT(
                        value_of_attribute(
                                DBI_ENTITY_SCALAR,
                                database_root,
                                "_/.users/root.usr/.work",
                                &avalue ) );
                token = *avalue.attr_value.v_entity;
                create_entity(token,
                                "dfd_diagram",
                                name,
                                &token);

                strcpy(fso_object,"f=_/.users/root.usr/.work/");
                strcat(fso_object,name);

/*****************************************************************
 *       Set up the dynp structure to locate HOME
 *****************************************************************/

                context.dy_tsig = 0;
                context.dy_stdinput = (char *) 0;
                context.dy_stdoutput = (char *) 0;
                context.dy_typoutput = 0;
                context.dy_stderror = (char *) 0;
                context.dy_typerror = 0;
                context.dy_refcurr = (char *)0;
                dynp = &context;
```

```
/*****************************************************************
 *   Set up the parameters to invoke the DFD tool's process
 *****************************************************************/

            params[0] = "ex.tool";
            params[1] = "w=_/.users/root.usr/.work/rigorous.dbwork";
            params[2] = fso_object;
            params[3] = "d=dfd_diagram";
            params[4] = "s=A4";
            params[5] = (char *)0;

            callp ("_/eclipse.tools/ex.tool",
                    params,
                    environ,
                    dynp,
                    &status);
            }
        return (EI_OK);
}
```

Bibliography

[Alde1] A. Alderson, B. Bott, "Overview of the Eclipse Programme", ***Eclipse - An integrated project support environment - F. Bott (editor).***

[Alde2] A. Alderson, A. Elliott, "The Eclipse Tool Builder's Kit and the HOOD Toolset", ***Software Engineering Environments - Research and Practice*** - K. Bennett (editor).

[Beer] S. Beer, et al, "The Design Editor", ***Eclipse - an integrated project support environment.***

[Bolo] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS".

[Boud] G. Boudier, et al , "An Overview of PCTE and PCTE+", ***Proceedings of the Third ACM Symposium on Software Development Environments,*** (Nov 1988).

[Bune] P. Buneman, et al, "An implementation technique for database query languages", ***ACM transactions on Database Systems***, (June 1982).

[Cart1] J. Cartmell, B. Passingham, "Eclipse Database Functional Specifications".

[Cart2] J. Cartmell, A. Alderson, "The Eclipse Two-Tier Database", ***Eclipse - An Integrated Project Support Environment*** - *F Bott (editor).*

[Chen] P.P. Chen, "The Entity-Relationship Model: towards a unified view of data", ***ACM Transactions on Database Systems,*** (March 1976).

[Dart] Susan A. Dart, et al, "Software Engineering Environments", ***IEEE Computer,*** (Nov 1987).

[Gree] M. Green, "Format Description Language - Functional Specification", Dec 1987.

[Hays] B. Hayselden, et al, "Message Handling Facilities Functional Specification".

[Huds] S.E. Hudson, King, "The Cactis Project: Database Support for Software Environments", ***IEEE Transactions on Software Engineering***, (June 1988).

[John] D.B. Johnson, "Help Eclipse User Guide", March 1988

[Lari] G. Larini, "Classified Specification of the AI's node a'la DNL test case", ***Copyright by the SPECS Consortium*** (P.CSELT.WP3.30) - July 1989.

[Luba] M.D. Lubars, "The IDeA Design Environment", ***Proceedings of the 11th International Conference on Software Engineering***,(May 1989).

[McLe] W.B. McLean, S.M. Jefferson, "Eclipse Public Tools Interface Functional Specification".

[Nest] J. R.Nestor, et al, "IDL - Interface Description Language : Formal Descrition 2nd Edition", *Carnegie-Mellon University, Computer Science Department.*

[Notk] David Notkin, "The Relationship Between Software Development Environments and the Software Process", ***Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,***(Nov 1988).

[Ober] P.A. Oberndorf, "The Common Ada Programming Support Environment (APSE) Interface Set (CAIS)", *IEEE Transactions on Software Engineering* (June 1988).

[PCTE 85] "PCTE - A Basis for a Portable Common Tool Environment", ***Esprit Technical Week*** (1985).

[PCTE 86] "PCTE Functional Specifications 1.4", Bull, GEC, ICL, Nixdorf, Olivetti, Siemens, (Sept 1986).

[PCTE 88] "PCTE - The Essential Base for Computer Aided Software Engineering", ***Brochure of the PCTE Interface Management Board (PIMB),*** (Nov 1988).

[Pene1] M.H. Penedo & W.E. Riddle, "Software Engineering Environment Architectures", ***IEEE Transactions on Software Engineering,***(June 1988).

[Pene2] M.H. Penedo, et al, "Object Management Issues for Software Engineering Environments - Workshop Report", ***Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments***, (Nov 1988).

[Pott1] S. Potter, et al, "Interaction with Eclipse", *Eclipse - An integrated Support Environment*" - F. Bott (editor).

[Pott2] S. Potter, et al, "Tool Builder's Guide to the Eclipse User Interface".

[Reed] R. Reed, et al, "A Formal Technique Environment for Telecommunications Software", ***Proceedings of SETSS '89***.

[Robs1] K. Robson, et al, "Design Editor Functional Specification" - July 1988.

[Robs2] K. Robson, "Graph Description Language - Functional Specification" - Sept 1988.

[Ship] D.W. Shipman, "The functional data model and the data language DAPLEX", *ACM Transactions on Database Systems*, (March 1981).

[Sten] Vic Stenning, "On the Role of an Environment"

[Stone] Stoneman Document, US Department of Defense (1980), ***Requirements for Ada Programming Support Environments***.

[Stre] T. Strelich, "The Software Life Cycle Support Environment (SLCSE) - A Computer Based Framework for Developing Software Systems", ***Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments***, (Nov 1988).

[Tedd] M. Tedd, "PCTE+ - The Evolution of PCTE", ***Software Engineering Environments - Research and Practice*** - K. Bennett (editor).

[Thom] I. Thomas , "The PCTE Initiative and the PACT Project", ***Esprit Technical Week 1988.***

[Tull] C.J. Tully, "Prospects for Future Environments: Introduction to Panel Session", ***Proceedings of the 9th International Conference on Software Engineering.***